DARIUSZ TEFELSKI, ANGELIKA TEFELSKA

LABVIEW AND OPEN SOURCE SOLUTIONS

PUBLISHER: DARIUSZ TEFELSKI

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

PUBLISHED BY DARIUSZ TEFELSKI

This project has been funded with support from the European Commission. This communication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

First printing, March 2020

ISBN: 978-83-955851-0-4

Contents

List of Figures

List of Tables

This book is dedicated to doc. dr Wiesław Tłaczała,
who introduced LabVIEW, negotiated LabVIEW Academy
program and teached Principles of Virtual
Instruments Design for many years among
students of Faculty of Physics at Warsaw
University of Technology.

Introduction

National Instruments LabVIEW is a very innovative way to create virtual instruments. It is a graphical programming language focused on data flow concept. It is very easy for engineers to use it, to build simple graphical applications to control measurement equipment, prepare test stands, acquire data, visualize and analyze data. It is very flexible, easy to understand, easy to read and modify. That is why it is very popular in scientific and engineering environments. However this is a commercial closed source product. Some aspects of LabVIEW gives opportunity to easily modify graphical code which is hierarchical and this possibility is similar to open source concepts. Still this possibility can be limited if such feature is necessary.

Big advantage of using LabVIEW is excellent support of National Instruments hardware such as multi-purpose data acquisition devices. In such combination preparation of virtual instruments is unbeatable in such aspects as time of deployment (rapid application development), quality and reliability. People trained in LabVIEW and National Instruments devices tend to be much more productive then any other solution.

LabVIEW is available for 3 system platforms: Microsoft Windows, Apple MacOS and GNU/Linux. The best integration and support is available for Microsoft Windows, but even on free operating system like on GNU/Linux distributions it can be operated. This is possible mainly thanks to major world scientific laboratories like CERN, where LabVIEW environment is used commonly on machines with GNU/Linux systems. However support for GNU/Linux system is limited and only part of hardware is fully supported. Similar problems occurs also for Mac users. So if you want to get most of LabVIEW and National Instruments products features, the Windows version is a way to go.

In figure 1 there is a view of front panel and block diagram of simple application developed on Windows platform and in figure 2 is the same application run on GNU/Linux platform. Both looks and behaves pretty sure the same. However Windows version is more polished and the interface is more agile. Also you can be sure that on

Windows platform you will notice less problems and errors (especially internal errors).

We have gained some experience working with students on Faculty of Physics in Warsaw University of Technology. We are teaching LabVIEW programming (laboratories on "Principles of Virtual Instruments Design" and "Advanced LabVIEW applications") as well as open source technologies (laboratories like "Principles of Microcontroller Systems", "Principles of Embedded Systems"). We are pretty sure, that such book can help students to improve their skills in programming and understanding of both worlds: closed and open source and point that connection between them is possible and also very fruitful. Open source solutions may be very cost effective, easy to implement and give possibility of reuse in the future and be modified by other people to adapt to other needs.

It is extremely important that young engineers could select proper tools for their goals. This is also important for people who are in charge of organizing the work.

This book was organized in chapters, which are dedicated for different experience in LabVIEW graphical programming and open source solutions. First chapter is a quick introduction to LabVIEW and we suggest to read it if you are just beginning the adventure in LabVIEW programming. It can be used also as a reminder for senior users, as it was designed as a help for students which take course of "Principles of Virtual Instruments Design". This chapter covers not only the basic but also usage of design patterns like state machine, producer/consumer etc. These are topics which are needed for exams for Certificate of LabVIEW Associated Developer and higher degrees. Students can take part on exams on Universities which have LabVIEW Academy programs and get such a degree from National Instruments. This is very motivating as such experts are sought in industry. Such knowledge is needed to get a well paid job.

Closed and open source approach

Working with software development we encounter two approaches. One is called closed source and another one is called open source.

Theses are completely different in their assumptions. Sometimes it is seen as open source solution as a newer approach which should change software development completely. Open source approach actually has come as an answer to closed source inefficiency and problems which arose on software maintenance and specifically on hardware changes and driver software which should be accordingly modified. In many situations such software was prepared once and later not modified. This rendered situations where some devices could not be

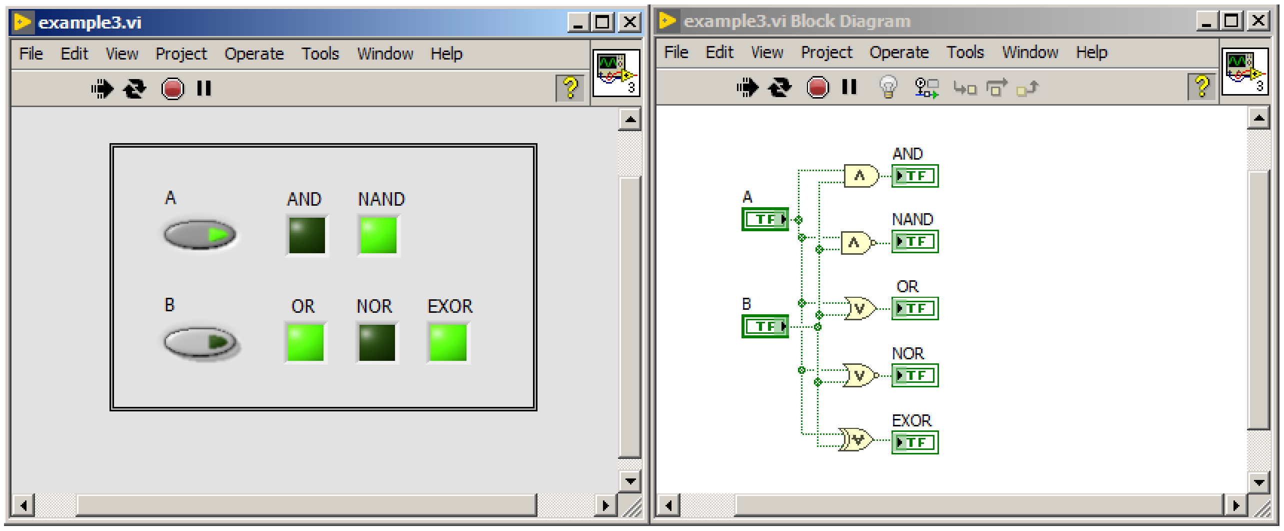

example3.vi
File Edit View Project Operate Tools Window Help
A
AND
NAND
B
OR
NOR
EXOR
example3.vi Block Diagram
File Edit View Project Operate Tools Window Help
A
B
AND
NAND
OR
NOR
EXOR

Figure 1: Simple LabVIEW application front panel and block diagram on Windows platform

operated on newer systems or these devices had limited features by software where they could operate much better if software could be modified. Open source is a fight to give possibilities to modify software which controls devices in a way users would want.

Let's identify also closed source approach. Closed source means that users are not allowed to explore and modify sources of software. Such software is distributed as binary files. According to assumptions, it should protect authors and enable them to earn money. Users usually get a license to use such software and this is only possible in strict ways. Such software is modified only by software vendors and of course only when they care to modify and develop this software further. There is always an anxiety, that such software development could be stopped.

Open source gives possibility for non-stoppable development because every one interested in development can do this. Everyone can download and compile source files as these are freely available. However there is a one problem. In open source approach it is not easy to get money from users. Sometimes support of users can be paid (this can be organized for large corporations), sometimes separate documentation, additional content, etc. In closed source approach, every user needs to pay money for license to use such software. Frequently it is needed to pay for single license or other business agreements could be prepared. This approach gives easily more money and allows to pay programmers for development.

However professional software licenses are expensive for casual users. Such licenses can be paid by large organizations, corporations, etc. but this is not suitable for single users. This is contradictory for software

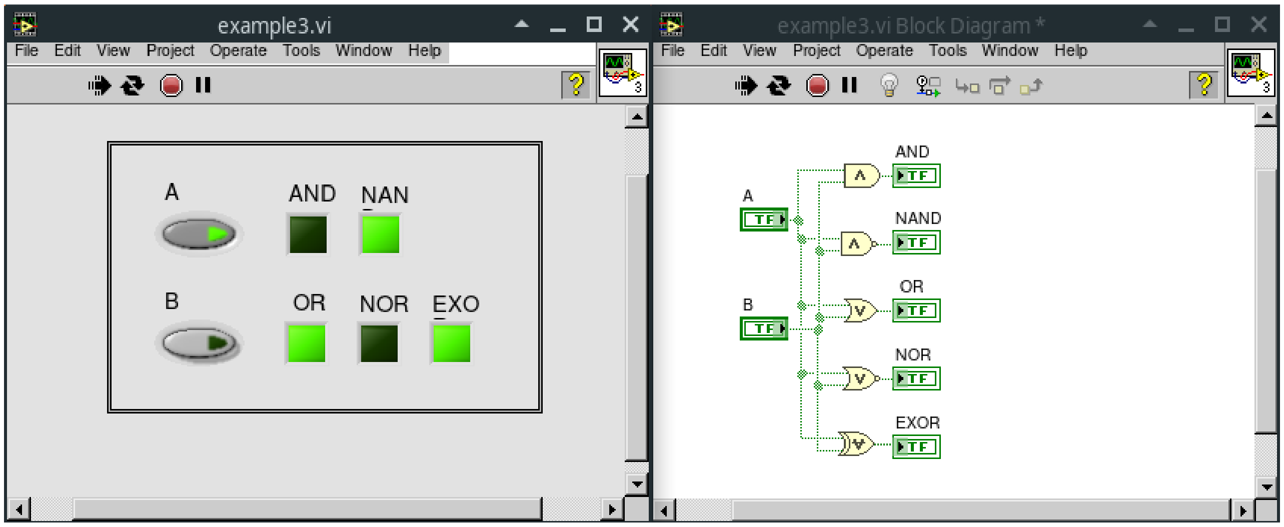

example3.vi
File Edit View Project Operate Tools Window Help
A
AND
NAN
B
OR
NOR
EXO
example3.vi Block Diagram *
File Edit View Project Operate Tools Window Help
AND
NAND
OR
NOR
EXOR

Figure 2: Simple LabVIEW application front panel and block diagram on GNU/Linux platform

spread. Single users are more interested in open source, because there are great communities which can help with software usage and development. In closed source approach normally software vendors don't listen to single user needs, in open source approach, every one may influence on software development.

LabVIEW for scientists and engineers

LabVIEW is famous as an easy, rapid and productive environment for software development mainly for controlling wide spectrum of control and measurement devices. In this environment engineers can prepare tests, calibrations, data analysis and many other tasks just using graphical language based on data flow paradigm.

It is invaluable tool for preparing graph, charts. There are buttons, scales, knobs and many other graphical user interface (GUI) widgets, users can easily implement just by dragging them from available palette. This is very convenient for preparing so-called "virtual instruments". Virtual instruments are just hardware devices, which doesn't have physical interface for users (like real knobs, displays, etc.). They are used with connection with computers or other control devices and actually software provides all necessary components to communicate with them, control them, display acquired data, and make calculations and analysis. These steps can be prepared in such way that prepared system will be extremely well suited for solving specific engineer problem. For virtual instrumentation many universal devices for data acquisition (DAQ) and signal generation were produced. These devices have excellent support in LabVIEW environment.

These mentioned features are welcome in scientific and engineering groups, because software development is not the main topic for them. Stability, easiness of usage, compatibility are important. Software should work as engineer is planning and debugging process should be minimalized as much as possible.

National Instruments LabVIEW environment comes in some flavours and there are also versions for students which are low-priced but limited by license in possibilities what user can do with prepared code.

Open source

Open source movement started because people have problems with hardware like printers, when drivers were not available for their systems. If hardware specification is known, there is available documentation, or source code for another architecture is available, programmers all over the world can prepare drivers for other systems (even if such vendors went bankrupt, or just not want to support such hardware any more). Many hardware vendors just kept as secret their software designs, communication protocols, etc. This is why so-called reverse engineering evolved. Just to reveal these secrets by testing, measurements and using other sophisticated techniques.

Licenses

There are many open source licenses. Some of them allow almost everything with source code, like Public Domain code. Other like BSD license allows almost everything, but they don't protect authors of code, and finally there are GPL licenses, which are more restrictive, by requirement of using them if derivative work is published. However they protect also authors of code. These legal issues exceeds this book topics. There are plenty of legal information over internet, just seek in case of need.

Cooperation

Mixing software licenses may cause headaches. However, there are many high quality open source codes available, easily downloadable over internet, that it is impossible to not use some of them. Many solutions are available only as open source. Even if there are proprietary solutions, frequently they are expensive and limited. If they have errors, sometimes vendors don't fix them or doesn't want to.

If you use open source libraries, the best option is also to release your software with open source license, however if it is not possible, then you should know, that various licenses like LGPL allows to use dynamic loadable libraries (DLL) or shared objects (.so) with such license along with your code under proprietary license. If you don't change such libraries, it is only need, to specify on some “about” software part, that such product is used, under LGPL license, and the best would be to point to the internet website where users can download source codes. This allows users even to replace such library, to study its code. So these are flexible possibilities granted for users of software, even if closed source parts co-exists.

Introduction to LabVIEW

Laboratory **V**irtual **I**nstrument **E**ngineering **W**orkbench (LabVIEW) is an programming environment for rapid and effective graphical user interface application building. LabVIEW was designed for dealing with measurement equipment, for data acquisition and for controlling devices using intuitive graphical interface. Engineers and scientists nowadays use LabVIEW also for data analysis and for simulation tasks.

The strength of LabVIEW is that it has wide set of libraries for communication with measurement equipment, data analysis and visualization. These libraries are either installed by default or they can be installed with *VI Package Manager* (VIPM) software, which is distributed with newer versions of LabVIEW together or can be also downloaded from the web. Moreover National Instruments supply wide set of drivers for measurement equipment like oscilloscopes, generators, multimeters, e.t.c.

LabVIEW is an environment based on "G" graphical programming language, where instead text line commands, we use set of icons and connecting wires. In textual programming languages, commands are executed line after line, whereas in LabVIEW we have dataflow paradigm, where data flow determined is by nodes. Using graphical language gives possibility of easy and intuitive creation of user interface for application execution control.

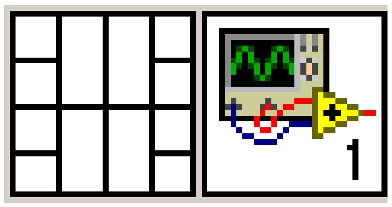


Figure 3: Icon and Connector Pane.

Programs prepared in LabVIEW are known as *virtual instruments* (VIs), because in their appearance they look as the real measurement devices. All LabVIEW programs consists of three elements: *Front Panel*, *Block Diagram* (in fig. 4) and *Icon and Connector Pane* in fig. 3). A Front Panel is the graphical user interface. A Block Diagrams is a space for placement of program graphical code. Icon and Connector Pane determines the look of a VI inside another VI. Moreover it also determines input and output connections, which are needed to provide for correct operation of VI.

Front Panel is a graphical user interface, which consists of input terminals called *controls* and output terminals called *indicators*. After clicking the right mouse button on the front panel, a *Controls palette*

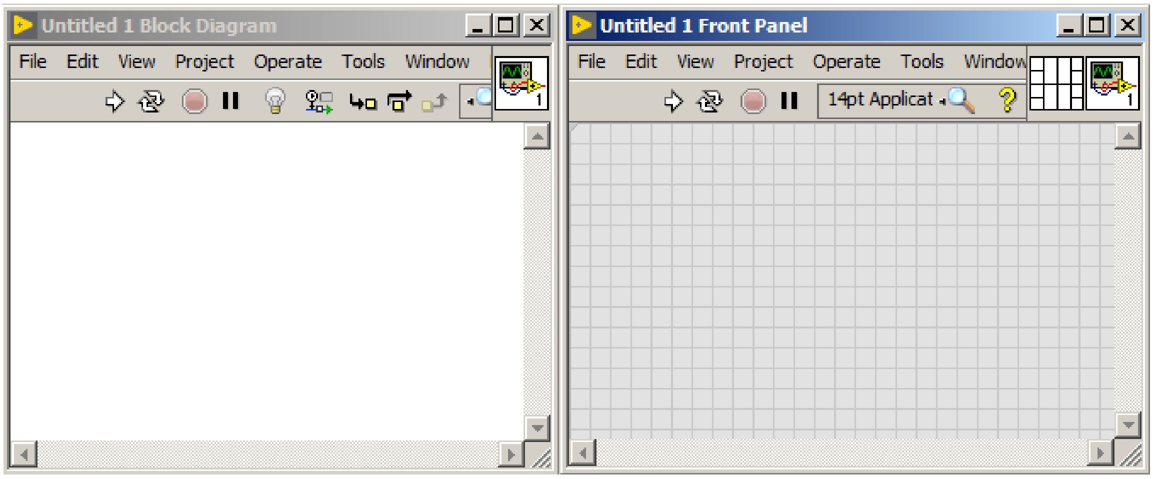

Untitled 1 Block Diagram
File Edit View Project Operate Tools Window
Untitled 1 Front Panel
File Edit View Project Operate Tools Window
14pt Applicat

Figure 4: The block diagram (left side) and front panel (right side).

will be displayed. (in fig. 5).

User can select *controls* and *indicators* from four palettes: *Modern, Silver, System and Classic*. Appearance of example *controls* and *indicators* in case of differently selected palette is presented in figure 6.

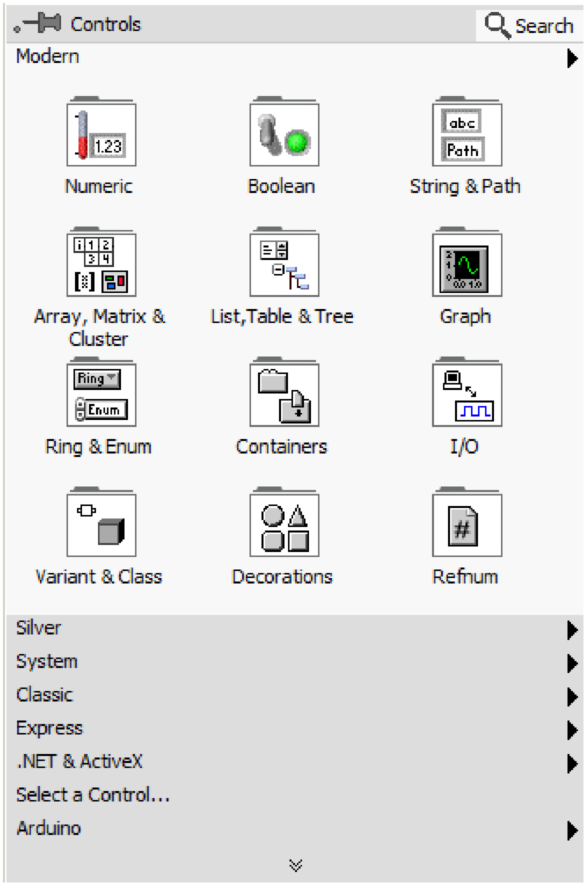


Figure 5: *Controls palette.*

The Silver palette has a wide set of available controls and indicators. The Modern palette has controls and indicators with modern look compared to classical palette. The Modern palette is recommended for the most applications, where the Silver palette for creation fancy front panels e.g. for customers. Classic palette is dedicated for applications, which should be a part of the bigger project or for applications, where controls and indicators would be customized. System palette is dedicated for applications, which will be executed on displays with only 16 colors available.

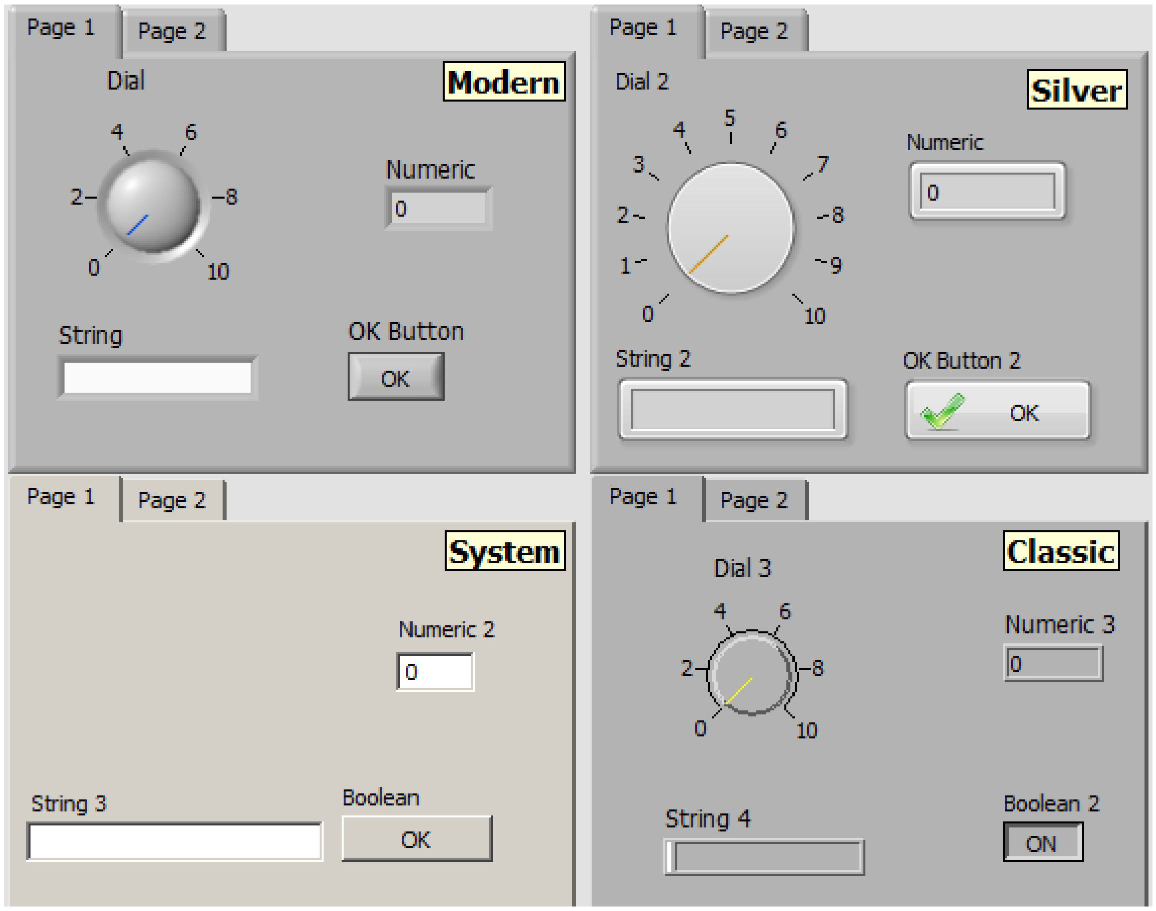


Figure 6: Appearance of controls and indicators in cases of different palette selected in LabVIEW.

Block diagram is a space, where programmer puts the code. To add functions, structures, loops or consts, one needs to click the right mouse button on block diagram, which then opens *Functions palette* (in fig. 7).

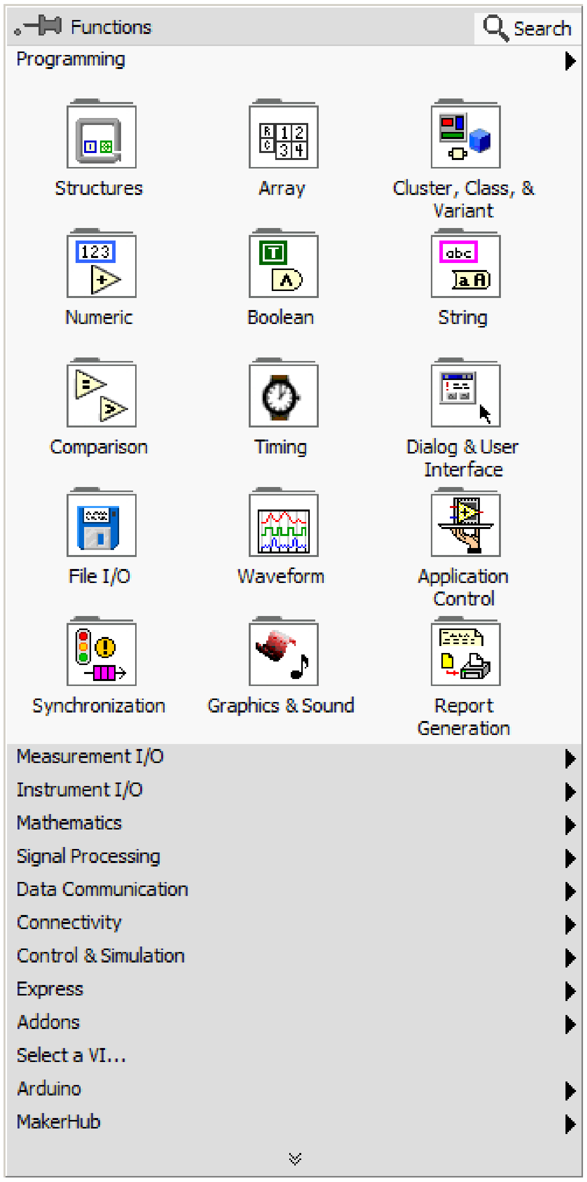


Figure 7: *Functions palette.*

Icon and Connector Pane defines appearance of VI icon. Every VI can be used inside another VI, in this case it is called "subVI". The equivalent of subVI is a function (or procedure) in textual programming languages. Default appearance of VI icon is presented in figure 3. When in selected VI there are more then one subVI with default icon, these icons can be distinguished from themselves by the number placed in the bottom right corner of the icon. A good practice is to change the look of VI icon, which makes that the code is more clear and understandable. To change the look of icon, one needs to click the right mouse button on the icon placed in top right corner of front panel

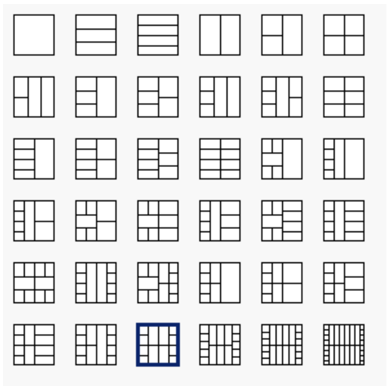


Figure 8: Connector pane patterns.

and select *Edit Icon...* option. The edit window is presented in figure 9. Connector Pane defines where in icon, the input and output terminals should be connected. Default configuration of connector pane consists of terminals in pattern: 4x2x2x4 (left side, top side, bottom side, right side). To change the number and arrangement of terminals one need to click the right mouse button on connector pane placed in top right corner of front panel and select *Patterns* option. Available patterns are presented in figure 8.

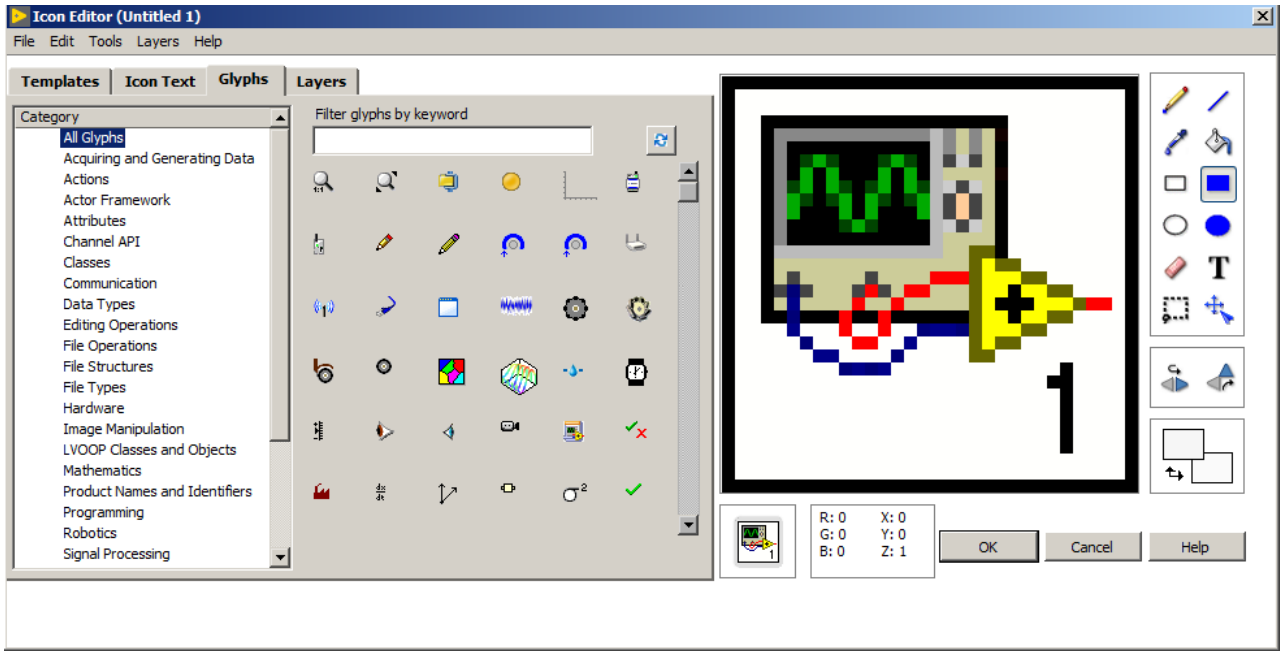


Figure 9: VI icon edit window.

Basic data types in LabVIEW

In LabVIEW we can use basic data types as: *numeric*, *boolean*, *string* and *enumeration type*.

Numeric data type

Data of the *numeric* type can represent values in various forms, e.g. integer numbers (*integer*) or floating point numbers (*real*). Available number representations are presented in table 1. Representation of any terminal can be changed by clicking of the right mouse button on terminal and selecting *Representation* option.

In case of the needs, numerical terminals appearance may be changed, which is presented in figure 10. Range of the selected control or indicator can be changed by clicking the right mouse button on the terminal and selecting *Properties* $\rightarrow$ *Data Entry*.

Numeric Data Type	Bits	Approximate Range
Single-precision, floating-point	32	(1.40e-45; 3.40e+38), (-3.40e+38, -1.40e-45)
Double-precision, floating-point	64	(4.94e-324;1.79e+308), (-1.79e+308; -4.94e-324)
Extended-precision, floating-point	128	(6.48e-4966; 1.19e+4932), (-1.19e+4932; -6.48e-4966)
Complex single-precision, floating-point	64	(1.40e-45; 3.40e+38), (-3.40e+38, -1.40e-45)
Complex double-precision, floating-point	128	(4.94e-324;1.79e+308), (-1.79e+308; -4.94e-324)
Complex extended-precision floating-point	256	(6.48e-4966; 1.19e+4932), (-1.19e+4932; -6.48e-4966)
Byte signed integer	8	(-128;127)
Word signed integer	16	(-32 768; 32 767)
Long signed integer	32	(-2 147 483 648; 2 147 483 647)
Quad signed integer	64	(–1e19;1e19)
Byte unsigned integer	8	(0;255)
Word unsigned integer	16	(0;65 535)
Long unsigned integer	32	(0; 4 294 967 295)
Quad unsigned integer	64	(0;2e19)
128-bit time stamp	128	(01/01/1600 00:00:00 UTC; 01/01/3001 00:00:00 UTC)

Table 1: Representation of numeric data types in LabVIEW. Source of the information: `https://zone.ni.com/`.

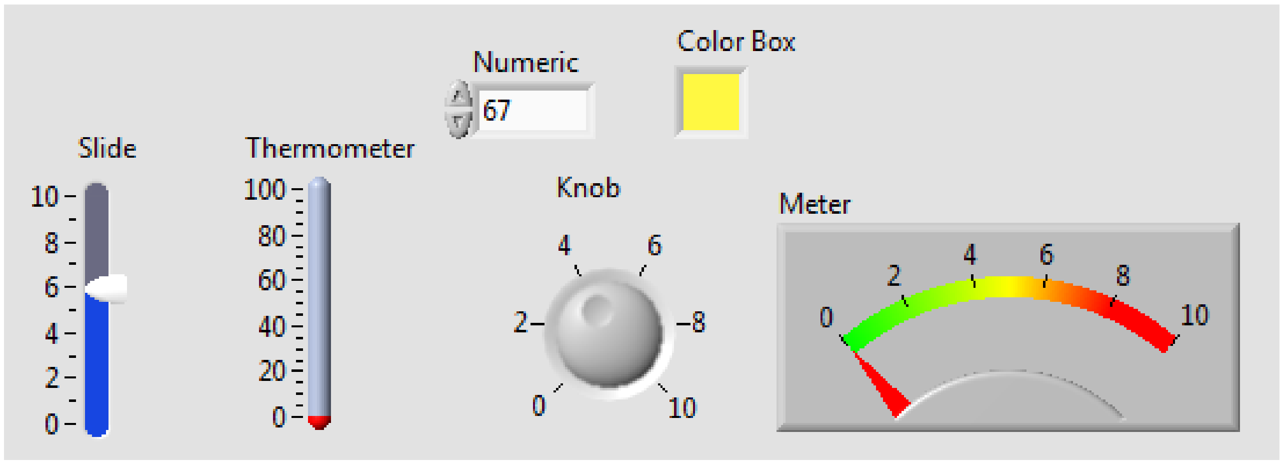


Figure 10: Controls and indicators appearance in LabVIEW.

In figure 11 there is an example program with numeric terminals. This program purpose is to calculate the area of an equilateral triangle from the given height and length of base. Input data are entered into two controls called *Length of base* and *Height*. Values from controls are propagated to inputs of *Multiply* function. To get the area of triangle, a result from multiply should be divided by 2, so output from *Multiply* function is connected to input of *Divide* function. On the second input of *Divide* function, a constant value "2" is provided. Result of *Divide* function is then propagated to input of indicator called *Area of triangle*.

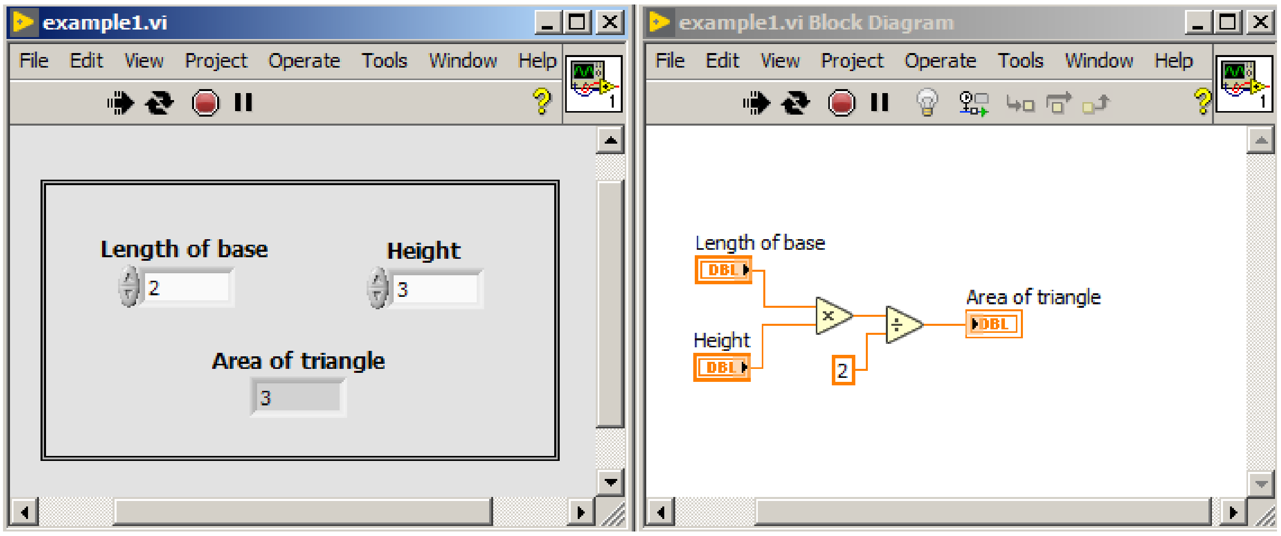

example1.vi
File Edit View Project Operate Tools Window Help
Length of base
2
Height
3
Area of triangle
3
example1.vi Block Diagram
File Edit View Project Operate Tools Window Help
Length of base
DBL
Height
DBL
2
Area of triangle
DBL

Figure 11: A program which calculates area of an equilateral triangle using height and length of base provided by user.

In the example above, terminals with the same representation were connected to a function. However if two terminals of different representation will be connected to a function, there we will have an automatic conversion of simpler representation (less bits) to the more complex one (more bits). It is designed in such a way, to not loose the precision of numerical values. The conversion example is presented in figure 12. Precision lose can be observed if we convert from double to integer, which is presented in figure 13.

In case of connecting singed integer and unsigned integer to a function, the return result value will be unsigned integer. In the place where wire is connected to function and where conversion occurs, a red dot called *Coercion Dot* is shown.

In many cases, a conversion from one representation to another is a necessity. In such cases we can use functions from *Functions Palette* → *Numeric* → *Conversion* explicitly.

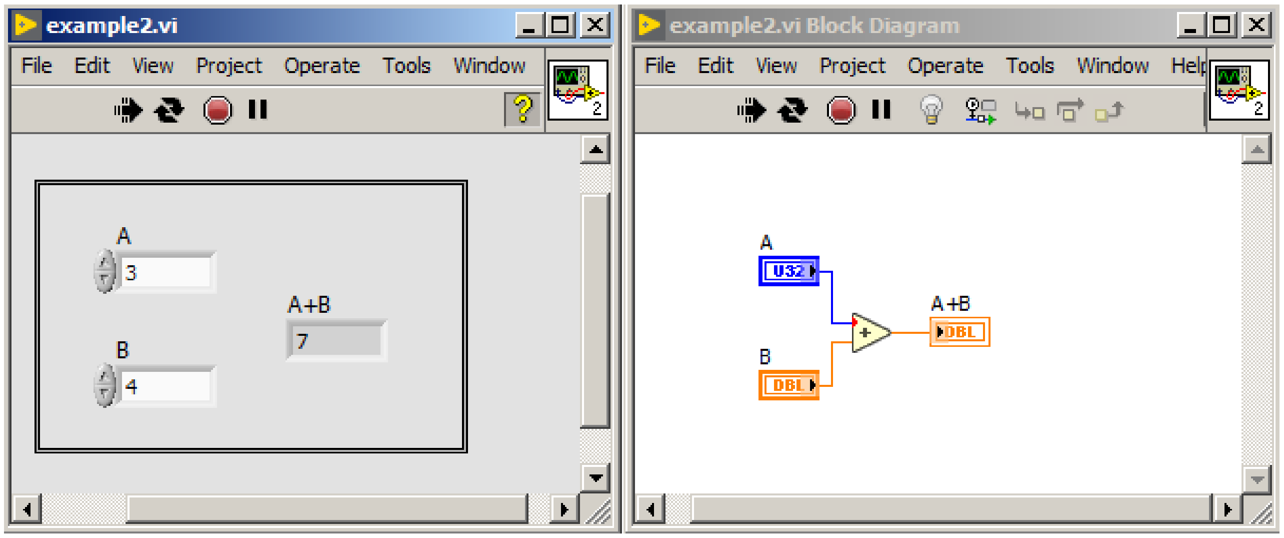

example2.vi
File Edit View Project Operate Tools Window
A
3
B
4
A+B
7
example2.vi Block Diagram
File Edit View Project Operate Tools Window Help
A
U32
B
DBL
A+B
DBL

Figure 12: An example which shows conversion of a number in order to perform the numerical operation.

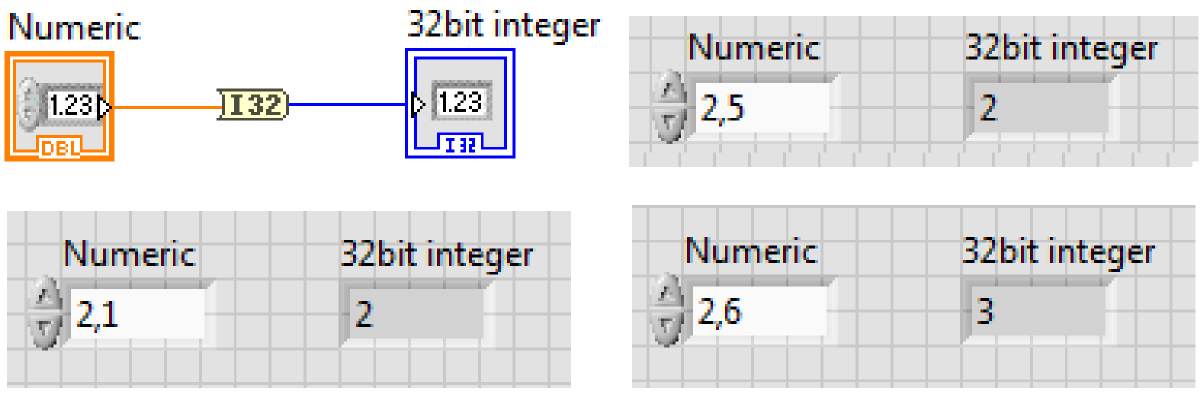


Figure 13: An example showing number conversion from double precision format to signed long integer.

Boolean data type

Next basic data type is *boolean*, which represents data with only two states: TRUE and FALSE or ON and OFF. Example controls and indicators are presented in figure 14.

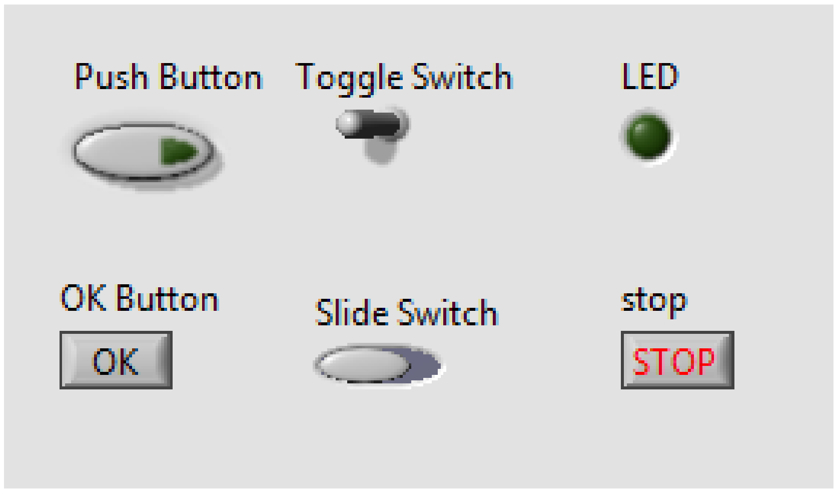


Figure 14: Example look of boolean controls and indicators.

In LabVIEW buttons have selectable mechanical actions. Available types of mechanical action are presented in figure 15. Individual graphs show what value the terminal will return after pressing the button and releasing it. Graphs for *switch* mechanical action consists of two lines. First line is marked by letter *m* (mechanical) and describes the moment of pressing the button (arrow downwards) and releasing the button (arrow upwards). Line marked by letter *v* shows the value, which the button returns in case of *m* line state. E.g. for *switch when pressed* mechanical action, in the moment of button press, the value changes from FALSE to TRUE. After releasing the button, the value read from terminal still will be TRUE. For *switch when released* mechanical action, the button returns TRUE value only after the release of button. For *switch until released* mechanical action, in the moment of button being pressed, the value will change from FALSE to TRUE until the moment of button being released.

Following lines describe *Latch* type mechanical actions, which are marked by *RD* letters. This line shows the moment of value readout from the button. Every readout of value is marked by cross. E.g. for *latch when pressed* mechanical action, in the moment of button press, the value changes from FALSE to TRUE until the time of first readout. So the event of button pressed will be read only once. Even if the following readout will take time before release of the button, the returned value will be FALSE. In case of *latch when released* mechanical action,

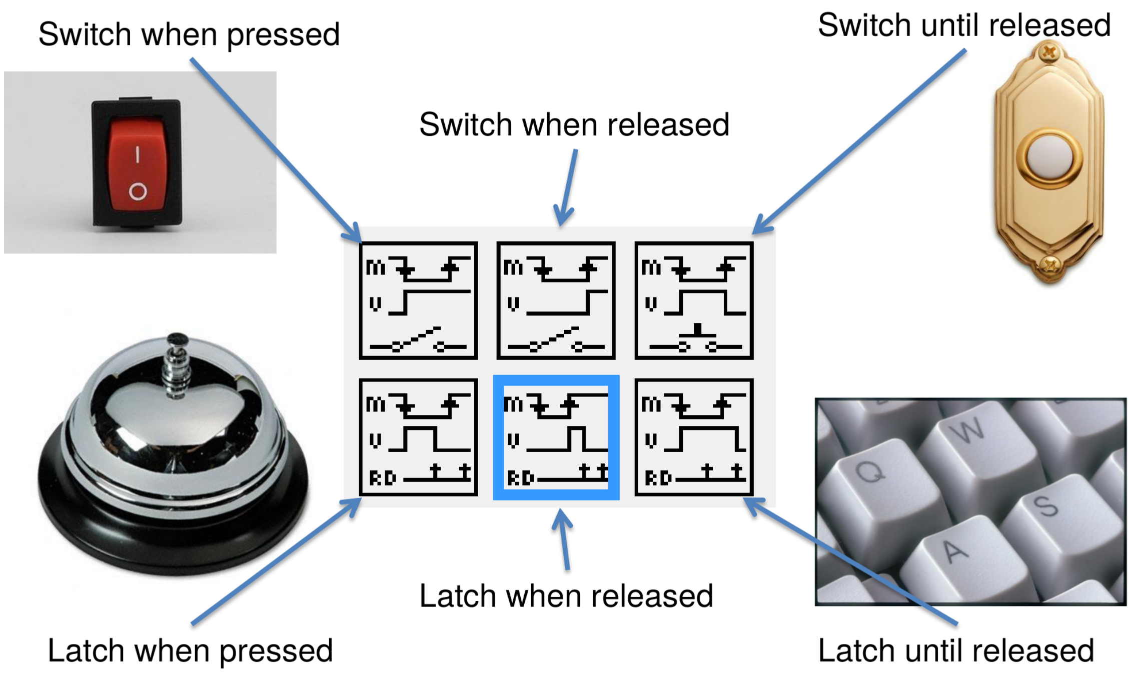


Figure 15: Available mechanical actions for buttons in LabVIEW.

value changes from FALSE to TRUE in the moment of releasing the button, and returns to FALSE just after the readout from terminal. For *latch until released* mechanical action, the value from button changes from FALSE to TRUE in the moment of button being pressed. Readouts of the value before releasing of the button doesn't affect on value. Only after releasing the button, the value changes to FALSE after reading the value from the button terminal.

In figure 16 an example program using boolean terminals is presented. Program shows the principles of logic gates (AND, OR, NAND, NOR, EXOR) operations. As input data, two controls named *A* and *B* were used. Results of logic operations are propagated to indicators named: *AND*, *OR*, *NAND*, *NOR* and *EXOR*.

String data type

The next basic data type is a string, which is a sequence of characters encoded in ASCII. In figure 17 there is a look of string control and string indicator presented.

Terminals of string type can display characters in four different styles: *Normal*, *Backslash Codes*, *Password* and *Hexadecimal*. To change the style of displayed characters, one needs to click the right mouse button on terminal and choose: *Properties* → *Appearance*. An example program with four indicators is presented in figure 18.

In figure 19 an example program with basic operations on strings is presented. On the beginning constants of the string type are con-

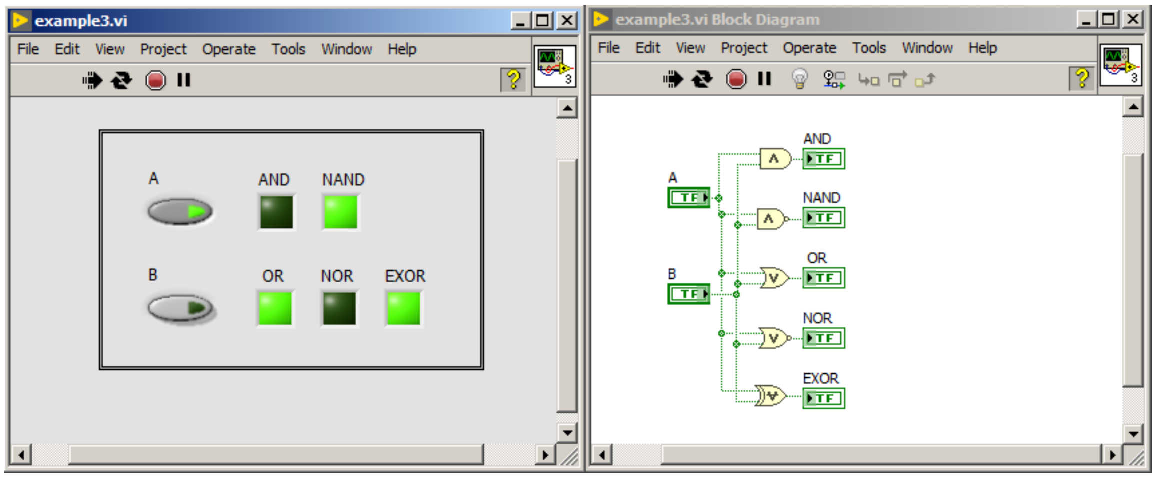

example3.vi
File Edit View Project Operate Tools Window Help
A
AND
NAND
B
OR
NOR
EXOR
example3.vi Block Diagram
File Edit View Project Operate Tools Window Help
AND
NAND
OR
NOR
EXOR
A
B

Figure 16: Logic gates operation demonstration in LabVIEW.

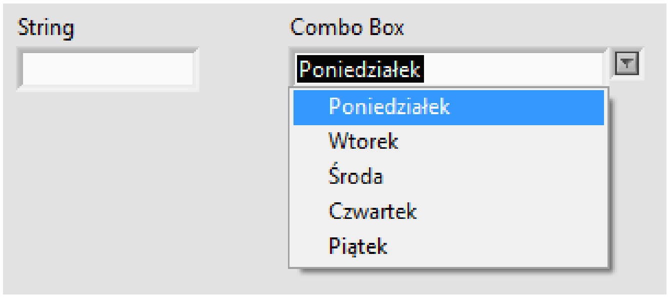
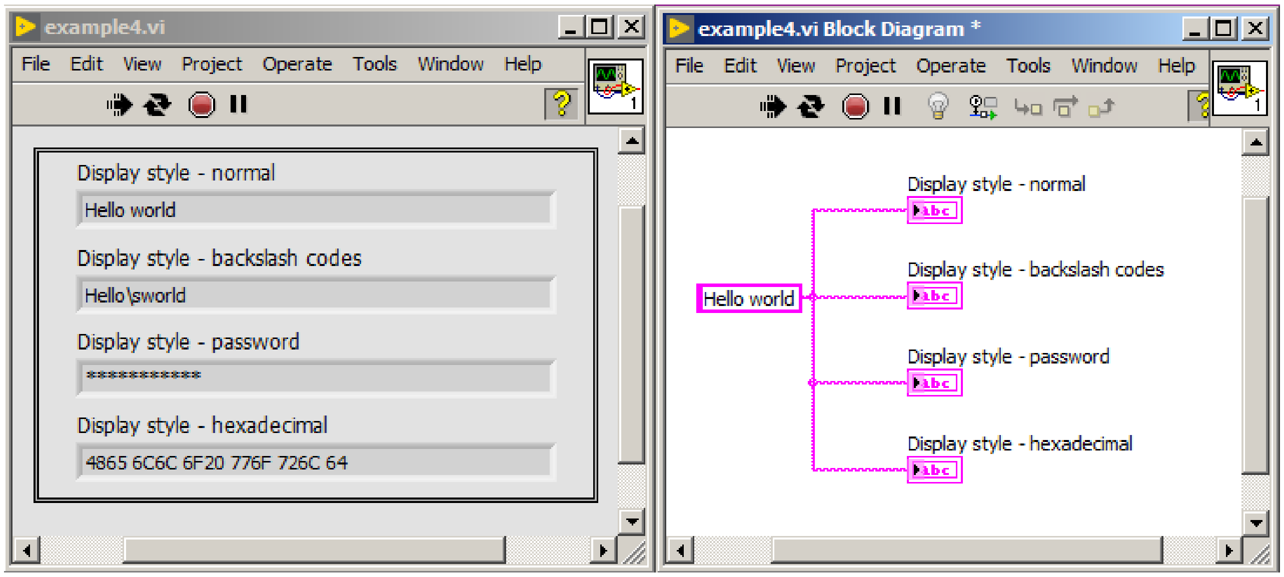


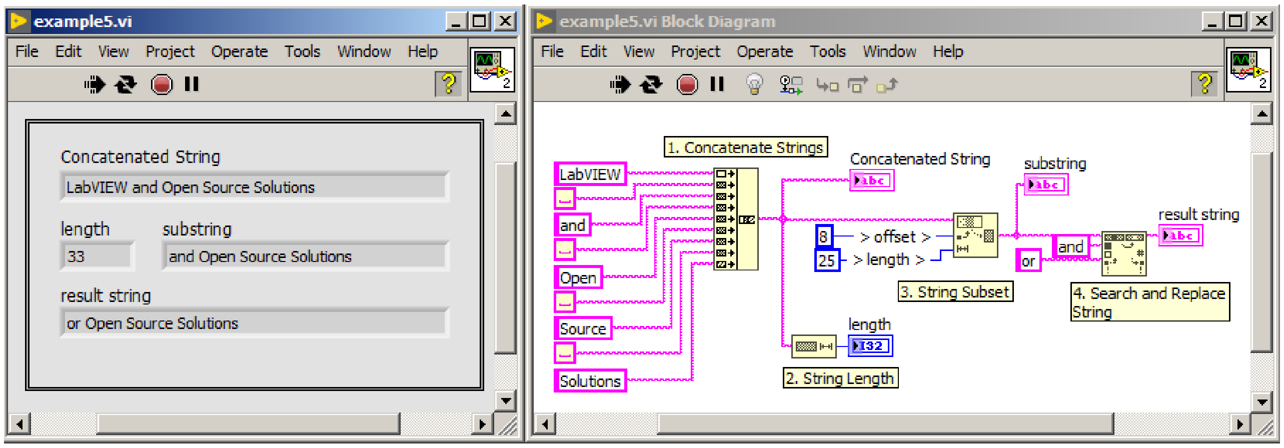
Figure 17: A look of example string control and string indicator.


example4.vi
File Edit View Project Operate Tools Window Help
example4.vi Block Diagram *
File Edit View Project Operate Tools Window Help
Hello world

```
Display style - normal
Hello world
Display style - backslash codes
Hello\sworld
Display style - password
***********
Display style - hexadecimal
4865 6C6C 6F20 776F 726C 64
```

Figure 18: Program which shows available styles of displaying characters in LabVIEW.

catenated into single string with a help of *Concatenate String* function. Next, the length of string is checked using *String Length* function. Moreover a part of the string is being extracted using *String Subset* function, which also have input parameters like *offset* (the position from where substring should begin) and *length* (how many characters should the substring have). If on the input of *String Subset* function a value will not be provided, then as a default, whole string starting from the offset position will be returned. Finally, a *Search and Replace String* function is shown, which searches provided string and replaces found fragment with another provided.



Concatenated String
LabVIEW and Open Source Solutions

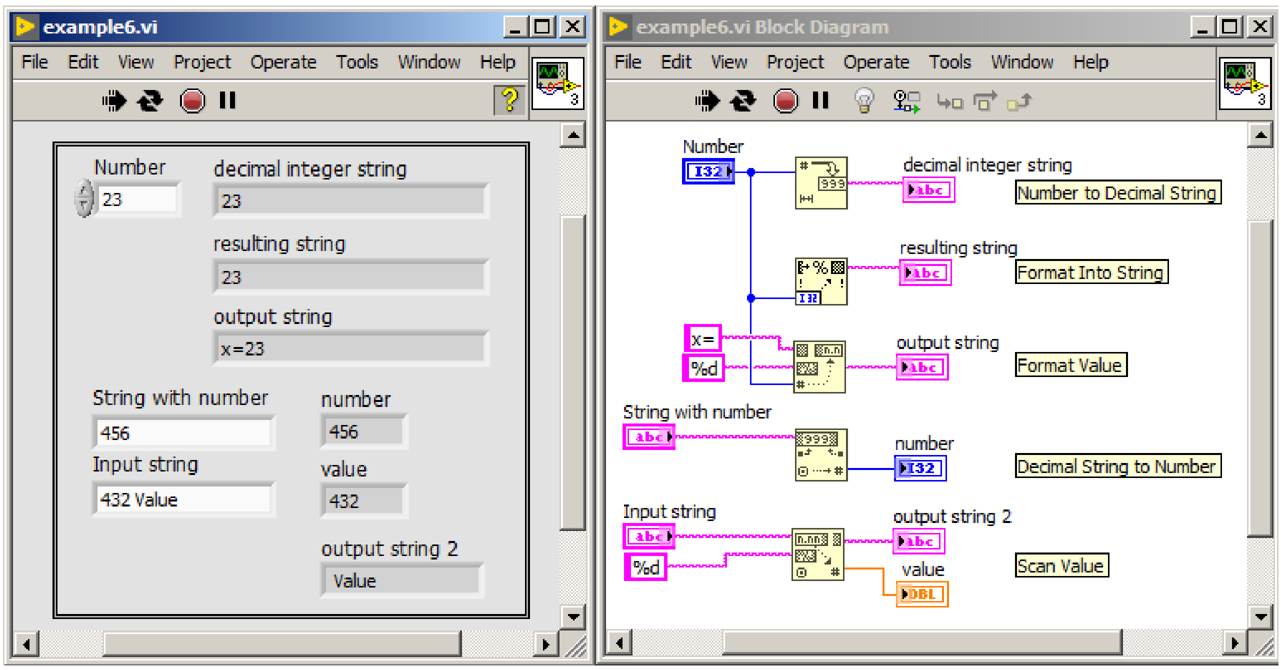
length
33

substring
and Open Source Solutions

result string
or Open Source Solutions

Figure 19: Demonstration of basic operations on strings in LabVIEW.

In figure 20 an example program which demonstrates different conversions methods of strings to numeric values and vice-versa was presented.



Number: 23
decimal integer string: 23
resulting string: 23
output string: x=23
String with number: 456
number: 456
Input string: 432 Value
value: 432
output string 2: Value

Figure 20: Program which demonstrate string conversion methods.

Enumeration type

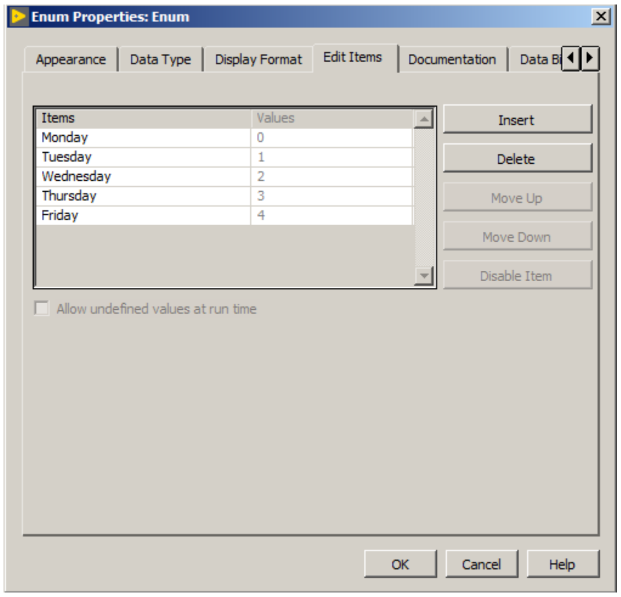
The last from basic data types is enumeration type: enum or ring. First of them assigns ordinal numbers to the entered names beginning from zero. Second of them allows to write own ordinal numbers to selected names. In figure 23 an example showing difference between enum and ring types is shown. In figure 21 and 22 adding of items to enum and ring is shown.

Every data type has specific style and different wire color on block diagram. In figure 24 different wire types assigned to data types are shown.

Array

An array is a set of elements of the same type, e.g. integer numbers. In LabVIEW, arrays may be of any type, e.g. array of strings, array of clusters, e.t.c. The only limit is that, you cannot create array of arrays. However multidimensional arrays are allowed. To create an array, one need to put object *Array* on front panel and put in them a control or indicator of selected type (presented in figure 25) and 26). Created array by default has one dimension. To create two dimensional array, one need to click the right mouse button on iterator and select *Add Dimension*.

An array can be also created by a function *Initialize Array*. An example of function usage is presented in figure 27. Moreover creation



Items	Values
Monday	0
Tuesday	1
Wednesday	2
Thursday	3
Friday	4

☐ Allow undefined values at run time

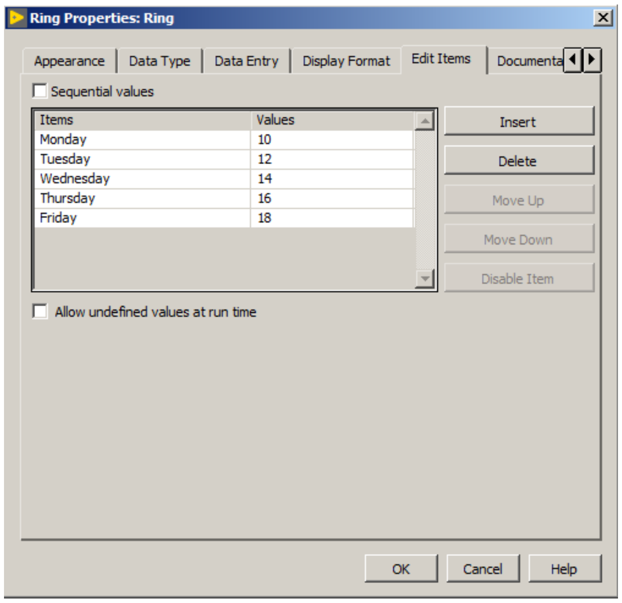
Figure 21: Elements added to enum type control.

and addition of elements to the array is possible with use of function *Build Array*, which is shown in figure 28.

In figure 29 an example is given, where it is shown how to get information about size of an array. In case of one dimensional array, function *Array Size* returns numeric value. In case of two dimensional array, function *Array Size* returns an array with two elements. The first element of this array is a number of rows, and the second element of this array is a number of columns.

In figure 30, an example is presented, in which it is shown hot to extract single element from one dimensional (1D) array or a column or a row from two dimensional array (2D). In such case, a function *Index Array* should be used, which accepts on input an array and an index number of the element. In case of 2D array, to receive an element, on input of function *Index Array*, one need to put an index number of row and an index number of column. If we supply only a row number, function *Index Array* will return all elements from this row (all columns). Similarly, when one supply only column number, function *Index Array* returns all elements from selected column (all rows).

To extract the piece of array, one need to use a function *Array Subset*, which accepts two arguments: index and length. First of them describes from which index number function should extract a piece of array. Second argument describes length of array that should be returned. If on input length no value will be supplied, then function *Array Subset* by default returns all of the elements of array from specified index number until the end of the array. An example of use of *Array Subset* function is presented in figure 31.



Items	Values
Monday	10
Tuesday	12
Wednesday	14
Thursday	16
Friday	18

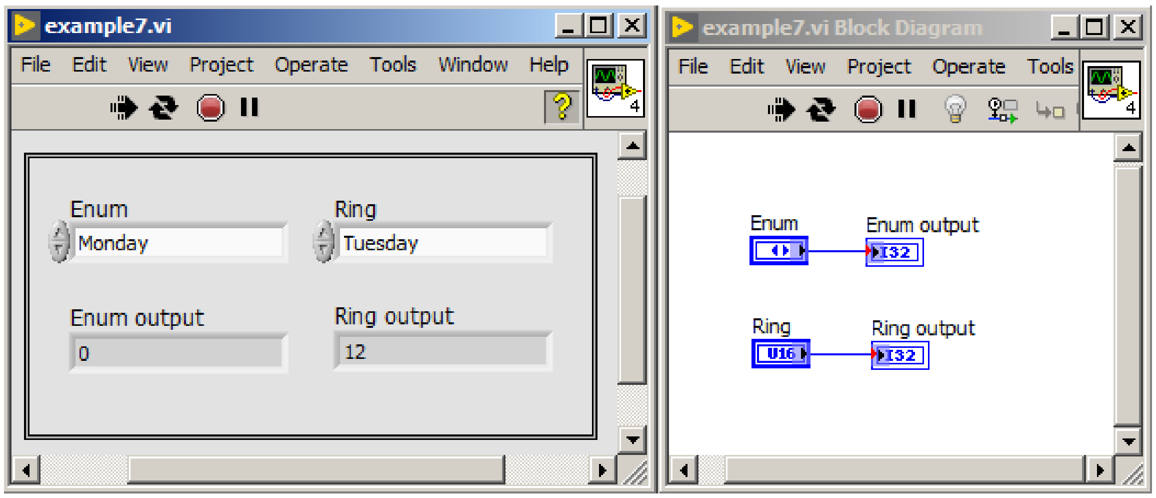
Figure 22: Elements added to ring type control.

To add new element to the array, one needs to use *Insert into Array* function or *Build Array* function (which will be described later). Function *Insert into Array* takes on input an index number, where new element should be placed in array. An example of function usage is presented in figure 32. Function *Insert into Array* may also be used for adding an array to another array. In such case on function input called *new element* one needs to connect an array.

To remove element from array one needs to use *Delete from Array* function, which takes on input:an array, length and index. The length parameter describes how many elements will be deleted from supplied index number. Function *Delete from Array* returns two arrays. First of them is an array with elements deleted. Second array is an array with deleted elements. In figure 33 there is an example of *Delete from Array* function usage.

Functions described above allow to perform basic operations on arrays. However in *Array* fold of functions palette there are more functions e.g. *Rotate 1D Array, Sort 1D Array, Split 1D Array, Reshape 1D Array* and *Max & Min*.

Arrays can be transferred to inputs of numerical functions, which recognize connected type of data and adapt their actions according to them (this is called polymorphism of functions). In figure 34 a program in which two arrays are connected to *Add* function is presented. In this case the element with zero index of “A” array is added to the element with zero index of “B” array. Similarly elements with index of one from arrays “A” and “B” will be added. The “A” array length is 3, but “B” array length is 2, so the third element of “A” array will


example7.vi
File Edit View Project Operate Tools Window Help
example7.vi Block Diagram
File Edit View Project Operate Tools
I32
U16
I32

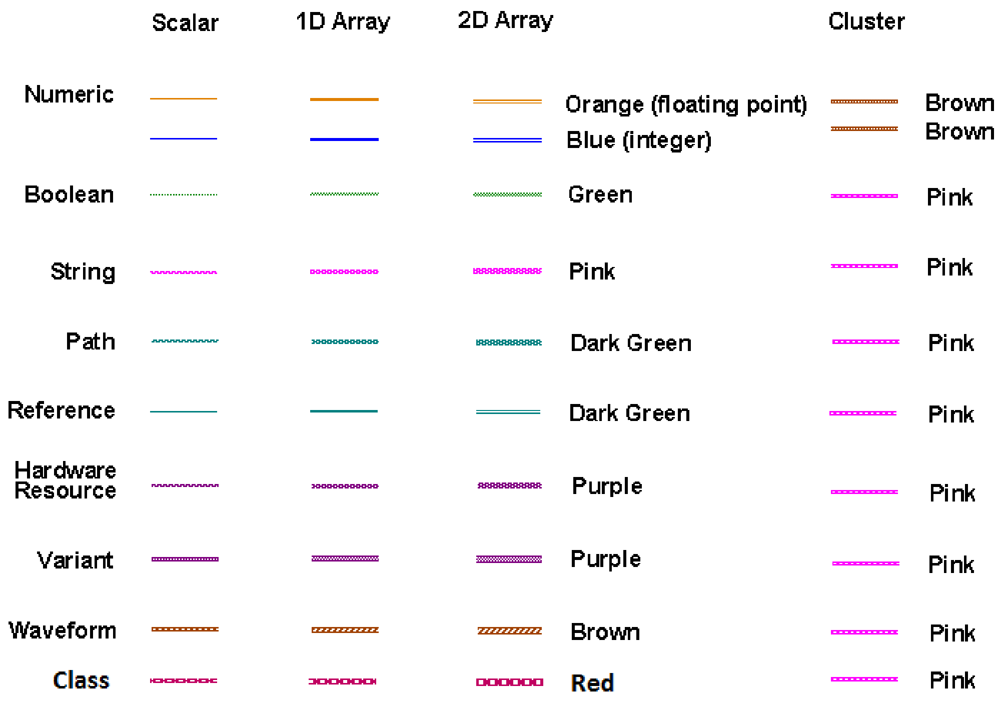
Enum: Monday

Ring: Tuesday

Enum output: 0

Ring output: 12

Figure 23: A program showing difference between enum and ring types.



	Scalar	1D Array	2D Array		Cluster	
Numeric	─	─	═	Orange (floating point)	═	Brown
	─	─	═	Blue (integer)	═	Brown
Boolean	┈	┈	┈	Green	─	Pink
String	∿	∿	∿	Pink	─	Pink
Path	∿	∿	∿	Dark Green	─	Pink
Reference	─	─	═	Dark Green	─	Pink
Hardware Resource	∿	∿	∿	Purple	─	Pink
Variant	─	─	─	Purple	─	Pink
Waveform	─	─	─	Brown	─	Pink
Class	─	─	─	Red	─	Pink

Figure 24: Types and colors of wires in case of different data types in LabVIEW. Source of information: `https://knowledge.ni.com`.

be discarded. This is because "B" array is smaller. *Add* function will return an array with 2 elements, which will be the result of sum of elements from "A" and "B" arrays. Similarly operation of subtraction can be achieved by using *Subtract* function.

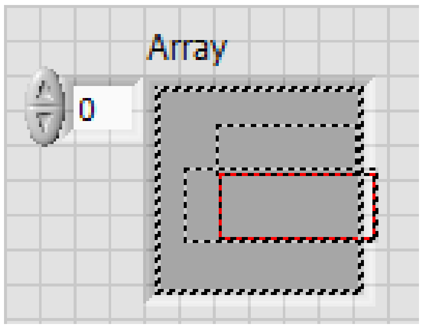


Figure 25: Array creation through putting numeric controls inside the array frame.

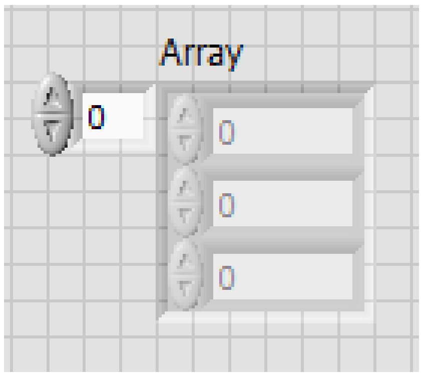


Figure 26: Created array of numeric type.

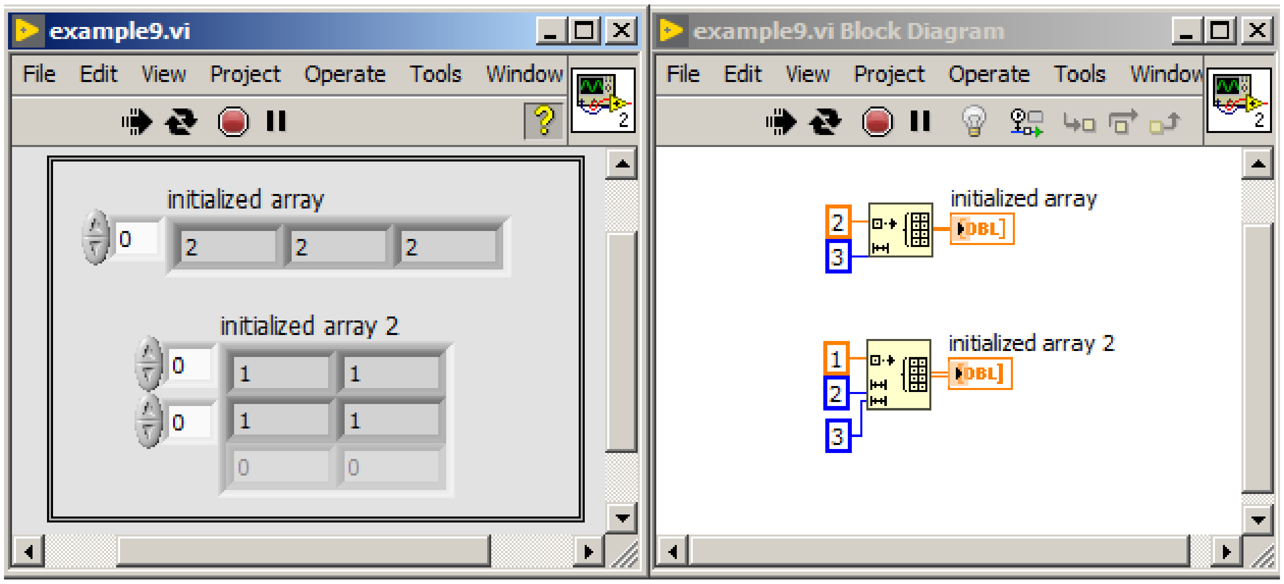

example9.vi
File Edit View Project Operate Tools Window
initialized array
0
2 2 2
initialized array 2
0
0
1 1
1 1
0 0
example9.vi Block Diagram
File Edit View Project Operate Tools Window
2
3
initialized array
DBL
1
2
3
initialized array 2
DBL

Figure 27: An array creation with use of function *Initialize Array*.

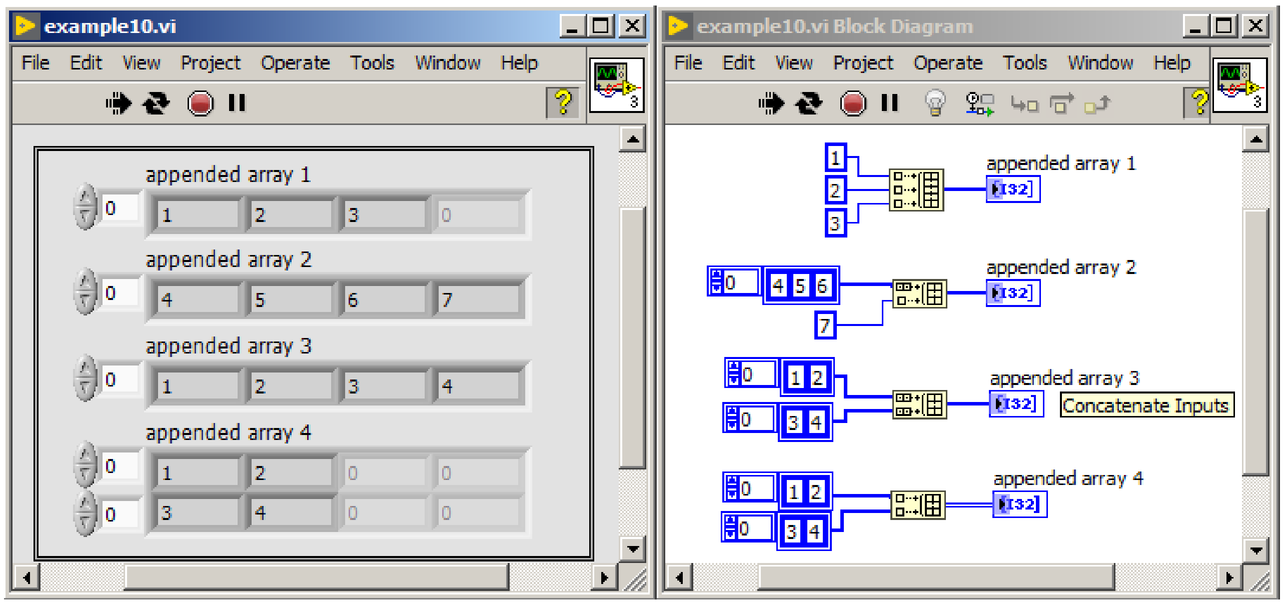


Figure 28: Creation and addition of elements to the array with use of function *Build Array*.

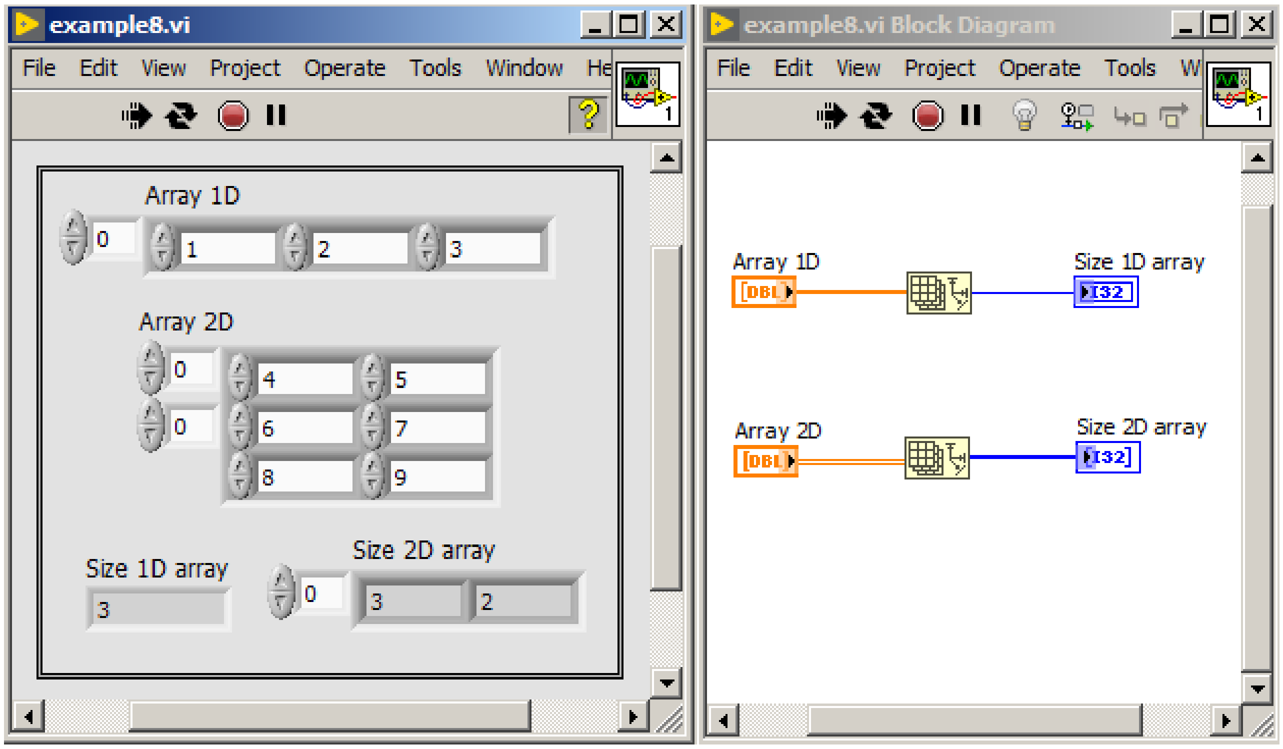

example8.vi
File Edit View Project Operate Tools Window He
Array 1D
Array 2D
Size 1D array
Size 2D array
example8.vi Block Diagram
DBL
I32

Figure 29: A program that shows the size of array in LabVIEW.

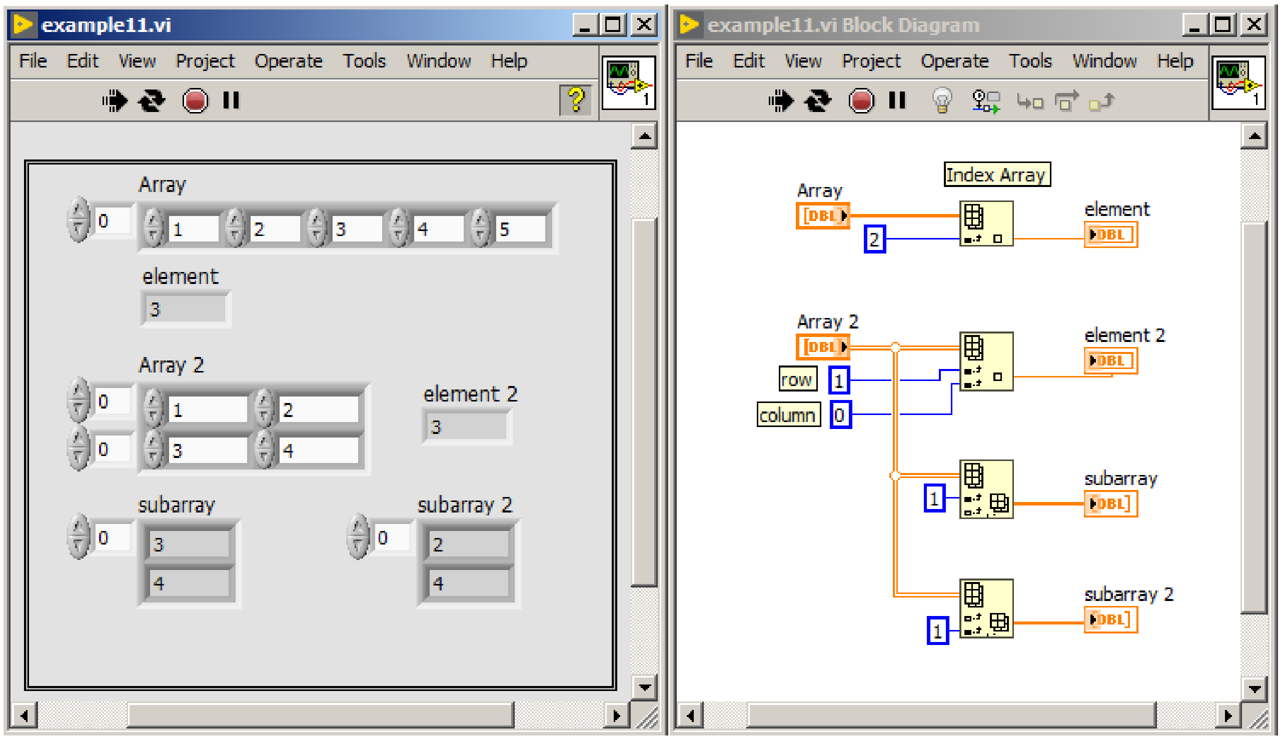


Figure 30: Program that extracts an element, a row or a column from array.

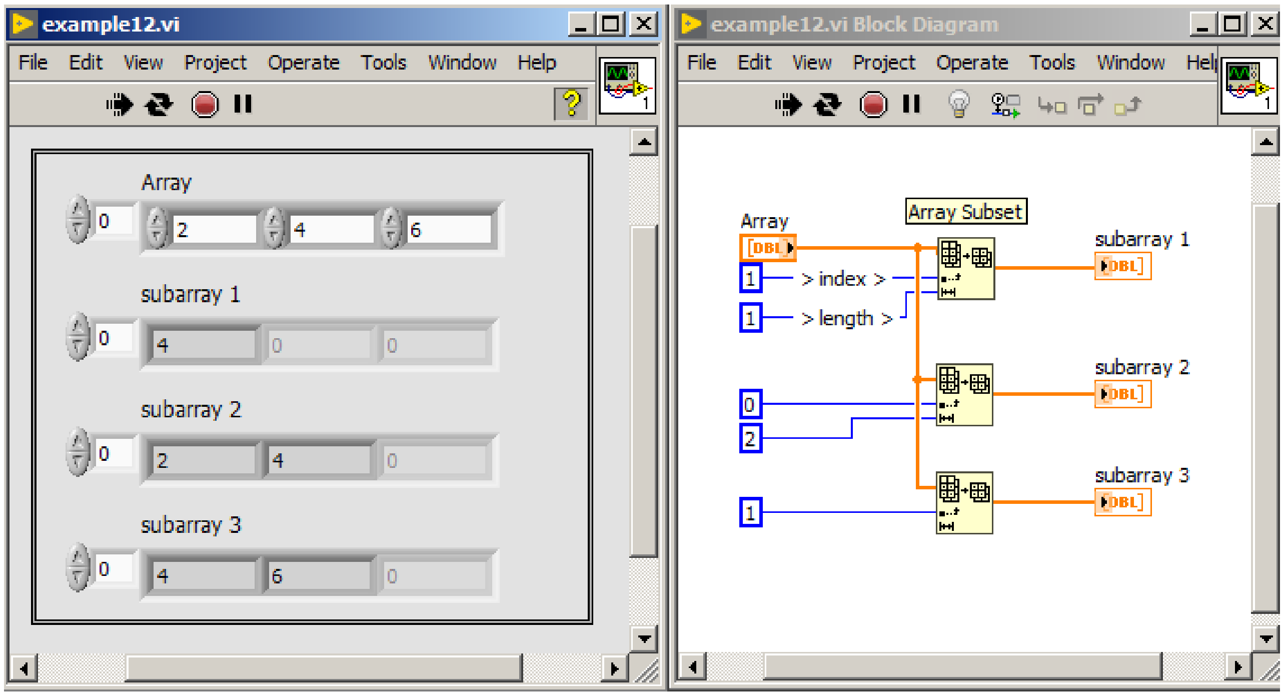


Figure 31: Program that extracts a subarray from array.

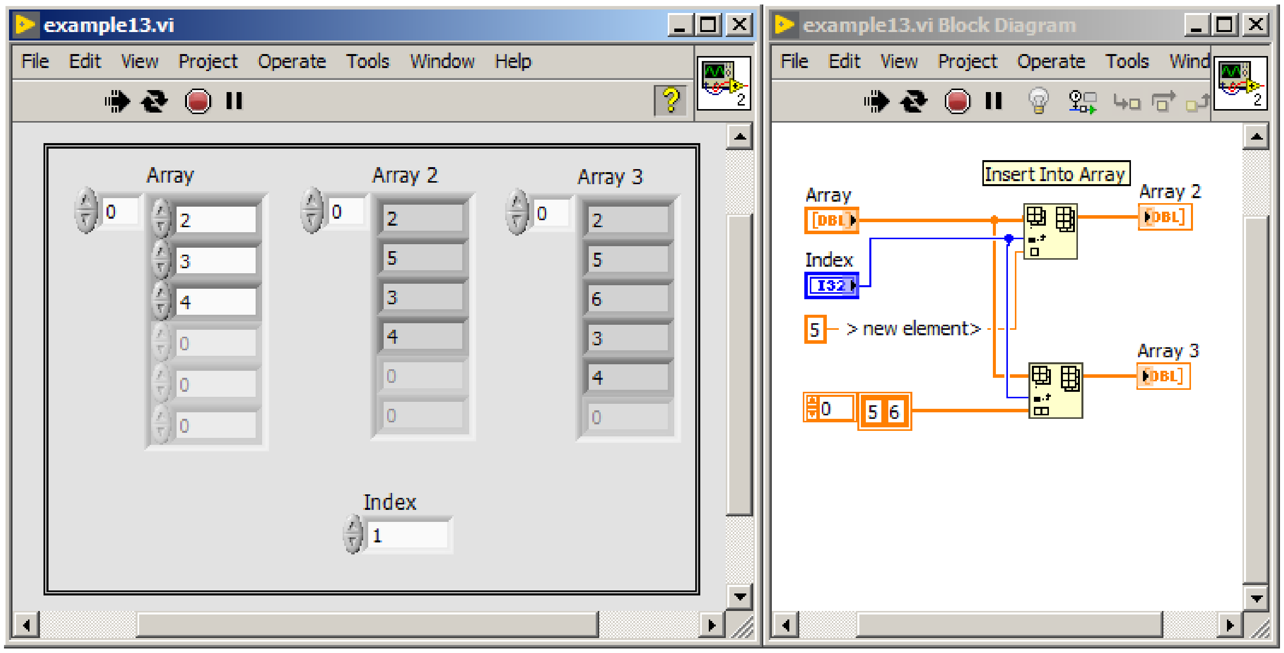


Figure 32: A program that demonstrates adding new element or array to existing array.

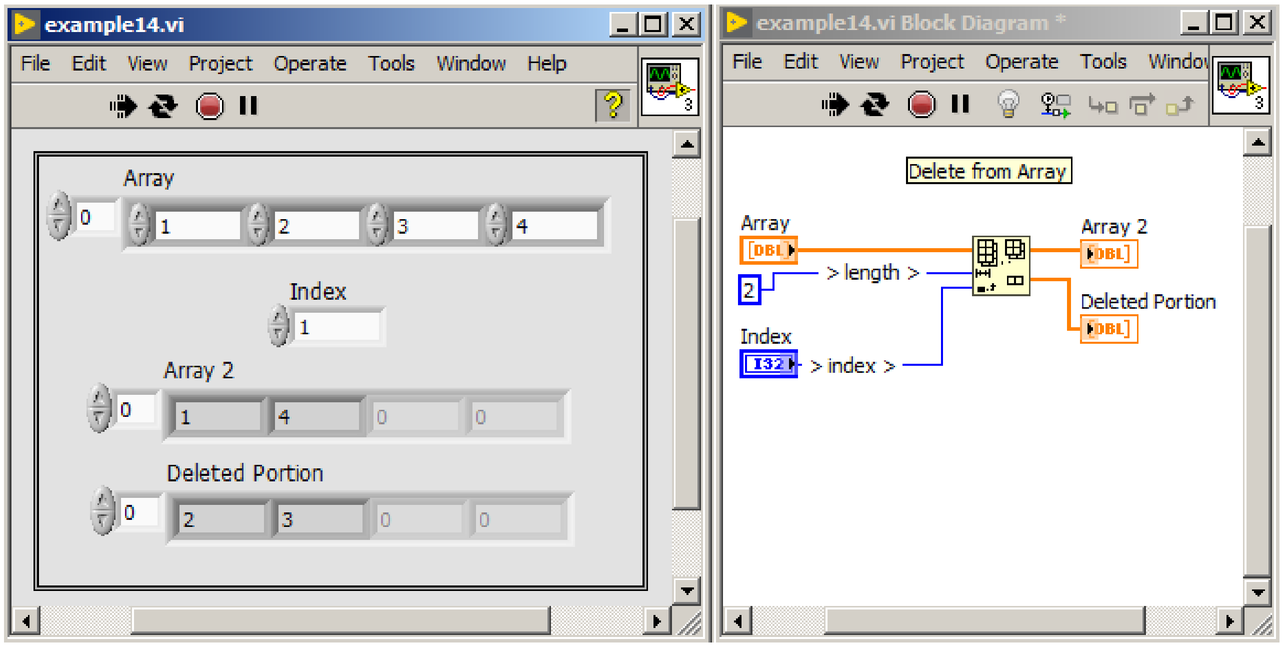


Figure 33: Program that demonstrates how to delete elements from array.

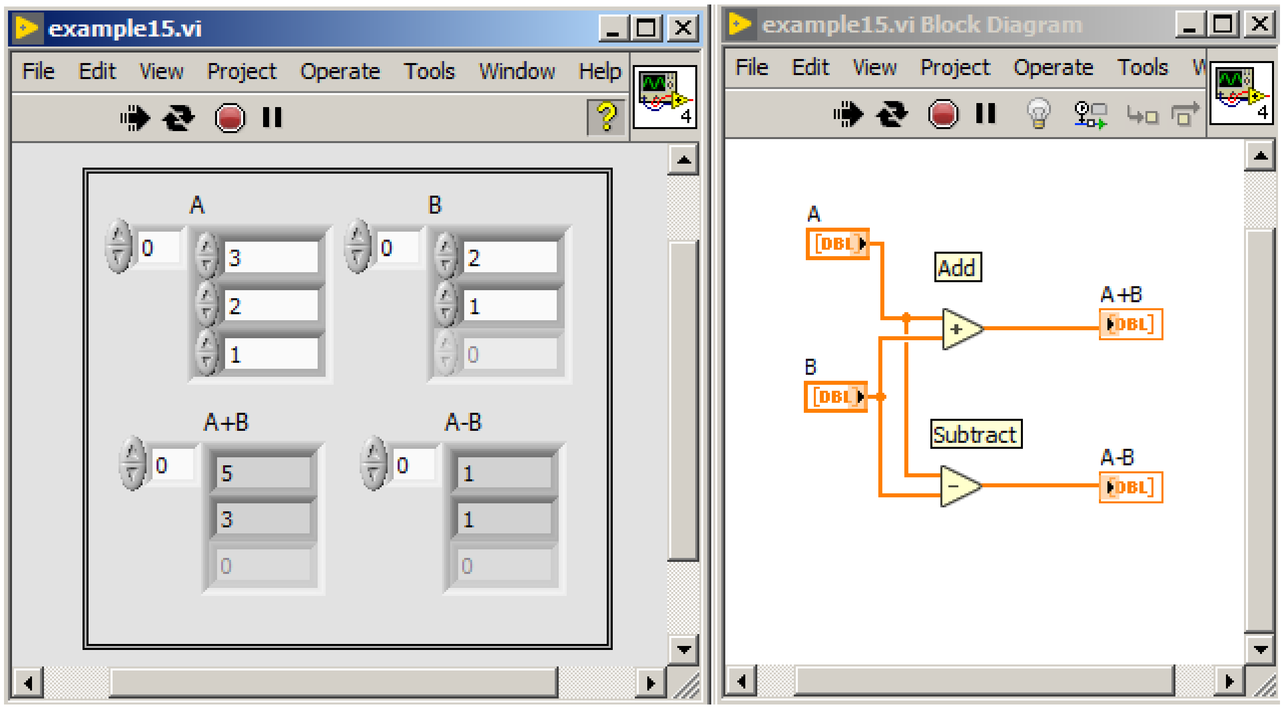

example15.vi
File Edit View Project Operate Tools Window Help
A
B
0
3
2
1
0
2
1
0
A+B
A-B
0
5
3
0
0
1
1
0
example15.vi Block Diagram
File Edit View Project Operate Tools
A
B
DBL
Add
Subtract
A+B
A-B

Figure 34: Program that demonstrates operation of numerical functions on arrays.

Clusters

Clusters are used for grouping data of different types, e.g. numerical together with logic values. They are equivalent to structures in textual programming languages. Elements in cluster can be of any type but elements in cluster should be either controls or either indicators. Clusters can be created in two ways. One of them is to put an *Cluster* object on front panel. *Cluster* is available in *Controls palette*→ *Array, Matrix & Cluster*. Next, selected elements should be dragged and dropped to *Cluster* object on front panel, which is shown in figure 35.

Cluster can also be created from existing controls or indicators using one of two functions: *Bundle* or *Bundle by Name*. In figure 36 an example of cluster creation with use of *Bundle* function is presented.

To modify an element in existing cluster, one needs also to use *Bundle* function, where existing cluster should be connected to top terminal, which is shown in figure 37.

To extract element from cluster, one needs to use *Unbundle* function, which is shown in figure 38.

Elements which are returned from *Unbundle* function are in such order how they were added to cluster. Sometimes it is convenient to change this order. To change order of the elements in cluster, one needs to click the right mouse button on the cluster border and select *Reorder Controls In Cluster* option. A window shown in figure 39 shows

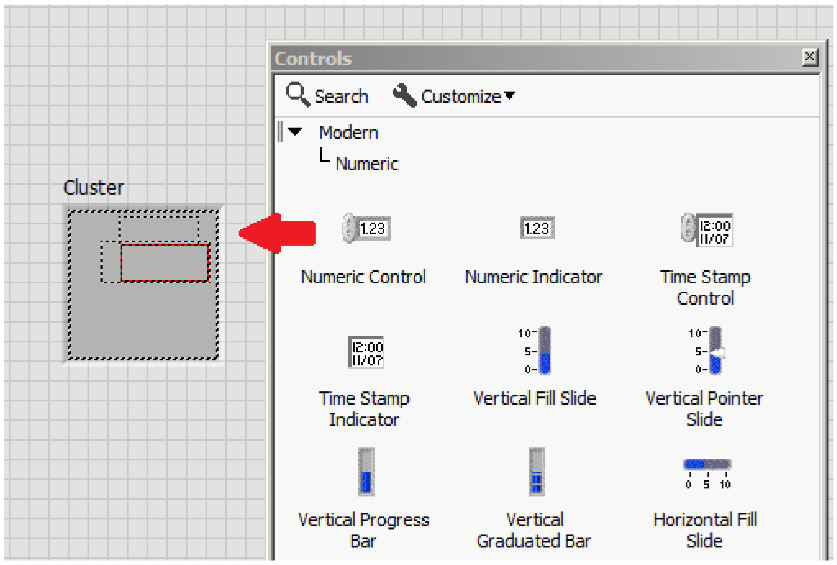


Figure 35: Adding an element to cluster on front panel.

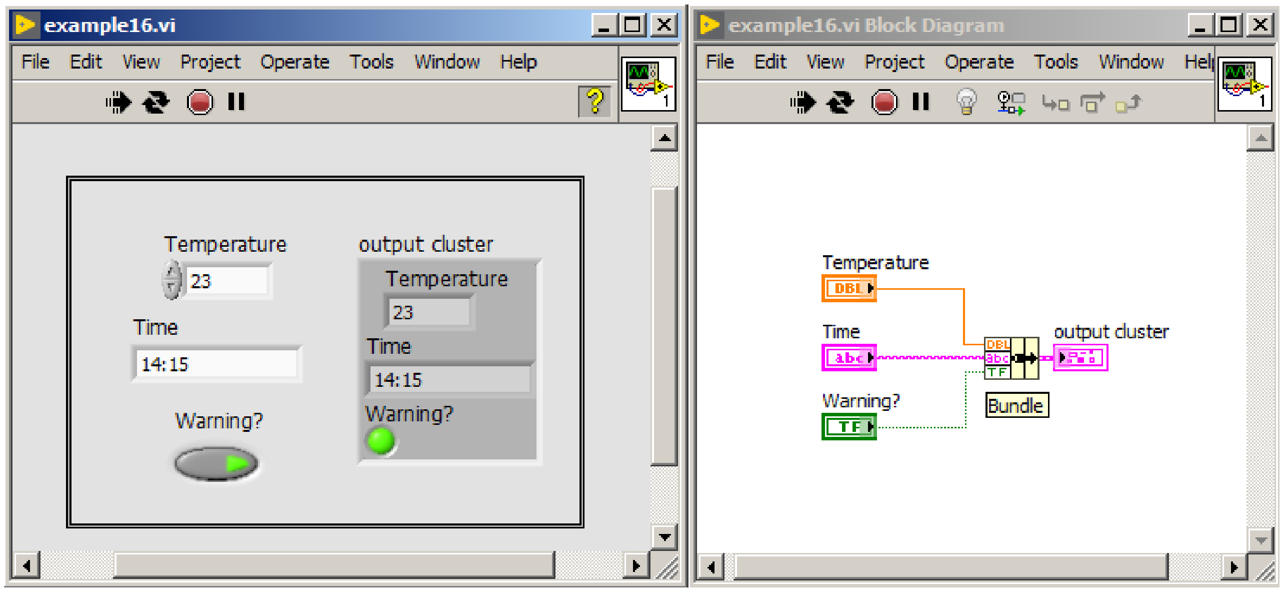

example16.vi
File Edit View Project Operate Tools Window Help
Temperature
23
Time
14:15
Warning?
output cluster
example16.vi Block Diagram
DBL
abc
TF
Bundle

Figure 36: Program which demonstrates cluster creation with use of existing controls and *Bundle* function.

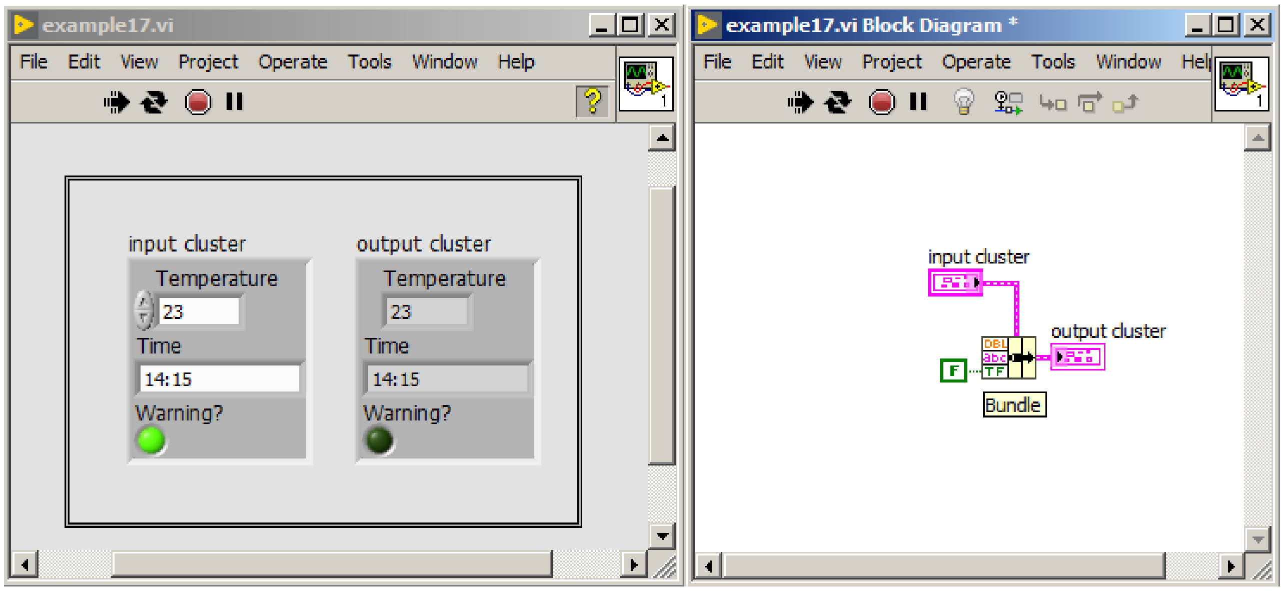


Figure 37: Program that shows cluster modification with use of *Bundle* function.

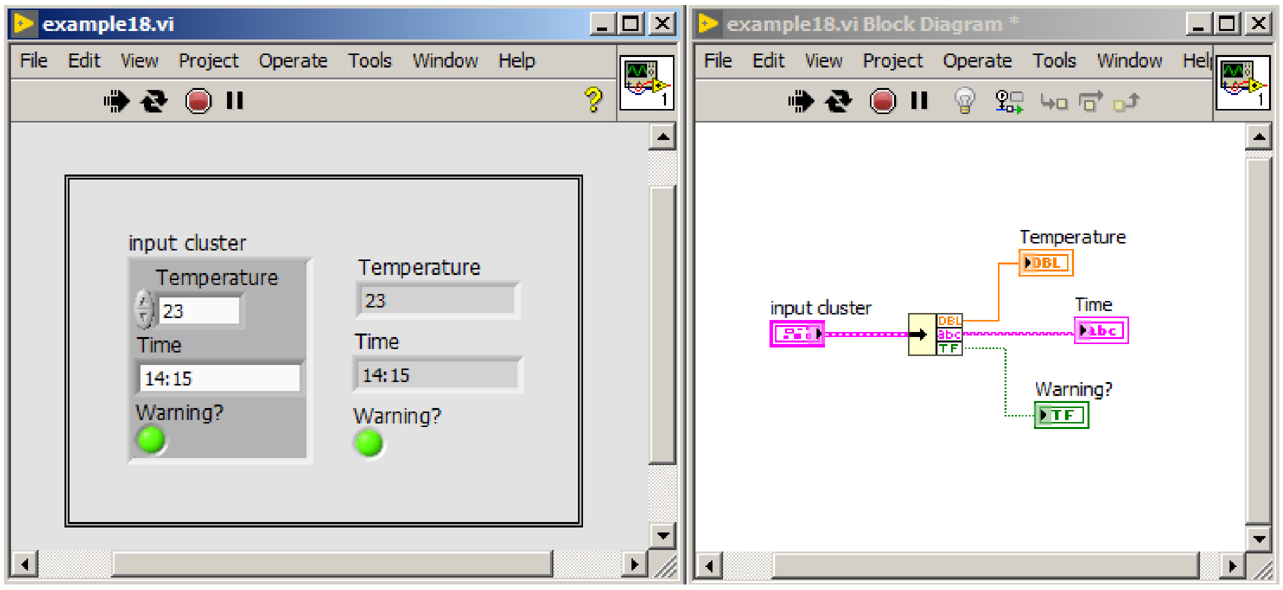

example18.vi
File Edit View Project Operate Tools Window Help
input cluster
Temperature
23
Time
14:15
Warning?
Temperature
23
Time
14:15
Warning?
example18.vi Block Diagram *
File Edit View Project Operate Tools Window Help
Temperature
input cluster
Time
Warning?

Figure 38: Program that shows extraction of element from cluster with help of *Unbundle* function.

up, where every cluster element is indicated by two numbers: one number is on a white background (actual element position in cluster), where second number is on a black background (new element position in cluster). After changes, new numbering should be confirmed by pressing "confirm" button.

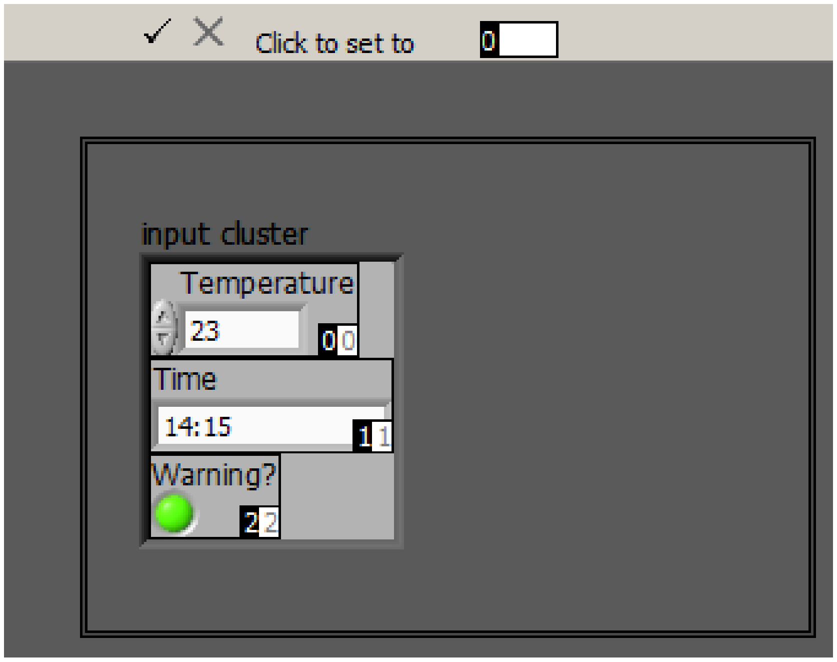


Figure 39: Change of elements order in cluster.

Creation, modification and extraction of elements from cluster with the help of *Bundle* or *Unbundle* functions may become cumbersome with a large number of elements in cluster. *Bundle* and *Unbundle* functions on input and output terminals shows only the data type, but not a name of control or indicator. In such cases, it is more convenient to use *Bundle by Name* and *Unbundle by Name* functions. In figure 40 a cluster creation with use of *Bundle by Name* function was presented.

Unbundle by Name function allows to extract selected elements from cluster. Such solution is particularly convenient when there is a large number of elements in cluster and we need only to extract one of them. An example of extraction of selected elements is shown in figure 41.

Type definition

In LabVIEW we can extensively customize controls and indicators for our needs, e.g. we can change scale, range, look of controls and indicators, and so on. If we would like to use customized control in many places, in case of further modifications, we should modify separately every instance of this control. Customization of such controls would be tedious and uncomfortable, so it is good to create *type definition* or

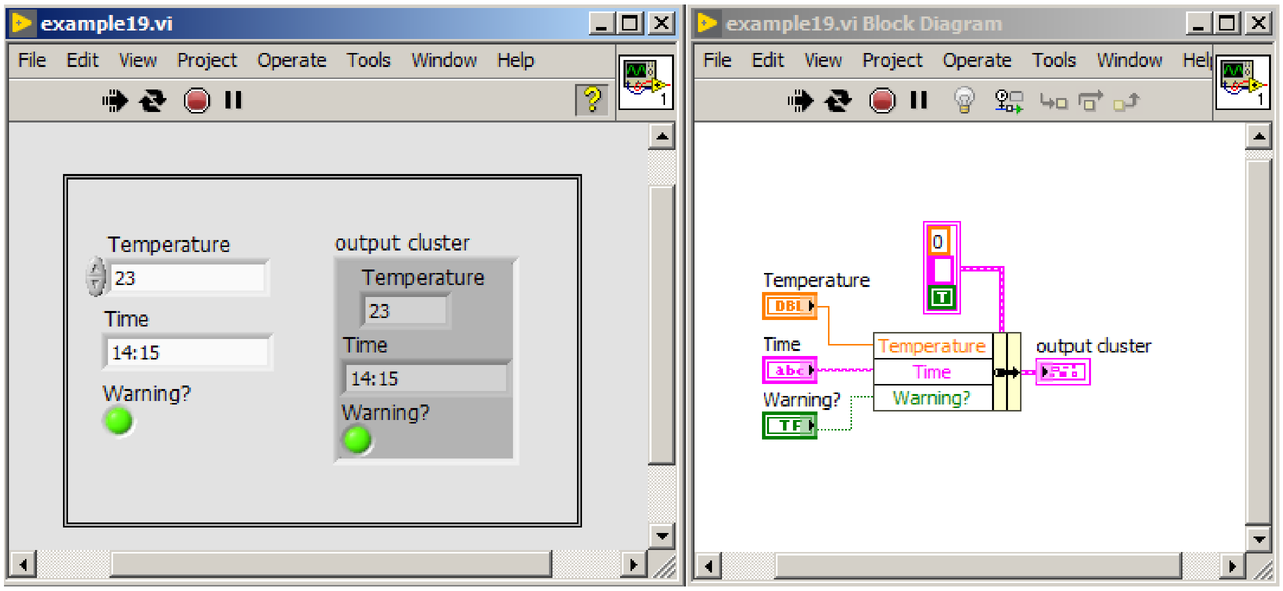


Figure 40: Program that shows cluster creation with use of *Bundle by Name* function.

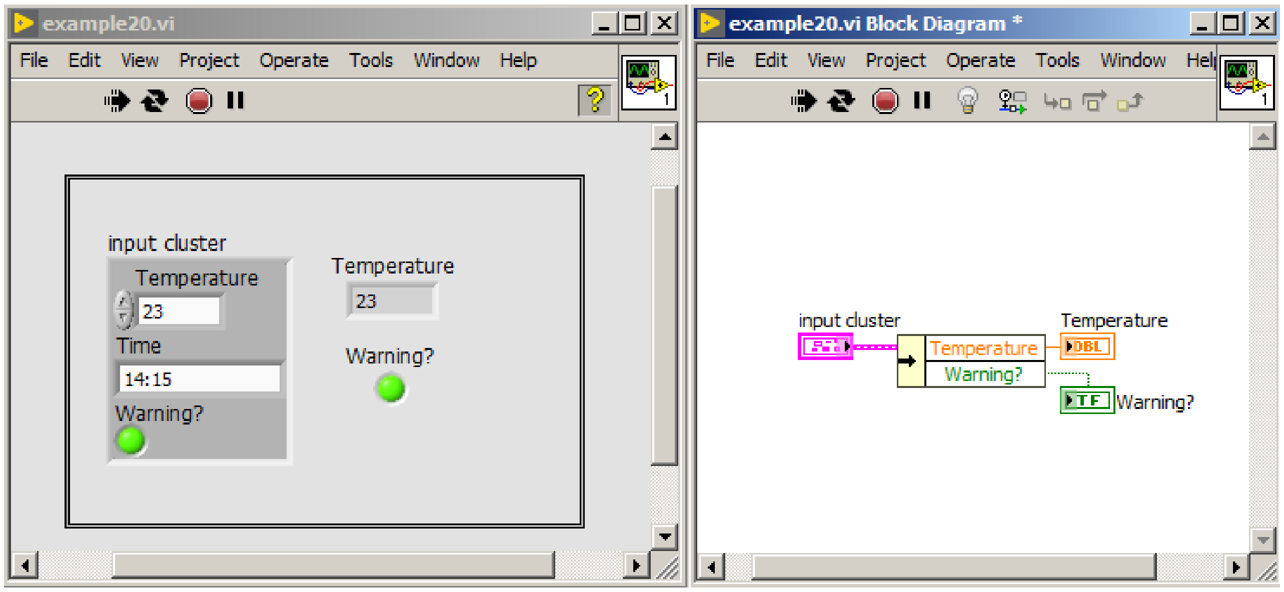


Figure 41: Program that demonstrates extraction of selected elements from cluster with use of *Unbundle by Name* function.

strict type definition.

If control is of type definition, then after data type change (e.g. from int32 to int64) or change of other property related to data type (e.g. addition of new element to enumeration type "enum"), all instances of control automatically will be also changed. However, still every instance would have unique properties like: caption, label, description, tip strip, default value, size, color, style of control or indicator.

If control is of strict type definition, then after change of control look, all instances will also change the look. Every instance will have unique: caption, label, description, tip strip and default value.

To set selected control as "type definition", one need to click the right mouse button on this control and select *Make Type Def.* option. Next again clink the right mouse button on this control and choose *Open Type Def.* option, which will open a window shown in figure 42. Now, it is possible to change parameters of control and to choose the type definition kind.

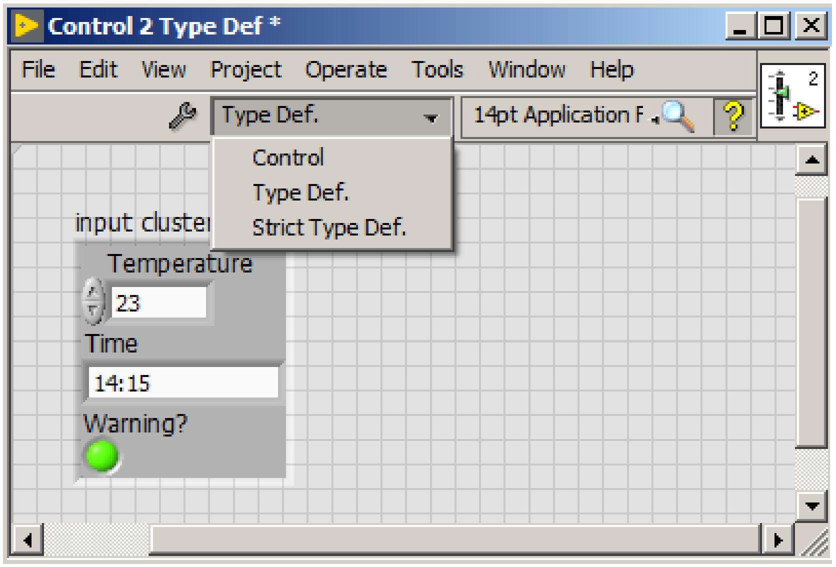


Figure 42: Modification of parameters of the control.

Loops and programming structures

Case structure

Case structure is equivalent of *if...else* structure in textual programming languages. It is used to take various actions in case of different conditions. Case structure consists of: case selector (where a condition is connected) and case label (possible states). To case selector, following data types can be connected: integer, boolean, string or enumerated type value. Components of case structure are presented in figure 43.

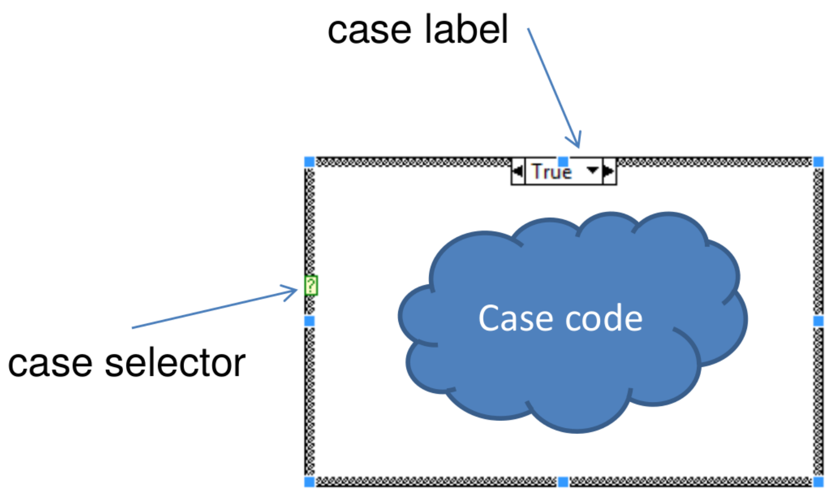


Figure 43: Components of the case structure.

In case of boolean value connected to case selector, two states *True* and *False* are possible. In figure 44 there is an example of case structure usage. To the case structure input, a button is connected. If the button will have *True* state, two values will be added, else subtraction operation will be performed.

In case of numerical value, string or enumerated type value connected to the case selector, it is needed to prepare actions for all possible states. In practice one of the frames is selected as *Default* and will be selected if any other condition is not met. In figure 45 an example of case structure usage for creation of simple calculator is presented. To the case selector, a numeric value is connected. In case of value of *Operation* control, specific case frame will be executed, so different numeric function will be selected like (addition, subtraction, multiplication, division),

If selected frame should execute not in case of single value, but in case of some range, so in case label one need to write range in such format: *starting value..ending value*, e.g. for range starting from 0 to 100, we should write 0..100.

Sometimes, it may happen that in frame of case structure there will be no value which should be returned and empty tunnel will be present. In such case program will not start, and we will get an error. As a solution for this you can click right mouse button on such tunnel and select *Use Default if Unwired* option (an example is presented in figure 46). In such case default value would be propagated to the indicator. Default values for basic data types in LabVIEW are shown in table 2.

For simple cases, instead case structure, one may use *Select* function, which have three input terminals. This function returns value connected to *True* (top terminal), if a *True* value is connected to the

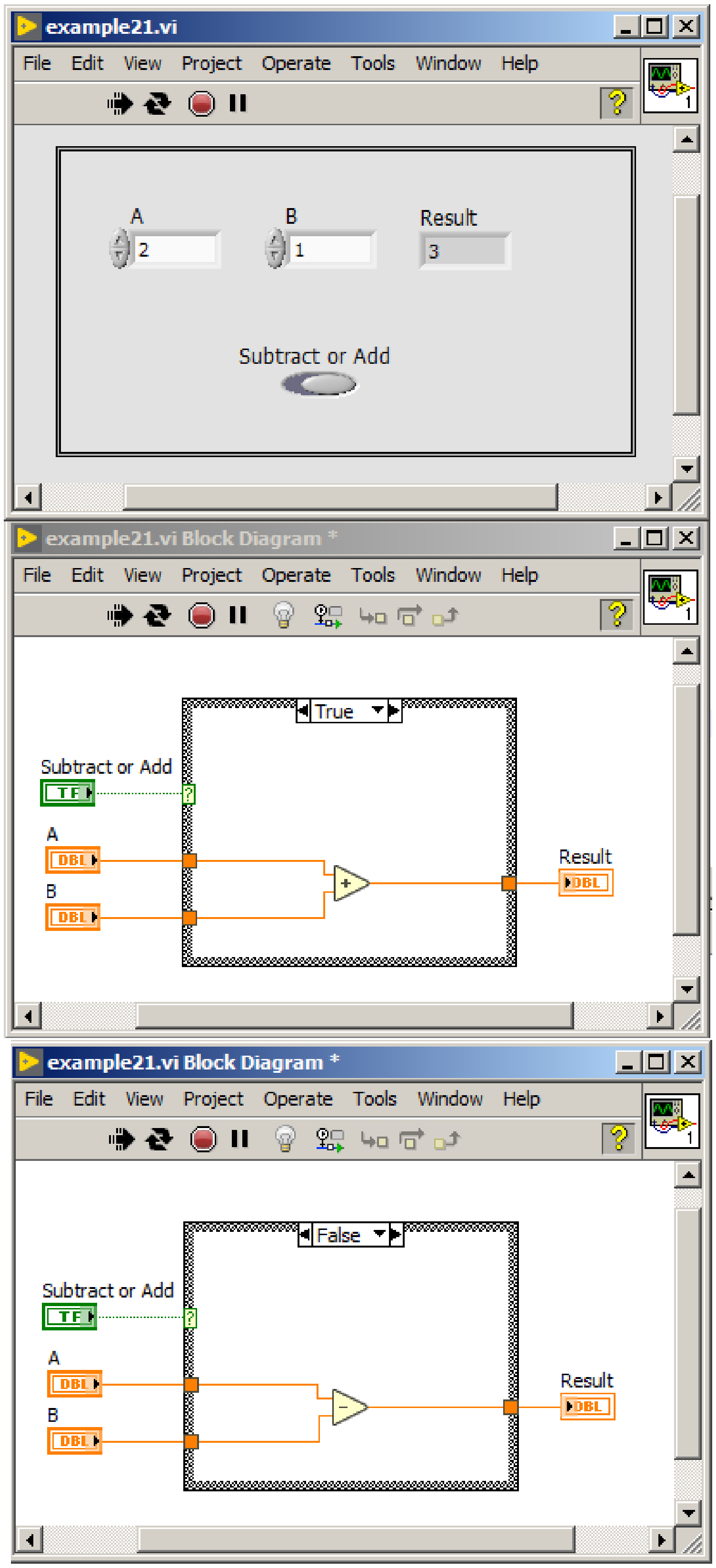


Figure 44: An example of case structure use with boolean value connected to the case selector

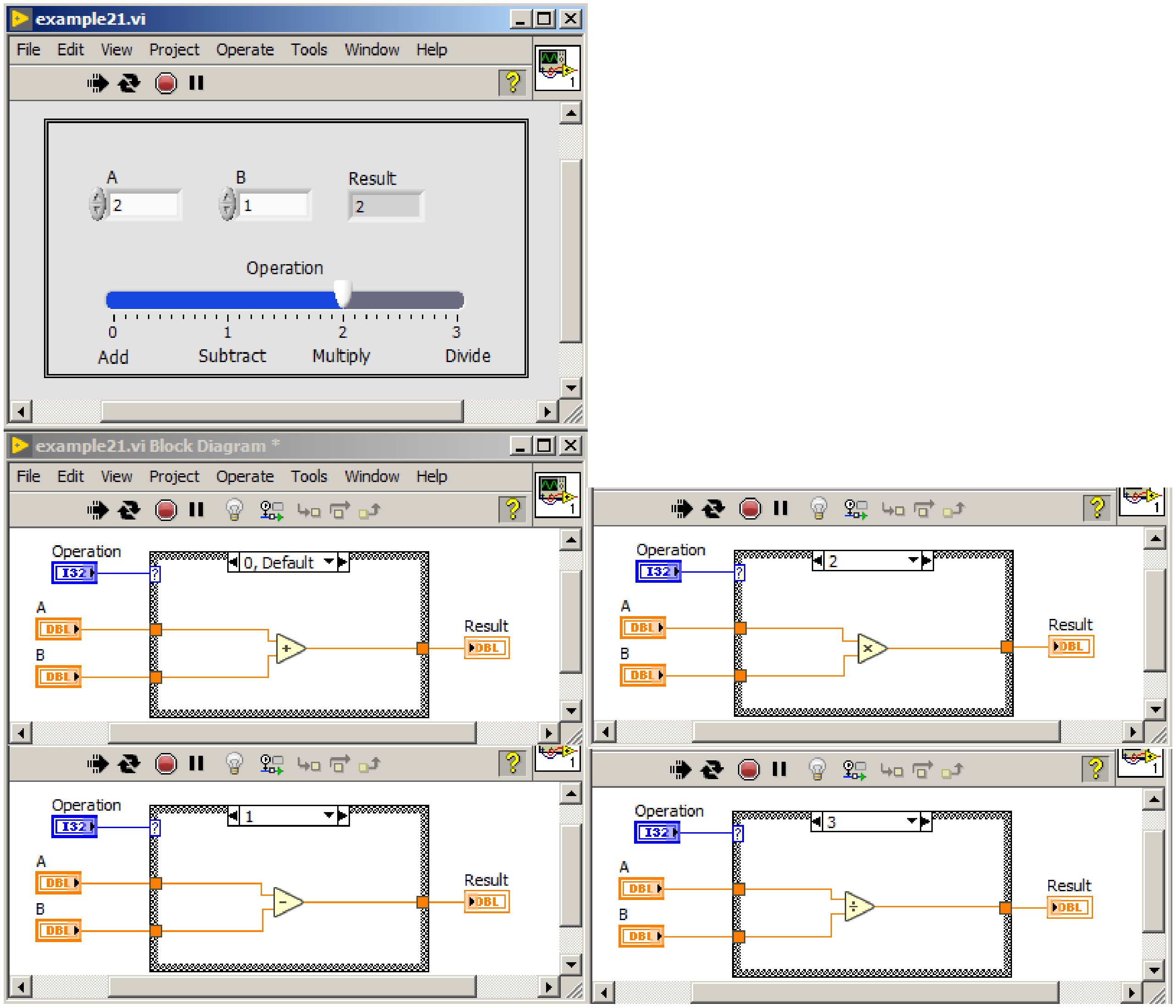


Figure 45: Simple calculator created with a help of case structure.

Data Type	Default Value
Numeric	0
Boolean	FALSE
String	empty ("")

Table 2: Default values of basic data types in LabVIEW. Source of information: "LabVIEW^{TM} Core 1 Course Manual" National Instruments, 2012.

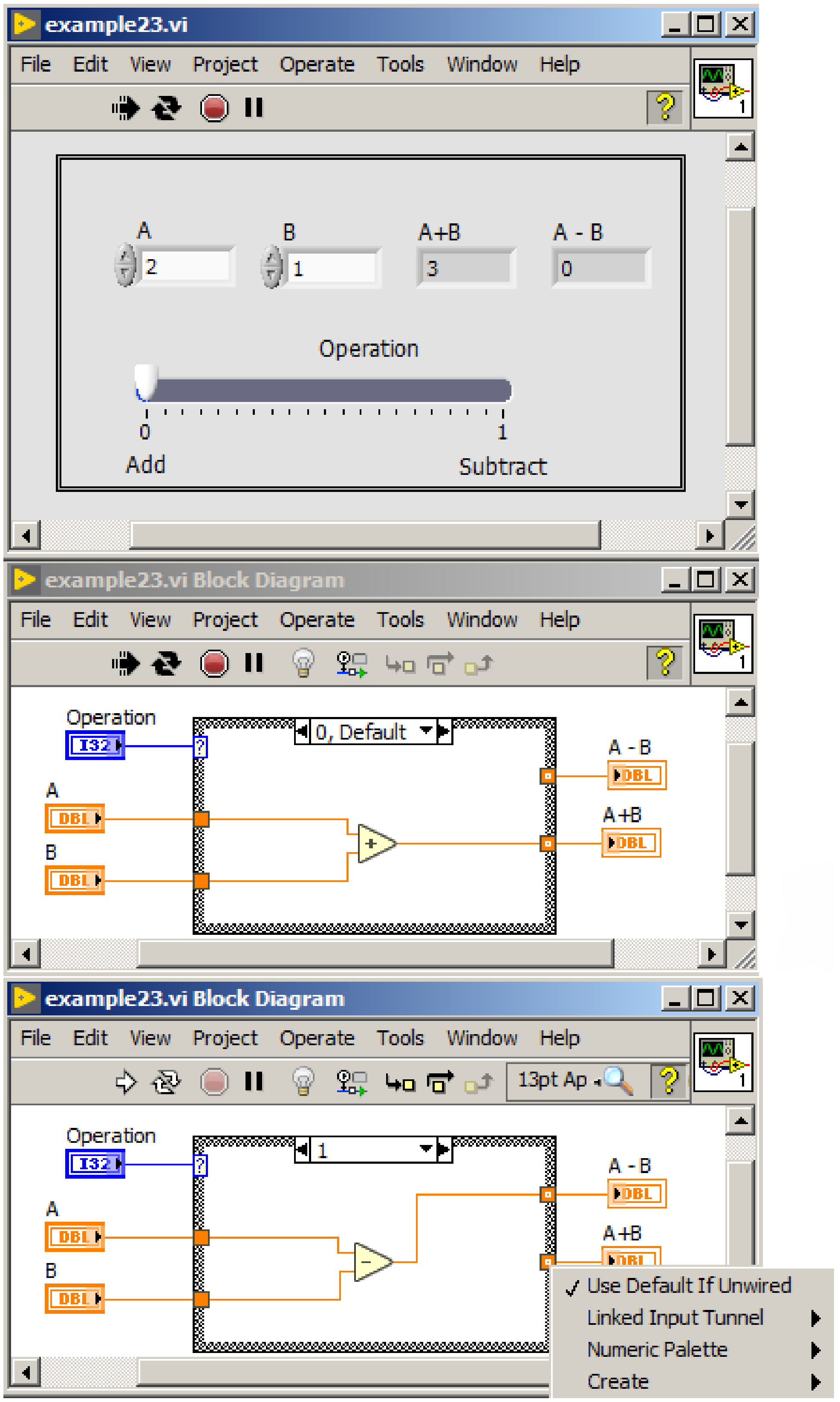

example23.vi
File Edit View Project Operate Tools Window Help
A
2
B
1
A+B
3
A - B
0
Operation
0
Add
1
Subtract
example23.vi Block Diagram
File Edit View Project Operate Tools Window Help
Operation
I32
A
DBL
B
DBL
0, Default
A - B
DBL
A+B
DBL
example23.vi Block Diagram
File Edit View Project Operate Tools Window Help
13pt Ap
Operation
I32
A
DBL
B
DBL
1
A - B
DBL
A+B
DBL
Use Default If Unwired
Linked Input Tunnel
Numeric Palette
Create

Figure 46: Simple calculator created with help of case structure with not used terminals.

middle input terminal. While to the middle terminal a *False* value would be propagated, so function will return a value connected to the *False* (bottom terminal). An example of *Select* function usage is presented in figure 47.

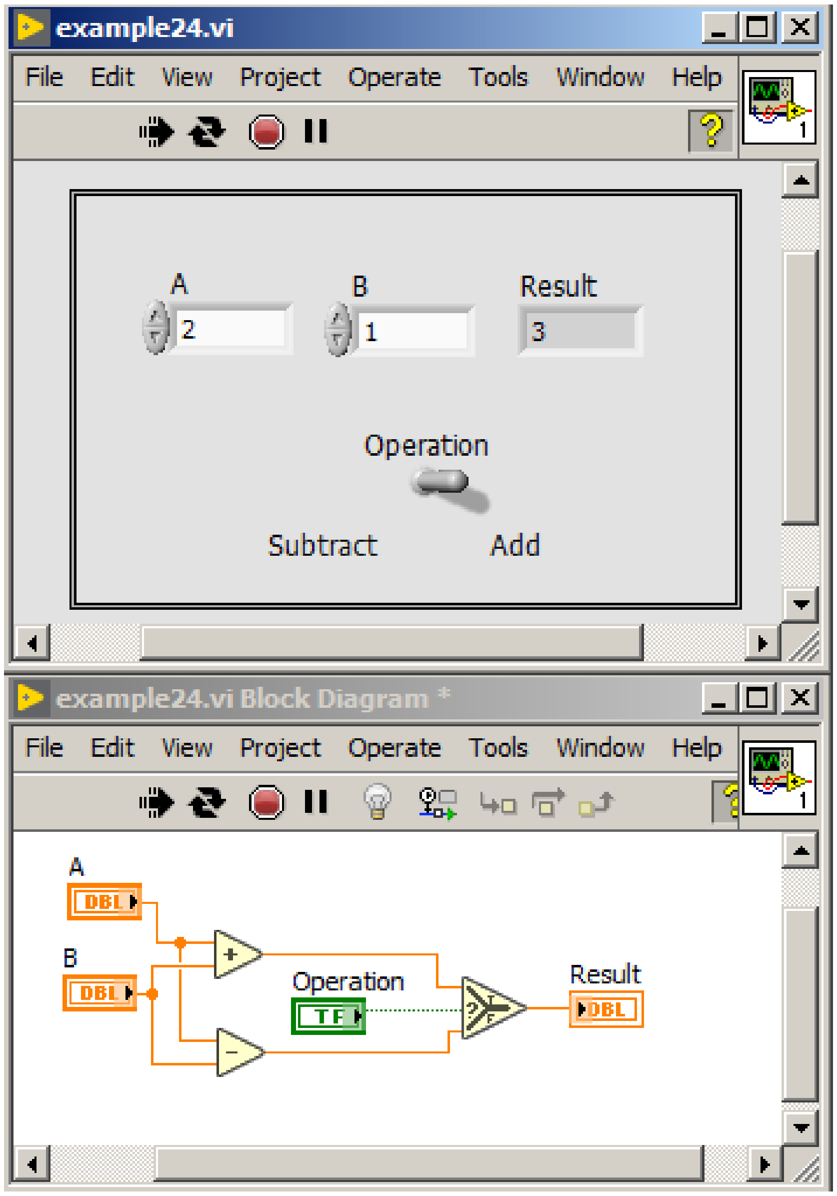

example24.vi
File Edit View Project Operate Tools Window Help
A
2
B
1
Result
3
Operation
Subtract
Add
example24.vi Block Diagram *
File Edit View Project Operate Tools Window Help
A
DBL
B
DBL
Operation
TF
Result
DBL

Figure 47: An example of *Select* function usage.

For loop

A for loop is used to execute some fragment of program multiple times. It consists of: *count* terminal and *iteration* terminal. First of them describes how many times the code inside the loop should be executed. While *iteration* terminal returns the number of finished iterations starting from zero.

An example program which demonstrates for loop usage is presented in figure 48. To count terminal input a value of 10 is connected, which means, that code inside the for loop will be executed 10 times. On the beginning of each iteration in loop, a random function tosses a value and passes it into input of chart of type: Waveform Chart.

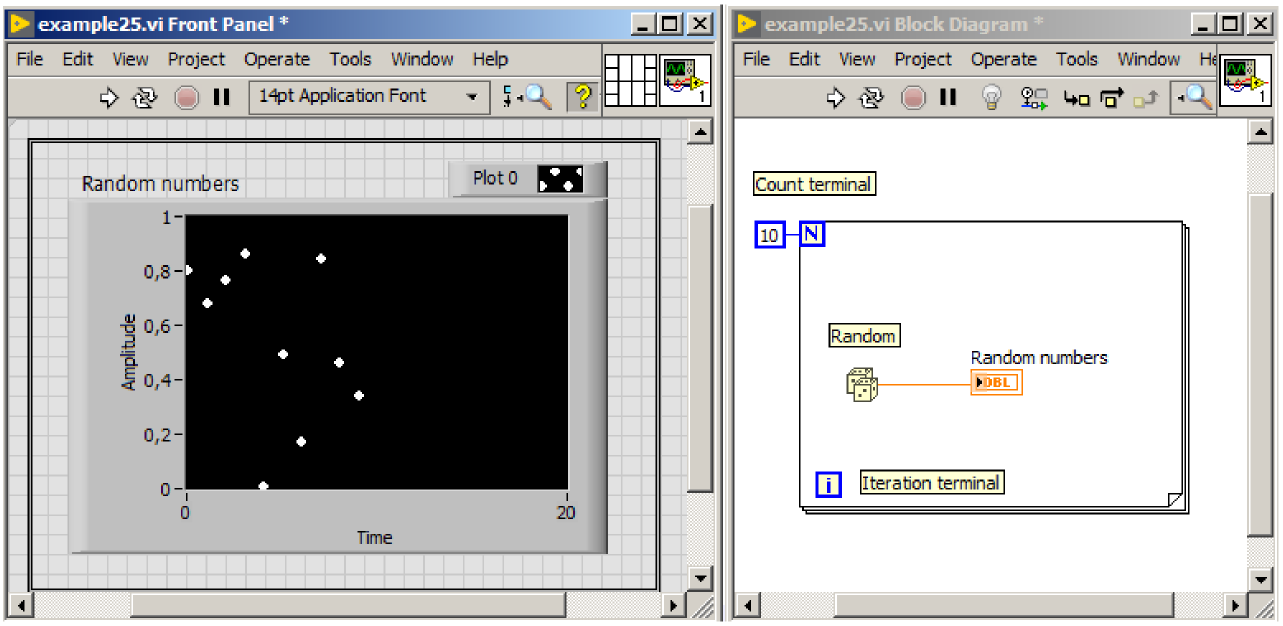

example25.vi Front Panel *
File Edit View Project Operate Tools Window Help
14pt Application Font
Random numbers
Plot 0
Amplitude
1
0,8
0,6
0,4
0,2
0
0
20
Time
example25.vi Block Diagram *
File Edit View Project Operate Tools Window He
Count terminal
10
N
Random
Random numbers
DBL
i
Iteration terminal

Figure 48: An example showing for loop usage, in which 10 random numbers are generated and displayed in chart.

Sometimes it is convenient to stop the for loop in example if error occurred or by user request. In such case, one need to click the right mouse button on for loop border and select *Conditional Terminal* option. Next to the newly created terminal, a logic value can be connected. In this case a *Stop if True* terminal is chosen. So a for loop may finish working during its repetitions if a *True* value is provided to conditional terminal. If we would like to stop repetitions by *False* value, it is needed to click the right mouse button on conditional terminal and select *Continue if True*.

An example in figure 48 was modified by adding a stop button. In the moment of button press by user, a for loop will end its work. Modified version of example is presented in figure 49.

Default tunneling type for the for loop is *Auto-indexing*. It means that, after connecting an array on for loop input, you don't need to add any value to count terminal. A for loop will execute exactly as many times as there are elements in array and in every iteration of for loop, next element of array will be available inside loop structure. Described situation is present in example in figure 50. In this example,

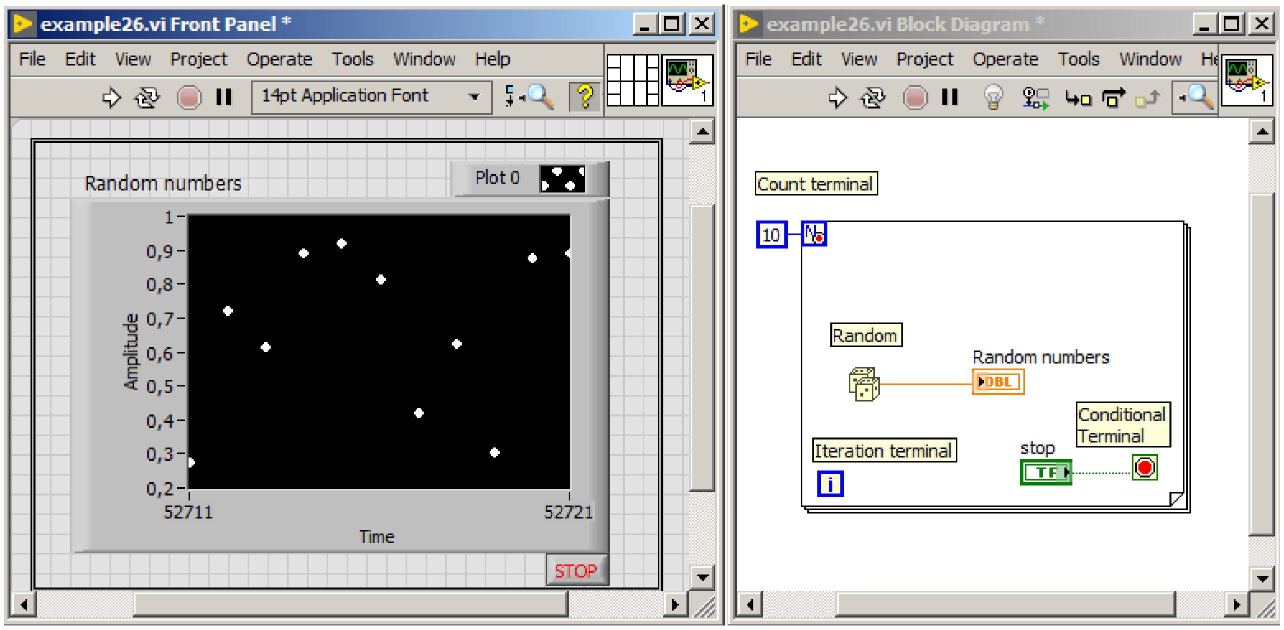


Figure 49: An example demonstrating for loop usage. 10 random numbers are tossed and displayed on chart. Program may finish its work after pressing the stop button.

in each iteration of for loop, a following value from array is read, and random number is added to this value. Result is passed to input of array named *Output array* which is placed outside for loop. As a result of auto-indexing tunneling, results are accumulated on loop border into the array and returned to *Output array* after finishing of for loop.

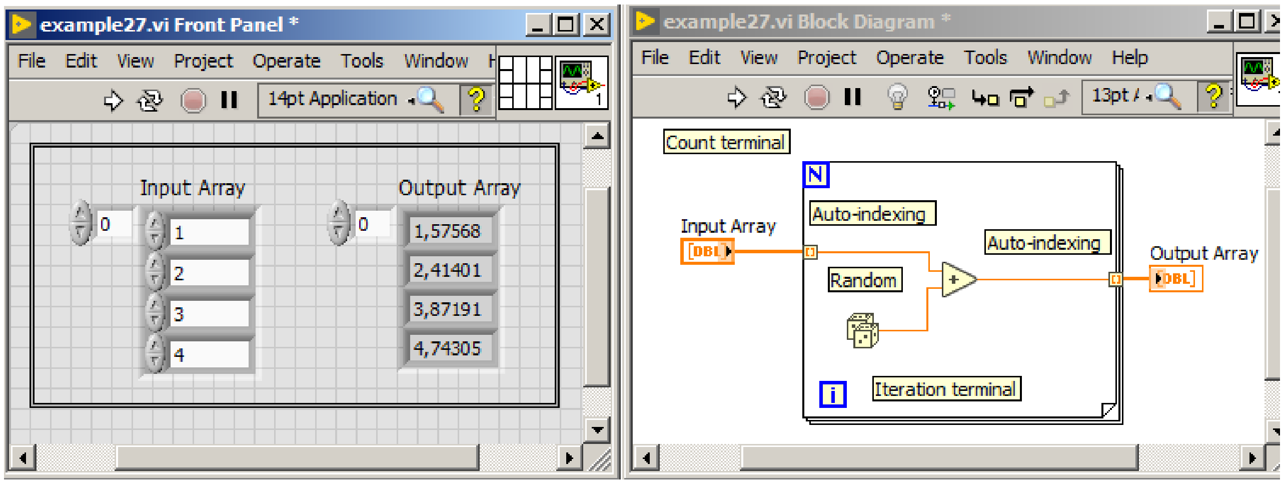


Figure 50: An example showing for loop usage with auto-indexing tunnel.

In case of arrays passed to for loop and a value passed to count terminal of for loop, the for loop will execute the least number of both conditions. In figure 51 an example is shown, in which two arrays are connected to for loop and a value of 2 is connected to count terminal. An *Input Array* consists of 4 elements and *Input Array* 2 consists of 3 elements. From these three numbers (4, 3 and 2 - count terminal), the least value is 2, so for loop will execute only two times.

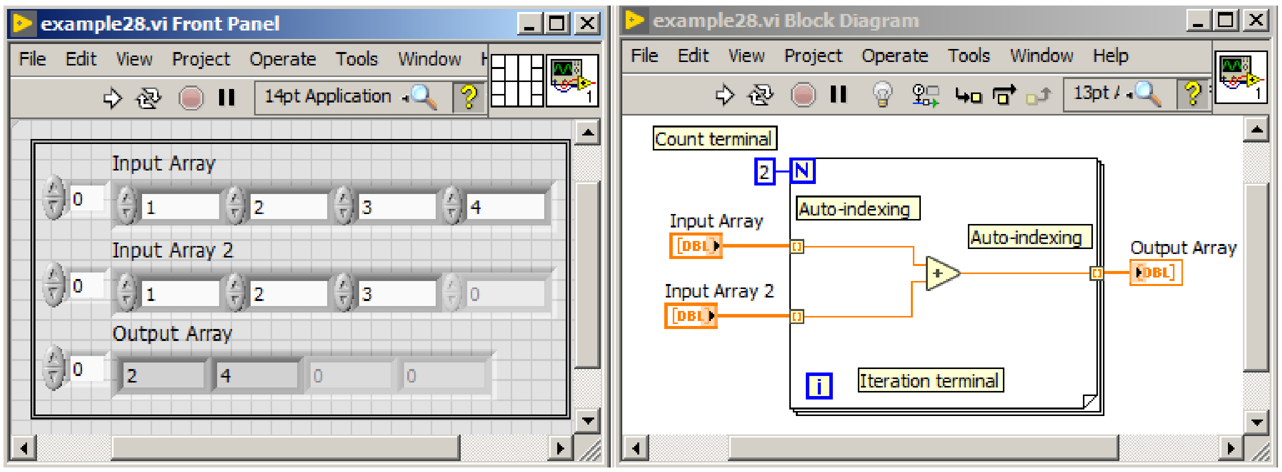


Figure 51: An example showing for loop working in case of multiple conditions affecting total number of iterations.

In LabVIEW there are three modes of tunneling: *Last Value*, *Indexing* and *Concatenating*. To change the tunneling mode, one needs to click the right mouse button on tunnel and select *Tunnel Mode* option.

While Loop

A while loop is used to repeat a fragment of code until some condition will be met. It consists of two terminals: *Loop Indicator* (which returns how many times a while loop was executed) and *Loop Condition* (condition when while loop should end which accepts values *True* and *False*). In figure 52, it is shown, that a loop finishes its work if on input *Loop Condition* a *True* value is passed (in case of *Stop if True* option selected) or *False* value is passed (in case of *Continue if True* option selected).

In figure 53 a while loop usage is presented. Program purpose is to generate random numbers from limited range (0;5) and comparison of them with user selected number. If both numbers are the same, program should end his work and should return an array of all generated values. In this program, a default tunneling mode was changed from *Last value* to *Indexing*. In case of this first mode, program would return last generated value only (which equals to user selected number). To return all values, a *Indexing* tunneling mode is necessary.

In table 3 a comparison of two types of loops: *for* and *while* is presented.

For Loop	While Loop
Executes N times or when condition occurs	Stop if condition is executed
Can execute 0 times	Must execute at least once
Tunnel mode is indexing by default	Tunnel mode is last value by default

Table 3: Comparison of two types of loops: *for* and *while*.

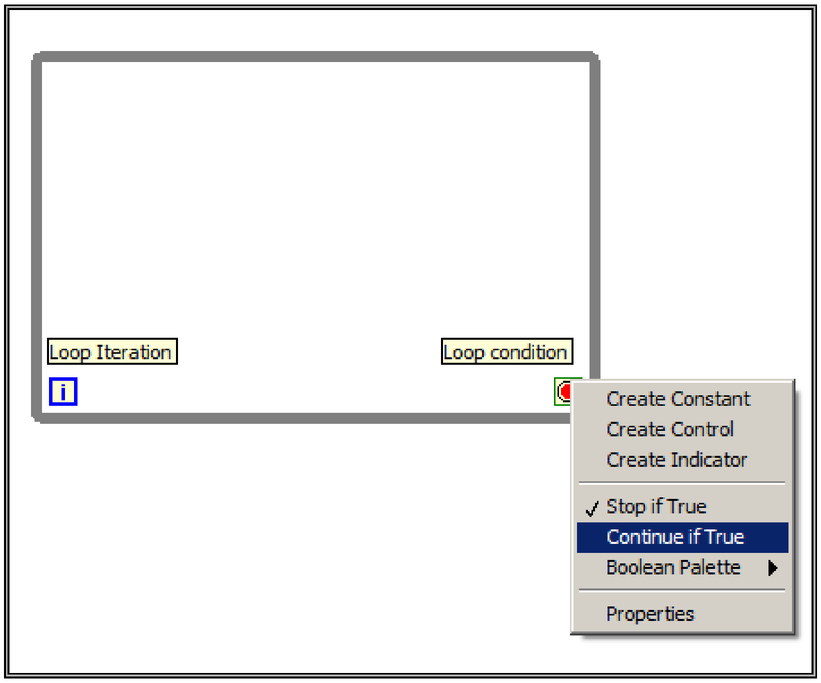


Figure 52: A while loop.

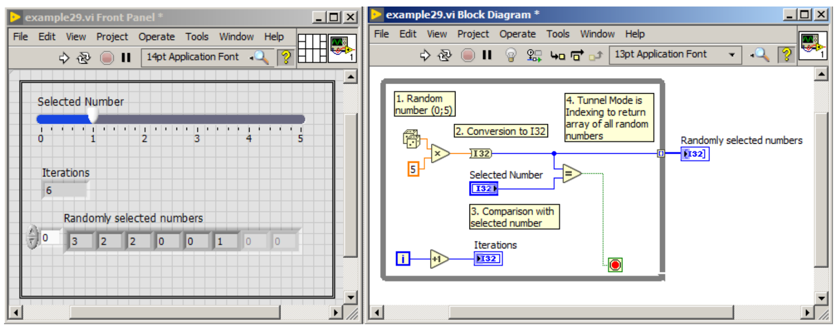

example29.vi Front Panel *
File Edit View Project Operate Tools Window Help
14pt Application Font
Selected Number
0 1 2 3 4 5
Iterations
6
Randomly selected numbers
0
3 2 2 0 0 1 0 0
example29.vi Block Diagram *
File Edit View Project Operate Tools Window Help
13pt Application Font
1. Random number (0;5)
2. Conversion to I32
4. Tunnel Mode is Indexing to return array of all random numbers
Randomly selected numbers
5
I32
Selected Number
3. Comparison with selected number
Iterations
i
+1

Figure 53: Program generates random number from range (0;5) as long as this number will be equal to user selected number.

Event structure

Event structure is used to call a specific fragment of program in case of an action, which will be performed by user, e.g. button click, mouse hover over chart, pressing a key. This structure consists of: *the Event Selector Label*, *the Timeout terminal*, *the Event Data Node* and *the Event Filter Node* (presented in figure 54). The event selector label defines event names, which are used in event structure. The timeout terminal defines how much time event structure will wait for event. Default value is -1, which means, that event structure waits forever for event. If event structure with timeout equal to -1 will be set inside while loop, then the next while loop iteration will not be started until the event configured in event structure will occur. The event data node is similar to cluster. It stores information about the event and its elements will change according to the event.

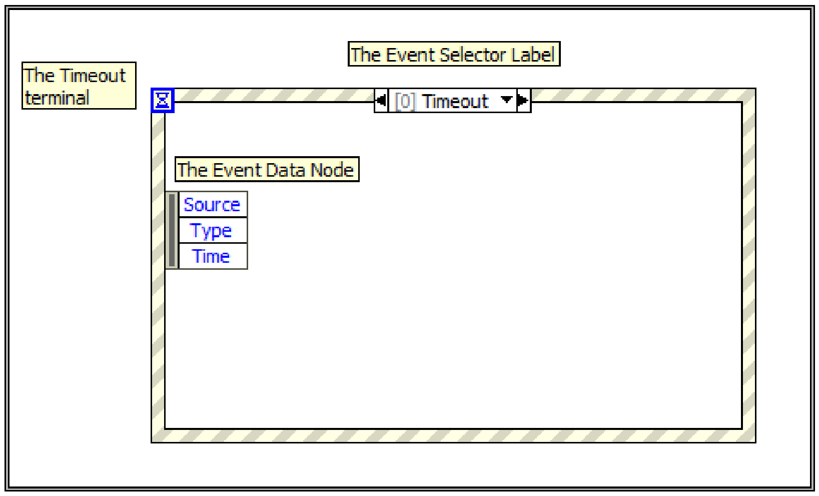
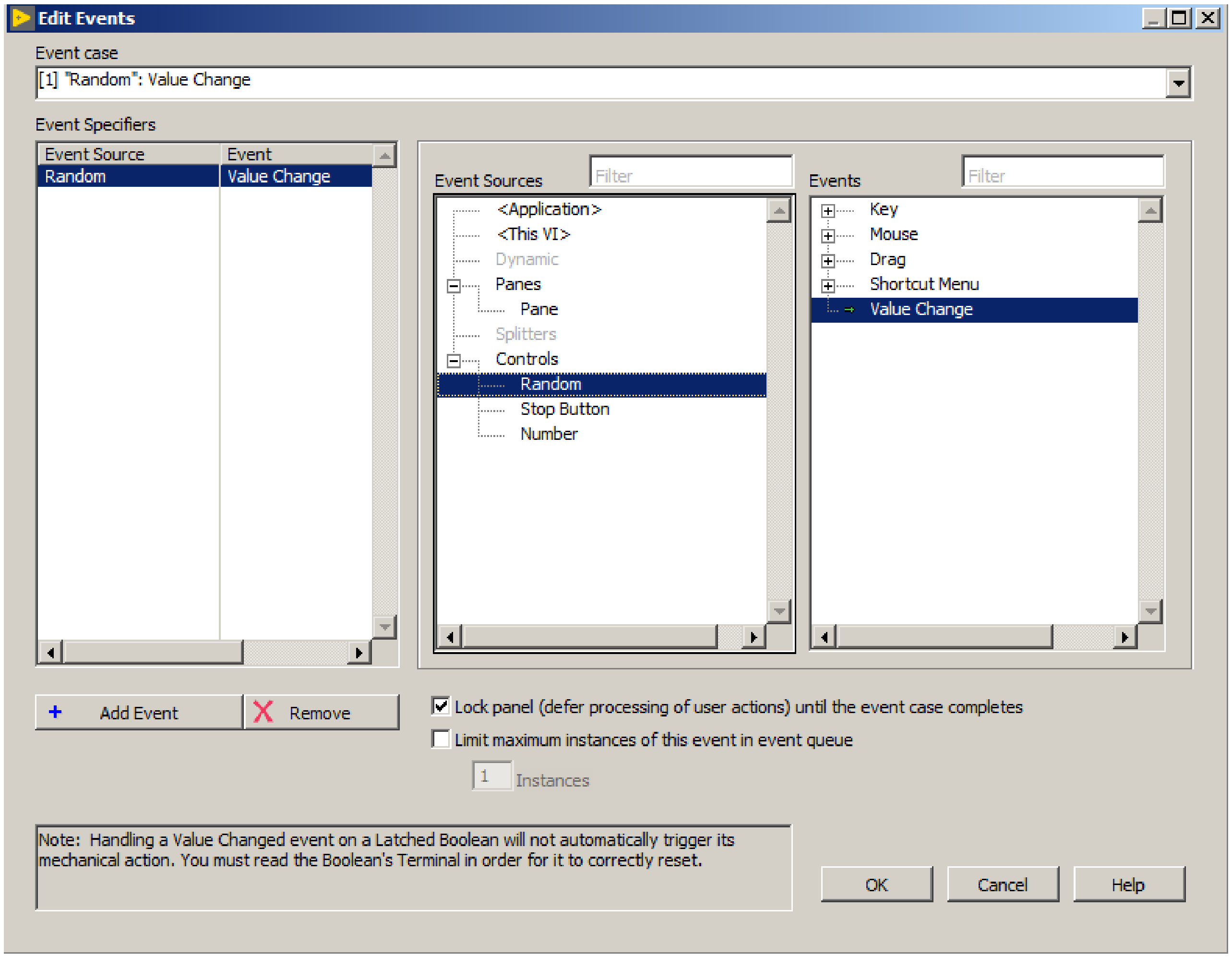


Figure 54: Event structure.

To add an event to the event structure, one needs click the right mouse button on the event selector label and choose *Add Event Case...*. A window presented in figure 55 will appear. In this window, in *Event Sources* tree view, there are all objects which may be configured as event sources. In *Events* tree view, there are all possible events that an object can generate. In figure 55, an event "Value Change" was configured to be generated, just when a value will change on button *Random*.

In figure 56 a program, which toss a random number from (0;1) range after *Random* button press is presented.


Edit Events

Event case: [1] "Random": Value Change

Event Specifiers

Event Source	Event
Random	Value Change

Event Sources
<Application>
<This VI>
Dynamic
Panes
Pane
Splitters
Controls
Random
Stop Button
Number

Events
Key
Mouse
Drag
Shortcut Menu
Value Change

[✓] Lock panel (defer processing of user actions) until the event case completes

[ ] Limit maximum instances of this event in event queue

1 Instances

Note: Handling a Value Changed event on a Latched Boolean will not automatically trigger its mechanical action. You must read the Boolean's Terminal in order for it to correctly reset.

Figure 55: Edit window for events in event structure.

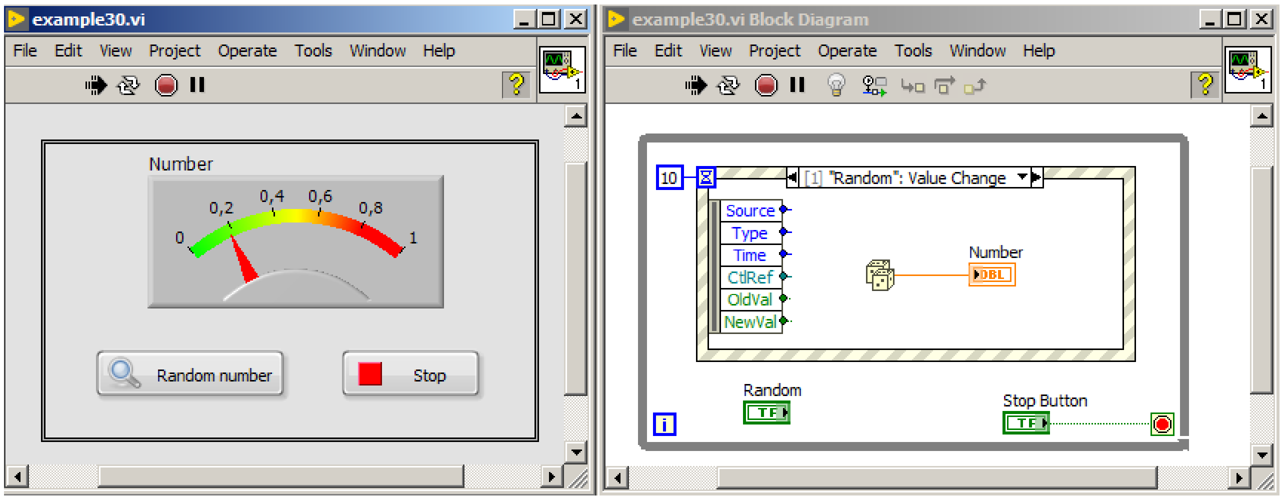

example30.vi
File Edit View Project Operate Tools Window Help
Number
0
0,2
0,4
0,6
0,8
1
Random number
Stop
example30.vi Block Diagram
File Edit View Project Operate Tools Window Help
10
[1] "Random": Value Change
Source
Type
Time
CtlRef
OldVal
NewVal
Number
DBL
Random
TF
Stop Button
TF
i

Figure 56: Program toss a random number from (0;1) range after *Random* button is pressed.

Flat Sequence

Flat Sequence gives possibility to control the order of operation execution in LabVIEW. It consists of frames, which are executed one after another. After selecting from function palette, flat sequence creates single frame. To add another frame, one needs to click the right mouse button on flat sequence and choose *Add Frame After*, which is presented in figure 57.

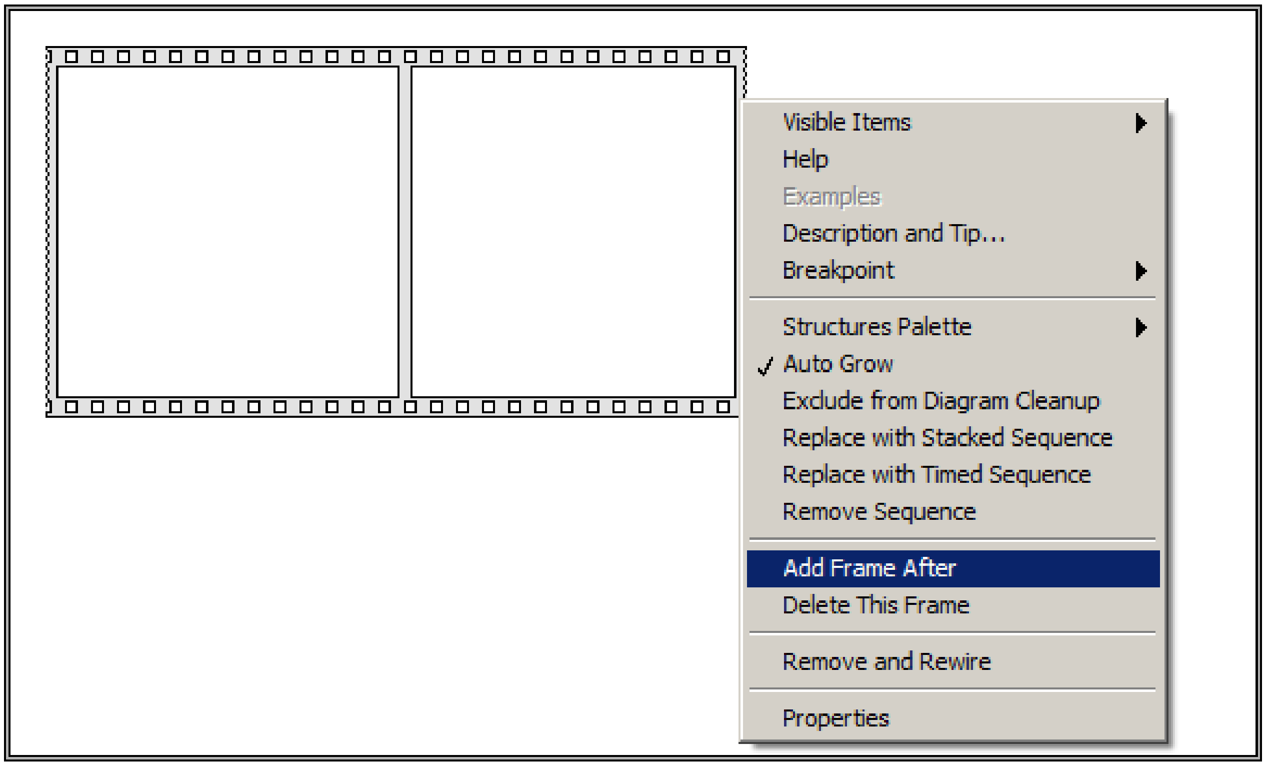


Figure 57: Flat sequence.

In figure 58 a program executing mathematical operation $A \cdot (A + B)$ with use of flat sequence is presented. In the beginning, only left frame will be executed, where A and B values will be added. Next the right frame will be executed, where result from previous one will be multiplied by A value.

Adding multiple frames in flat sequence will create big and unreadable diagram, so it is important to limit flat sequence usage to necessary minimum. However in such case control might be realised with state machine design pattern. More on this subject in chapter .

subVI

Repeating fragments of program should be enclosed in subVI, which is similar to function in textual programming language. In figure 59 a

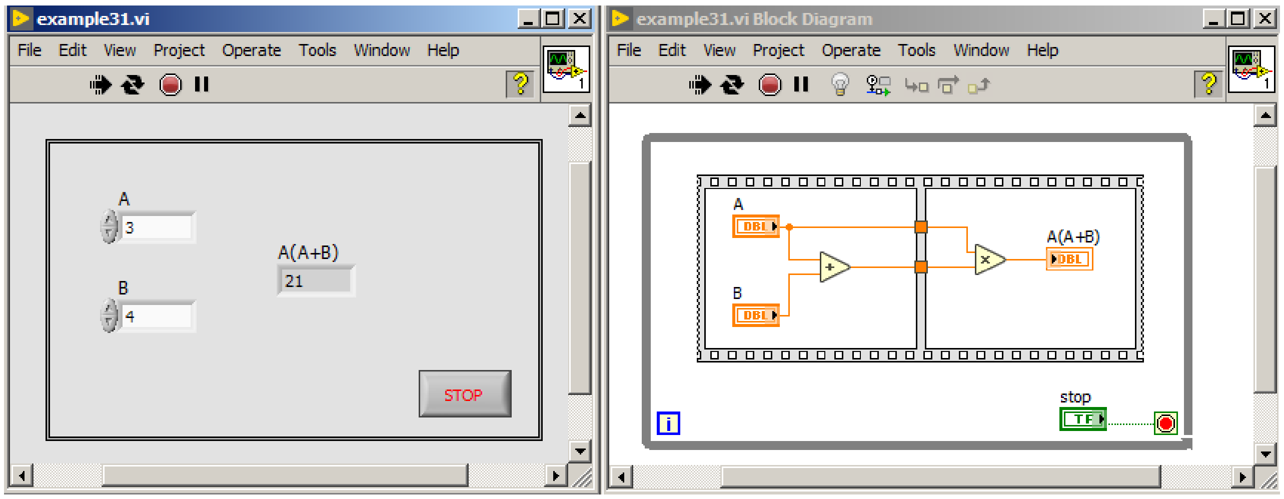

example31.vi
File Edit View Project Operate Tools Window Help
A
3
B
4
A(A+B)
21
STOP
example31.vi Block Diagram
A
B
A(A+B)
DBL
stop
TF

Figure 58: Program demonstrating flat sequence operation.

program, which calculates the mean value from array elements *Array 1* and *Array 2* is presented. After mean value calculation, program compares if the mean value from *Array 1* is bigger than mean value from *Array 2*. Using the *Feedback Node* on lower input of *Add* function gives possibility to pass the result of addition from previous *for loop* iteration to input of the function. In first iteration *Feedback Node* returns 0.

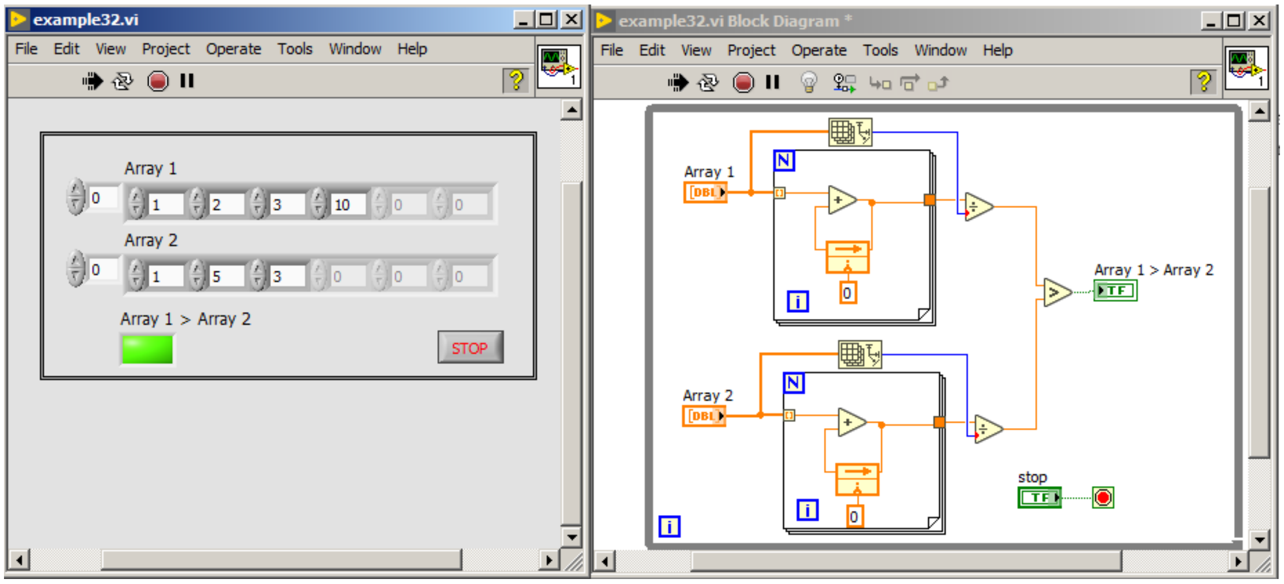


Figure 59: Program, which compares mean values from two arrays.

Presented program is not optimal (block diagram is not clean visually), because for two arrays the same operations are duplicated. To correct this, the reusable code fragment should be enclosed in subVI. To do so, reusable code is selected and *Edit* → *Create subVI* option is chosen. Created subVI is also used for the second array. The look of subVI is accordingly modified. The result is presented in figure 60.

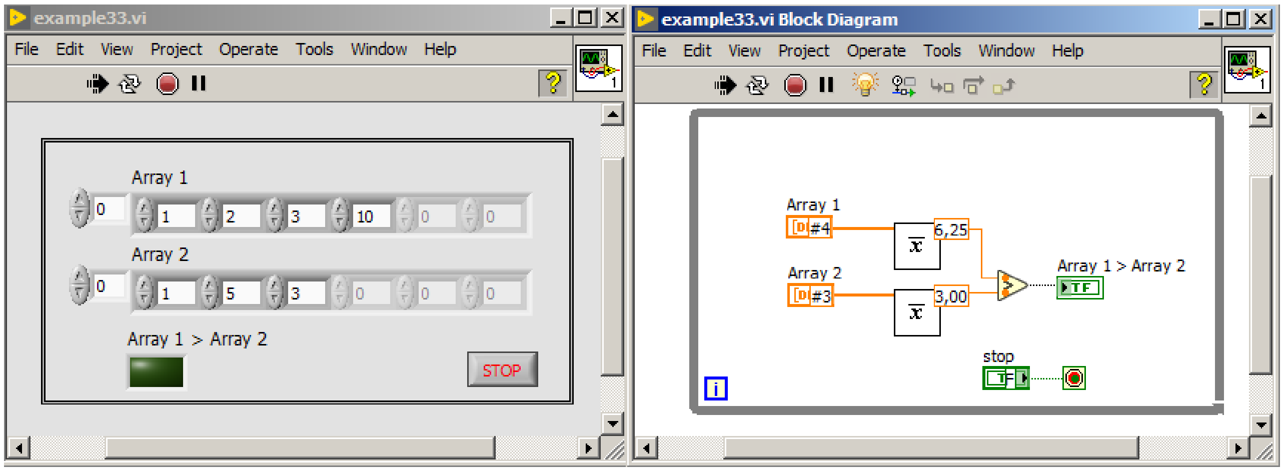
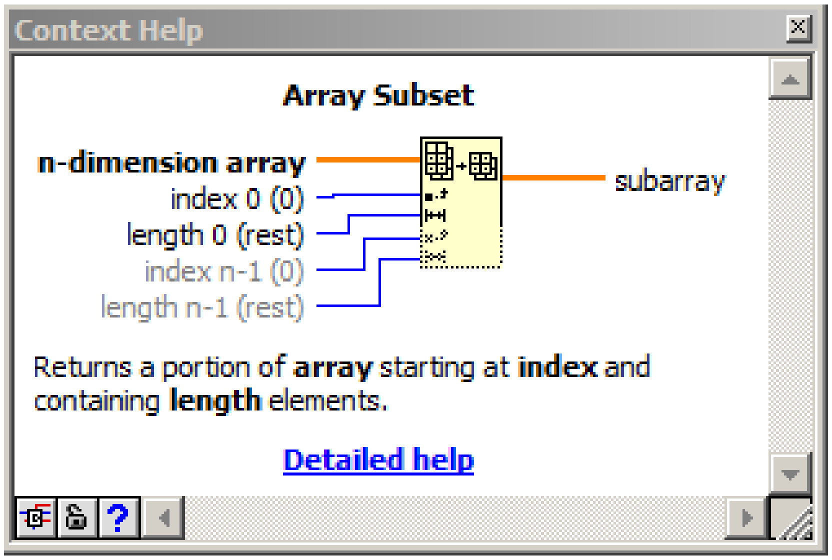


Figure 60: Program, which compares mean values from two arrays. Repeatable fragment of code replaced by subVI.

Another way to create subVI from existing VI is by connecting terminals from connector pane. To do so, on front panel using wire spool tool one needs to click the selected control or indicator and next on free terminal of connector pane. After saving this VI, it can be put in another VI (as a subVI) just by drag and drop technique - dragging the subVI icon (top right part of window) to the block diagram of another VI.

When creating subVI is it advised to use 4x2x2x4 (left side, top side, bottom side, right side) connector pane pattern of terminals. It is a convention that terminals on the left side are for inputs (there are mostly controls connected) and terminals on the right side are outputs (there are indicators connected). An error cluster by convention comes into bottom left corner of subVI and goes out from bottom right corner. A reference by convention comes into top left corner of subVI and goes out from top right corner. In such way subVIs can be connected in cascade way easily.

Sometimes created subVI is so general, that not always all input terminals are used. It is a good practise to precise which terminals are *Required*, *Recommended* or *Optional*. In case of this specification, following terminals will be displayed in different styles in context help window. Required connections will be shown with bold text, recommended connections will be shown in plain text and optional connections will be visible as dimmed text. In figure 61 a subVI with required, recommended as well as optional terminals is presented.



Returns a portion of **array** starting at **index** and containing **length** elements.

Figure 61: Context help window demonstrating subVI with required, recommended and optional terminals.

Time control

When a program controlling hardware devices is created, it is important to control time between following steps, e.g. for setting parameters of some specific oscilloscope one needs to wait about 100 ms between commands, which are sent. In such case functions from *Timing* fold from function palette are needed.

Two basic functions for time control are: *Wait (ms)* and *Wait Until Next ms Multiple*. First of them generates delay equal to specified value on input (in milliseconds). In figure 62 an idea of *Wait (ms)* function operation is presented, where a while loop with *Wait (ms)* function inside is used. If parallel code execution (inside while loop) will take more time than specified delay of *Wait (ms)* function, then following iteration of while loop will start immediately just after finishing all the operations in while loop frame. However if parallel code will take less time then specified delay (e.g. 5 ms), so the next iteration would start after 10 ms, just after finishing of *Wait (ms)* function.

Second function *Wait Until Next ms Multiple* waits for multiple times of the time period specified on input. Idea of this function operation is presented in figure 63. Code inside while loop will take 15 ms. So function *Wait Until Next ms Multiple* will wait until first multiplication of 10 ms possible to stop working. The nearest possible multiplication of 10 ms is 20 ms (because parallel task will take 15 ms). So next iteration of while loop will start after 20 ms of elapsed time.

Using both functions: *Wait (ms)* as well as *Wait Until Next ms Multiple* provides time to the processor to perform other tasks like e.g. graphical user interface handling.

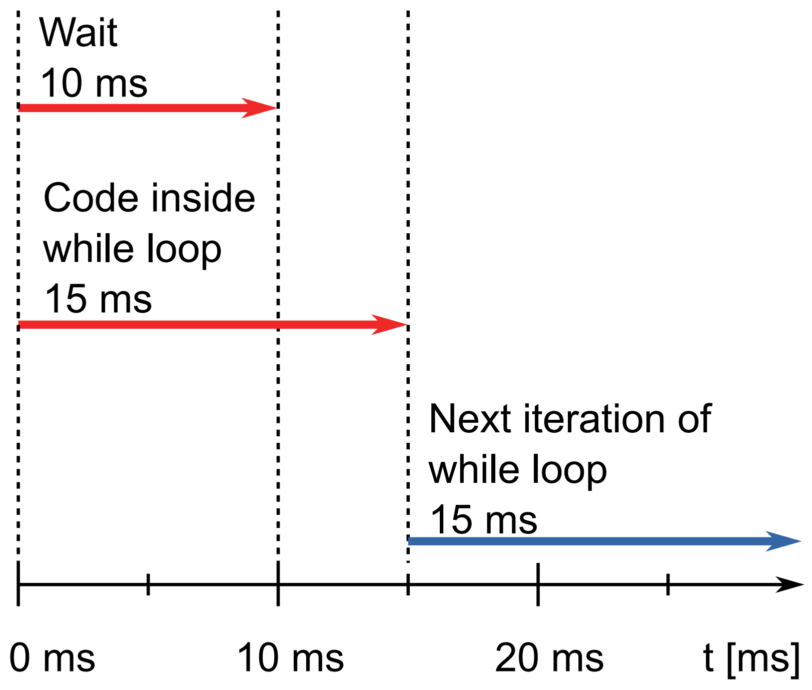


Figure 62: Principle of *Wait (ms)* function operation.

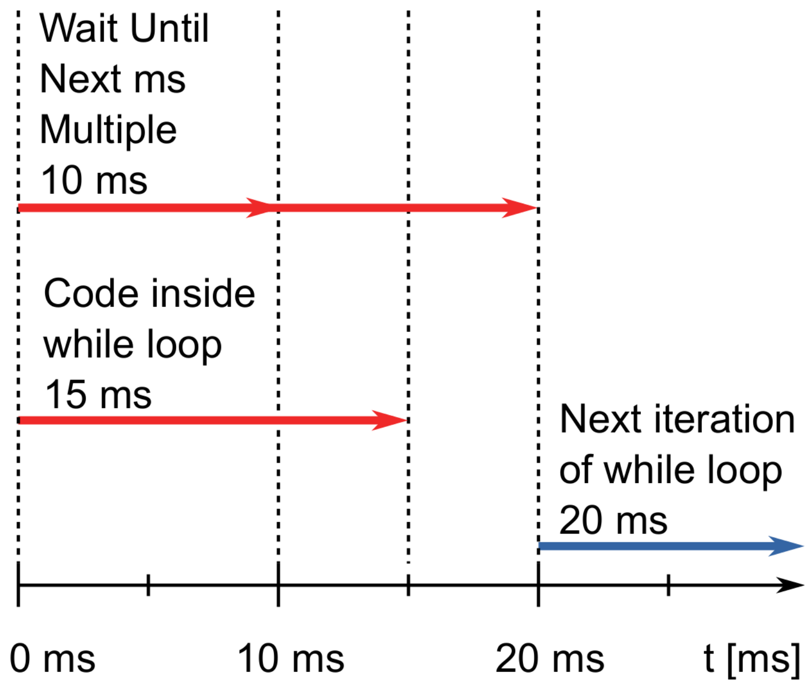


Figure 63: Principle of *Wait Until Next ms Multiple* function operation.

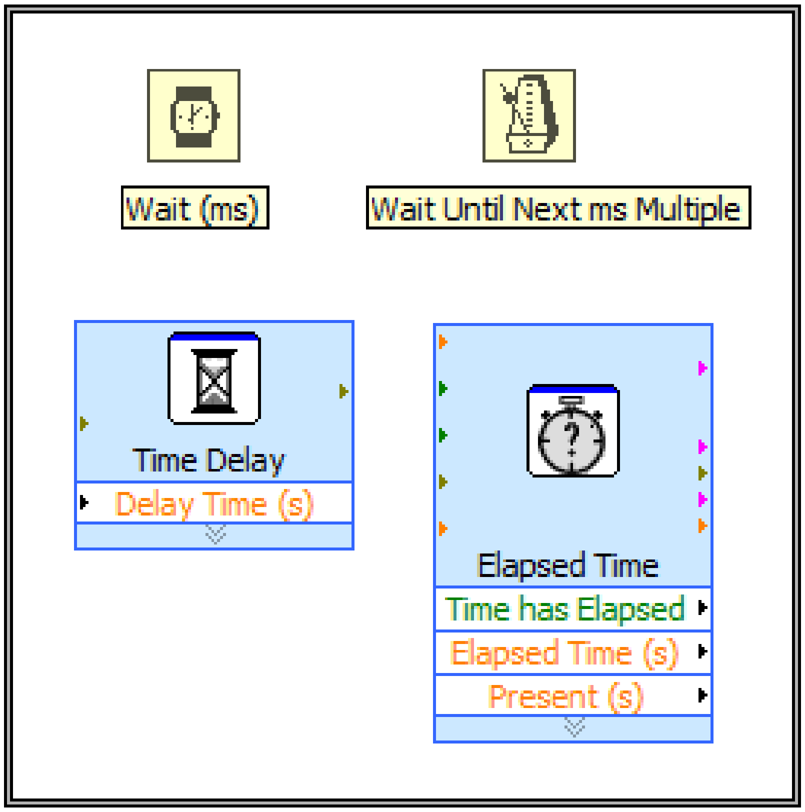


Figure 64: Basic functions to control time available in LabVIEW.

Besides basic functions to control time, there are other functions, like so called express VI functions: *Time Delay* and *Elapsed Time*. First of the works similarly to *Wait (ms)* function. However second of them returns information, how much time elapsed from selected moment. Both of them doesn't provide time to the processor to perform other tasks - this in contrast to the *Wait (ms)* and *Wait Until Next ms Multiple* functions. Time control functions are presented in figure 64.

Charts

Charts allow visualization of collected data and their preliminary assessment. In LabVIEW there are many different types of charts, which are available in fold: *Controls Palette* $\rightarrow$ *Graph*. Most popular charts are: *Waveform Chart*, *Waveform Graph* and *XY Graph*.

First of them displays real-time point by point, with points spaced the same distance on the X scale. All points are stored in memory inside Waveform Chart. In figure 65 three possible ways to visualize data in Waveform Chart are presented:

1. **Strip Chart** - points are displayed continuously and X axis is shifted from right to left, when chart is full. In this way new points appears on the right side, and old points disappear on the left side of chart.

2. **Scope Chart** - points are displayed continuously and when chart is full, new points start to appear on the left side, after cleaning whole chart.

3. **Sweep Chart** - points are displayed continuously and new point is separated from other points with red vertical line. When chart is full, new points will appear on the left side, but it will replace old data without cleaning whole chart.

To change the way of chart displaying one needs to click the right mouse button on chart and select: *Advanced*$\rightarrow$*Update Mode*.

As shown in figure 65, each point one by one are passed directly to the input of waveform chart. However waveform chart allows also adding more than one series of points. In such case one needs to connect points from different series into cluster using *Bundle* function, as shown in figure 66.

Next chart type is *Waveform Graph*, which displays whole series of points instead adding point after point. The easiest way to add points to waveform graph is to pass 1D array with points. They are added to the chart starting from x=0 and incrementing x by +1 after each point. Moreover waveform graph allows displaying points from any x value and increasing x axis for any const value. In such case, on input of waveform graph, one needs pass a cluster consisting of: x value, x

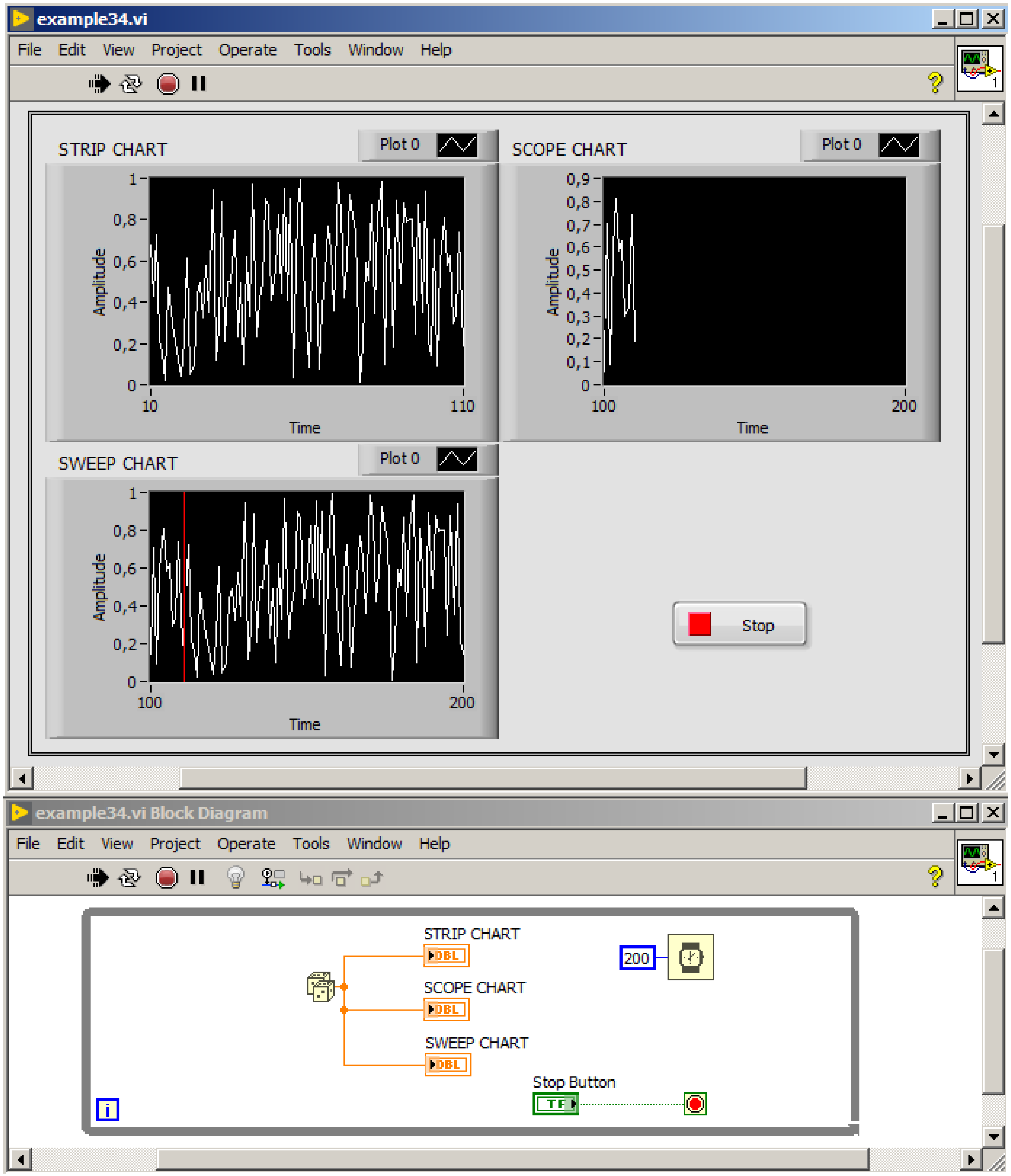


Figure 65: Program that demonstrates the different modes for displaying data on *Waveform Chart*.

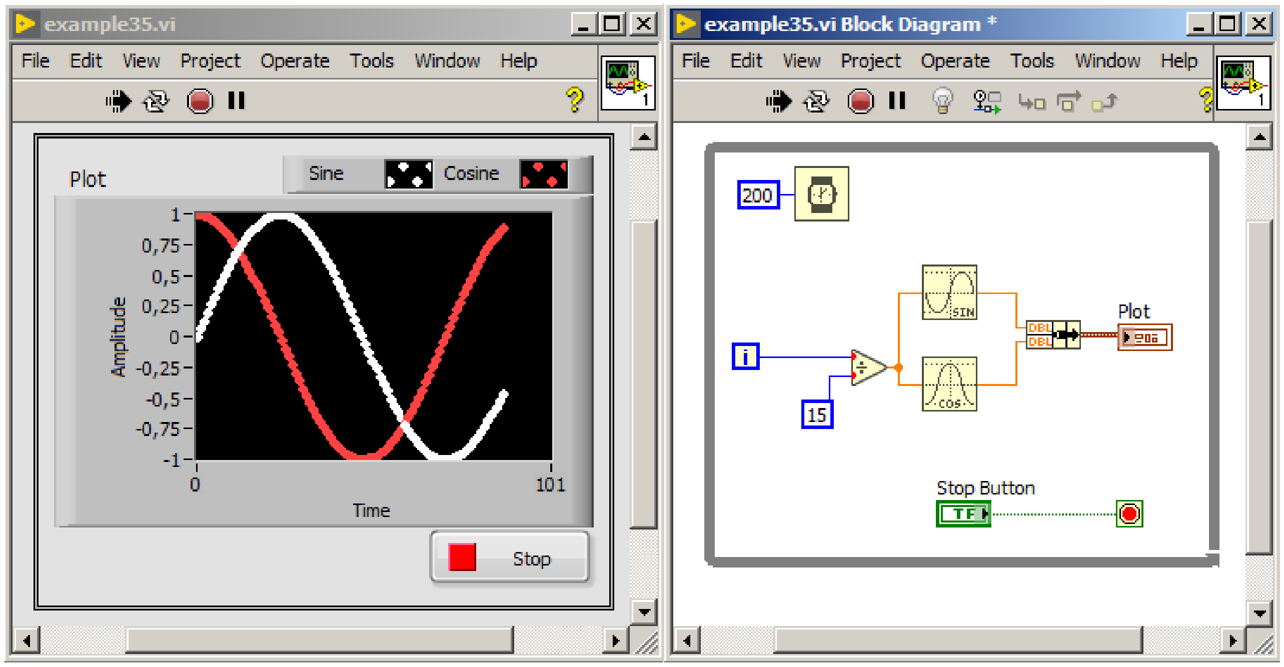


Figure 66: Program that demonstrates displaying two series of points on *Waveform Chart*.

increment, 1D array of y values. Both methods are presented in figure 67.

It is possible to pass multiple point series to charts of waveform graph type in different ways:

1. by passing 2D array, where in the first row there are points (y values) for the first series, in second row there are points (y values) for the second series, e.t.c.

2. by passing a cluster, which consists of (placed in this order): x value, x increment, 2D array of structure mentioned above.

3. by passing an array of clusters, which particular elements are: x value, x increment, 1D array of y values.

4. by passing 1D array of clusters where each cluster consists of y value array. This method is used mainly in cases where multiple measurement series differ in number of points.

Methods of waveform graph creation with two series of data are presented in figure 68.

The last from basic chart types is *XY Graph*, which allows to display unevenly distributed points about the x axis (scatter plot). In other words, both y and x values needs to be passed to the input of *XY Graph*. Charts with one measurement series can be created in two ways:

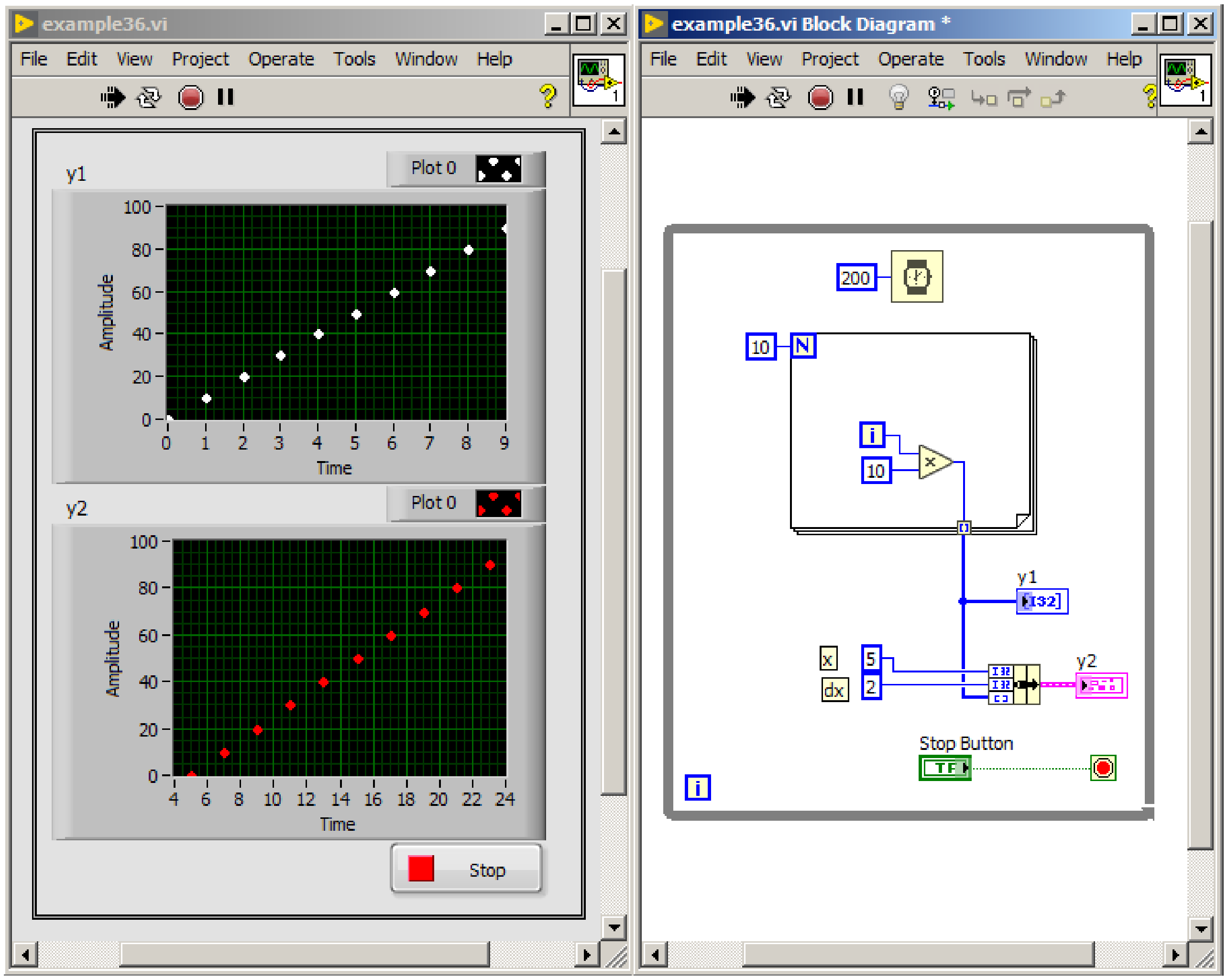

example36.vi
File Edit View Project Operate Tools Window Help
y1
Plot 0
Amplitude
Time
y2
Stop
example36.vi Block Diagram *
200
10
N
i
10
y1
I32
x
dx
5
2
y2
Stop Button
TF
i

Figure 67: Program that demonstrates ways of data passing to chart of *Waveform Graph* type.

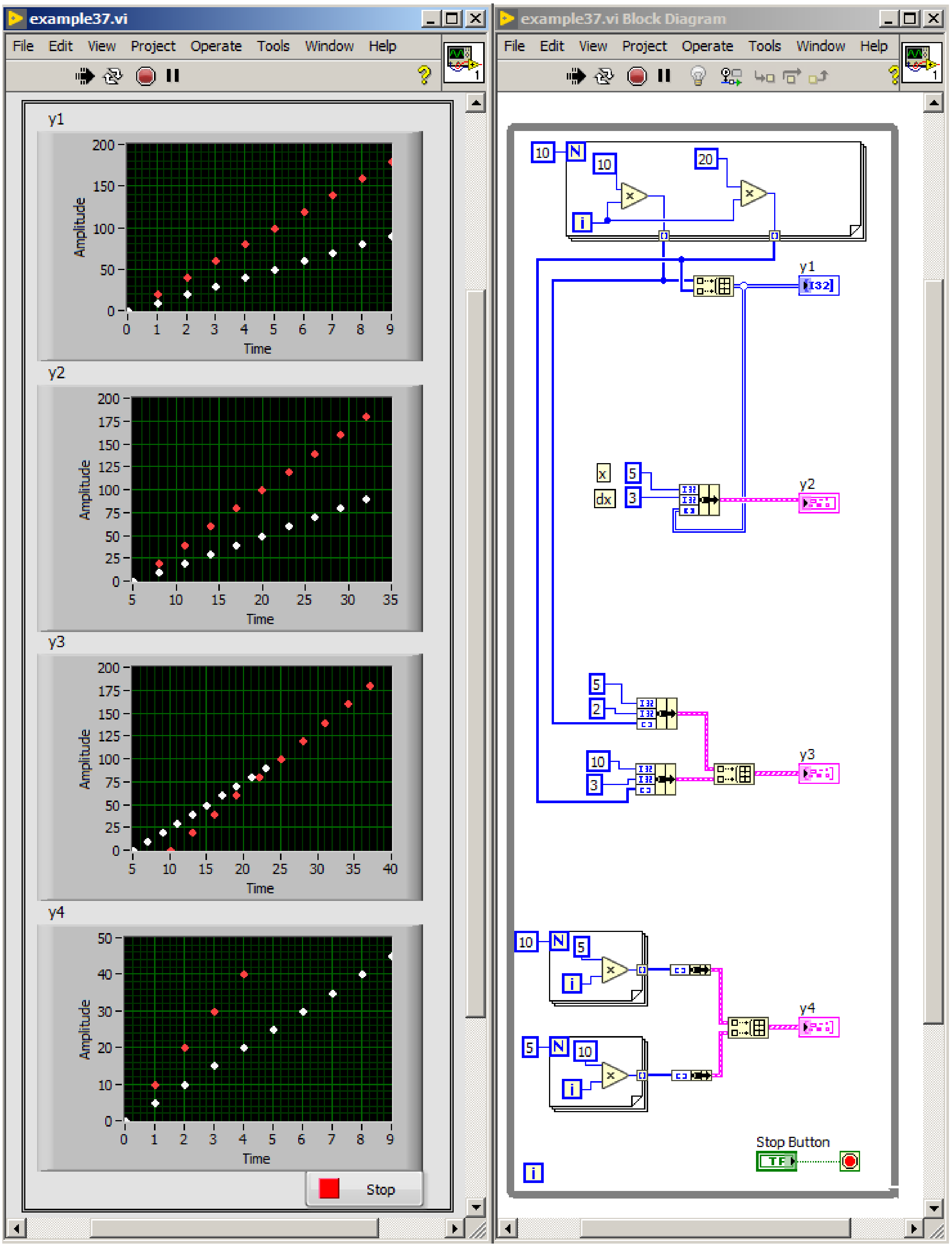

example37.vi
File Edit View Project Operate Tools Window Help
y1
Amplitude
Time
y2
y3
y4
Stop
example37.vi Block Diagram
10
N
i
20
x
dx
5
3
2
Stop Button
TF

Figure 68: Program that demonstrates different ways of passing data to charts of type *Waveform Graph* using points from two measurement series.

1. by passing cluster which consists of array of x values and array of y values,
2. by passing array of clusters, where each cluster consists of two values: x and y.

Described methods of XY graph filling with data are presented in figure 69.

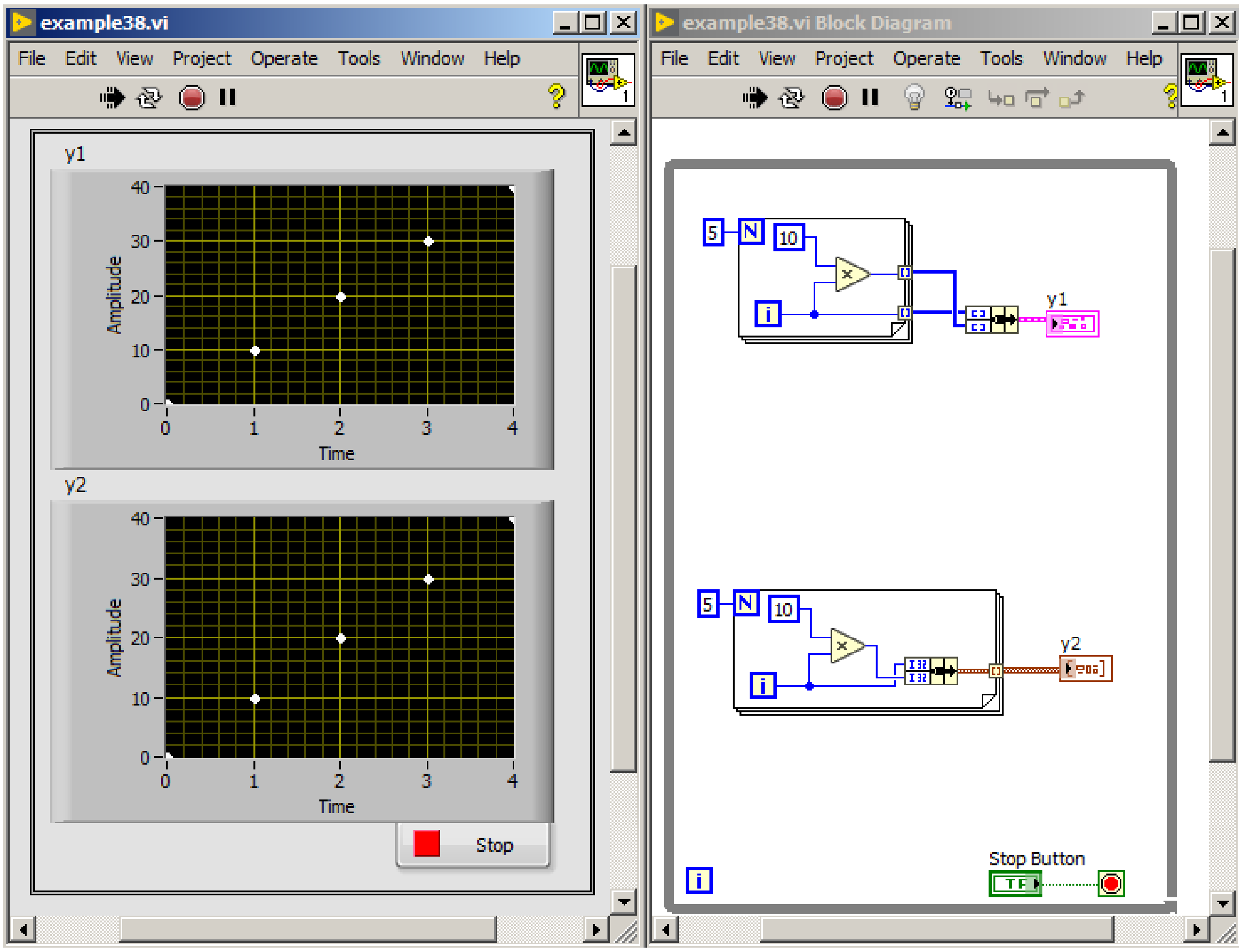
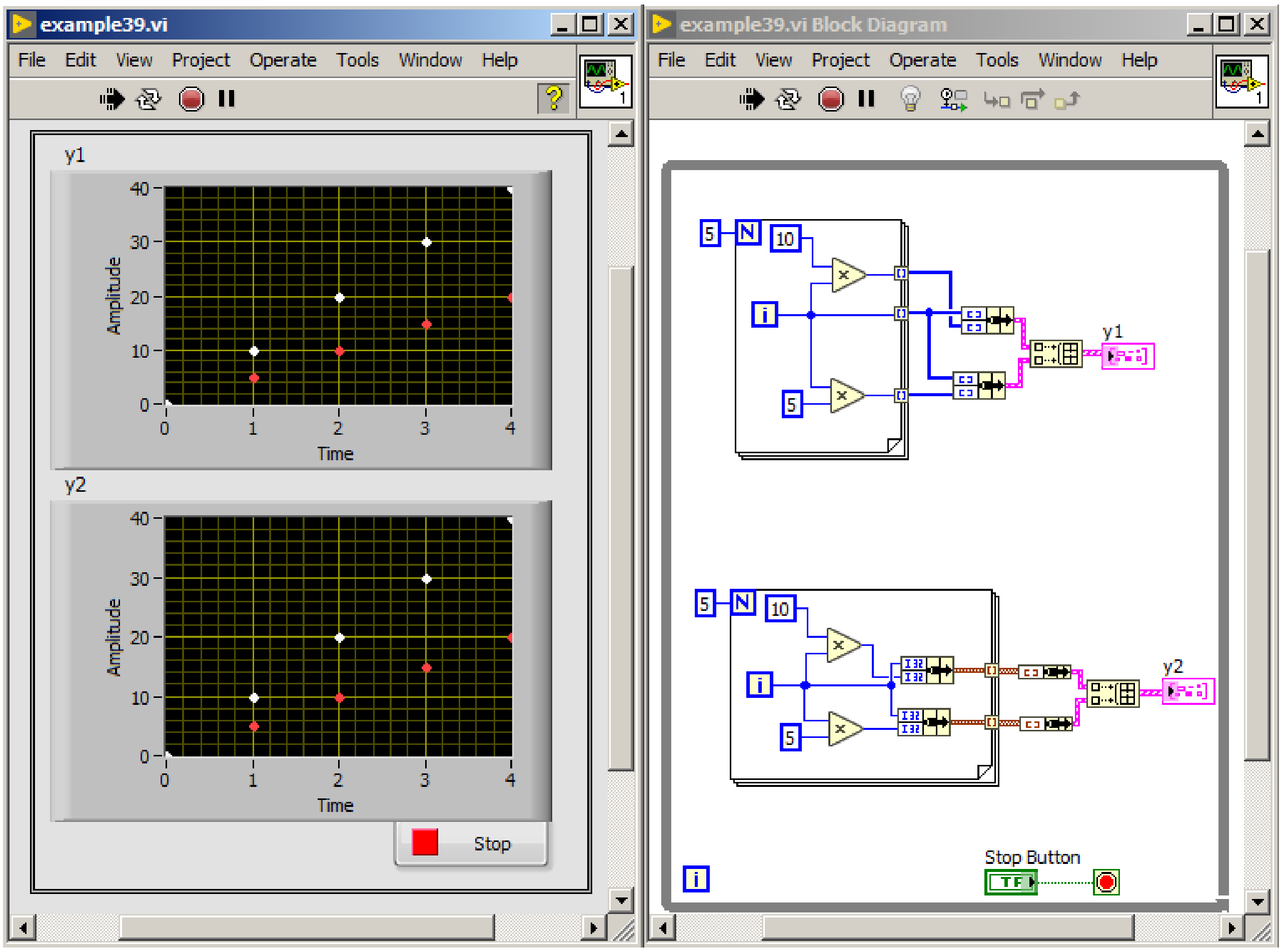

example38.vi
File Edit View Project Operate Tools Window Help
y1
Amplitude
Time
y2
Stop
example38.vi Block Diagram
5
N
10
i
y1
y2
Stop Button
TF

Figure 69: Program that demonstrates two ways of passing data to *XY Graph*.

To add more point series to the *XY Graph*, one needs to:

1. pass an array of clusters with x and y arrays inside.
2. pass an array of clusters, where each cluster consists of array of clusters with x and y values for one point.

Described methods of passing data to *XY Graph* with multiple point series are presented in figure 70.



example39.vi
File Edit View Project Operate Tools Window Help
y1
Amplitude
Time
y2
Stop
example39.vi Block Diagram
5
N
10
i
y1
y2
Stop Button

Figure 70: Program that demonstrates ways of data passing to *XY Graph* with two point series.

Shift Register

Shift register is used to pass values between one loop iteration and the next iteration. To add the shift register, one needs to click the right mouse button on the loop and select *Add Shift Register*. Shift register consists of two terminals: left (which returns value from previous loop iteration) and right (which receives value and passes it to the left terminal in next loop iteration).

In figure 71 a basic program using shift register is presented. To the right terminal of shift register a value from iterator is passed, where from the left terminal a value is read and passed to *result* indicator. To the left terminal an initial value equal to 0 is also connected. In table 4 results shown in *result* indicator and values stored in terminals of shift register are presented. Pay attention to the fact, that in "0" loop iteration, *result* indicator takes the 0 value, because this was the initial value connected to the left terminal. In "1" loop iteration, *result* indicator still takes the 0 value, because to the right terminal a 0 value was passed in "0" loop iteration. Running this program multiple times doesn't change the behaviour, because shift register is always initialized with 0 value.

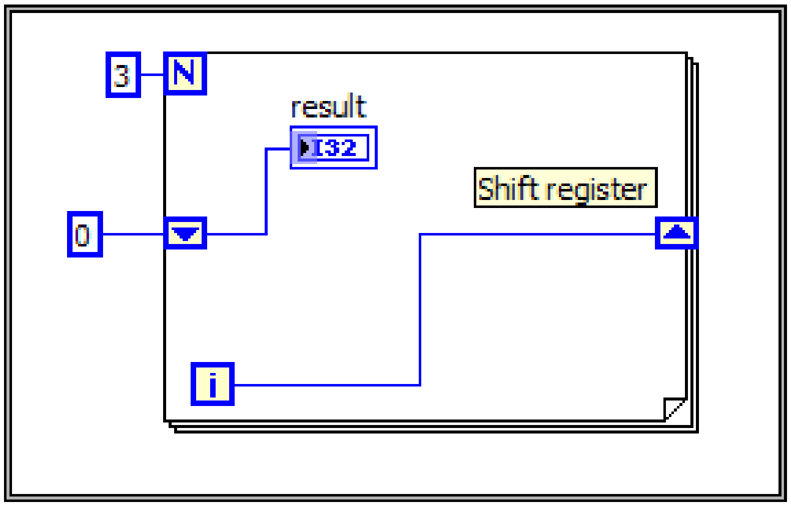


Figure 71: Program that demonstrates principles of shift register operation.

i	left terminal	result indicator	right terminal
0	0	0	0
1	0	0	1
2	1	1	2

Table 4: Values stored in shift register and in indicator after starting the program presented in figure 71.

In figure 72 a program using two shift register is presented. In both cases to the right terminal of shift register, a result from addition of two numbers: value from left terminal and for loop iterator is passed. Upper shift register is initialized with 0 value, which means, that always after starting the program, the left terminal will return 0 value. However in bottom shift register, the left terminal is not initialized. Used here is the fact that the shift register remembers the last value passed to the shift register from the previous program run and is is the initial value the next time you start the program. The situation described is best demonstrated by the results from table 5.

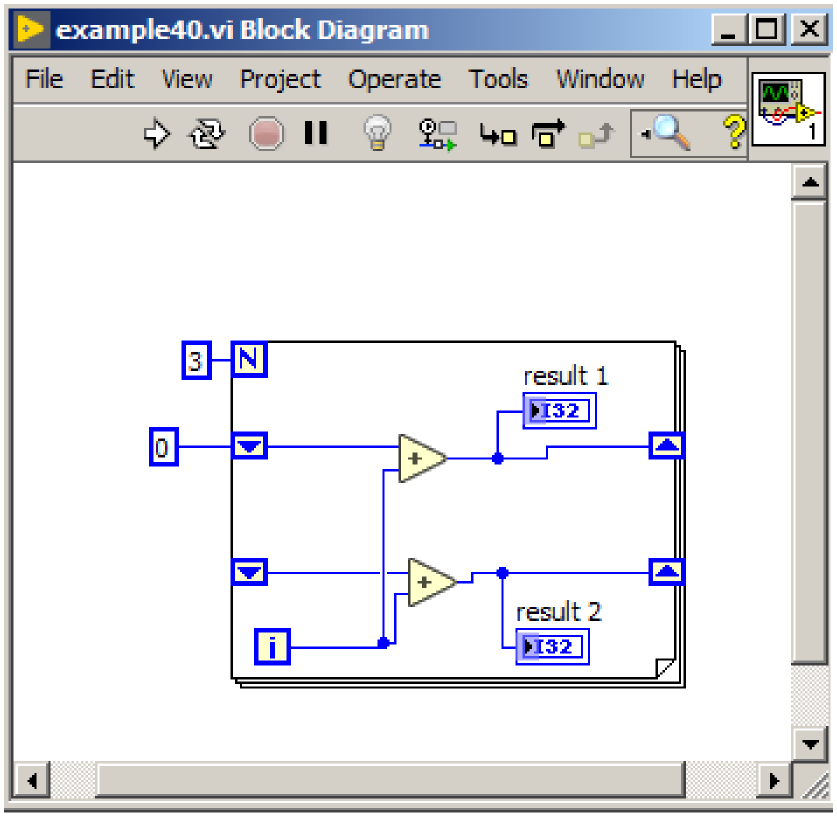


Figure 72: Program that demonstrates principles of shift register operation without initial value specified.

First program run						
i	left terminal	result 1 indicator	right terminal	left terminal	result 2 indicator	right terminal
0	0	0 (0+0)	0	0	0 (0+0)	0
1	0	1 (0+1)	1	0	1 (0+1)	1
2	1	3 (1+2)	2	1	3 (1+2)	3
Second program run						
0	0	0 (0+0)	0	3	3 (3+0)	3
1	0	1 (0+1)	1	3	4 (3+1)	4
2	1	3 (1+2)	3	4	6 (4+2)	6

Table 5: Values stored in shift registers and indicators after starting the program presented in figure 72.

Shift register can also pass values from several previous iterations.

In such case, one needs to grab the left terminal of shift register and stretch it to the desired number of terminals. In figure 73 a program with left terminal of shift register stretched to 3 terminals is presented. The top terminal will return value from *i-1* for loop iteration, middle terminal will return value from *i-2* for loop iteration and bottom terminal will return value from *i-3* for loop iteration. In table 6 results from running the program are presented.

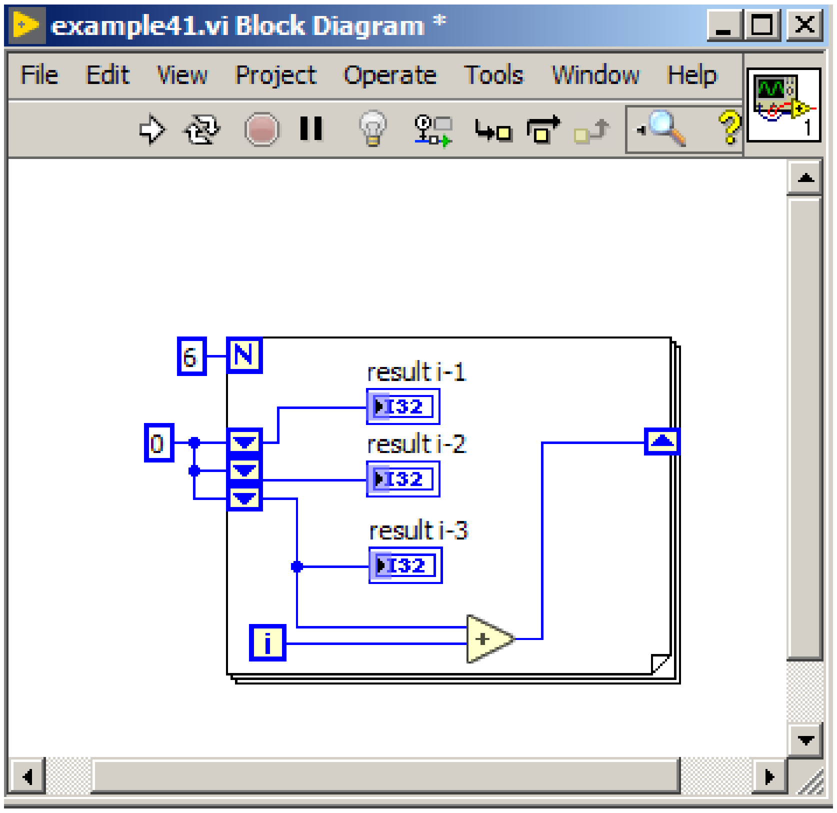


Figure 73: Program that demonstrates principle of shift register operation with several terminals.

i	result i-1	result i-2	result i-3	right terminal
0	0	0	0	0 (0+0)
1	0	0	0	1 (0+1)
2	1	0	0	2 (0+2)
3	2	1	0	3 (0+3)
4	3	2	1	5 (1+4)
5	5	3	2	7 (2+5)

Table 6: Values stored in shift register and in indicators after starting the program presented in figure 73.

Control of Graphical User Interface

Property Nodes

Property Nodes allow access to object properties from the level of block diagram. It is a tool particularly useful in situations, where there is a need to change object properties depending on the input data. For example if we measure ambient temperature with temperature sensor and we display the result on the chart and we would like to change color of the line from green to red when maximal allowed temperature is exceeded. In such situation, we can use *property nodes*. Another example is when we protect certain data with a password, if the user enters the correct password, the program reveals hidden data fields.

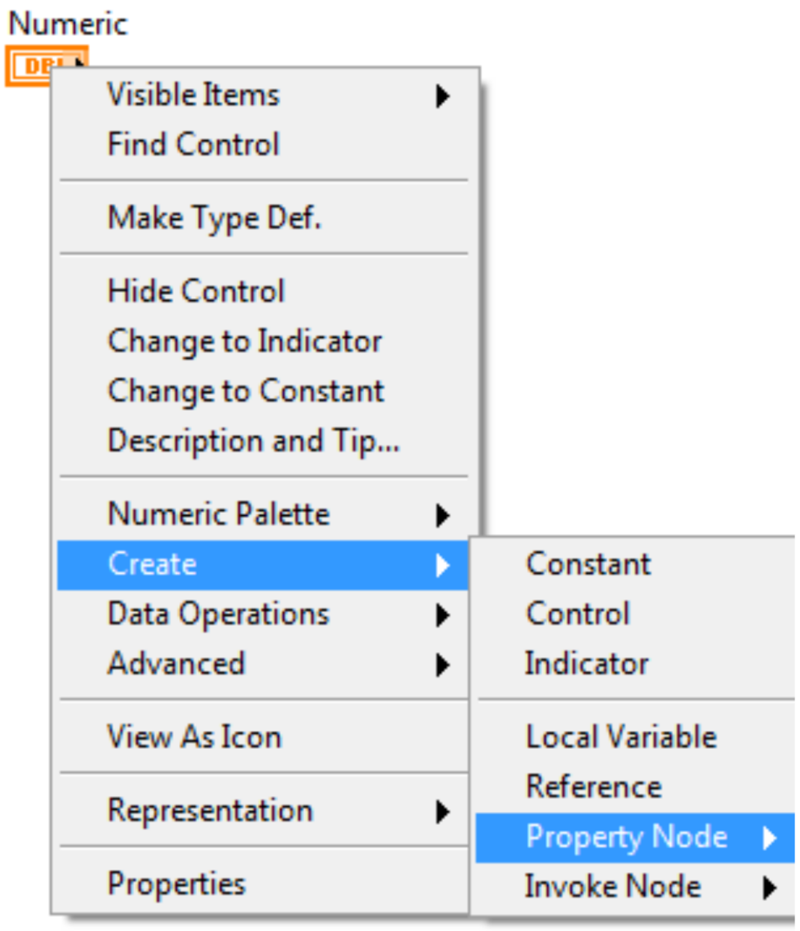


Figure 74: *Property Node* creation.

All control and indicator properties, which may be changed with use of property window or context menu may also be changed programmatically with use of *Property Nodes*. To create *Property Node*, one needs to click the right mouse button on control or indicator terminal, and choose from menu **Create** → **Property Node** option, which is presented in figure 74. Next from a list interesting property should be chosen. Each control has properties assigned hierarchically - starting from general properties, common to many different objects, and ending with specific properties.

After choosing particular property, specific property node terminal will be created, which is by default only for reading. To write a value to such property, one needs to click the right mouse button on property node terminal and next choose **Change to Write**.

An example program using property node is presented in figure 75. Depending on the value selected from the Slider, filling of the Slider will change its color to one of two possible: green or red. If value will be higher then 5, the red color will be passed to *Fill Color* property node and thus filling of the slider will change its color to red. Otherwise filling of the slider will be in green color.

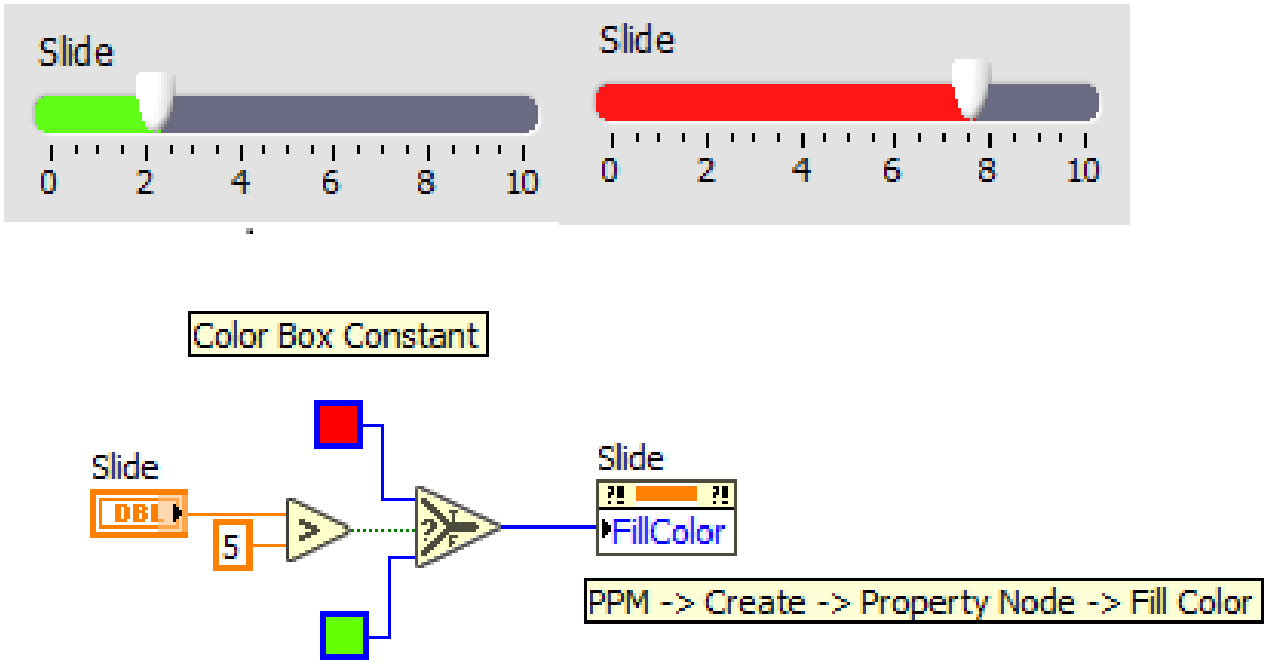


Figure 75: An example program with use of *Property Node*.

In many applications more than one property of particular object needs to be modified. In such case, there is no needed to create two separate terminals of property node, but existing terminal can be stretched to add more positions. By default, next visible terminals will appear as following from property node list. To select interesting property, one needs to hover mouse pointer (in the form of hand) over undesirable property terminal and after clicking the right mouse button, select the desirable property. An example is presented in figure 76.

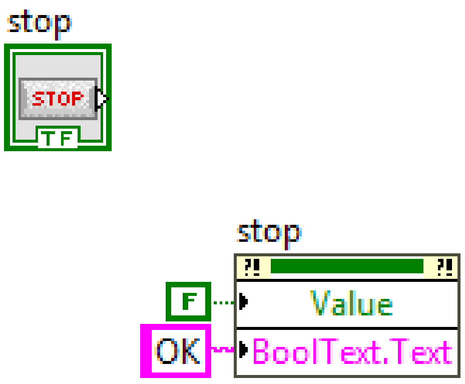


Figure 76: An example of property node with two property terminals usage.

Not all property nodes allow to write and read its values. Some properties are only for reading, e.g. *Label*, and some only for write, e.g. *value signaling property*. Some object properties are stored in clusters, e.g. control position in window, so to read the position of a control, a *unbundle* function is needed. However to write such property, first a cluster have to be created, and then should be passed to specific property node.

Invoke Nodes

Invoke nodes are used to call methods on objects. In contrast to property nodes, one invoke node can only implement one method. First element in invoke node terminal is a method name, then method parameters are displayed: required on white background and optional on gray background. If a method returns a value, then invoke node will have terminal which returns a value. An example invoke node is presented in figure 77.

To create invoke node, one needs to click the right mouse button on specific object and select **Create** → **Invoke Node**. From displayed list an interesting method should be selected. The Invoke Node creation method is shown in figure 78.

Control References

Methods of creation of Property Nodes or Invoke Nodes described above bind a given object positioned on the front panel with specific terminals of property nodes or invoke nodes on the block diagram. In this case we are talking about implicitly linked property nodes or

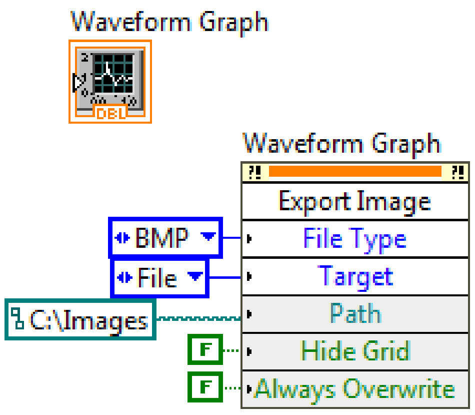


Figure 77: An example of *Invoke Node*.

invoke nodes. In case of subVI creation we will lose such link of property/invoke node with given object. We can solve this problem with use of a reference to object.

If subVI is created by selecting specific part of block diagram, and next selecting **Edit** → **Create SubVI**, then references to objects will be automatically created. But if subVI is created by saving a VI to a file, a property node from functions palette must be created. In such case select: **Functions palette** → **Programming** → **Application Control** → **Property Node**. Next click the right mouse button on object, which you want to bind with property node and select **Create** → **Reference**. The created reference connect to the upper input of property node terminal and next select interesting property for read or write. An example is presented in figure 79. Such created property node is called explicitly linked property node. Described method may also be used for invoke node creation.

Variables

Local Variables

Local variables are usually used to pass data between parallel loops. To better understand their operation, let's consider the following case. In figure 80 a program is presented, which generates two cosine waveforms with different frequencies. Both while loops should end their work after pressing a stop button. However, is this really happening?

The *stop* button is placed outside of while loop on block diagram. So its value is passed only once on the beginning of program. If user clicks the button in the next iterations of while loop, the changed value from *stop* button won't be passed inside the loop. In this way two loops working indefinitely were created. Therefore the question arises,

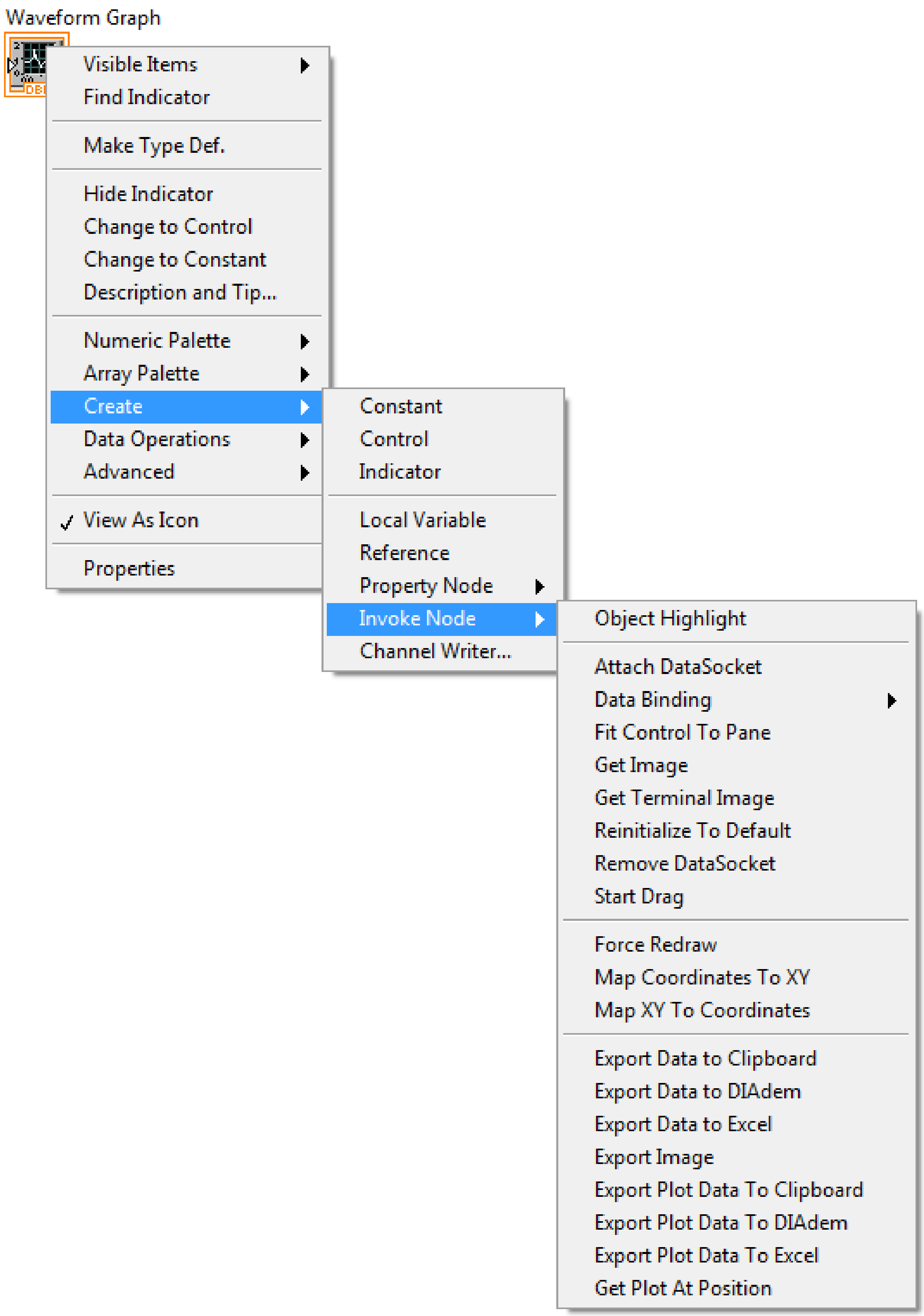


Figure 78: Invoke Node creation method.

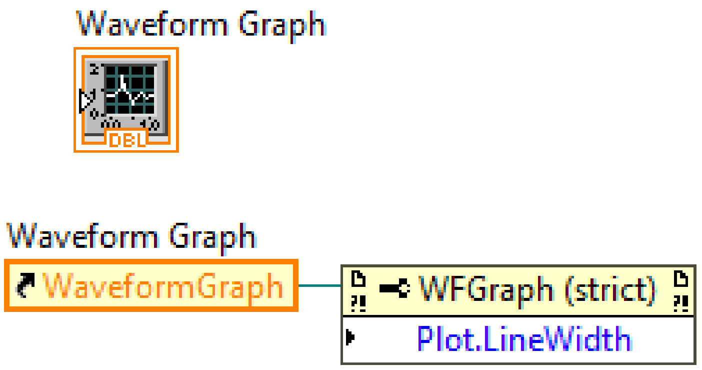


Figure 79: An example of creation of explicitly linked property node.

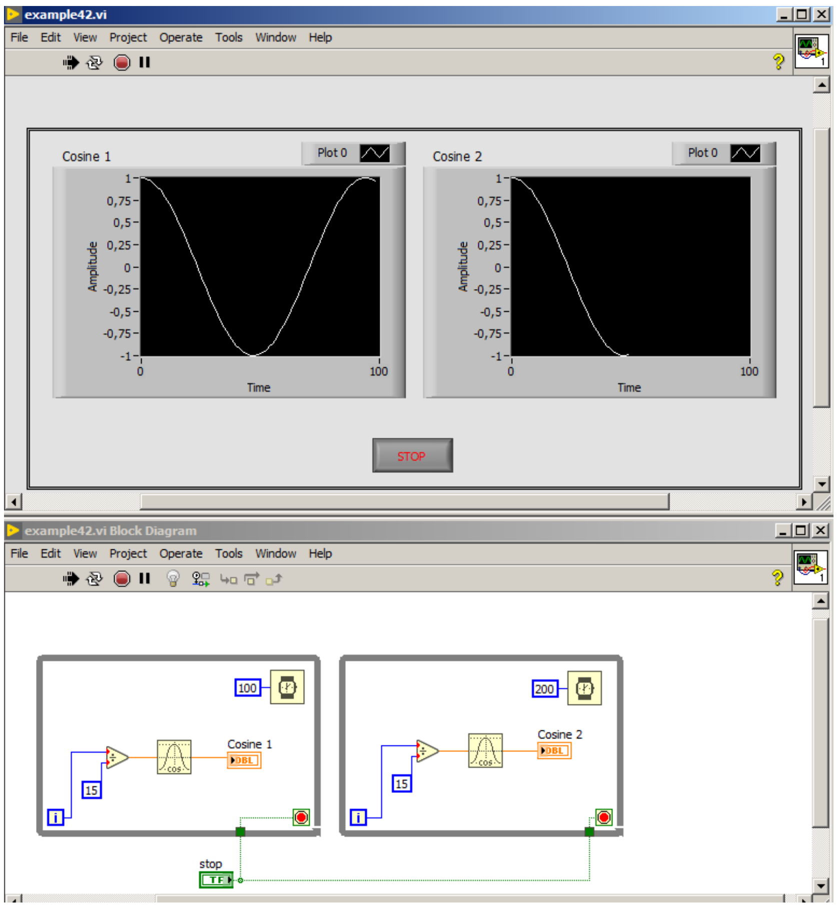


Figure 80: Program that demonstrates generation of two cosine waveforms with different frequencies. Both loops should end their work in the moment of stop button press. Attention: This solution is not correct! It is here only for didactic purpose.

whether it is enough to move the *stop* button inside one of while loops, which is presented in figure 81?

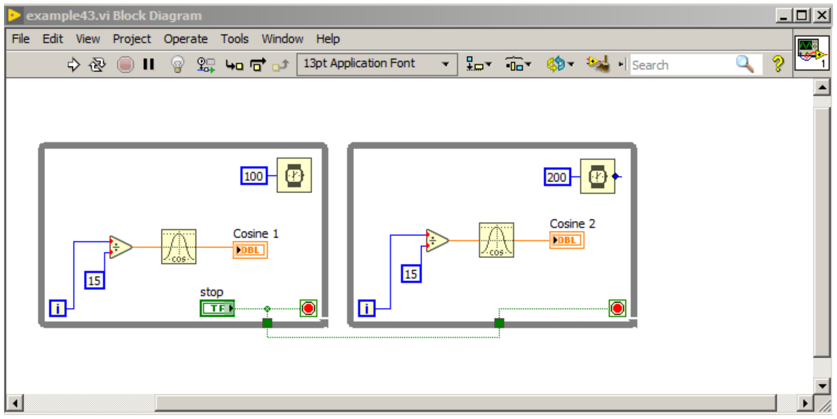

example43.vi Block Diagram
File Edit View Project Operate Tools Window Help
13pt Application Font
Search
100
200
Cosine 1
Cosine 2
15
stop
DBL
TF
i

Figure 81: Program, which demonstrates generation of two cosine waveforms with different frequencies. This is next potential solution. Attention: this solution is also not correct! This example is for didactic purpose only.

In this solution, while loop generates cosine waveform and fills the *Cosine 1* chart. The right loop doesn't start because it still waits for input logical value on terminal connected to *stop* button. After clicking the *stop* button, left while loop ends working and sends a *True* value to right while loop, which enables it to work. Right while loop will work only single iteration, because the condition to end the loop is met. Therefore how to pass the state of the *stop* button on an ongoing basis to the right while loop? That's what local variables are used for. They allow to read or write a value from a given control or indicator placed on front panel in many places on the block diagram.

One of the methods to create local variable is to click the right mouse button on control or indicator and select *Create* → *Local Variable*. The correct solution to the discussed case is shown in figure 82. Stop button control is connected with stop condition of the left while loop, and created local variable to stop button control is connected with stop condition of the right while loop. By default, created local variable is in write only mode. To read from local variable, one needs to click the right mouse button on local variable and choose *Change to Read*. It is important, that if we use local variable with a button, then mechanical action of the button needs to be *switch*.

Local variables are useful in communication between parallel loops, however they have a major disadvantage. Let's assume, that in one loop we would like to increase the value from *Numeric* control by 1, and in other loop to decrease the value by 1. For ease of reference the result is also placed on the chart: *Value*. In each loop a delay was added. If delays are the same, the *Numeric* control should have value equal to 0. *Numeric* control value is shown in figure 83.

Not always *Numeric* value is first incremented by 1 and later de-

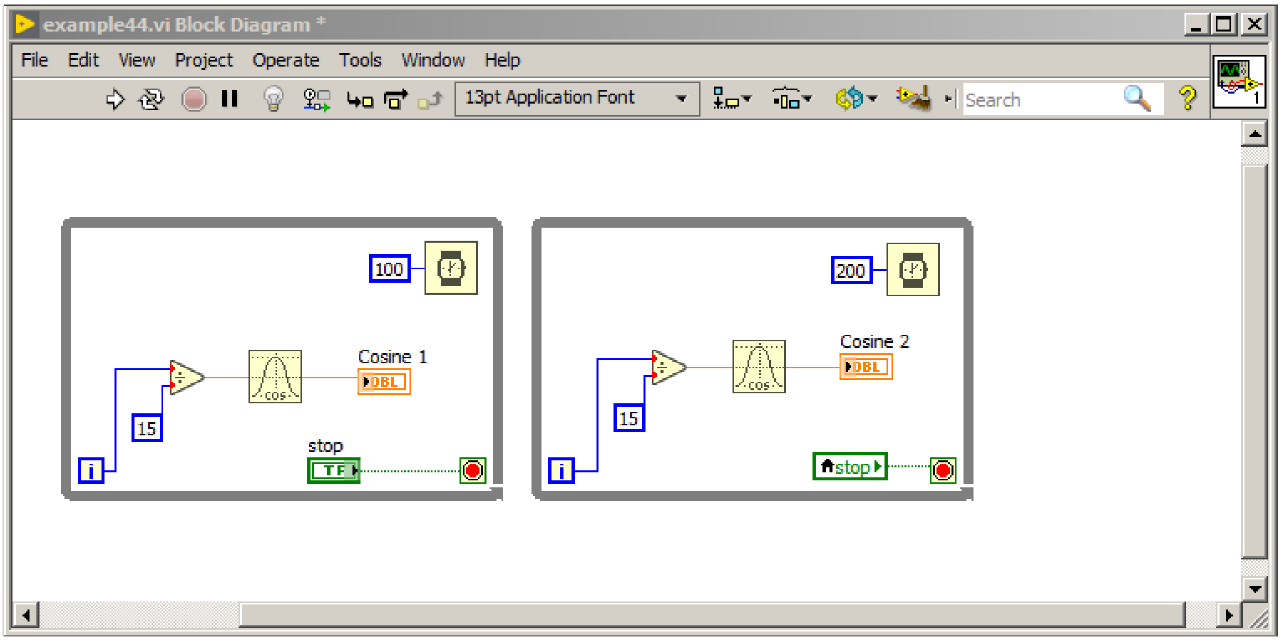

example44.vi Block Diagram *
File Edit View Project Operate Tools Window Help
13pt Application Font
Search
100
Cosine 1
DBL
15
stop
TF
i
200
Cosine 2
DBL
15
stop
i

Figure 82: Program which generates two cosine waveforms with different frequencies. Correct solution of stop button.

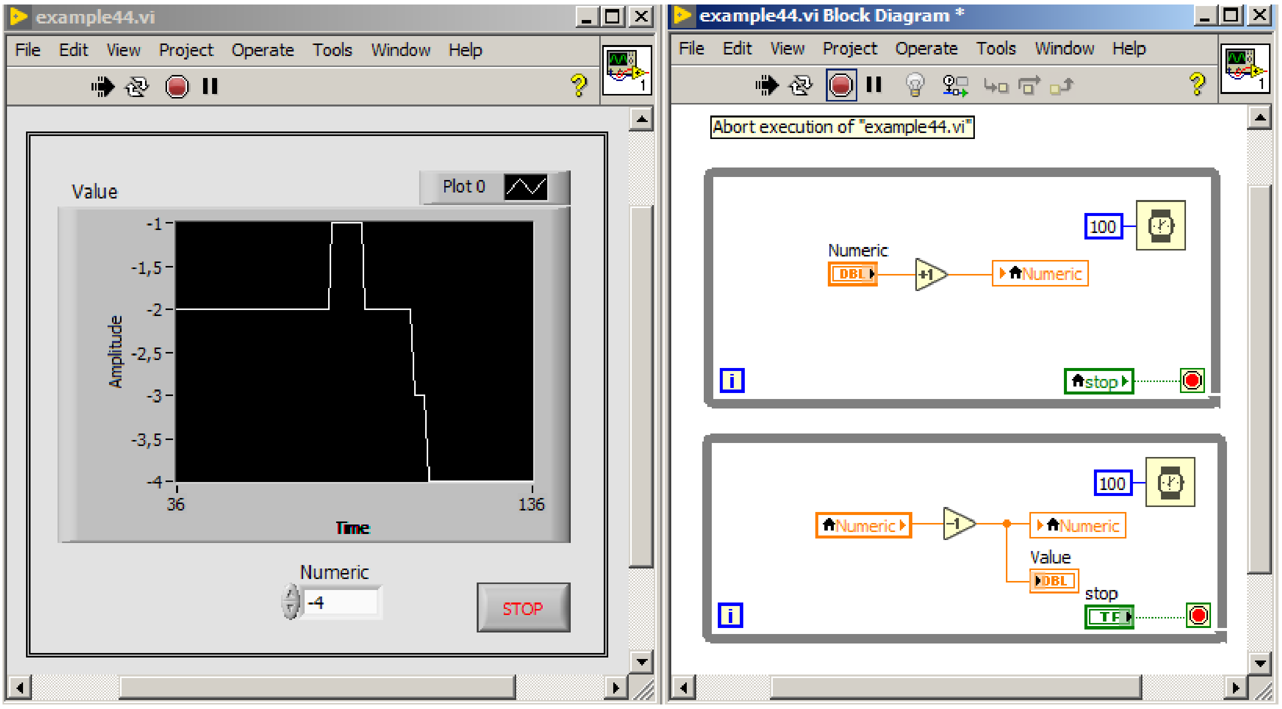


Figure 83: Program, which demonstrates race condition phenomenon.

creased by 1. If delay in one loop will change, the more difficult will be to predict, which loop will finish first. Thus, we see, that we are not able to predict what value will be on *Numeric* control. Such phenomenon is called "Race Conditions" and it is the result of lack of control on data flow. To prevent such situation, avoid writing value to control or indicator in many places of the program.

Global Variables

Local variables may be used in single VI only. To pass data between different VIs, a global variable is used. To create global variable, one needs to select *Functions palette* → *Structures* → *Global Variable*. Next click the left mouse button on created terminal. A global variable front panel will open. Given global variable may consists of single or multiple objects. Objects are added to global variable from *controls palette* just as on any VI front panel. After saving global variable and clicking again on its terminal, a list with available objects will be visible as shown in figure 84.

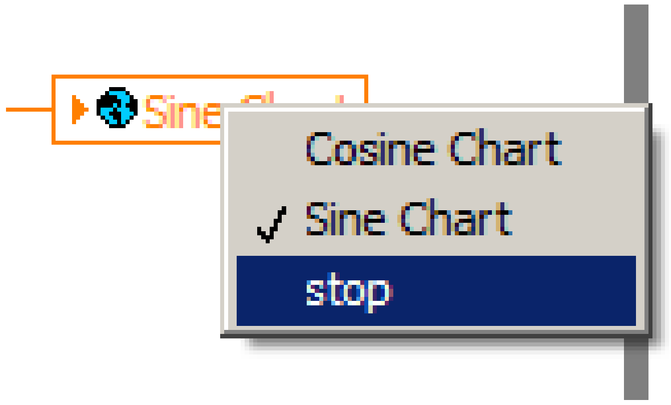


Figure 84: An example list of objects put in single global variable.

In figure 85 a program using global variable is presented. In first VI, sinus waveform is generated and passed to global variable named *Sine Chart*. Moreover from *Stop* global variable, a logic value is read and passed to stop condition of while loop. Second VI is similar, but generates cosine's waveform. Third VI consists of enum, in which user selects, which waveform is displayed on Waveform Chart named *Plot*. On block diagram, an enum value is read. In case of selected waveform, different case structure frame will be executed. For each waveform in case structure frame, a chart is read from global variable. Moreover *stop* button value is passed to global variable to stop all VIs in the same time. Outside while loop there are icons of two other VIs, just to run them in the same time as main VI.

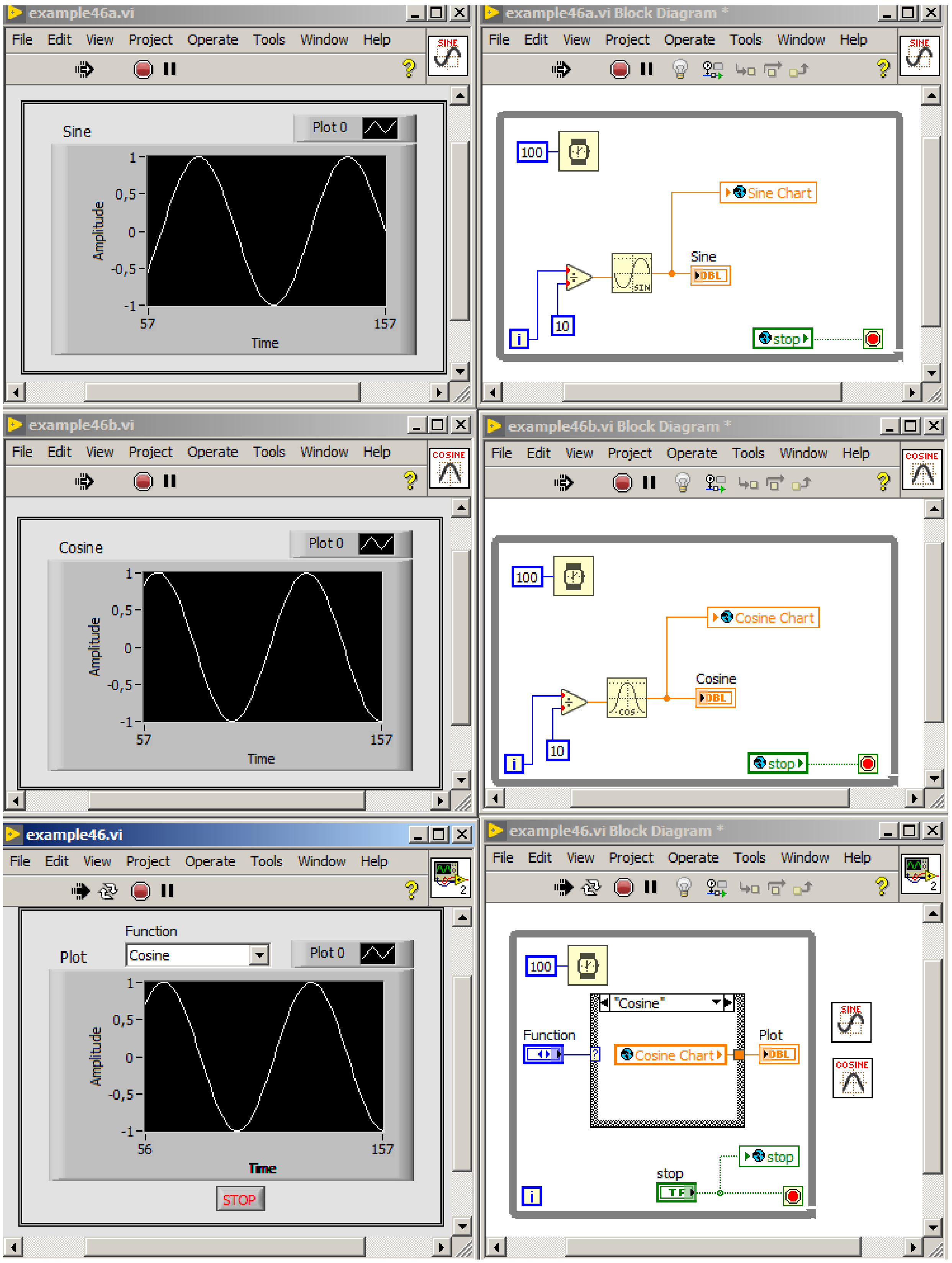

example46a.vi
example46a.vi Block Diagram *
File Edit View Project Operate Tools Window Help
Sine
Plot 0
Amplitude
Time
Sine Chart
stop
example46b.vi
example46b.vi Block Diagram *
Cosine
Cosine Chart
example46.vi
example46.vi Block Diagram *
Function
Plot
STOP
"Cosine"

Figure 85: Program, that demonstrates passing different waveforms from VIs to main VI with use of global variables.

Shared Variables

Shared variables are used to pass data between different VIs in the same project and also to pass data between different devices connected through network.

When such variable is being created, a name and a type (presented in figure 86 and 87) must be defined. Write or read from shared variable is possible by dragging icon of global variable from project window to block diagram of given VI. By default shared variable is for read only. To make it writable, one needs to click the right mouse button on shared variable, and then select **Access Mode** → **Write**.

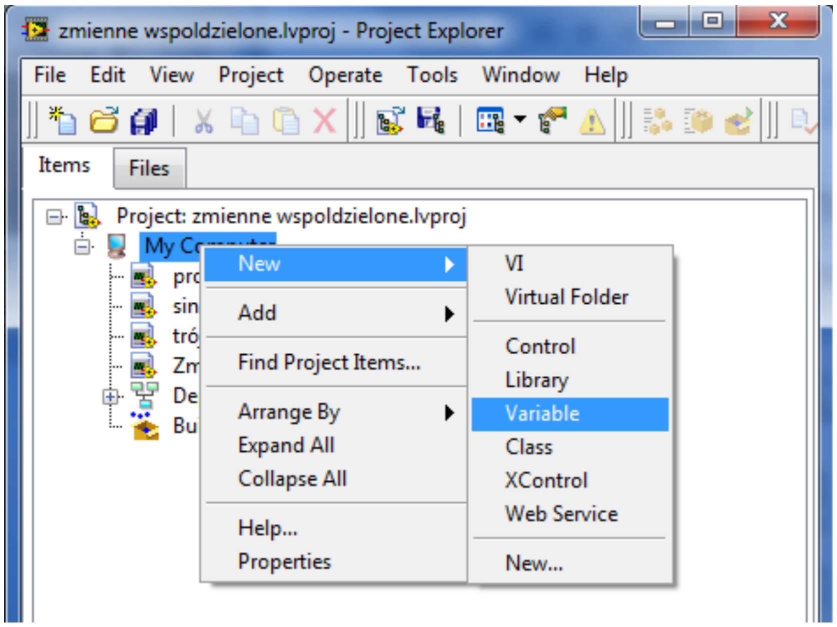


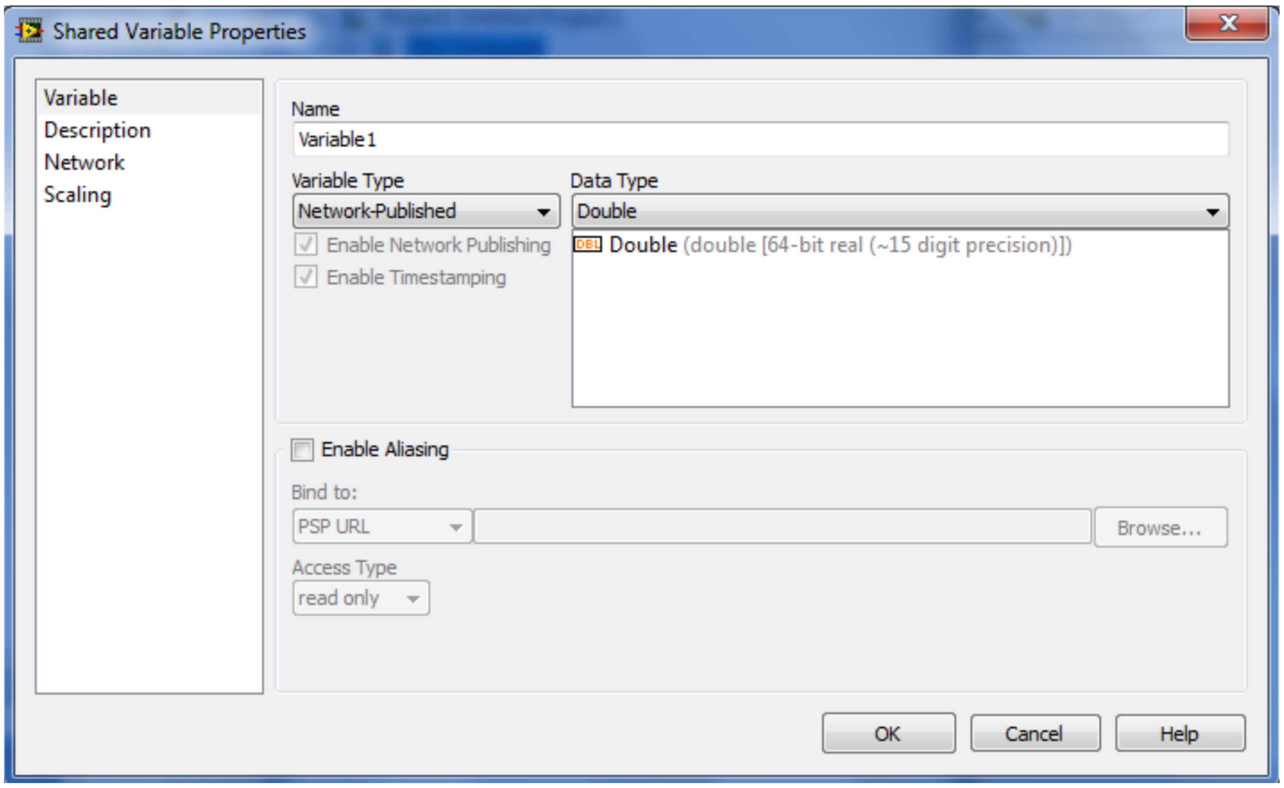
Figure 86: Shared Variable creation.

Shared variables should be used with care, because their usage don't imply the order of program execution. Because of this, when shared variables are used some not intended behaviour may happen like, e.g.:

- reading twice the same value from shared variable
- miss of shared variable value change

Functional Global Variables

Functional Global Variable (FGV) is a variable built on subVI with shift register basis. It uses the fact, that shift register stores value from last time running. Functional Global Variable besides value storing may



Name: Variable 1
Variable Type: Network-Published
Data Type: Double
☑ Enable Network Publishing
☑ Enable Timestamping
Double (double [64-bit real (~15 digit precision)])
☐ Enable Aliasing
Bind to: PSP URL
Access Type: read only

Figure 87: Shared Variable property window.

also perform operations on data, like changing its format (e.g. we can pass a scalar value to FGV, and read an array of data from FGV).

Functional Global Variable consists of while loop with shift register and case structure, which is shown in figure 88. A while loop has stop condition connected to constant true value - so, the code inside while loop will run only once when such subVI will be executed with variable. While loop is used to create shift register, which stores the data. FGV variable besides data values also may accept commands, which are realized as type definition (strict type definition) for enum type. Basic three commands realized by Functional Global Variable are: initialization, write, read. In practice Functional Global Variables frequently have other commands defined, as data analysis, write to file, change data format, e.t.c.

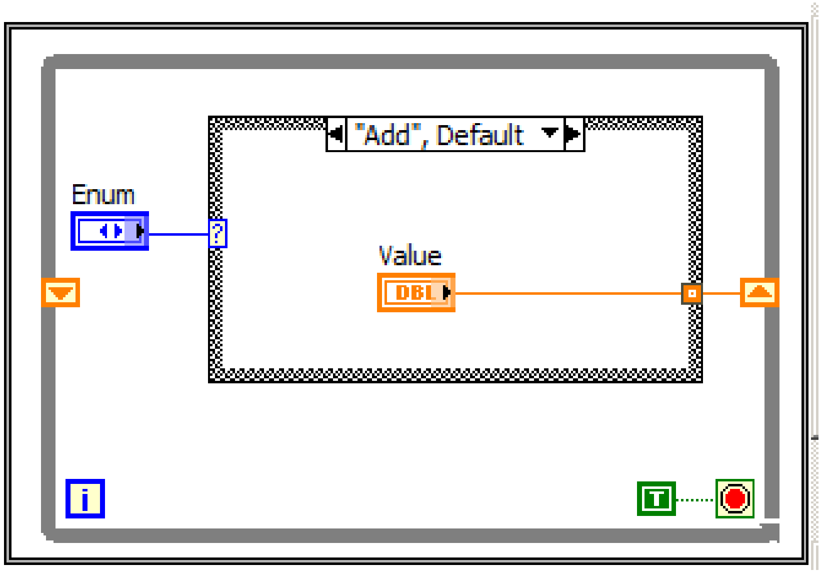


Figure 88: Functional Global Variable.

FGV variable is one of two most popular ways of protection of critical diagram parts from *race conditions* phenomenon, because FGV variable is *non-reentrant VI*. In practise, this means, that there is single place in memory, where data from subVI are stored. Such subVI cannot be run parallel. Other threads must always wait when such subVI will finish then they can call it.

When Functional Global Variable is created, it is worth to remember about some shortcuts in LabVIEW. After clicking the Case structure, there is an option “Add case for every value”, which allows for automatic creation of all cases in structure.

Write and read from files

In many situations it is necessary to read or write data to files. Every operation on file consists of three steps: open/create file, read/write data, close reference to file. Operation schema is presented in figure 89.

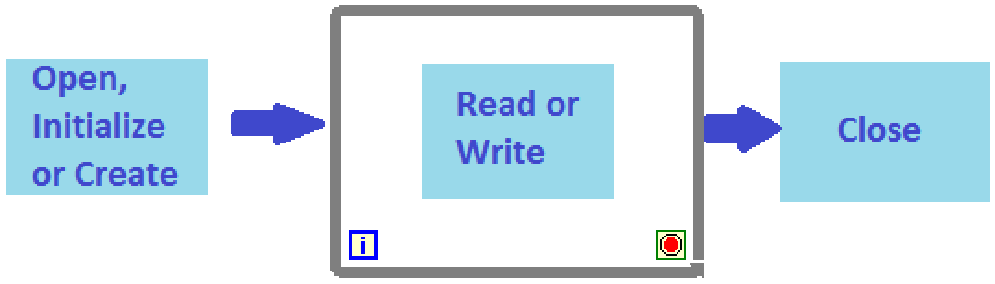


Figure 89: Schema of operation on file.

In LabVIEW it is possible to read and write data to following file formats: Binary, ASCII, LVM and TDMS. Two latest file formats are characteristic for LabVIEW. First of them (LVM) is based on ASCII file format, in which elements are separated by tabs. Such design makes loading the file to the Spreadsheet easier. TDMS file is designed by National Instruments and is based on binary file format. Besides values (data) it also stores object properties.

Available functions to work with files in LabVIEW are divided in two categories:

1. High-Level File I/O - consists of functions, which take three steps of file operations in one block. There is no need to open file separately, read/write and close reference to file. High-Level functions are convenient to use, however they are less effective.

2. Low-Level File I/O - consists of functions, to realize particular steps of file operations. These functions are much more effective and they are recommended in situations, where speed of data read/write has high importance.

The High-Level File I/O includes: Write Delimited Spreadsheet, Read Delimited Spreadsheet, Write To Measurement File and Read From Measurement File. In figure 90 a program, which toss 5 random values and write them to text file with use of *Write Delimited Spreadsheet* is presented. Results from text file are read then with *Read Delimited Spreadsheet* function and passed to array. Notice that all three steps: file creation, write/read and file reference close are realized by single function.

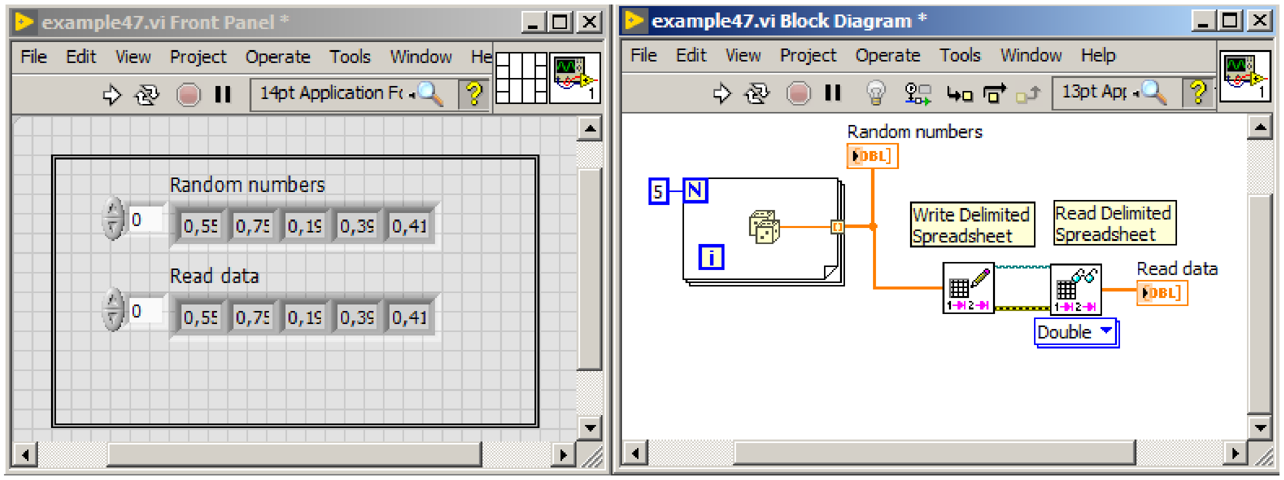

example47.vi Front Panel *
File Edit View Project Operate Tools Window He
14pt Application Fo
Random numbers
0
0,55 0,75 0,19 0,39 0,41
Read data
0
0,55 0,75 0,19 0,39 0,41
example47.vi Block Diagram *
File Edit View Project Operate Tools Window Help
13pt App
Random numbers
DBL
5
N
i
Write Delimited Spreadsheet
Read Delimited Spreadsheet
Read data
DBL
Double

Figure 90: Program, that writes random numbers to text file, then read them from file with use of High-Level File I/O function.

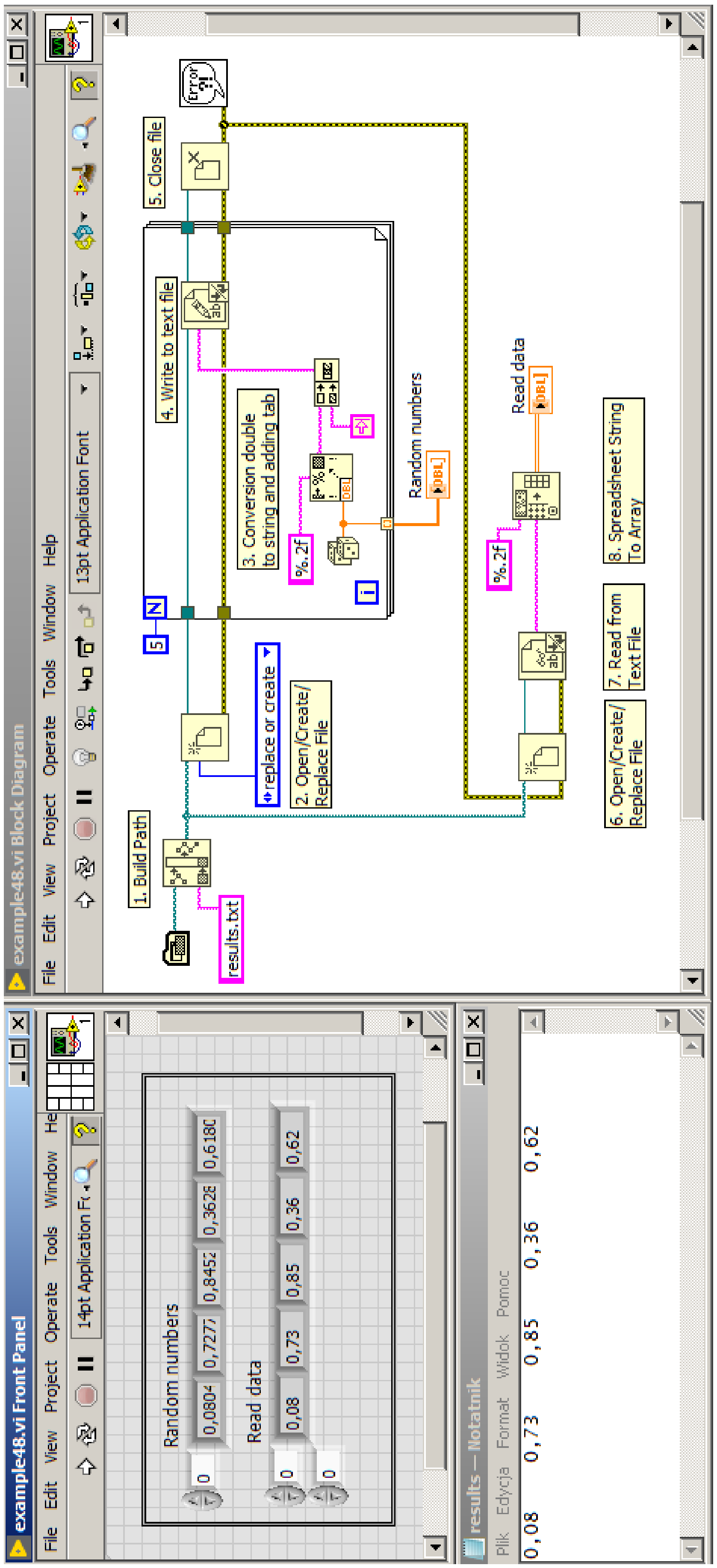
Functions: Write to Measurement File and Read From Measurement File works in similar way to Write/Read to Delimited Spreadsheet. The difference is that, we use them to write data in LVM or TDMS in contrast to ASCII file.

In figure 91 analogous program to presented in figure 90 is presented, but with use of Low-Level File I/O functions. As it is seen, code size is bigger, however speed of write and read operations is also higher.

Error handling

Important part of every professional program is correct error handling. User can generate many errors, e.g. give value outside allowed range, give path to non-existing file or divide numerical value by 0. In LabVIEW there are two types of error handling: automatic and manual. Automatic error handling is used by default and as it is triggered, it stops program run, highlights function/subVI, which caused an error and displays error list. Second possibility is to use manual error handling, which is used in example presented in figure 91. It uses error cluster, which is propagated through all functions having such possibility, e.g. Open/Create/Replace File, Write to Text File, e.t.c. When error occurs, program will not stop immediately, but error code and description of error will be passed in error cluster to the final error handling function e.g. *Simple Error Handler*, which will display information about error in the form of a dialog window.

Error cluster consists of: status (logic value), code (32-bit integer) and source (string). Status become *True* if error occurred or *False* if warning occurred. Every type of error has its individual error code.



```
0,08	0,73	0,85	0,36	0,62
```

Figure 91: Program, that writes random numbers to text file, and then it reads them from file with help of Low-Level File I/O functions.

Error clusters can be joined, by *Merge Error* function, which returns only first error or warning. It doesn't return full information about several errors. More examples about manual error handling are presented in section *State Machine* ().

Design patterns

Every professionally written program uses design patterns. In LabVIEW there are some important design patterns, which should be known by programmers. Most popular are: State Machine Design Pattern, Master/Slave Design Pattern and Producer/Consumer Design Pattern.

State machine

State machine consists of while loop, case structure with enum type connected to case selector and shift register. As an example of state machine usage, we will use washing machine simulator, which block diagram is presented in figure 92.

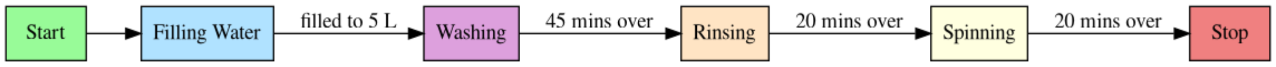


Figure 92: Block diagram of washing machine simulator.

Following states of washing machine are put in separate frames of case structure. An enum has a list of all states, and shift register is used to pass next value from enum to the input of case selector. Using a state machine allows to control execution of program step by step. Moreover each step is divided, so it reduces the space taken by code on block diagram and makes code better readable.

In figures 93, 94, 95, 96, 97 and 98 following states of washing machine simulator are presented. To display, which state is actually executed, a cluster with LED diodes (logic values) is used. Moreover *Water Level* numerical indicator is used, which simulates filling washing machine with water. Then there is *Time* indicator, which displays elapsed time in each state. Presented example is only simulation, however it demonstrates very well the idea of state machine design pattern. In practice, state machine design pattern is used very often. State machine may be use e.g. for reading data from some measurement device in one state, than in another state it may analyze and visualize data. On next state the data might be saved to file.

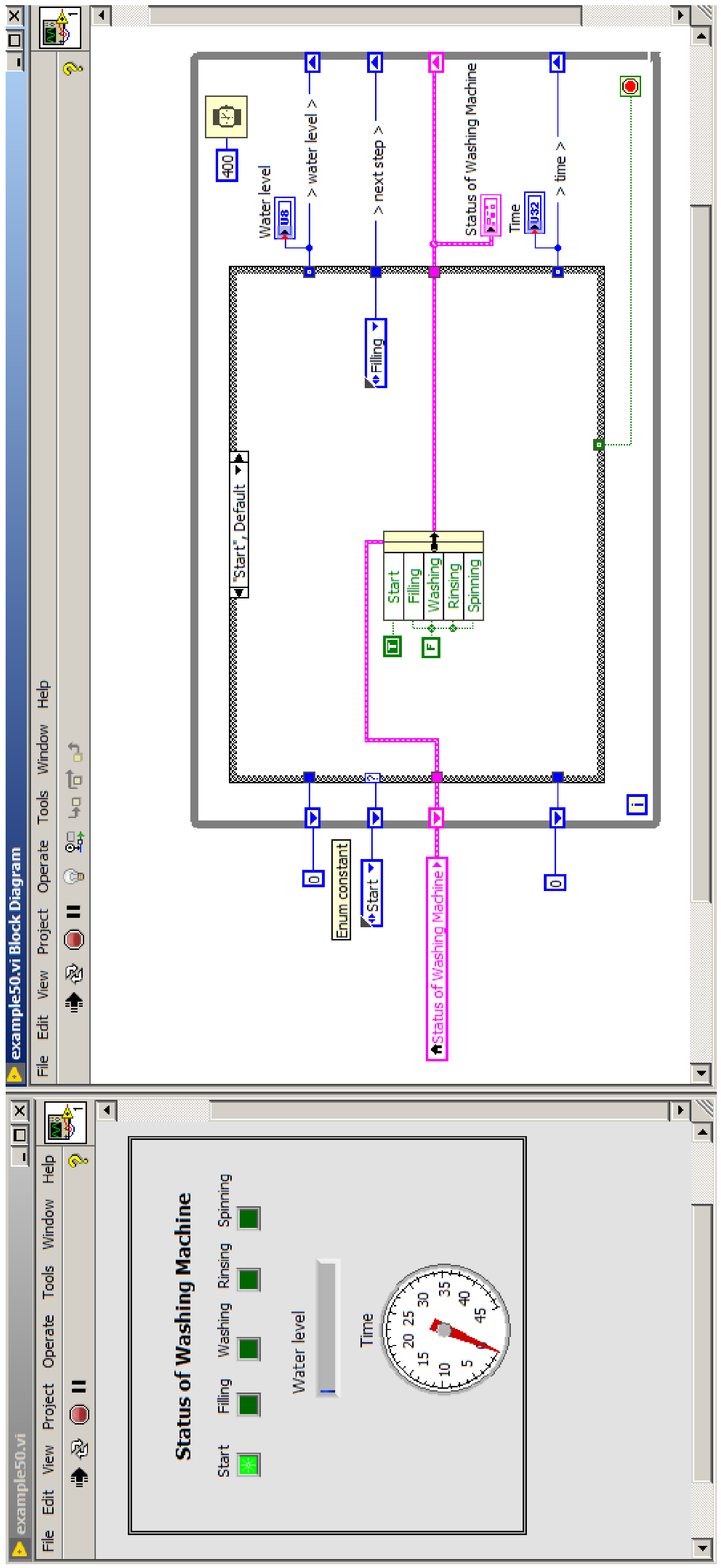


Figure 93: Implementation of *Start* state of washing machine simulator.

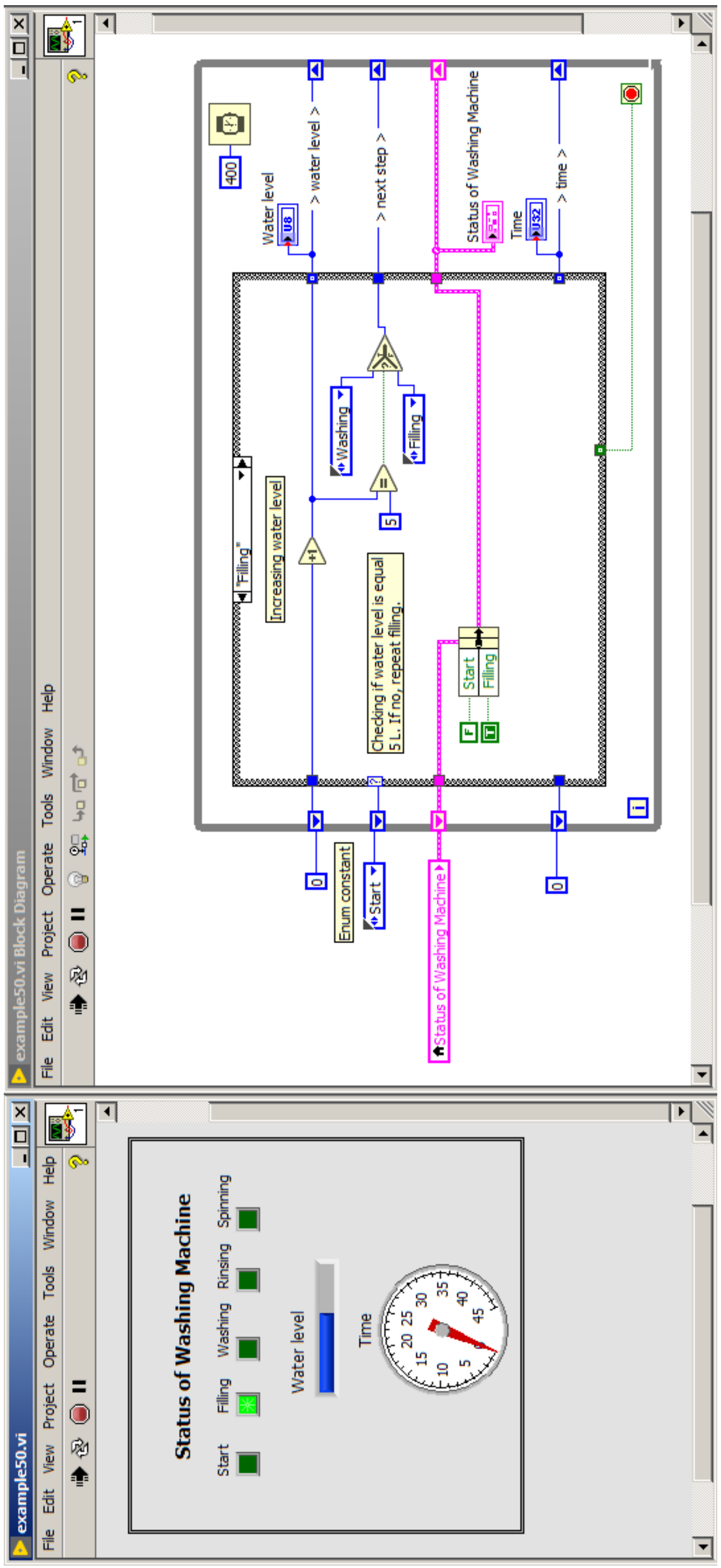

example50.vi Block Diagram
File Edit View Project Operate Tools Window Help
Water level
U8
> water level >
> next step >
Status of Washing Machine
Time
U32
> time >
400
"Filling"
Increasing water level
+1
Washing
Filling
5
Checking if water level is equal 5 L. If no, repeat filling.
Start
Filling
F
T
0
Enum constant
Start
Status of Washing Machine
0
i
example50.vi
File Edit View Project Operate Tools Window Help
Status of Washing Machine
Start
Filling
Washing
Rinsing
Spinning
Water level
Time
5
10
15
20
25
30
35
40
45

Figure 94: Implementation of *Filling Water* state of washing machine simulator.

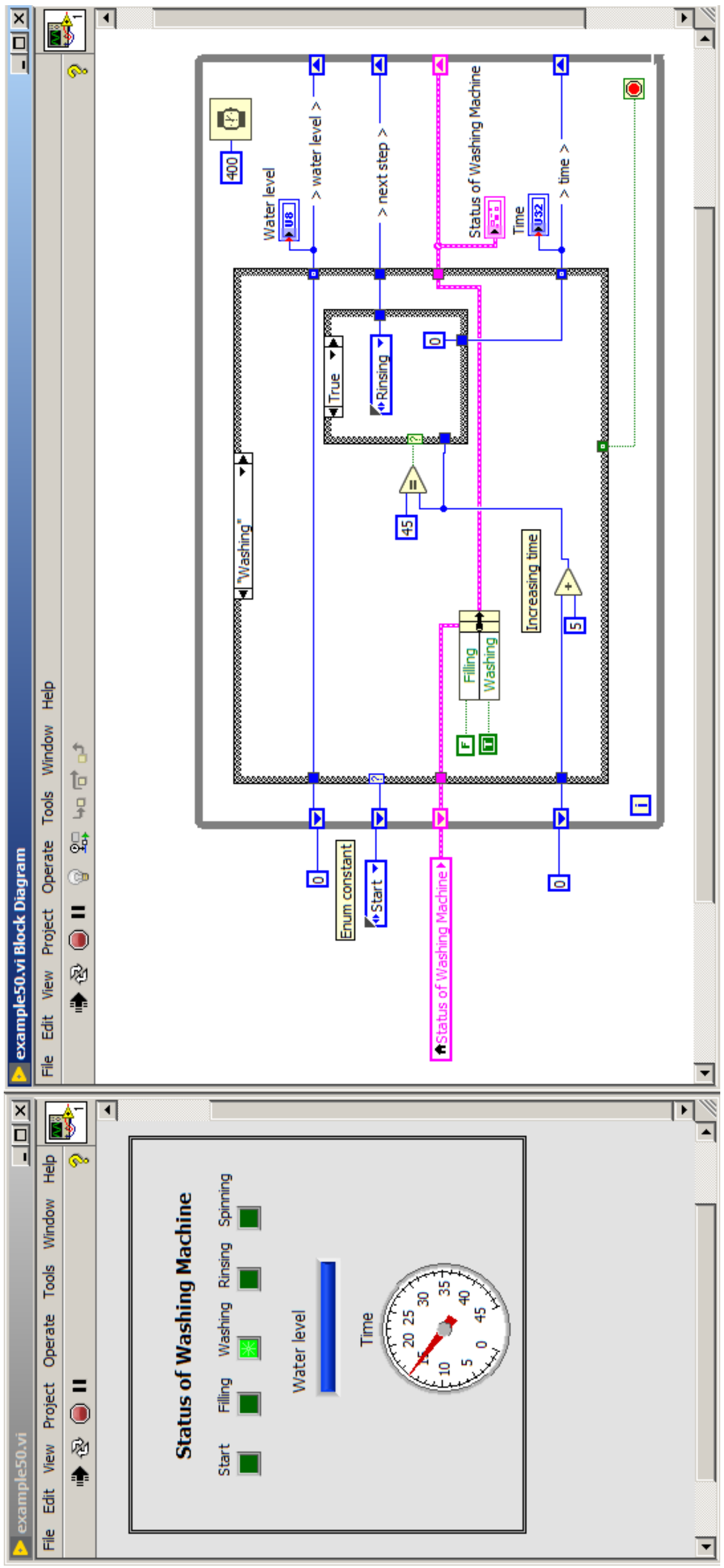

example50.vi Block Diagram
File Edit View Project Operate Tools Window Help
400
Water level
U8
> water level >
> next step >
Status of Washing Machine
Time
U32
> time >
"Washing"
True
Rinsing
0
45
Filling
Washing
F
T
Increasing time
5
0
Enum constant
Start
Status of Washing Machine
0
example50.vi
File Edit View Project Operate Tools Window Help
Status of Washing Machine
Start
Filling
Washing
Rinsing
Spinning
Water level
Time

Figure 95: Implementation of *Washing* state of washing machine simulator.

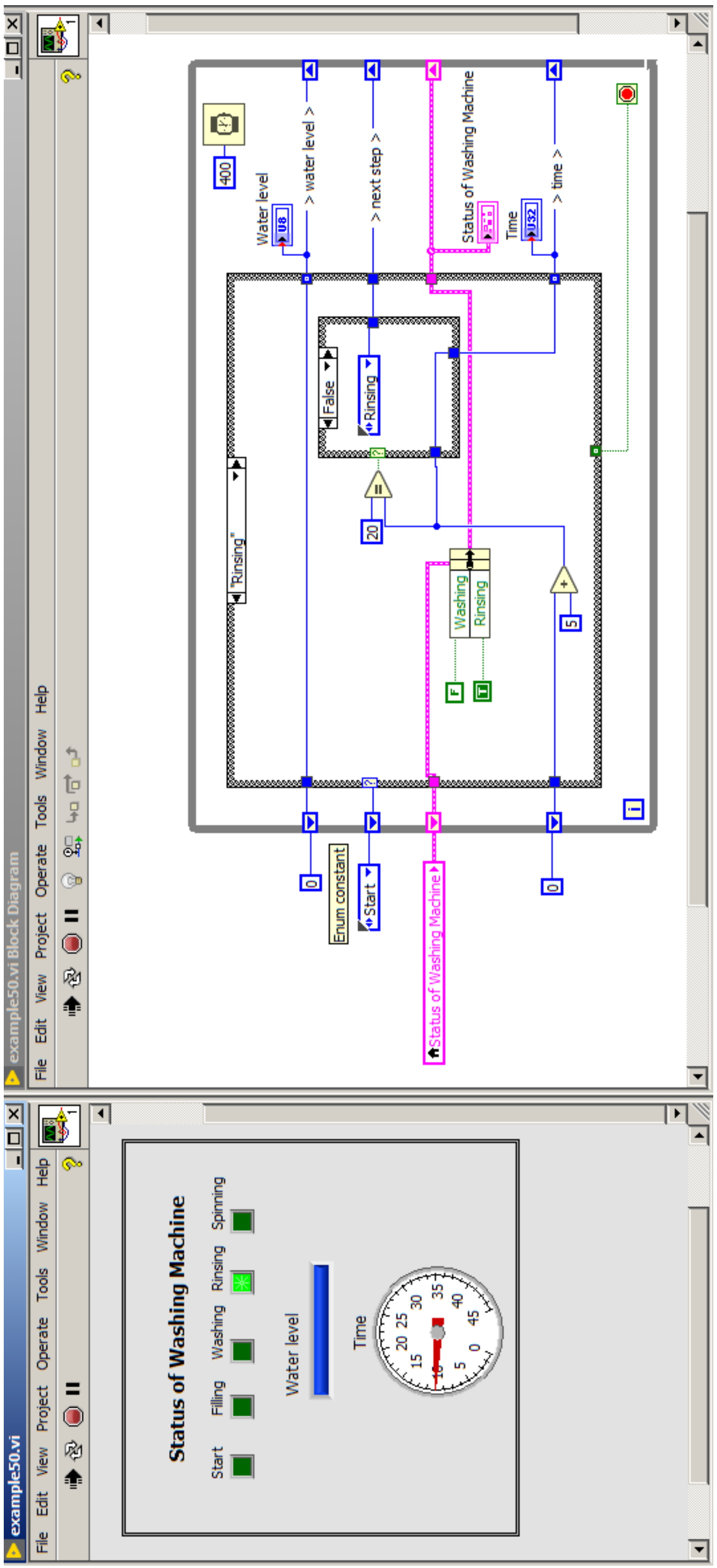


Figure 96: Implementation of *Rinsing* state of washing machine simulator.

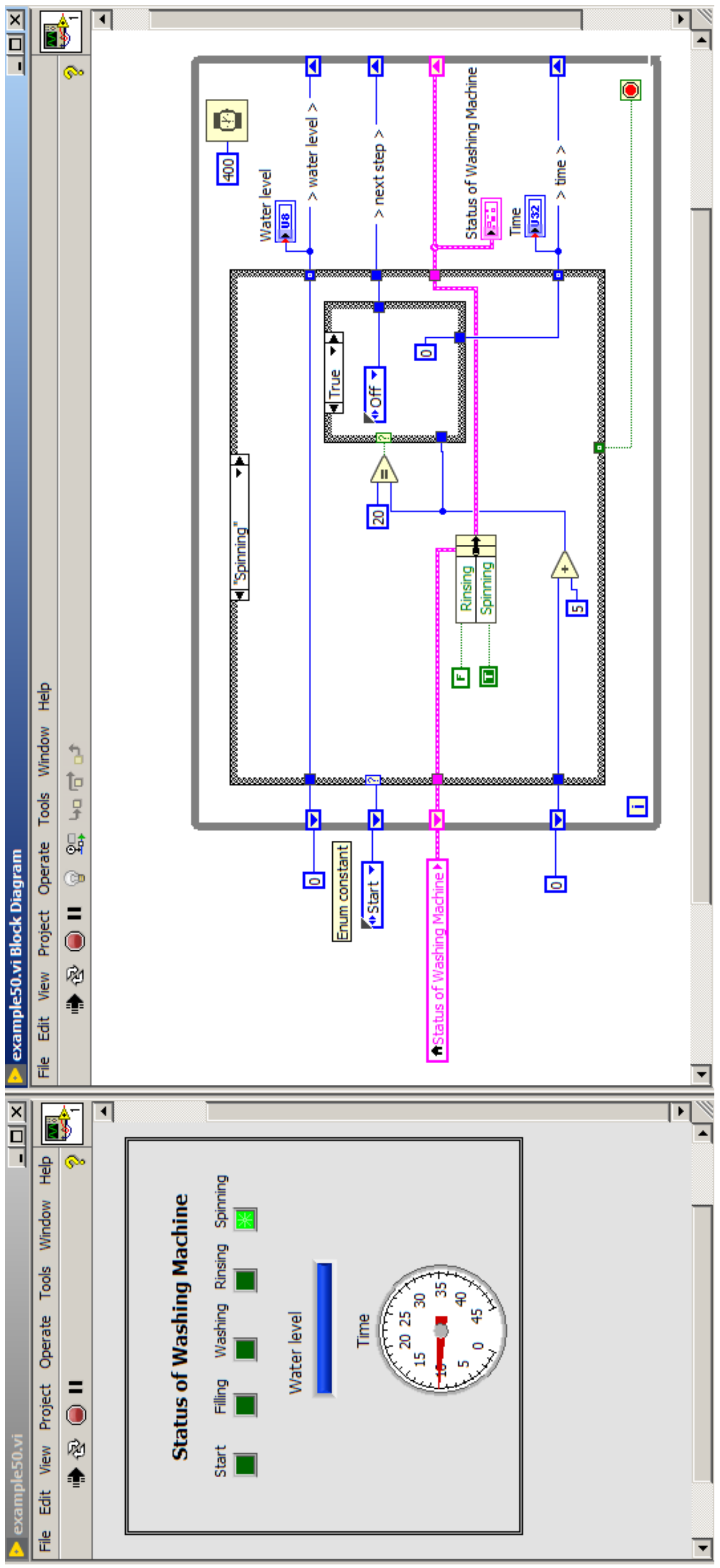

example50.vi Block Diagram
File Edit View Project Operate Tools Window Help
Water level
U8
> water level >
400
> next step >
Status of Washing Machine
Time
U32
> time >
"Spinning"
True
Off
0
20
Rinsing
Spinning
F
T
5
0
Enum constant
Start
Status of Washing Machine
0
example50.vi
File Edit View Project Operate Tools Window Help
Status of Washing Machine
Start
Filling
Washing
Rinsing
Spinning
Water level
Time
0
5
10
15
20
25
30
35
40
45

Figure 97: Implementation of *Spinning* state of washing machine simulator.

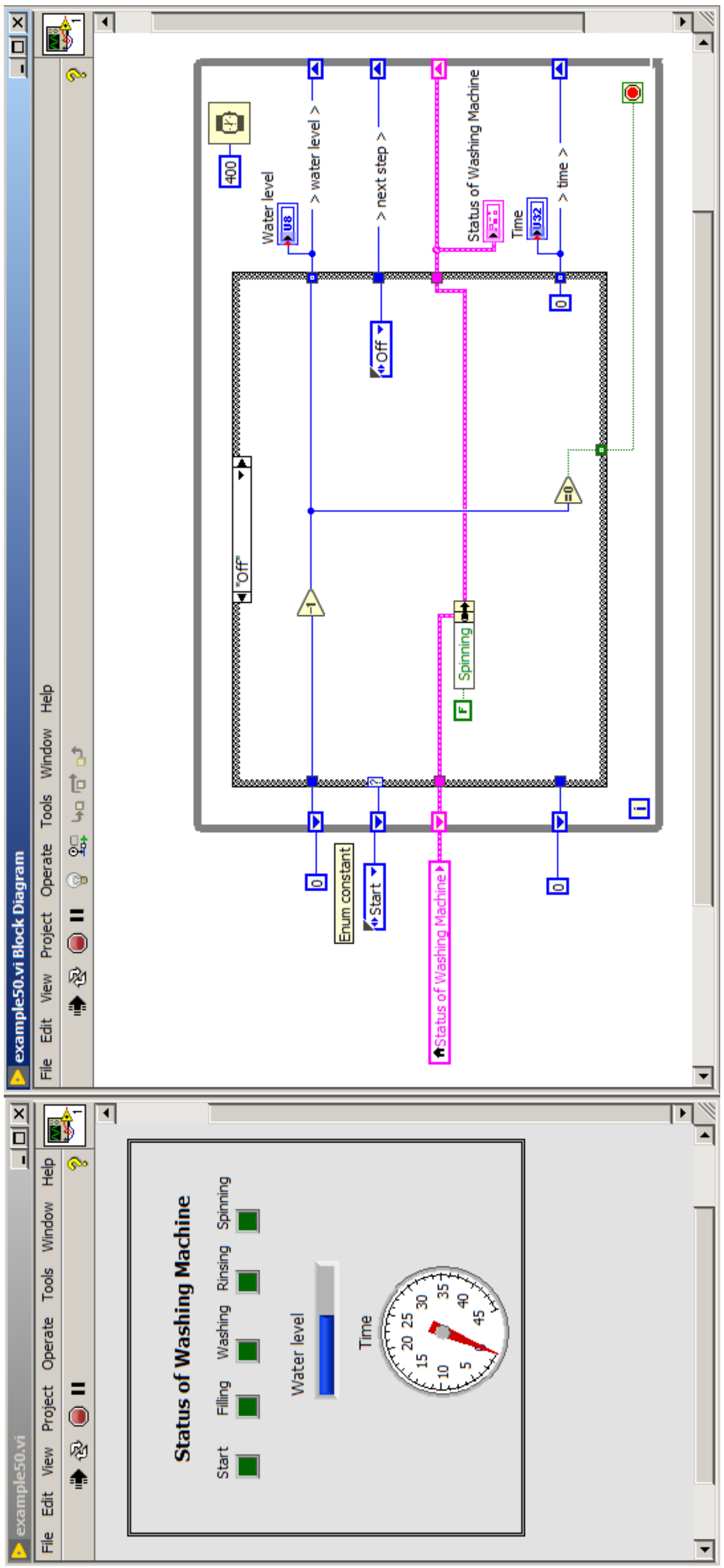


Figure 98: Implementation of *Stop* state of washing machine simulator.

Master/Slave Design Pattern

To communicate between parallel loops a local variable might be used. However it has significant disadvantage. If two loops are executed with different frequencies, the race condition phenomenon occurs. One of alternatives to local variable is notifier - a tool to synchronize two independent processes. It works similar way as local or global variables. In one place a value is passed to notifier, and in another place a value is read. But working of loop, where a value is read from notifier is hold until data arrives to notifier.

In figure 99 an example program showing differences between local variable and notifier work principles. In upper loop a sinus waveform is generated and passed to *Sine 1* chart and to *Value* indicator. Moreover, a notifier is created with use of *Obtain Notifier* function, and in the middle of upper loop sinus value is passed to notifier with use of *Send Notification* function. In the middle loop a value from *Value* indicator is read with use of local variable and passed to *Sine 2* chart. In lower loop a *Wait on Notification* function is used, which holds the while loop until new data are added to notifier in upper loop. Next it passes read data to *Sine 3* chart. Also in each loop a delay is added. Wherein the upper loop has a larger delay then the other two. As seen in figure 99, local variable from time to time reads twice the same value and *Sine2* chart differs significantly from *Sine 1* chart. On the other hand *Sine 3* chart is identical to *Sine 1* chart, because lower while loop is in hold until next value appears in notifier, so there is no risk of reading the same value multiple times.

Because of a feature of synchronization of multiple while loops with use of Notifier, it is used in Master/Slave design pattern. This design pattern is used when there is a need of performing some actions in parallel and with different frequencies. Master/Slave design patter consists of only one master loop and one or multiple slave loops. Master loop controls of slave loops operation and communicates with them through Notifier. Each loop realizes separate task, what makes the program more modular. Master/Slave design pattern is so popular, that there is no need to create it from scratch, but you can use a ready-made template, which is presented in figure 100. To do so, a *File* → *New* → *From Template* → *Frameworks* → *Design Patterns* → *Master/Slave Design Pattern* should be chosen.

Master/Slave design pattern is frequently used in cases, where in master loop, a graphical user interface is processed, and in slave loops data is processed, backed or commands to external devices are sent. However presented design pattern has a major disadvantage: slave loops always must work faster then master loop, because notifier doesn't have a data buffer.

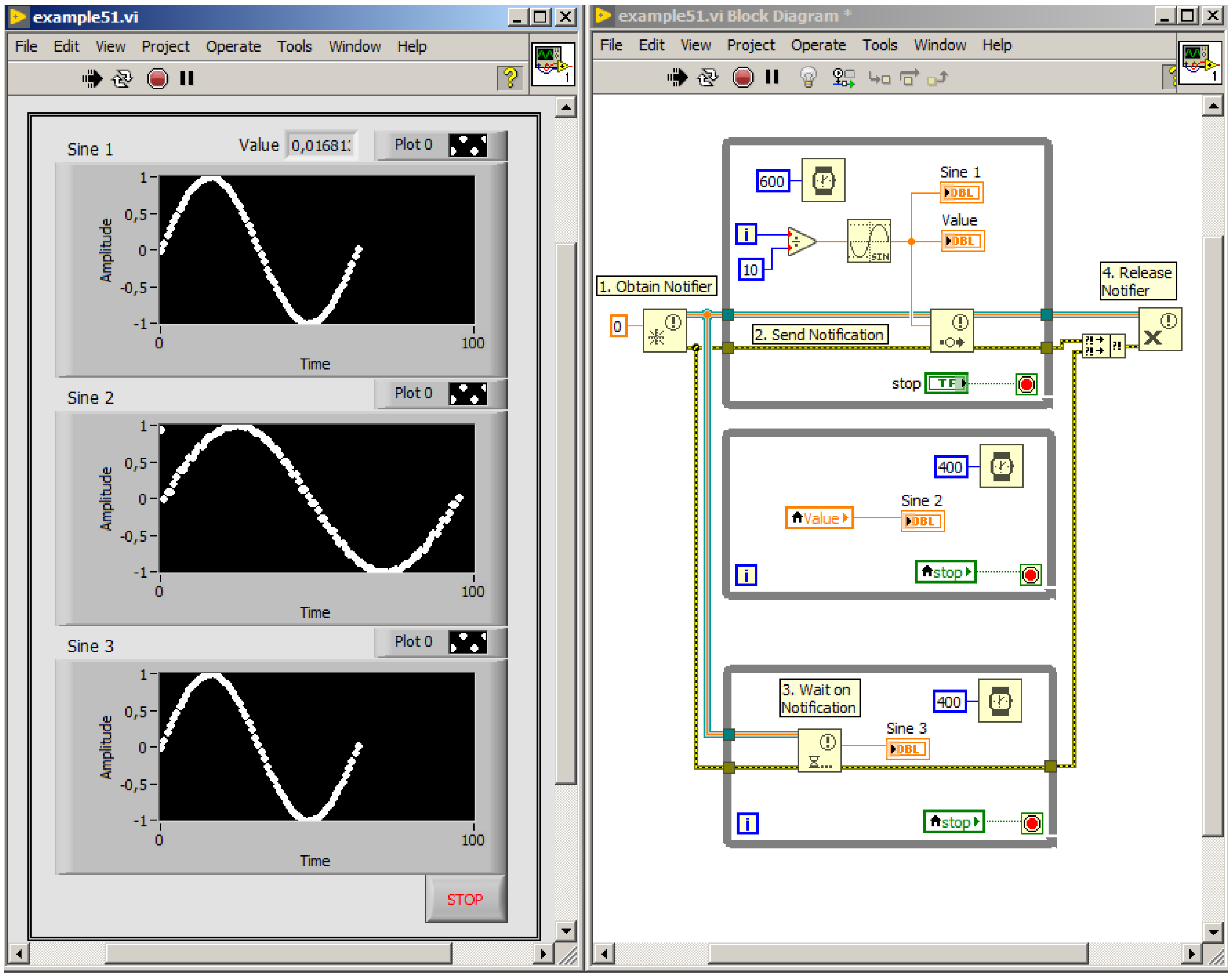


Figure 99: Program that demonstrates difference of operation of local variable and notifier.

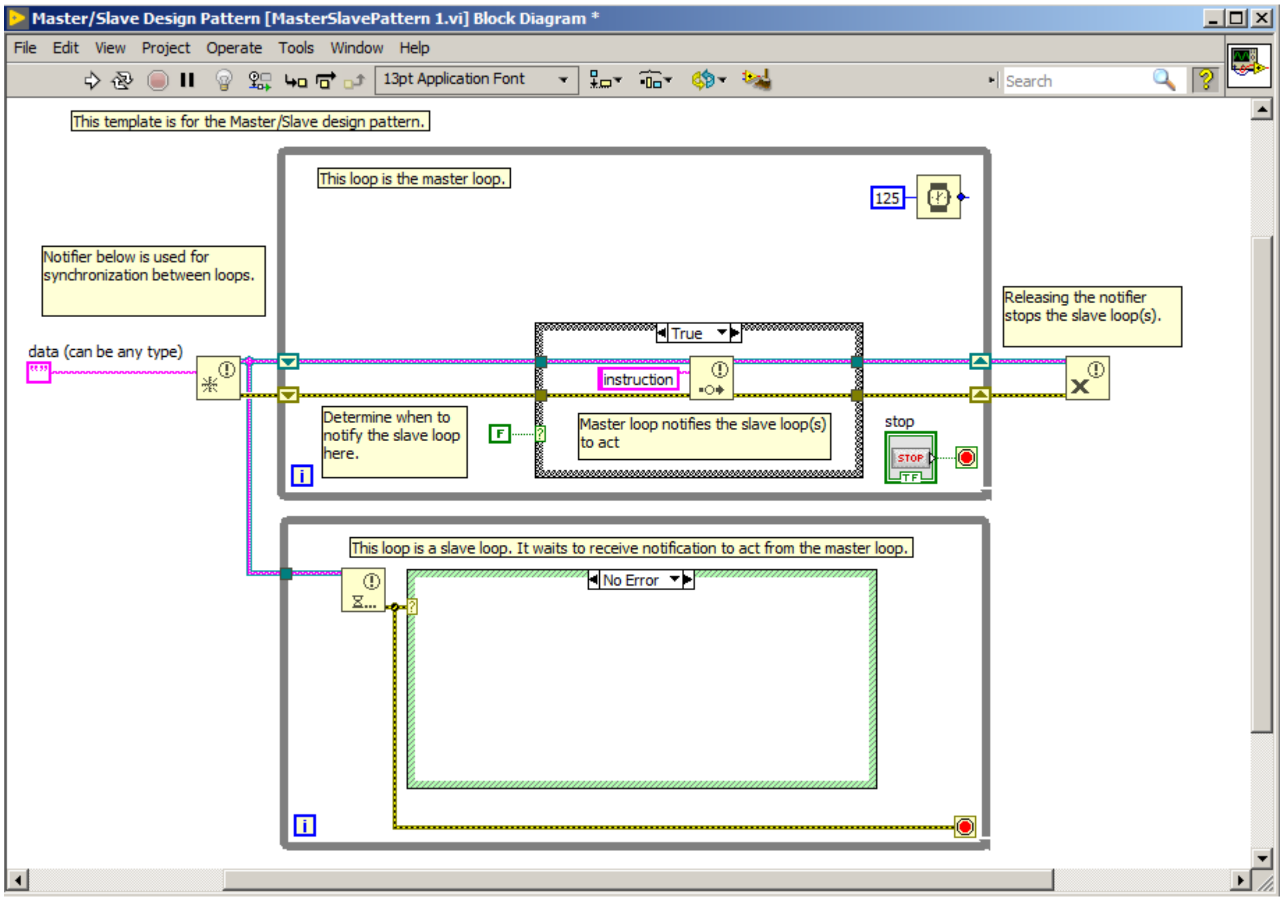

Master/Slave Design Pattern [MasterSlavePattern 1.vi] Block Diagram *
File Edit View Project Operate Tools Window Help
13pt Application Font
Search
This template is for the Master/Slave design pattern.
This loop is the master loop.
125
Notifier below is used for synchronization between loops.
data (can be any type)
True
instruction
Determine when to notify the slave loop here.
F
Master loop notifies the slave loop(s) to act
stop
STOP
Releasing the notifier stops the slave loop(s).
This loop is a slave loop. It waits to receive notification to act from the master loop.
No Error

Figure 100: Master/Slave Design Pattern template.

Producer/Consumer Design Pattern

In previous chapter, a notifier was described, which has no data buffer. As an alternative to notifier is a queue, which works as FIFO (first in, first out) buffer. Data, which enters the queue first, will also leave the queue first, which is shown in figure 101. In the queue, any data types can be stored.

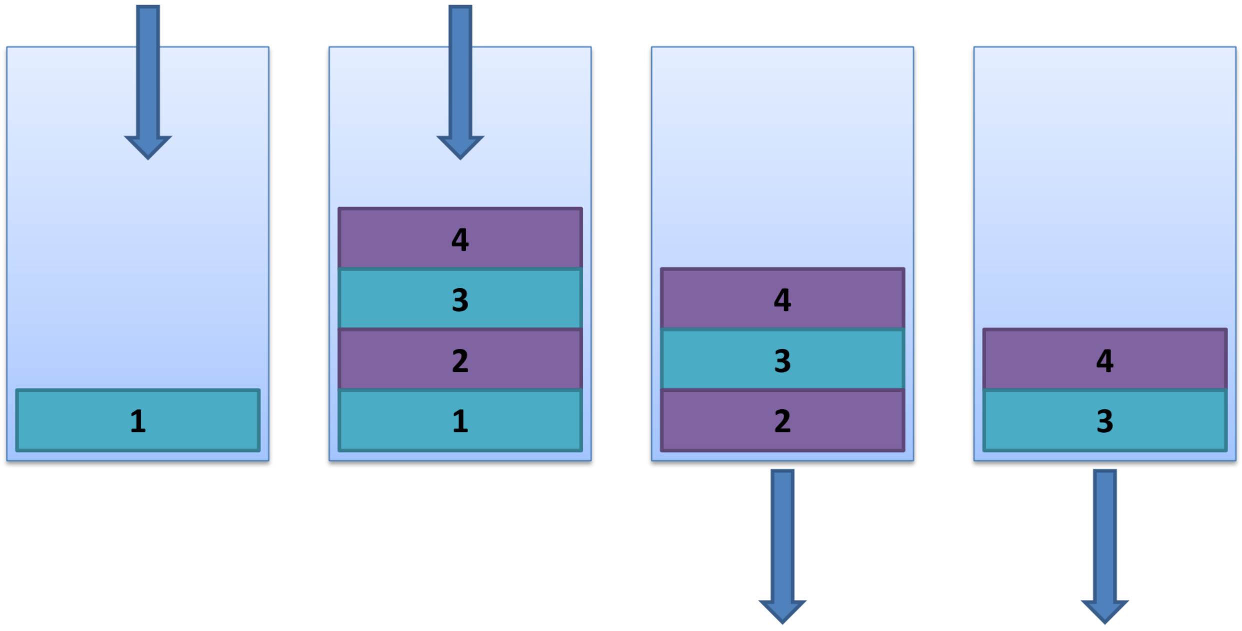


Figure 101: Principle of FIFO buffer operation.

Basic functions used to create and operate the queue (presented also in figure 102) are:

- **Obtain Queue** - Queue initialization. It returns a reference to queue.
- **Enqueue Element** - Adds an item to the end of queue.
- **Dequeue Element** - Returns and removes the currently read item from the queue.
- **Preview Queue Element** - Peek on next item in queue, without removal.
- **Get Queue Status** - Returns actual number of items in the queue.
- **Release Queue** - Releases a reference to the queue.

To understand the difference between notifier and queue, another loop is added to the example in figure 99. A queue was created wit use of *Obtain Queue* function. Next in upper loop, a sinus value is

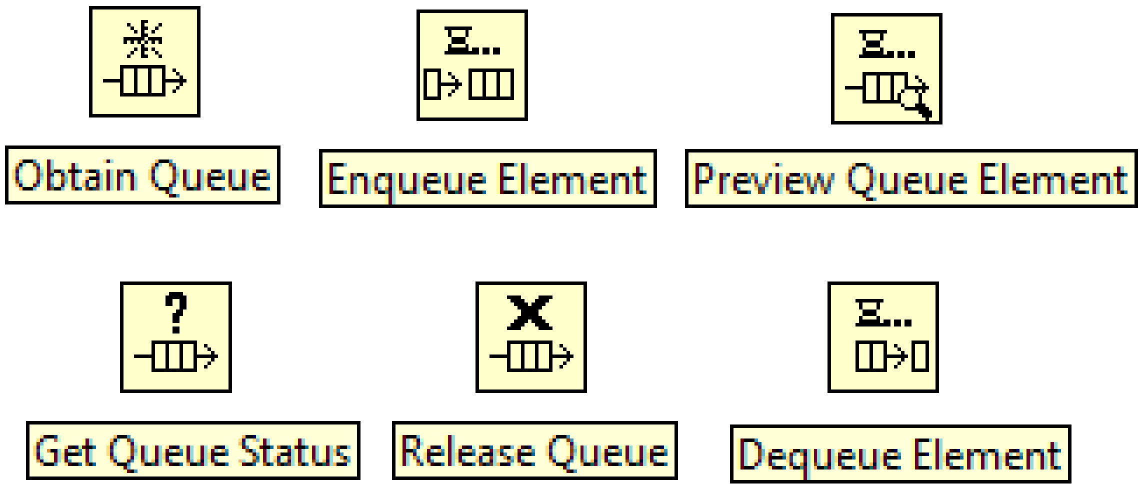


Figure 102: Basic functions to operate the queue.

passed to the queue with use of *Enqueue Element* function. In bottom loop a *Dequeue Element* is used, which holds loop operation until there are data to read. To observe the difference between the notifier and the queue, delays are changed. Upper loop has two times less delay than lower loops. The finished program is presented in figure 103. After starting the program, it is visible that charts: *Sine 2* and *Sine 3* have two times less points than *Sine 1* chart. Local variable and notifier doesn't have a data buffer, which causes data loss (these loops do not manage to read data before overwriting it by the upper loop). However the queue has a buffer and as a result *Sine 4* chart obtain all points generated in upper loop, but it takes some time.

Therefore, the queue is an effective tool to synchronize two loops independently of their frequency of operation. The Master/Slave design pattern has been modified by replacing notifier with queue. Such new design pattern is called Producer/Consumer Design Pattern. To create such design pattern, one needs to select *File* → *New* → *VI* → *From Template* → *Frameworks* → *Design Pattern*. On the list there will be two variants of Producer/Consumer design pattern called *Data* (presented in figure 104) and *Events* (presented in figure 105).

Producer/Consumer design pattern is based on two loop types: Producer Loop and Consumer Loop. First of them is used to "produce" data, e.g. reading data from measurement device. Second loop is used to process data, e.g. data analysis and visualization on charts. Communication between loops is provided by queue, which enables data buffering. Additionally, loops can work in different frequencies,

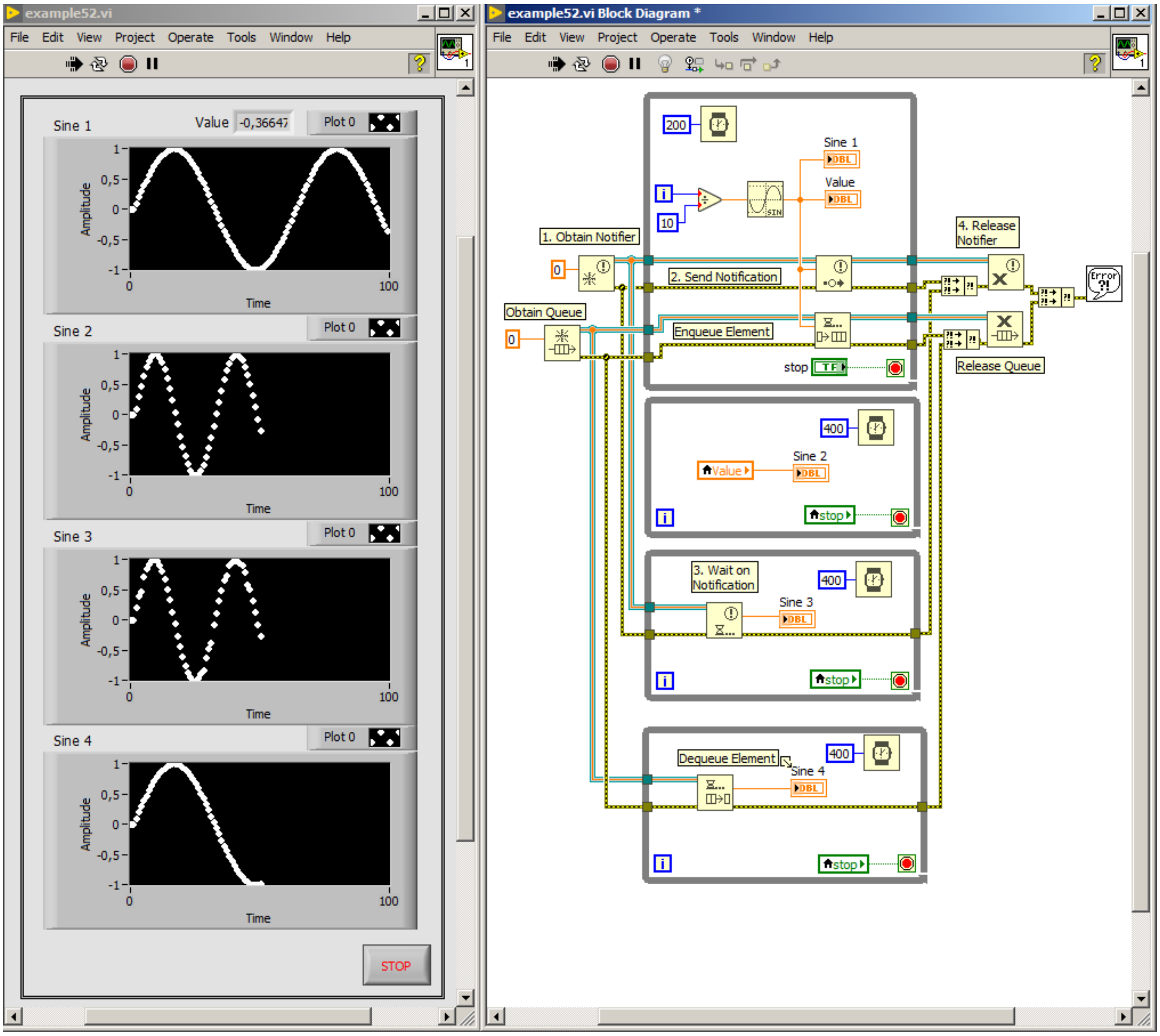
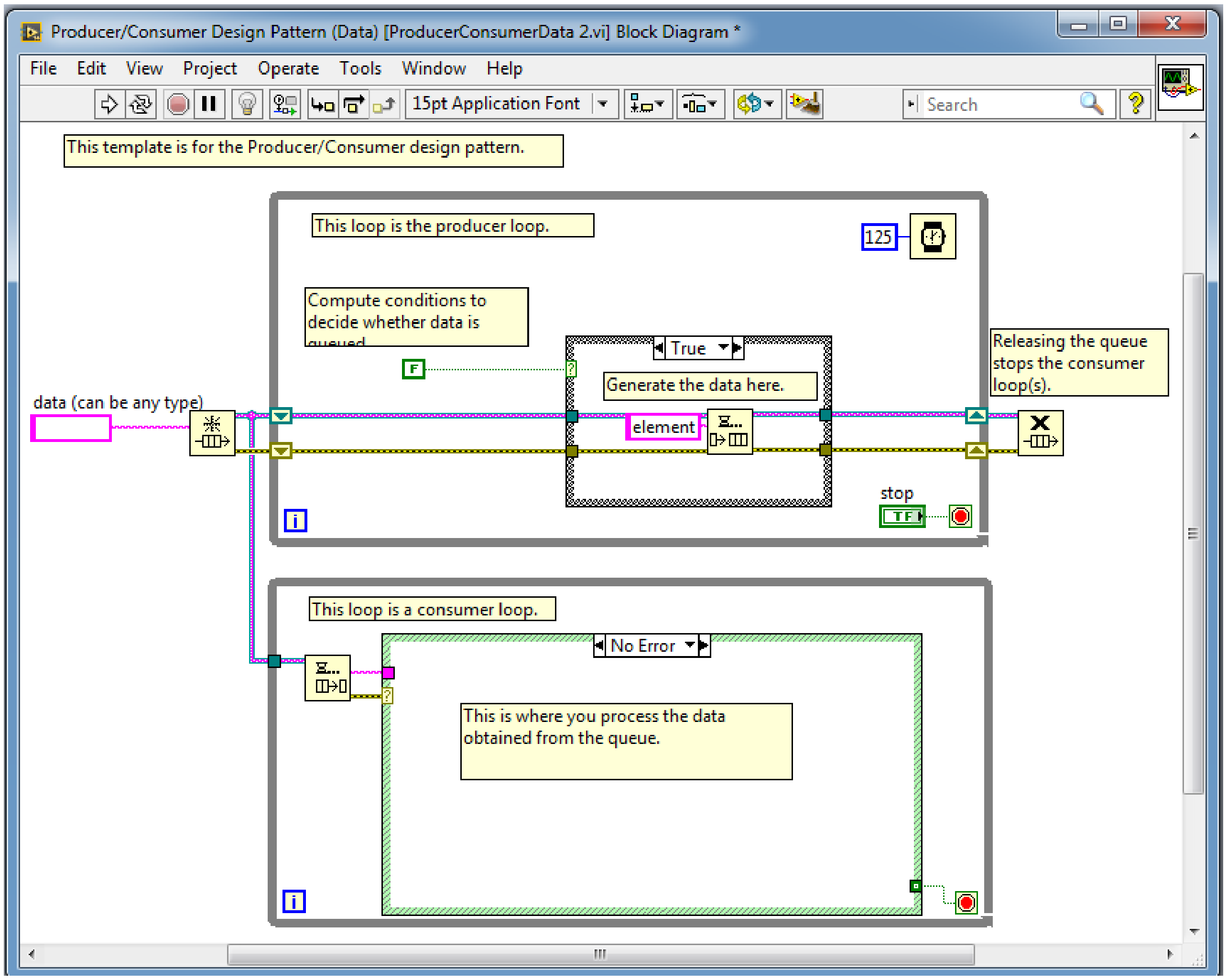


Figure 103: Program that demonstrates the difference between local variable, notifier and queue.

without worrying on data loss. Using Producer/Consumer is particularly recommended in data acquisition and data processing on demand. However there is one drawback, only one consumer loop can be used together with producer loop, this is because of buffer usage.


Producer/Consumer Design Pattern (Data) [ProducerConsumerData 2.vi] Block Diagram *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
data (can be any type)
element
True
F
125
stop
No Error

This template is for the Producer/Consumer design pattern.

This loop is the producer loop.

Compute conditions to decide whether data is queued

Generate the data here.

Releasing the queue stops the consumer loop(s).

This loop is a consumer loop.

This is where you process the data obtained from the queue.

Figure 104: Producer/Consumer (Data) Design Pattern template.

Producer/Consumer (Event) design pattern is a modification of previously presented design pattern by adding event structure to the Producer loop. This modification is used to process data on user demand. Using Producer/Consumer (Event) design pattern makes program more responsive (in sense of interactivity with graphical user interface).

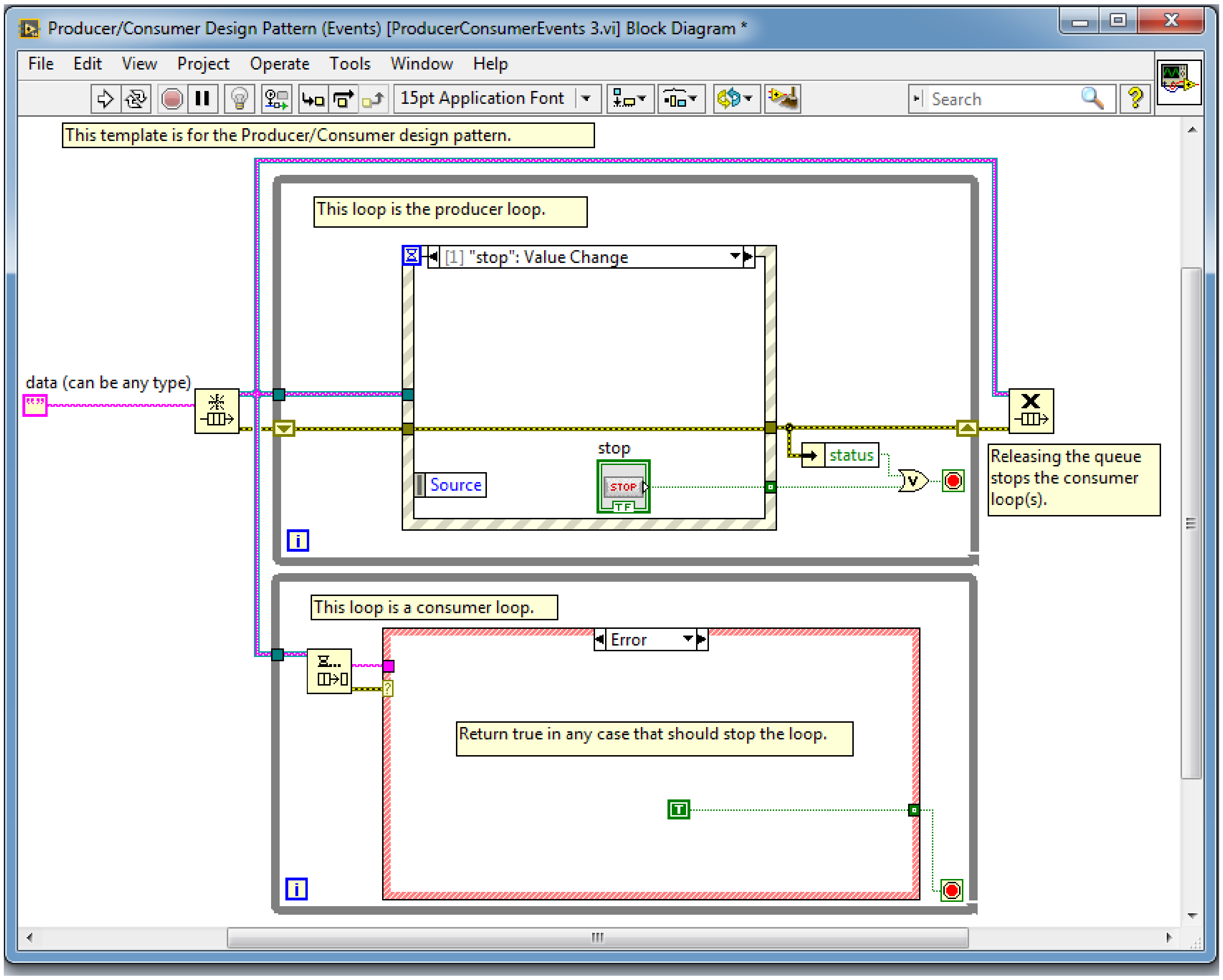

Producer/Consumer Design Pattern (Events) [ProducerConsumerEvents 3.vi] Block Diagram *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
This template is for the Producer/Consumer design pattern.
This loop is the producer loop.
[1] "stop": Value Change
data (can be any type)
stop
Source
status
Releasing the queue stops the consumer loop(s).
This loop is a consumer loop.
Error
Return true in any case that should stop the loop.

Figure 105: Producer/Consumer (Event) Design Pattern template.

Code documentation

Well prepared code documentation allows multiple programmers to work on the given project and allows program extension after many months. Code documentation should consists of following information: ideas, procedure descriptions, reference materials.

A list of useful elements for creating documentation:

1. *Labels*:
   - description of functionality of controls/indicators,
   - displayed in *Context Help* window and connector pane,
   - the name should include the default value and unit,
   - if control has limited range, please insert the free label to inform user about it.
2. *Captions*:
   - additional description of control/indicator functionality, which is displayed only on front panel,
   - use caption to describe functionality of controls/indicators and short label to save space in block diagram,
   - caption appears as tip strip in *Context Help* window.
3. *Tip strip* - additional description on yellow block, when user is focused on control/indicator.
4. *VI properties* should include the following items:
   - An overview of the VI
   - Instructions for using VI
   - Description of inputs and outputs.
5. *Free labels and comments*
6. *Wiring description*

LabVIEW interfacing techniques

Main goal of LabVIEW environment is to create an easy, cost effective way to produce professional grade applications for controlling measurement equipment. National Instruments LabVIEW is a real companion for scientists and engineers. Thanks to broad set of communication functions, analysis and visualization tools it makes almost perfect environment for rapid application development, testing and deployment. Ways of controlling hardware with use of LabVIEW environment are presented in this chapter.

Many commercial equipment like multimeters, oscilloscopes, generators, e.t.c. already have libraries to communicate with devices using LabVIEW environment. Typically there are functions to: initialize equipment, configuration of operation mode, then there are loops which control continuous work e.g. data acquisition. It can be prepared in design master/slave or producer/consumer design pattern. And finally at the end, one needs to close device and pass information about errors. The typical program structure for accessing hardware (measurement device) is presented in figure 106.

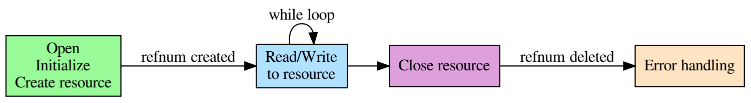


Figure 106: Typical program structure for accessing hardware

Drivers

The term drivers applies to distributed by National Instruments or third party LabVIEW functions (blocks) which are used for control of hardware devices connected to PC. This is the most convenient option, because such functions are specific and optimized for devices and are easy to use - user needs to drag and drop them on block diagram and connect wires. Frequently example applications using these drivers are also supported which allows for rapid development of any functionality user demands from data acquisition system. If such drivers are installed, they new functions appear in the Function Palette and are available to users. National Instruments devices have well defined interface for measurement equipment which is called DAQmx. Knowing how to use such drivers, user can practically program whole spectrum of available devices. In National Instruments portfolio there are also highly modular devices like CompactRIO, real-time controllers which use internally real-time Linux kernel. LabVIEW program can be easily deployed on such device.

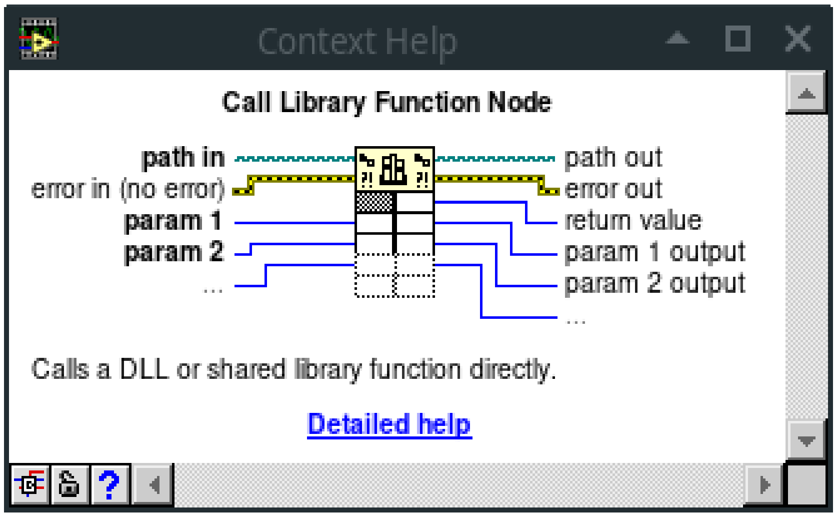
Nowadays many vendors of measurement equipment prepare LabVIEW functions (blocks), which helps a lot in development of measurement system. Of course we can expect support of National Instruments on such drivers for National Instruments produced devices, however in third party products it may be quite different. Bear in mind, that sometimes support only applies to specific versions of LabVIEW and some old drivers may not work in new environment or vice-versa. This is quite problematic in third party products.

To control hardware devices, common practise for good hardware vendors is to distribute C language libraries (drivers) for PC computer for their devices. These libraries consists of functions which can control every aspect of the device. The C API is a standard. So sooner or later, users who deal with measuring system software need to read, understand and write C code. In some areas, C++ code is also used, however C++ is not the best solution for such low-level software, because of its complexity of object oriented programming and different calling convention from C (symbol naming is mangled in compiled code). This is why in C++ we need to export functions as extern "C" if we want to cooperate with other code. In C++ we need to prepare "proper interface" if we want to make such code usable from other languages like LabVIEW also.

But what to do, if we don't have already prepared LabVIEW functions in third party product? There is a way of calling C functions in LabVIEW, which can be used for preparation of such interface. This will be explained in following chapter.

Calling external libraries

To call a function in external library, one need to know the name of function, and types of all function arguments and type of return value and use *Call Library Function Node* function, which is available in *Functions Palette* → *Connectivity* → *Libraries & Executables* → *Call Library Function Node*. Context help window for this function is presented in figure 107



Calls a DLL or shared library function directly.

Figure 107: Context help window for *Call Library Function Node*.

An example program which uses a function from external library is presented in figure 108. There, in function *Call Library Function Node* a names option was chosen in right click menu, to show on block diagram the name of function and names of arguments and return value.

In this example a LabVIEW for Linux (version 2017) was used. However it works almost the same on Windows platform. One visible difference is that external shared libraries on Windows are called "Dynamically Linked Libraries" and have a *.dll* suffix, whereas on Linux they are called "Shared Objects" and have a *.so* suffix.

Having a library it is not enough to use it. First we need to know what functions we need to use, how they should be called, in which order, what data structures should be prepared and how. In simple cases it is easy, but in complex situations, documentation to library is needed badly. It is also important to have header files for library (with *.h* suffix). This will help to reveal function signatures and argument types.

LabVIEW has limited possibilities of argument types we can use to call functions and receive return values. If e.g. library functions pass structures, then you will need to prepare some kind of a wrapper - another library prepared in C language by you, that will accept simple arguments from LabVIEW and communicate with functions from problematic library. These however exceeds topic of this book. If library is big and complicated, the work to interface LabVIEW with external library functions may be tedious and time consuming. However calling functions from external libraries may be the fastest method to acquire data from hardware devices. Accepted types of return values

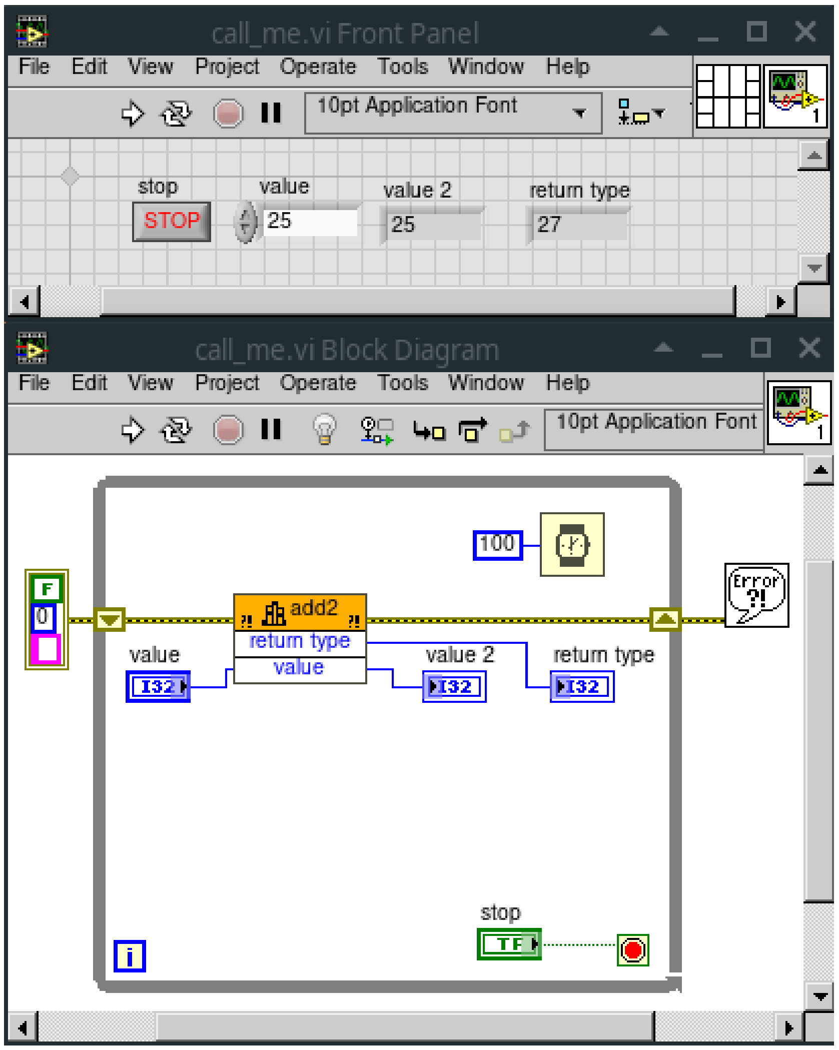


Figure 108: Program demonstrating use of *Call Library Function Node*.

in LabVIEW are: void (no return), string and numeric. Numeric types can be further configured as:

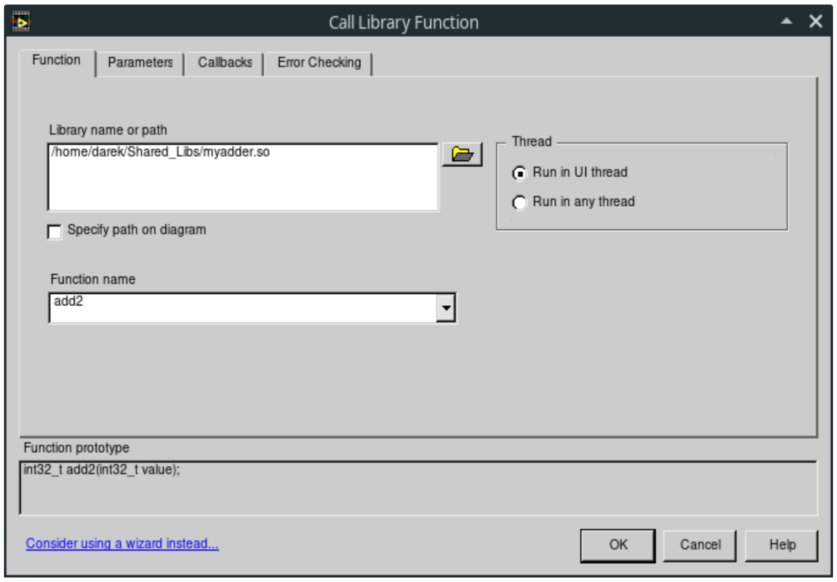
- Signed 8-bit Integer
- Signed 16-bit Integer
- Signed 32-bit Integer
- Signed 64-bit Integer
- Signed Pointer-sized Integer
- Unsigned 8-bit Integer
- Unsigned 16-bit Integer
- Unsigned 32-bit Integer
- Unsigned 64-bit Integer
- Unsigned Pointer-sized Integer
- 4-byte Single (Float)
- 8-byte Double

In case of string type, we have then an option of:

- C String Pointer
- Pascal String Pointer
- String Handle (not available in Linux)
- String Handle Pointer (not available in Linux)

Pascal String has length specifier and in classic definition has max. 255 chars, however, to create C-string (which is an variable array of chars, ended with zero value char), LabVIEW may need to preallocate some memory in their specific string type. In such case we can give a hint to LabVIEW specifying "Minimum size" in chars (bytes).

To configure *Call Library Function Node* one needs to click the right mouse button on *Call Library Function Node* and choose *Configure...* from menu. In fold *Function* Library or path should be filled to point the external library (.dll in Windows or .so in Linux). A function name should be also chosen there. On the bottom of window a function prototype is displayed. In windows version there is also *Calling convention* selection, where C uses __cdecl calling convention (standard C) and stdcall (WINAPI) used e.g. on Windows GUI libraries. There is also a thread select radio button. It is important to select if called functions should run in UI (User Interface) thread (which may be blocked if such



Library name or path: /home/darek/Shared_Libs/myadder.so

Specify path on diagram

Thread: [✓ Run in UI thread] Run in any thread

Function name: add2

Function prototype: int32_t add2(int32_t value);

Figure 109: Configuration of *Call Library Function Node*. Function fold.

function takes time) or any other thread. This configuration window is present in figure 109.

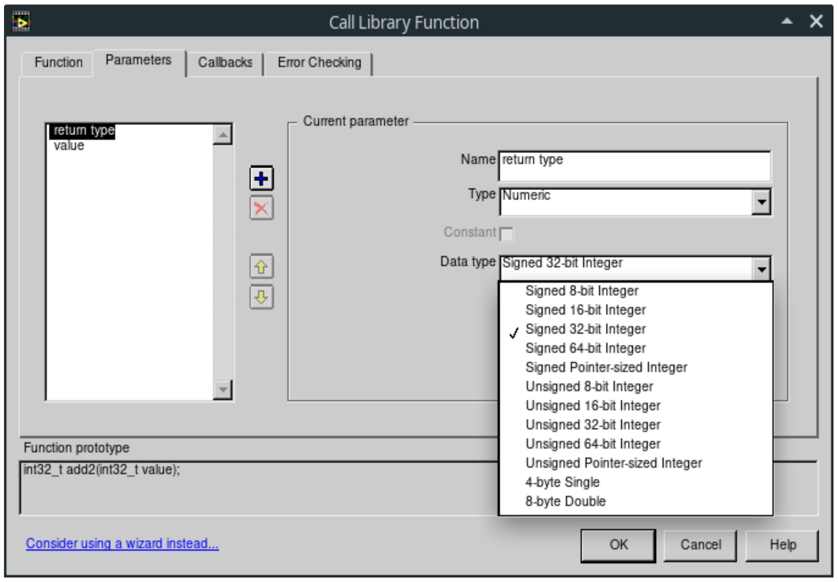
Then we need to specify function return value and its arguments, which is presented in figure 110 and 111. Each item can be added by clicking the black "plus" button and deleted by the red "cross" button. Items can be also arranged up and down using buttons with arrows.

In case of arguments, we have an option to pass data by Value or by Pointer to Value. The second option, allows function to modify the value, which is passed from LabVIEW side (some place in memory, which pointer points to it). That is why, in *Call Library Function Node*, we have an option to write argument (left side) and read it (right side) of icon.

If someone wants to prepare a wrapper in C, there is also a convenience option in right click menu, which allows to save prototype functions to C file, so you can then use these to prepare and compile correct wrapper.

Range of argument types is broader. There are following types: Numeric, Array, String, Waveform, Digital Waveform, Digital Data, ActiveX, Adapt to Type, Instance Data Pointer. However most of them are types defined and used in National Instruments libraries e.g. in LabWindows/CVI. In case of Array, we also need to specify dimension, data type (of single cell), array format(whether it is Array Data Pointer, Array Handle or Array Handle Pointer and Minimum size.

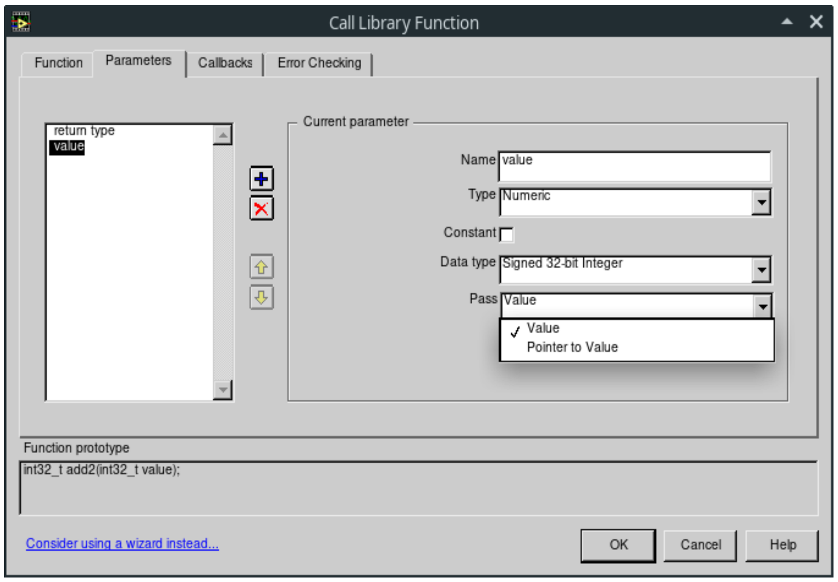
C code for library used in example is presented in listing 3. This is accompanied with header file in listing 2 and Makefile in listing 1 to easily compile under GNU/Linux system with use of GCC compiler.



Name: return type
Type: Numeric
Constant
Data type: Signed 32-bit Integer

Function prototype
int32_t add2(int32_t value);

Figure 110: Configuration of *Call Library Function Node*. Parameters fold (return value).



Name: value
Type: Numeric
Constant
Data type: Signed 32-bit Integer
Pass: Value

Function prototype
int32_t add2(int32_t value);

Figure 111: Configuration of *Call Library Function Node*. Parameters fold (arguments).

Similarly it can be prepared under Windows environment using Open Source tools such as MinGW or Cygwin or using closed source tools like Microsoft Visual C compiler. In our example if we build example library by invoking *make* command, we should get ready shared object library: myadder.so. This can be then specified in *Call Library Node* which is shown in figure 109. After specifying return value and argument type, which is present in figure 110 and 111 test program is almost ready. The complete example is presented in figure 108. If we run the program, we can see that LabVIEW calls the external library function, and whatever number we enter to "value" control, we will get value increased by 2 in "return type" indicator. This is how a very simple C function from the external library works.

Listing 1: Makefile : Makefile for example external library

```
myadder.so: myadder.o
        gcc -shared -o myadder.so myadder.o

myadder.o: myadder.c
        gcc -c -Wall -Werror myadder.c

.phony: clean

clean:
        rm -f myadder.so myadder.o *~
```

Listing 2: myadder.h : Header file for example external library

```
#ifndef _MYADDER
#define _MYADDER

extern int add2(int value);

#endif
```

Listing 3: myadder.c : C source file for example external library

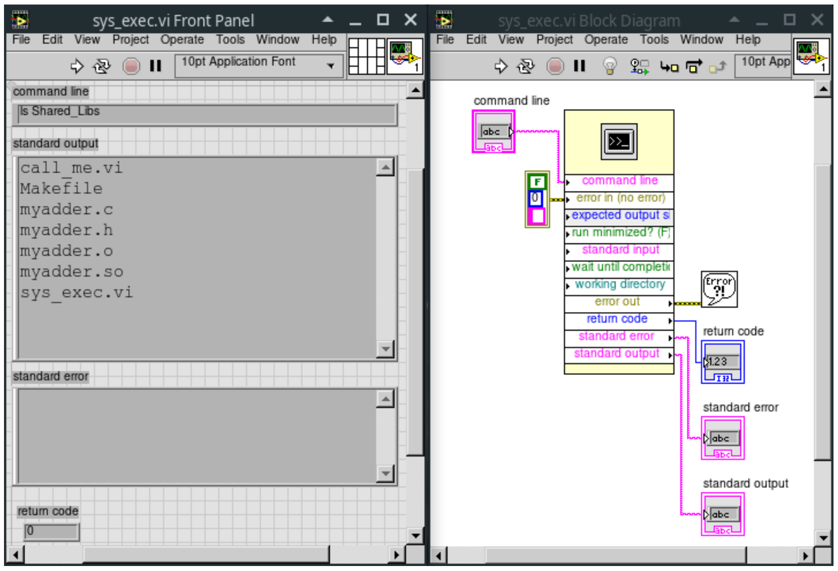
```c
#include "myadder.h"

int add2(int value){
  return value+2;
}
```

Executing system commands

Sometimes we can get or prepare external program which will communicate with hardware or prepare complex calculations, simulations, e.t.c. If such program can accept command line parameters and can return data to standard output, we can use another LabVIEW function to call such a program or any other tool available in system, which is named *System Exec*. This function looks more comfortable if option "View as icon" in right-click menu is disabled. Then we can expand named fields, where *command line* is just a command line one could use in terminal window to run this command. If we want to pass something to the running program, we can do it via *standard input* field. Results from program can be viewed from *standard output* or *standard error* depending on message nature and program design. This will work if *wait until completion* is enabled (default behaviour). Finally there is error code output, which is of integer type and is a normal return code from any program which is executed in system. This may be used for diagnostics. In *System Exec* function we can also specify *working directory* which will be used by program, *expected output size* which may improve memory efficiency (this should be more than actual size). Default value is 4096 characters (bytes). We have also typical in LabVIEW error cluster input and output. An example program which calls "ls" - standard command to list files in directory on GNU/Linux system is presented in figure 112. Similar results could be achieved on Windows platform with use of "dir" command. Consider also calling shell scripts which will actually run programs. The possibilities are virtually limitless, e.g. is possible, to run Windows program, using "Wine" environment and execute it from "LabVIEW for Linux". These possibilities are really great for engineers who interface between Open and Closed Source solutions.

VISA

Sometimes measurement equipment doesn't have drivers in LabVIEW or C library, but has well defined interface and well defined format of



```
ls Shared_Libs
```

```
call_me.vi
Makefile
myadder.c
myadder.h
myadder.o
myadder.so
sys_exec.vi
```

```
0
```

Figure 112: Program that demonstrates calling another program using *System Exec* function in LabVIEW. Here an "ls" command is called on GNU/Linux system.

data transmission. Together with device manual, user can easily program such device using ready LabVIEW functions for communication over several interfaces like:

- Serial
- Parallel
- GPIB (IEEE-488.2)
- VXI
- Ethernet
- USB

Also standardized protocol for communication with measurement equipment was developed. It is named *SCPI* for Standard Commands for Programmable Instruments.

In LabVIEW there is a special High Level API (Application Programming Interface) called VISA (Virtual Instrument Software Architecture) which is a common way to access different hardware interfaces like Serial, Parallel or GPIB. In VISA toolkit, which is placed in *Function Palette* $\rightarrow$ *Instrument IO* $\rightarrow$ *VISA*, there are common functions like Write, Read and specific for bus (interface) are placed in *Advanced* $\rightarrow$ *Bus specific*.

Although there are many interfaces which could be used to communicate with equipment, the most simple interface, which can be used for Open Source projects easily is actually a Serial interface. Many boards like Arduino, although connected via USB interface, in reality uses CDC (Communication Device Class) which is recognized and used by system as typical Serial interface. In VISA for every device on every interface there is unique identifier called Instrument Descriptor:

- Asynchronous serial - ASRL[device][::INSTR]
- GPIB - GPIB[device]::primary address[::secondary address][::INSTR]
- VXI - VXI[device]::VXI logical address[::INSTR]

where device is device number, in some interfaces there are addresses, and last part specifies VISA session type (INSTR or Event).

In case of serial communication first, serial interface has to be configured. There are specific parameters like serial transmission speed (baud rate), format of data frame like bit stop number, how many data bits, parity bit. Everything must match on both side of transmission channel.

Ethernet communication

Very useful technique is socket communication. It can be used not only to control equipment over network, but also to communicate with server or client program on the same machine (as Inter Process Communication). For that in LabVIEW there are separate functions located in *Data Communication → Protocols*. Here we have separate *TCP* and *UDP* folders and also ready functions for e-mail sending *SMTP Email* as well as Hypertext Pages download *HTTP Client* and File Transfer Protocol *FTP*.

Nowadays there is a very popular library which helps proper development of networked software. It is called *ZeroMQ* `https://zeromq.org` - Zero Message Queue. The main goal is to create not prone to errors network communication with already predefined templates for network protocols. There is plenty of different scenarios for client/server, publisher/subscriber e.t.c. This library is completely Open Source - LGPL licensed. It can be installed using VI Package Manager. A wide range of possibilities and ease of use predestine the use of this library for cooperation with other software or external devices that are able to communicate according to these protocols. This means mostly System on Chip devices, which operates on GNU/Linux system like Raspberry PI, Beaglebone, FPGA devices with hard CPU cores like Xilinx Zynq, e.t.c.

If we develop some devices for Internet of Things, it is worth to consider the *MQTT* library, which allows LabVIEW application to connect to Message broker like Eclipse *Mosquitto* `https://mosquitto.org`.

Such library was prepared by *daq.io* `https://www.daq.io`. Source for library is available in GitHub `https://github.com/DAQIO/LVMQTT`. This is a way to use cloud based data storage for e.g. data acquired from sensors, which can be spread all over the world.

Open source platforms

Arduino boards

Arduino is an open-source electronic platform, which heart is an 8-bit microcontroller Atmel AVR. This platform was created as a simple and inexpensive tool for prototyping for students without the electronic and programming background. By its simplicity, it gained supporters around the world, especially in so-called makers. Due to the growing demand Arduino began to develop by adding new versions of boards dedicated to various purposes, e.g. Internet of Things (IoT), medical measurements, construction of 3D printers or embedded systems. The list of Arduino board versions is long but the most common used boards are shown in figure 113.

The Arduino UNO was the first USB Arduino board. It is a reference model for Arduino platform. The heart of this board is the ATmega328P microcontroller. The Arduino UNO is equipped with 14 digital input/output pins and 6 analog inputs. The 6 of 14 digital pins also can be used as PWM outputs (Source: `arduino.cc`).

The Arduino Leonardo board is dedicated to projects, when USB communication is needed. Especially, this board can be connected to a computer as a mouse or keyboard. The heart of Arduino Leonardo is the ATmega32u4 microcontroller and it is equipped with 20 digital input/output pins, in which 7 of them can be used as PWM outputs, and 12 analog inputs (Source: `arduino.cc`).

The Arduino Mega 2560 is based on the ATmega 2560 microcontroller and it was designed for more complex projects especially for the 3D printers and robotics projects. This board is equipped with 54 inputs/outputs, in which 15 of them can work as PWM outputs, and there are 16 analog inputs.

The Arduino LilyPad is special version dedicated to wearables projects, based on the ATmega328 microcontroller with 9 digital inputs/outputs and 4 analog inputs. This board can be attached to the textile by special conductive thread. The Arduino LilyPad can be washed by hand with a mild detergent after removing the batteries and other power supplies. Then the board should be dried according to the description of this board (Source: arduino.cc). The disadvantage of the Arduino LilyPad is lack of stabilized power supply, which the user have to deliver.

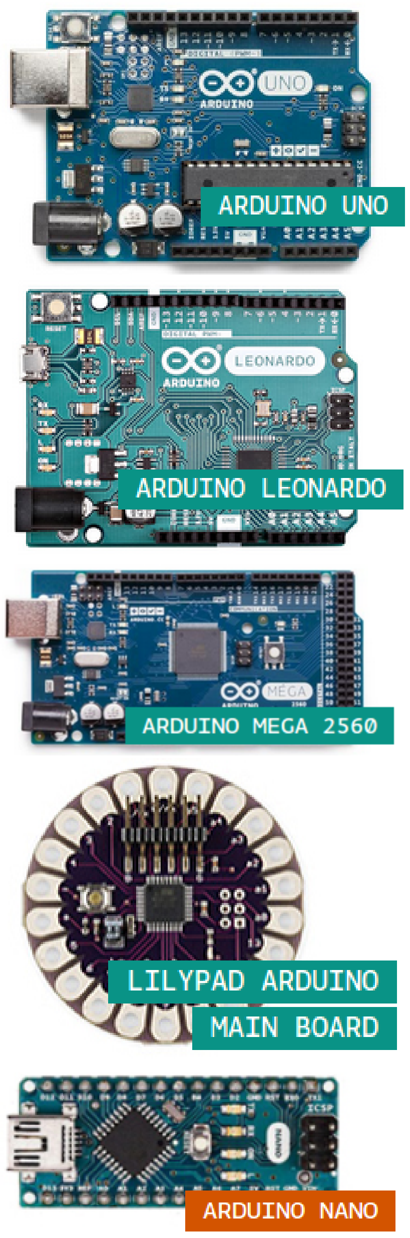


Figure 113: The most popular Arduino Boards: Uno, Leonardo, Mega 2560, LilyPad and Nano. Source: arduino.cc

The Arduino Nano is the smallest board with dimensions: 18 x 45 mm. The heart of this board is the ATmega328 and it is equipped with 22 digital inputs/outputs, in which 6 of them can be used as PWM outputs, and there are 8 analog inputs. The size of the board is compatible with breadboards.

The building of the Arduino UNO board is shown in figure 114 and the Arduino Mega 2560 in figure 115.

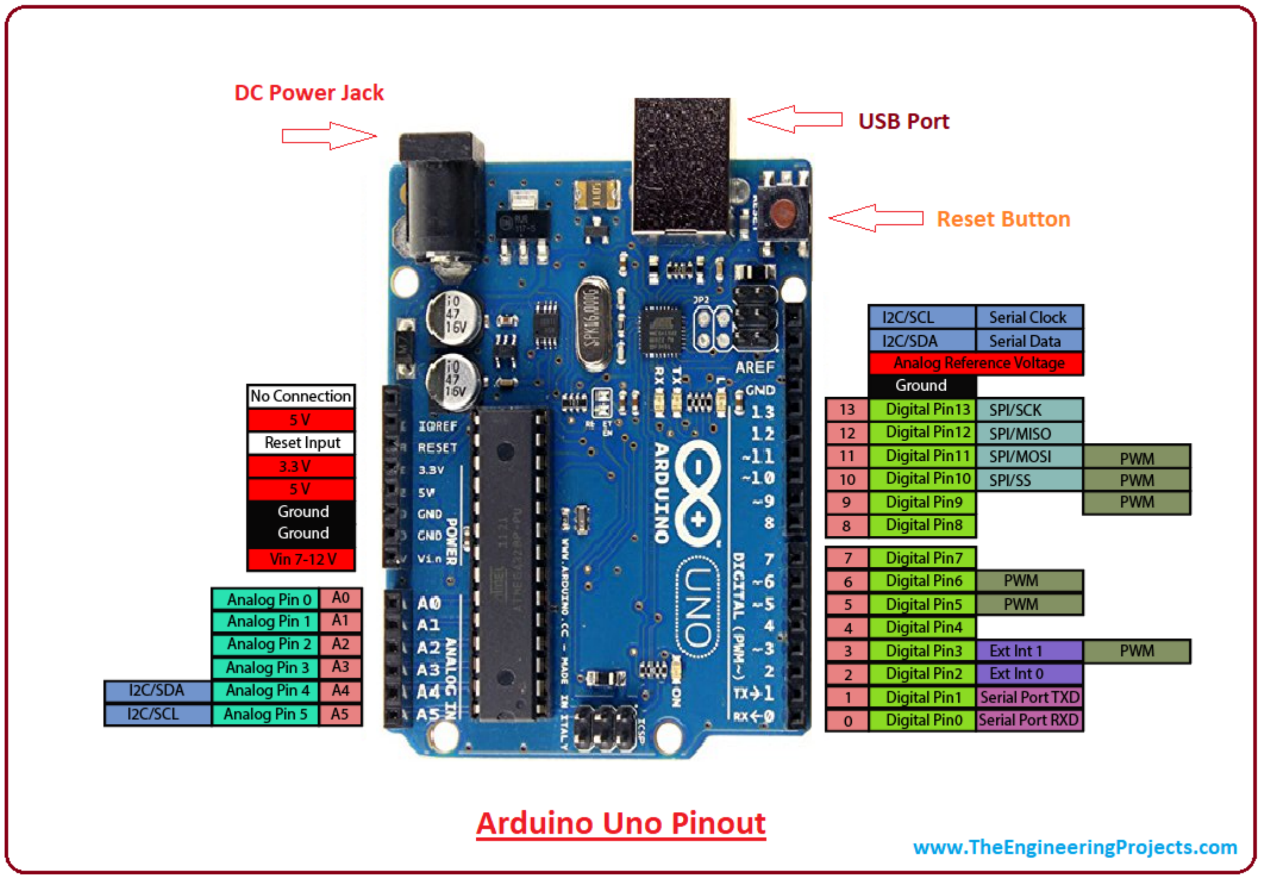


Figure 114: The Arduino UNO pinout. Source `https://www.theengineeringprojects.com`

The main programming software dedicated to the Arduino Board is the Arduino integrated development environment (Arduino IDE). The Arduino IDE supports the C++ language but the code must be written according to a specific scheme. Additionally, the Arduino boards can be programmed using block-based programming languages as Snap4Arduino (dedicated for child) or graphical programming languages as LabVIEW (LIFA and LINX libraries).

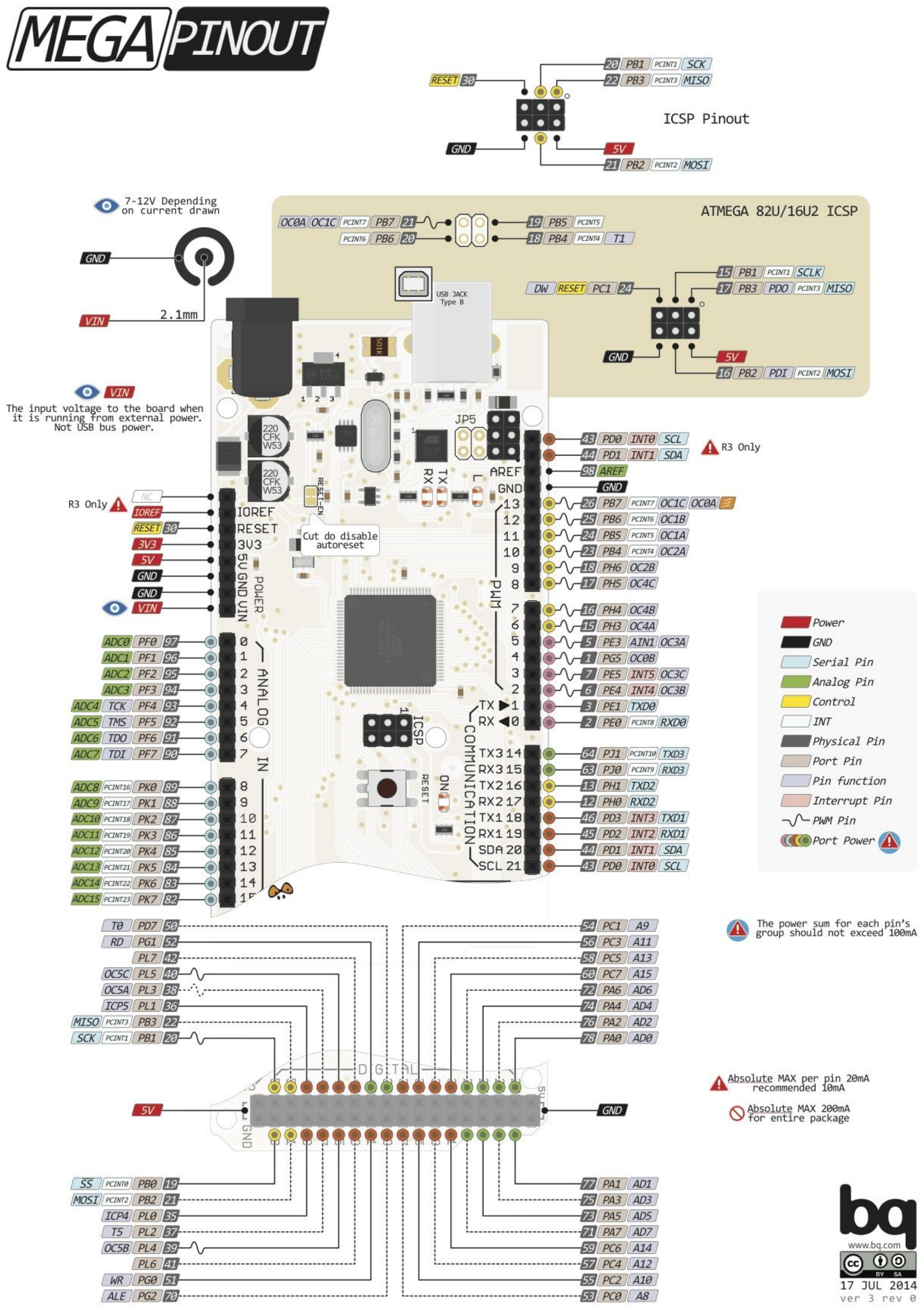
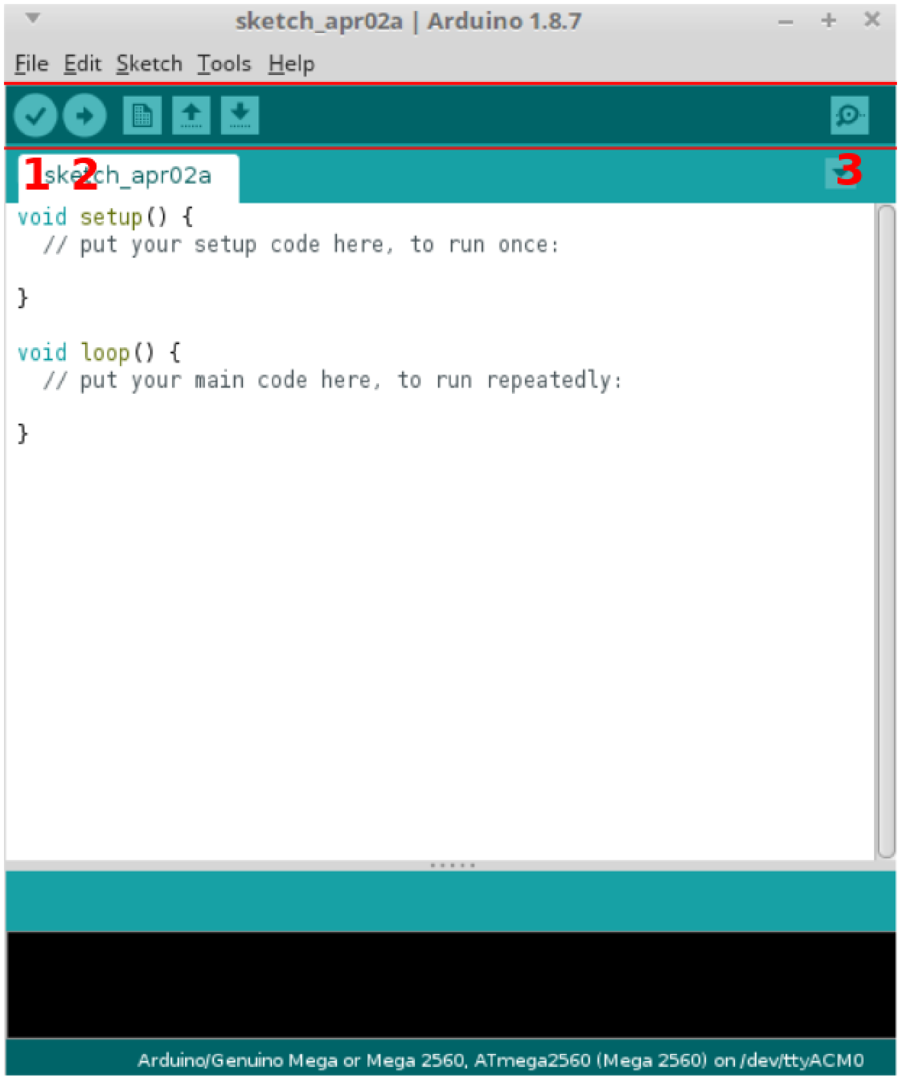

MEGA PINOUT
ICSP Pinout
ATMEGA 82U/16U2 ICSP
7-12V Depending on current drawn
2.1mm
The input voltage to the board when it is running from external power. Not USB bus power.
R3 Only
Cut do disable autoreset
Power
GND
Serial Pin
Analog Pin
Control
INT
Physical Pin
Port Pin
Pin function
Interrupt Pin
PWM Pin
Port Power
The power sum for each pin's group should not exceed 100mA
Absolute MAX per pin 20mA recommended 10mA
Absolute MAX 200mA for entire package
www.bq.com
17 JUL 2014
ver 3 rev 0

Figure 115: The Arduino Mega 2560 pinout. Source https://www.pinterest.com/

Arduino IDE

The Arduino integrated development environment (Arduino IDE) is dedicated software for Arduino boards, which can be download from page https://www.arduino.cc. The Arduino IDE window is shown in figure 116. After running the software, open the *Tools* section and choose board and port from the list.



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Arduino/Genuino Mega or Mega 2560, ATmega2560 (Mega 2560) on /dev/ttyACM0

Figure 116: The Arduino IDE window, where 1 - button to verify the code, 2 - button to upload the code to Arduino board and 3 - button to open the Serial Monitor.

All the programs always consist of two basic functions:

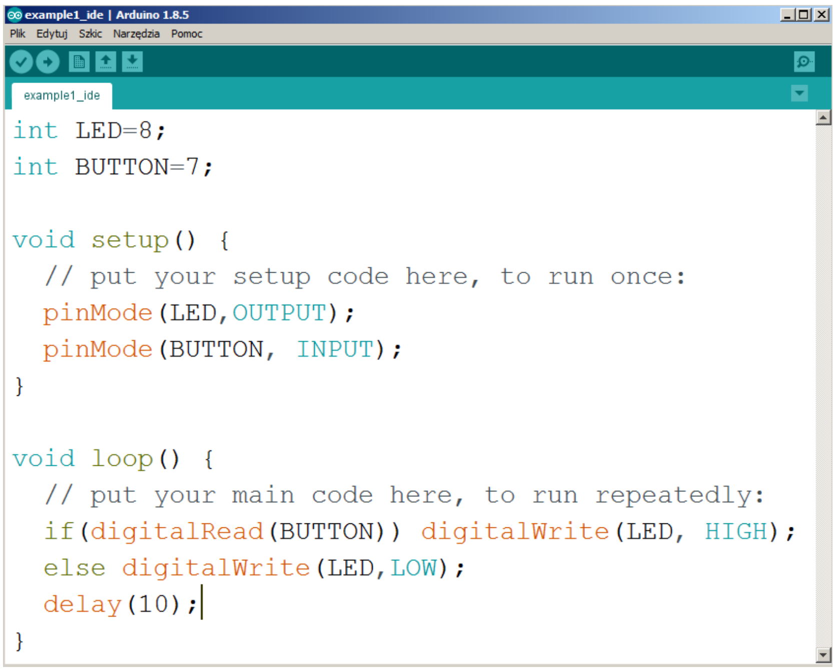
- *setup* - run only ones at the beginning of the program.
- *loop* - run constantly. All instructions put here will be repeated many times.

Users can extend programs with own functions but these two are mandatory.

The Arduino IDE allows to run examples, which are helpful to understand how the libraries work. Such example can be opened after clicking on *File→Examples* and then the user will see the list of libraries and examples for them.

Arduino Libraries

Standard library in Arduino IDE consists functions to deal with digital and analog pins and to generate Pulse Width Modulation (PWM) signals. In figure 117 there is a connection scheme of setup consisting of LED diode connected to pin #8 and button connected to pin #7. A capacitor is connected to button to compensate for the effect of contact jitter.

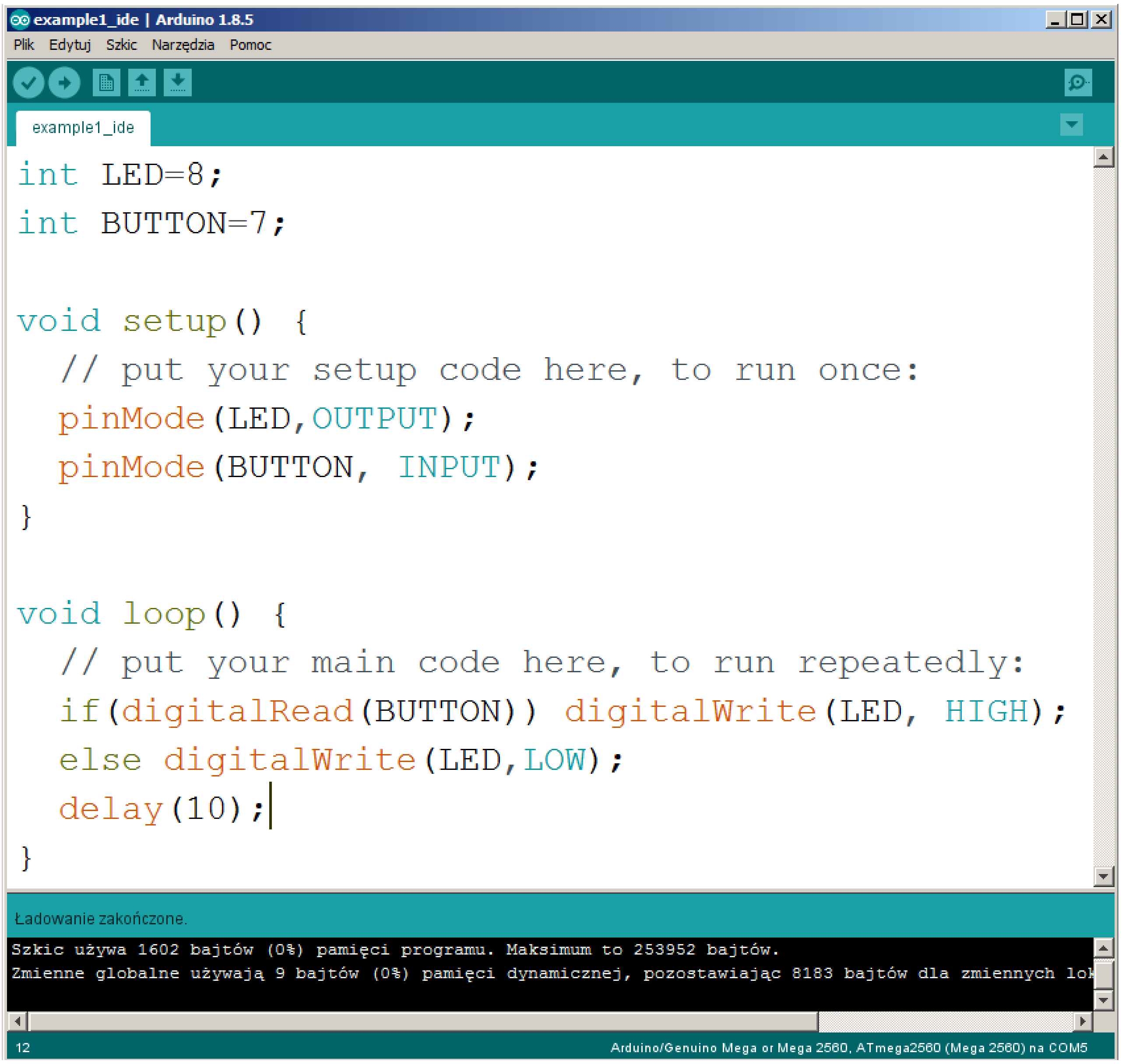

example1_ide | Arduino 1.8.5
Plik Edytuj Szkic Narzędzia Pomoc
example1_ide

```
int LED=8;
int BUTTON=7;

void setup() {
  // put your setup code here, to run once:
  pinMode(LED,OUTPUT);
  pinMode(BUTTON, INPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
  if(digitalRead(BUTTON)) digitalWrite(LED, HIGH);
  else digitalWrite(LED,LOW);
  delay(10);
}
```

Figure 117: Scheme of setup for example in figure 118.

In figure 118 there is a program, that should lighten up LED diode in the moment of button press. The program begins from creation two integer variables named *LED* and *BUTTON*, which stores the pin numbers. Next in function *setup*, a ***pinMode(pin number, OUTPUT/INPUT)*** function is called, which sets up the pin for input or output. From button, a value is read, so the pin mode is set as input. While pin connected to LED diode will send *HIGH* Boolean value (LED on) or *LOW* (LED off), so the pin mode is set as output.

Next in function *loop* a value is read from button thanks to ***digitalRead(pin number)*** function. If read value is *HIGH* (button is pressed), then a *HIGH* signal is sent to LED diode with help of ***digitalWrite(pin***


example1_ide | Arduino 1.8.5
Plik Edytuj Szkic Narzędzia Pomoc
example1_ide

```
int LED=8;
int BUTTON=7;

void setup() {
  // put your setup code here, to run once:
  pinMode(LED,OUTPUT);
  pinMode(BUTTON, INPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
  if(digitalRead(BUTTON)) digitalWrite(LED, HIGH);
  else digitalWrite(LED,LOW);
  delay(10);
}
```

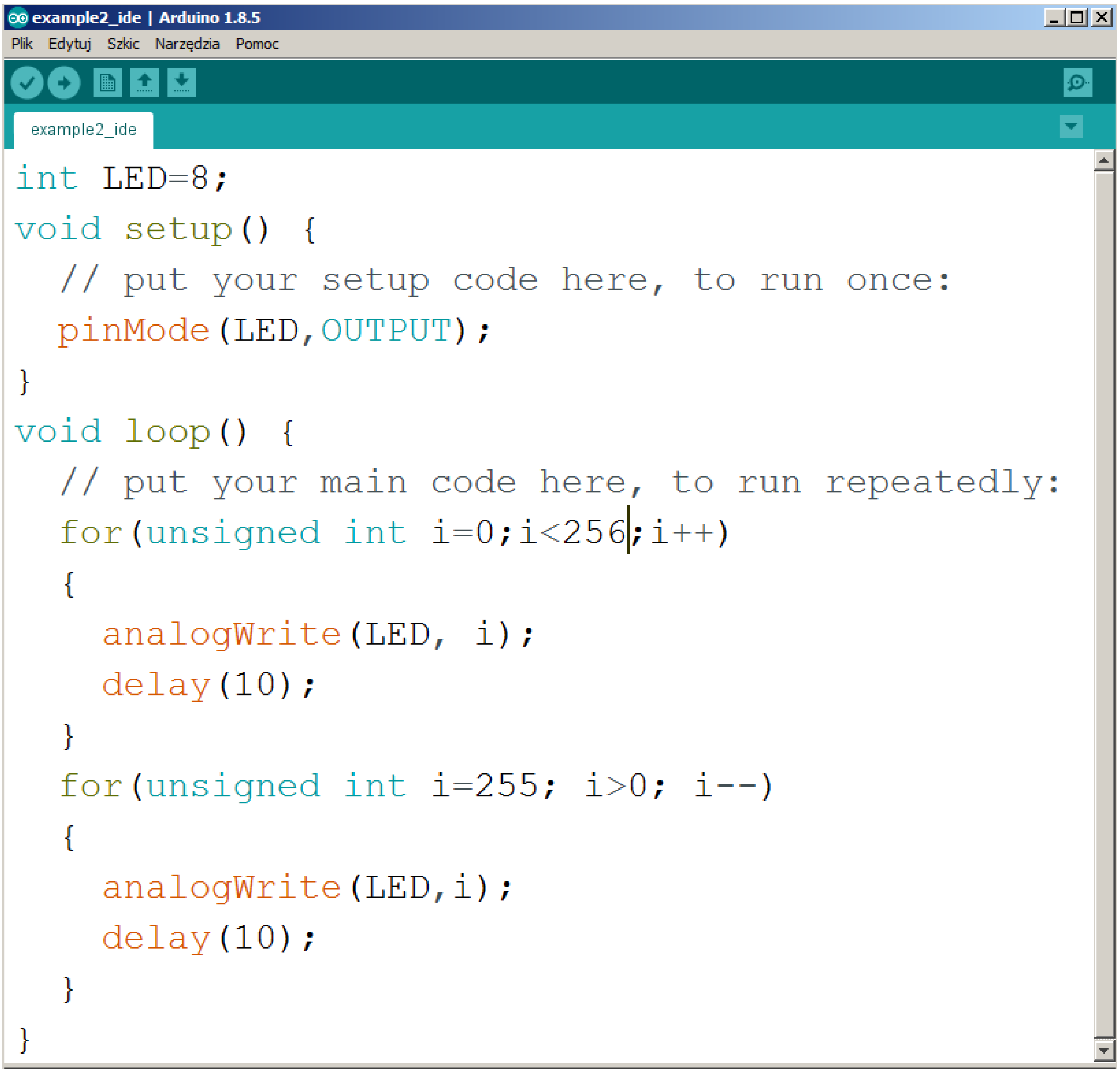
```
Ładowanie zakończone.
Szkic używa 1602 bajtów (0%) pamięci programu. Maksimum to 253952 bajtów.
Zmienne globalne używają 9 bajtów (0%) pamięci dynamicznej, pozostawiając 8183 bajtów dla zmiennych lok
```

```
12                Arduino/Genuino Mega or Mega 2560, ATmega2560 (Mega 2560) na COM5
```

Figure 118: Program which turns LED on when button is pressed, prepared in Arduino IDE environment.

number, HIGH/LOW) function. Otherwise on this pin a *LOW* signal is sent. At the end, small delay (10 ms) is added with help of *delay(time in ms)* function.

Digital pins marked as PWM (*Pulse Width Modulation*) give possibility of digital signal generation with different filling. Function to generate PWM signal is ***analogWrite(pin number, filling)***. Wherein filling accepts value from [0;255] range. Value 255 means that PWM signal has 100% filling. From scheme **??** a button was removed.


example2_ide | Arduino 1.8.5
Plik Edytuj Szkic Narzędzia Pomoc
example2_ide

```
int LED=8;
void setup() {
  // put your setup code here, to run once:
  pinMode(LED,OUTPUT);
}
void loop() {
  // put your main code here, to run repeatedly:
  for(unsigned int i=0;i<256;i++)
  {
    analogWrite(LED, i);
    delay(10);
  }
  for(unsigned int i=255; i>0; i--)
  {
    analogWrite(LED,i);
    delay(10);
  }
}
```

Figure 119: Program which changes filling of PWM signal connected to LED diode in Arduino IDE.

Next a program presented in figure 119 was written, which gradually changes PWM filling from 0 to 255 on pin connected to LED diode. After reaching maximum filling, program gradually decreases filling from 255 to 0. Human eye is not capable to distinguishing *LOW* and *HIGH* states if they are changing fast enough, so when filling is increasing it seems that LED diode is shining stronger, and when filling is decreasing, it seems that LED diode is shining less. In reality we don't regulate light intensity, but we change the time of *HIGH* state - so the time, when LED shines. Program starts by creating integer variable, which stores LED pin number. Next in function *setup*, a *pinMode* function is called, which sets #8 pin as output. In *loop* function, a for loop is created, where *i* variable changes by 1 in the range [0;255]. Value of *i* variable is passed as filling to *analogWrite* function. In this manner PWM signal is generated, which increases by 1 every 10 ms until 255 value is reached. Next for loop is decreasing *i* value by 1 in every iteration until zero value is reached.

The last signal type, which can be read by Arduino board is analog signal. To demonstrate reading analog value, a system was assembled, presented in figure 120.

In figure 121 there is a program, which reads analog signal (voltage value) provided by LM35 temperature sensor. On the basis of voltage read, a temperature in Celsius centigrade is calculated. Value of temperature in Celsius centigrade is passed to serial port monitor. On the beginning, a variable is created, which stores analog port number used for LM35. Next in *setup* function, serial port is initialized with help of ***Serial(baud rate)*** function. In *loop* function analog value is read with help of ***analogRead(pin number)*** function. *analogRead* function returns value from range [0;1023] (as the AD converter in microcontroller on Arduino board has 10-bit resolution) and maximal voltage on standard Arduino setup is 5 V. The value 1023 means the 5 V. So to get the value of voltage we need to multiply the read value by 5 and divide by 1023. [1] Voltage calculation is performed in line 13 of source code. Next obtained voltage value is multiplied by 100 (according to LM35 manual) to get the temperature value. Temperature value is displayed in serial monitor with use of ***Serial.println(value)*** function. Results are presented in figure 122. To see temperature changes it is convenient to use serial plotter instead serial monitor. Such results are visible in figure 123.

[1] correctly we should divide by 1024 - this is stated in reference manual for AVR microcontrollers. Value 1023 is referred to Vref - 1 LSB, not the Vref

Arduino IDE is the most popular environment to program Arduino boards. Many examples and libraries ready for multiple sensors can be found in Internet. Convenient website for library searching is `https://www.arduinolibraries.info`. To demonstrate how to use ready library, an ultrasonic range sensor was connected to Arduino board according to figure 124.

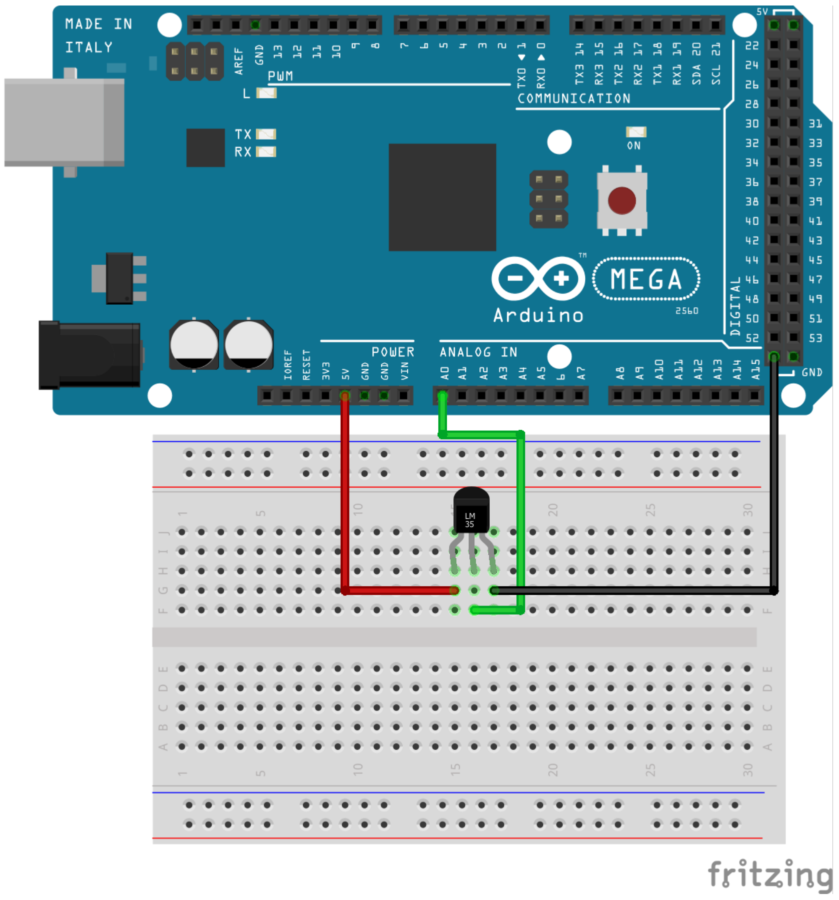
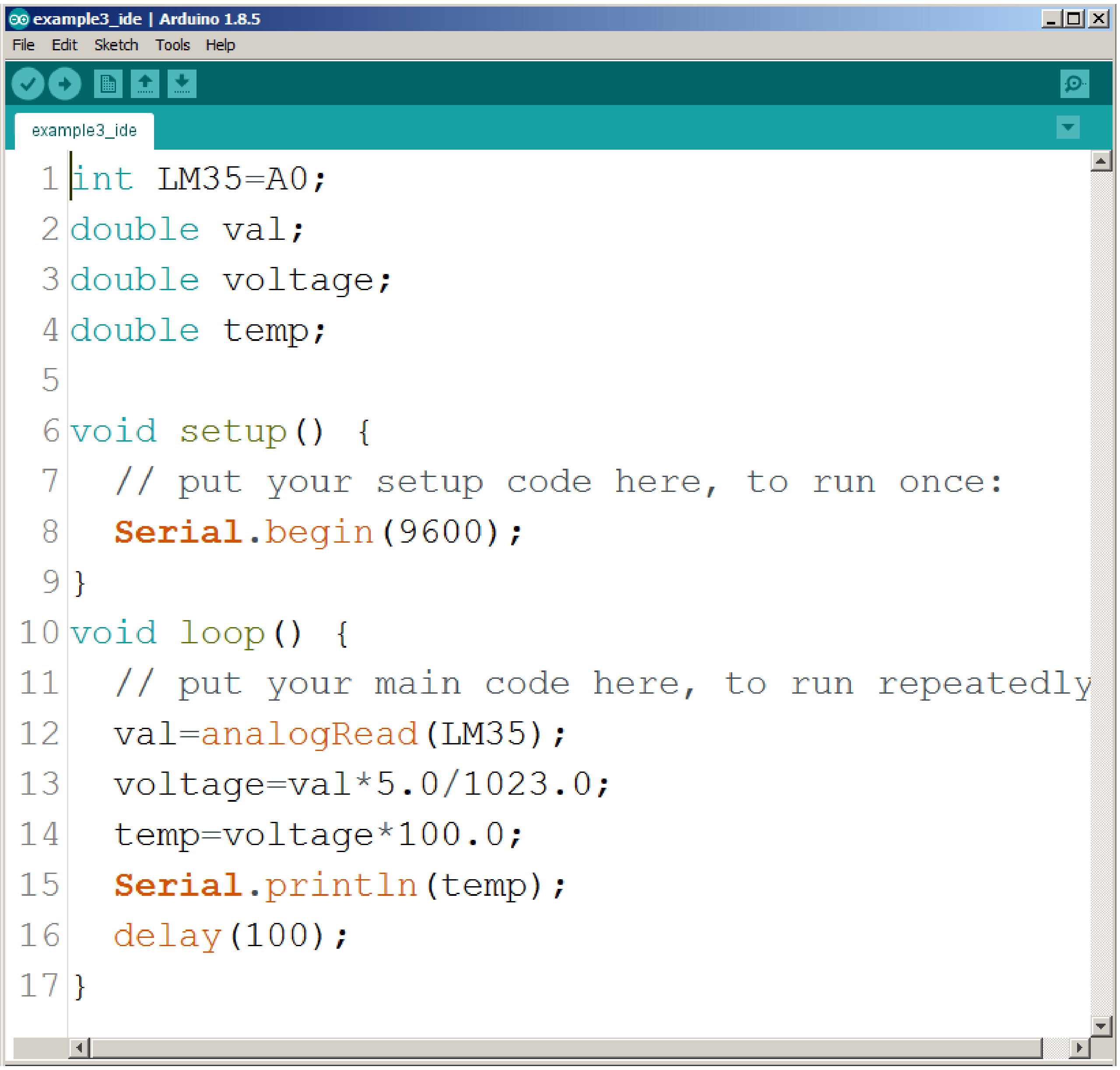
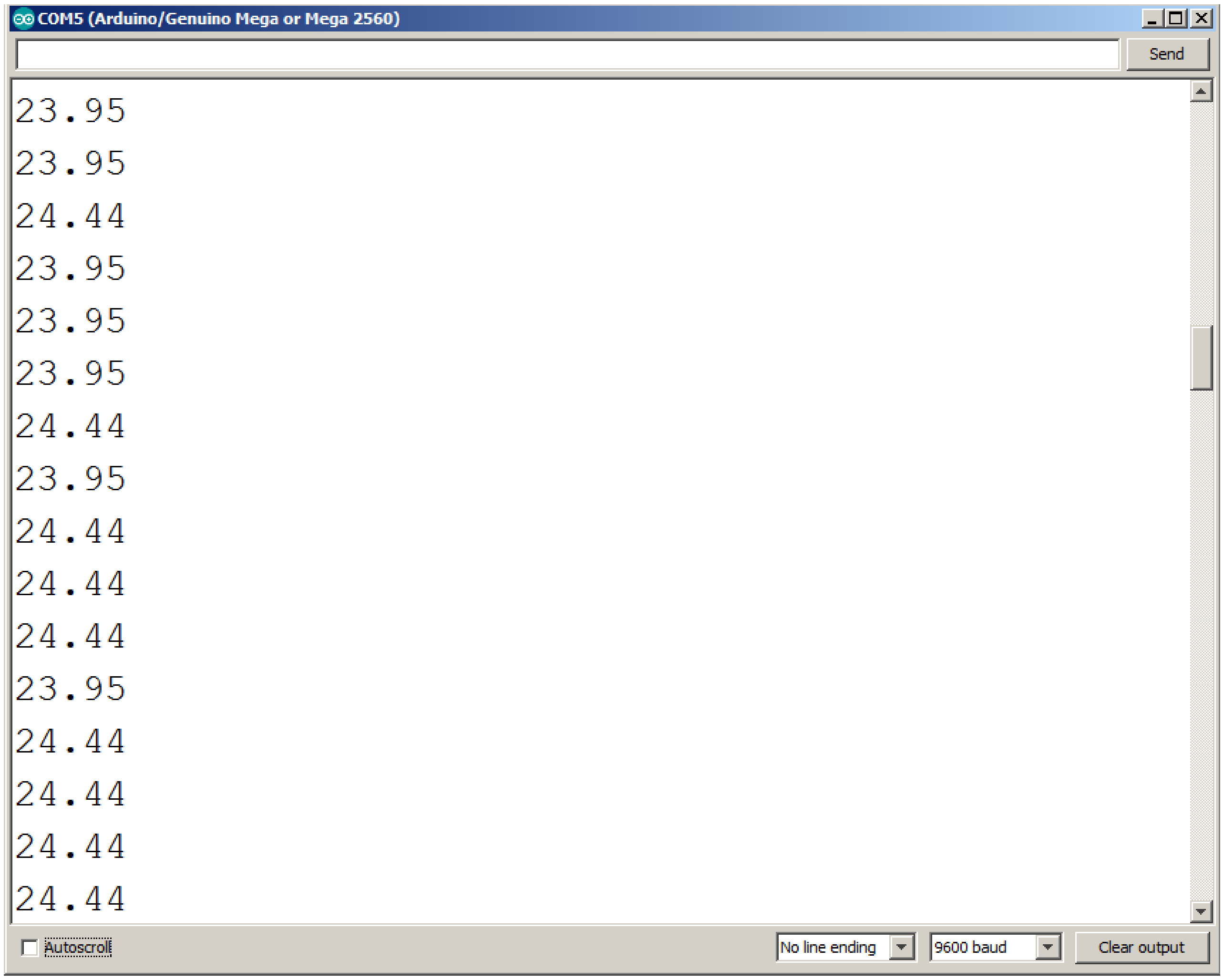


Figure 120: A scheme for setup for example in figure 121.



```
int LM35=A0;
double val;
double voltage;
double temp;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
}
void loop() {
  // put your main code here, to run repeatedly
  val=analogRead(LM35);
  voltage=val*5.0/1023.0;
  temp=voltage*100.0;
  Serial.println(temp);
  delay(100);
}
```

Figure 121: Program, that reads temperature from LM35 sensor, prepared in Arudino IDE.



```
23.95
23.95
24.44
23.95
23.95
23.95
24.44
23.95
24.44
24.44
24.44
23.95
24.44
24.44
24.44
24.44
```

Figure 122: Result of program from figure 121 in serial monitor.

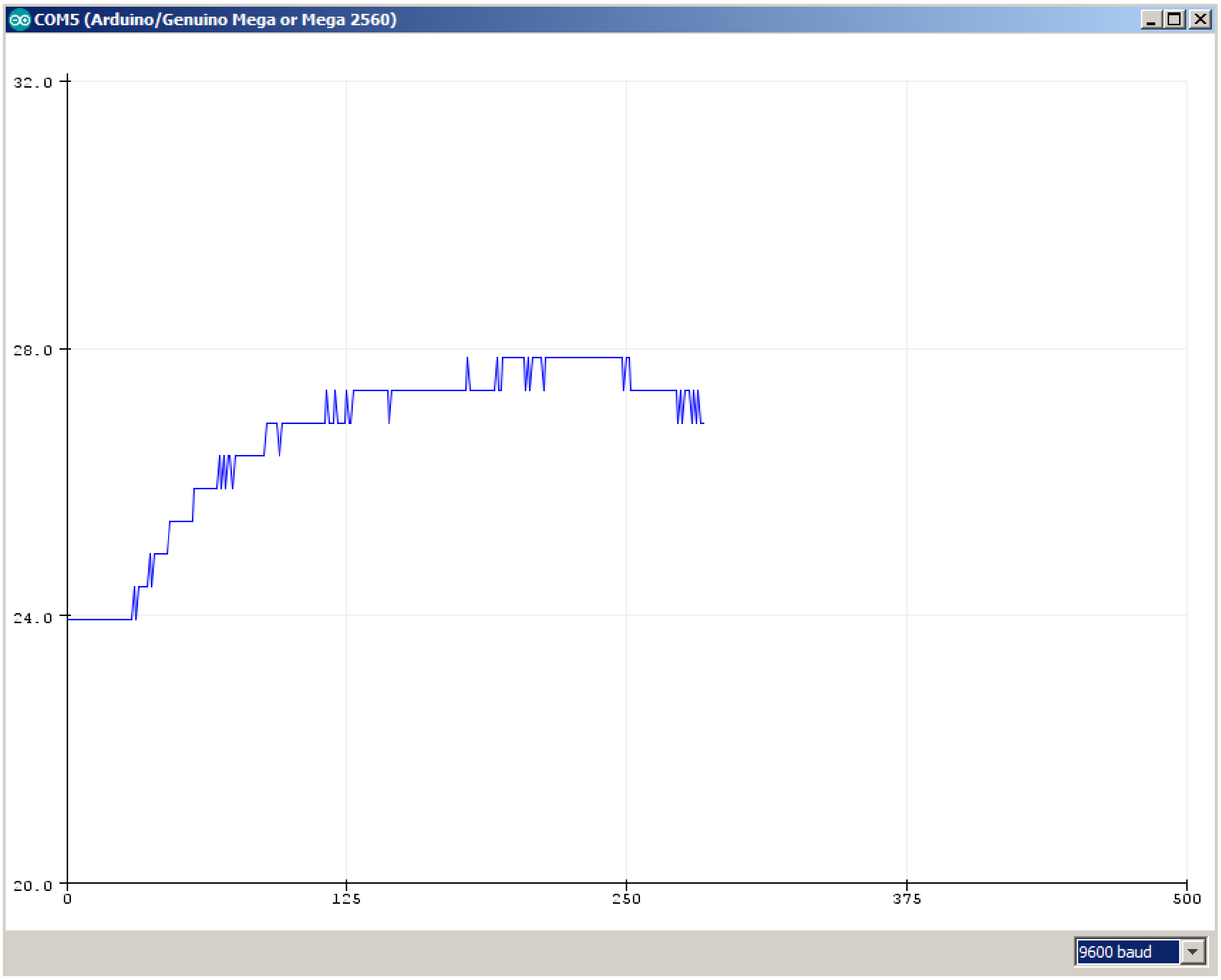


Figure 123: Result of program from figure 121 in serial plotter.

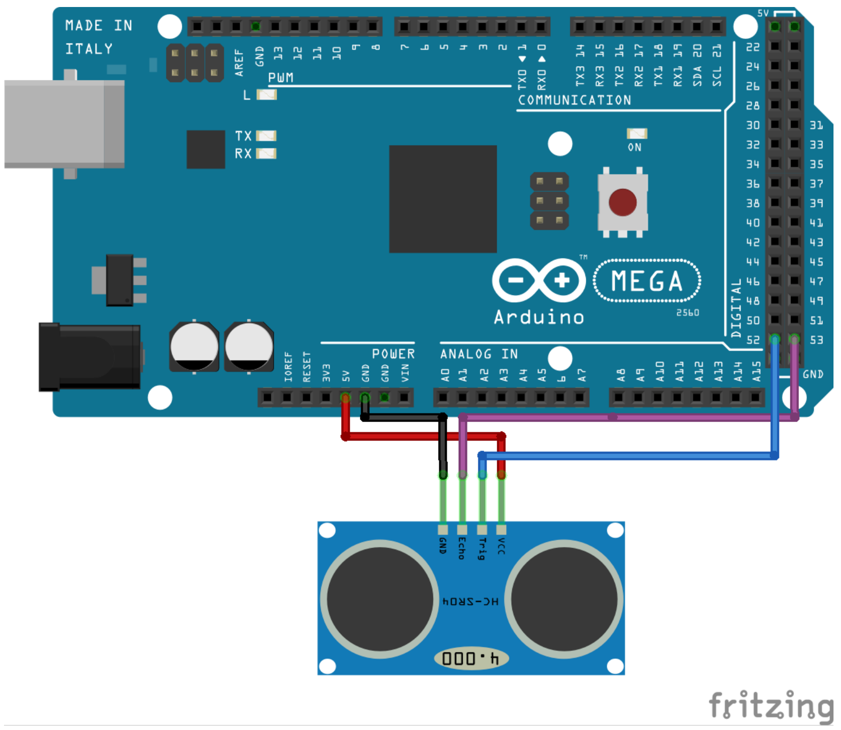


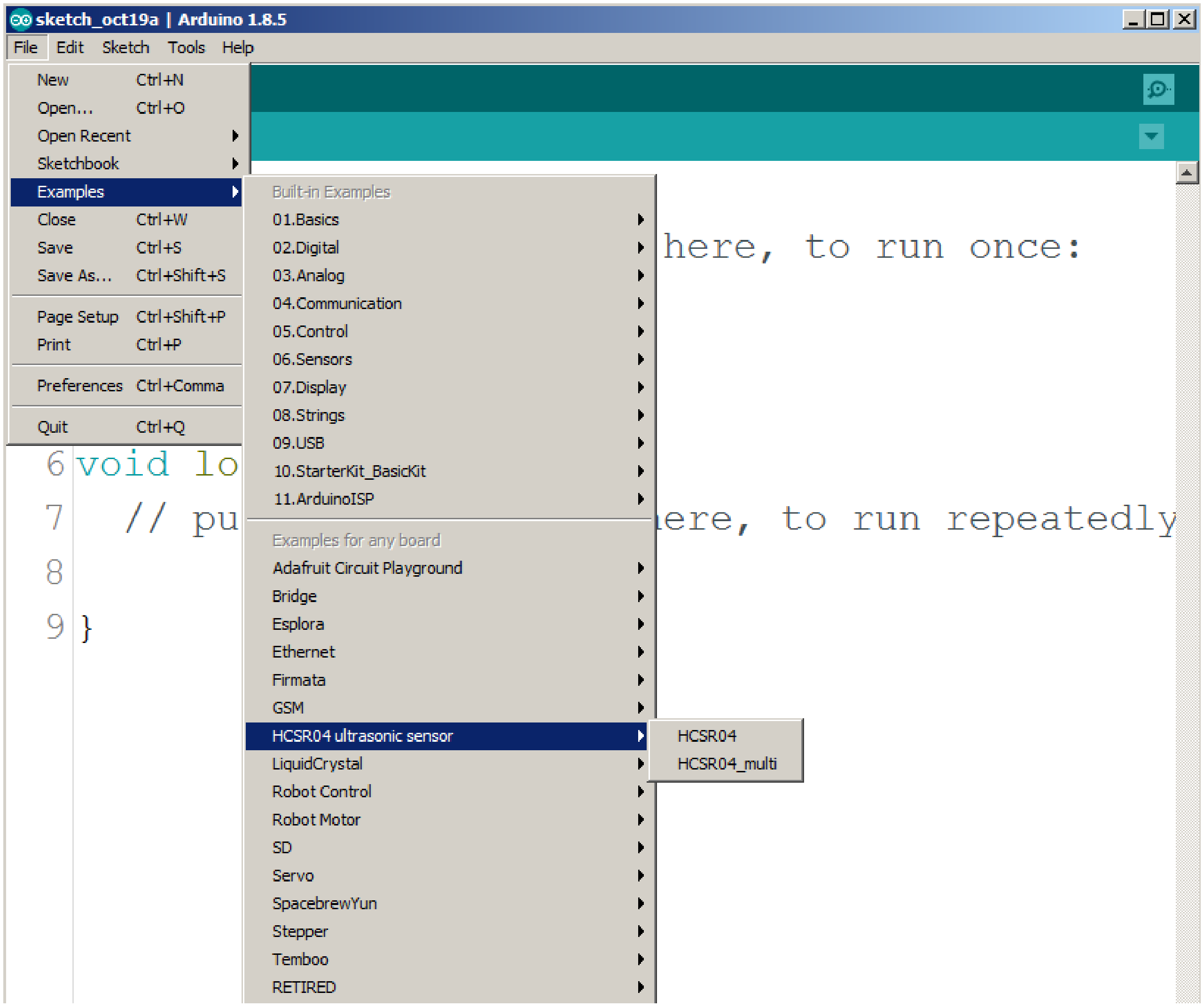
Figure 124: Scheme of setup to example in figure 126.

From `https://www.arduinolibraries.info/libraries/hcsr04-ultrasonic-sensor` website a library for ultrasonic range sensor support was downloaded and unpacked in folder *Arduino/libraries*. Next Arduino IDE environment was restarted. After choosing *File→Examples* one could find ready examples which is shown in figure 125. From the list a *HCSR04.ino* was chosen. Next pin numbers for "trigger" and "echo" in hc object were updated, and example was programmed (in figure 126).

Results from running *HCSR04.ino* example are presented in figure 127. As it is clearly seen, using external libraries is easy. One needs to find a library first, than upload it to *Arduino/libraries* and then check if example code works.

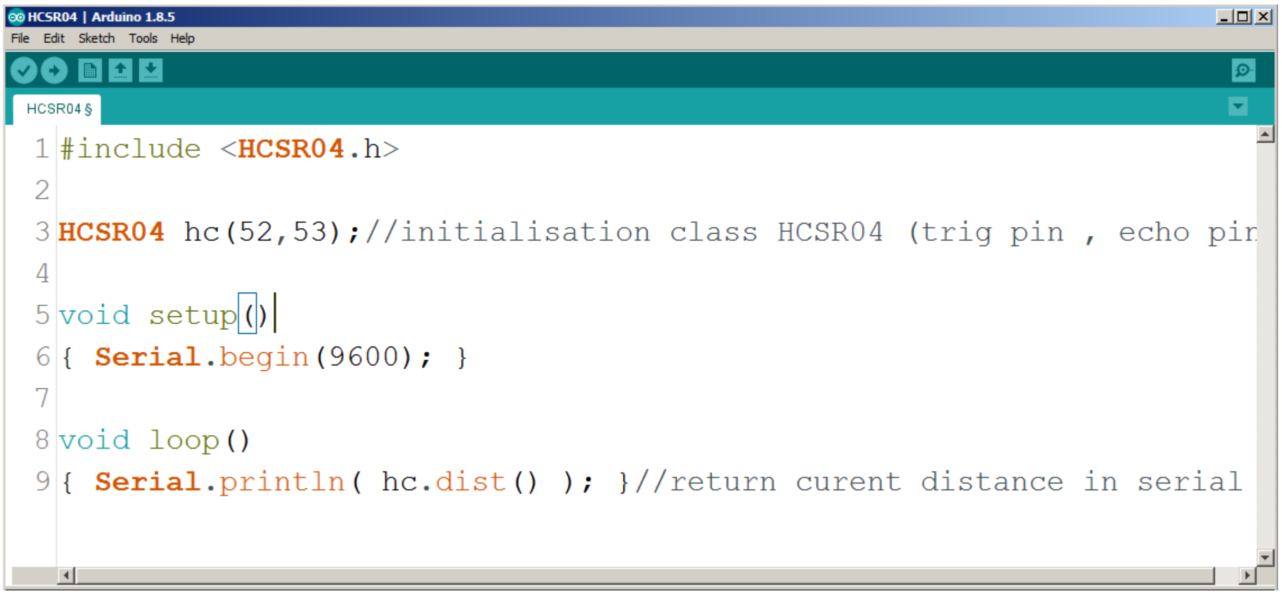
Besides `https://www.arduinolibraries.info` website, many example libraries can be found on `https://github.com`. For example let us use KA-Nucleo-Weather shield (presented in figure 147), which contains a LPS331 sensor. Library for pressure sensor is available on GitHub: `https://github.com/pololu/lps331-arduino`). After downloading library, it should be unpacked to *Arduino/libraries* directory. Then *SerialMetric.ino* example was run, which is shown in figure 128.

Results from *SerialMetric.ino* example are shown in figure 129. In Arduino IDE there are plenty of possibilities to quickly and easily program readout from sensors. Thanks to a wide range of libraries, available on-line, there is no need to write low-level code.



```
here, to run once:
void lo
  // pu         ere, to run repeatedly
}
```

Figure 125: Available examples in installed library for ultrasonic range sensor support.



```
#include <HCSR04.h>

HCSR04 hc(52,53);//initialisation class HCSR04 (trig pin , echo pin

void setup()
{ Serial.begin(9600); }

void loop()
{ Serial.println( hc.dist() ); }//return curent distance in serial
```

Figure 126: Example program that reads distance from ultrasonic range sensor.

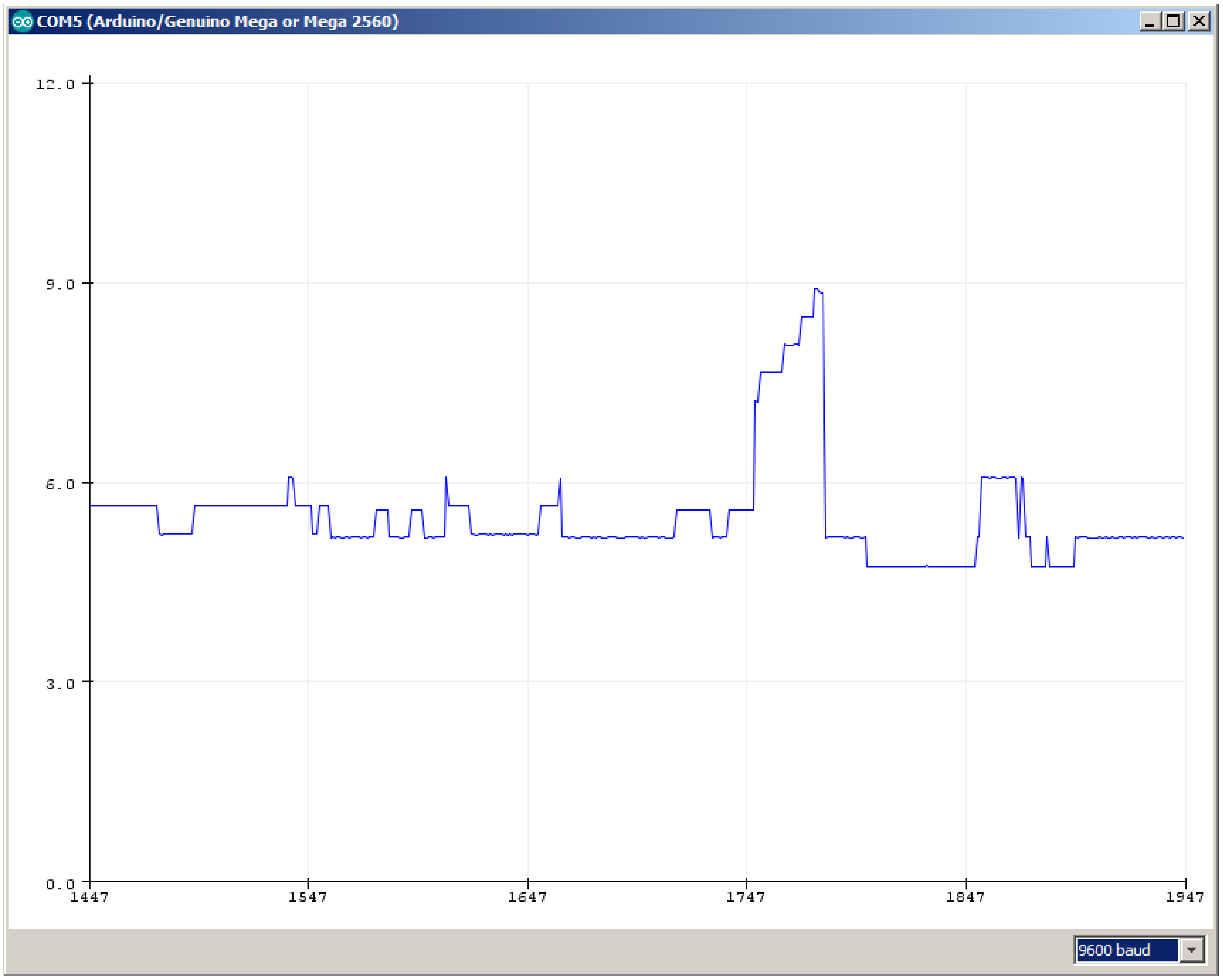
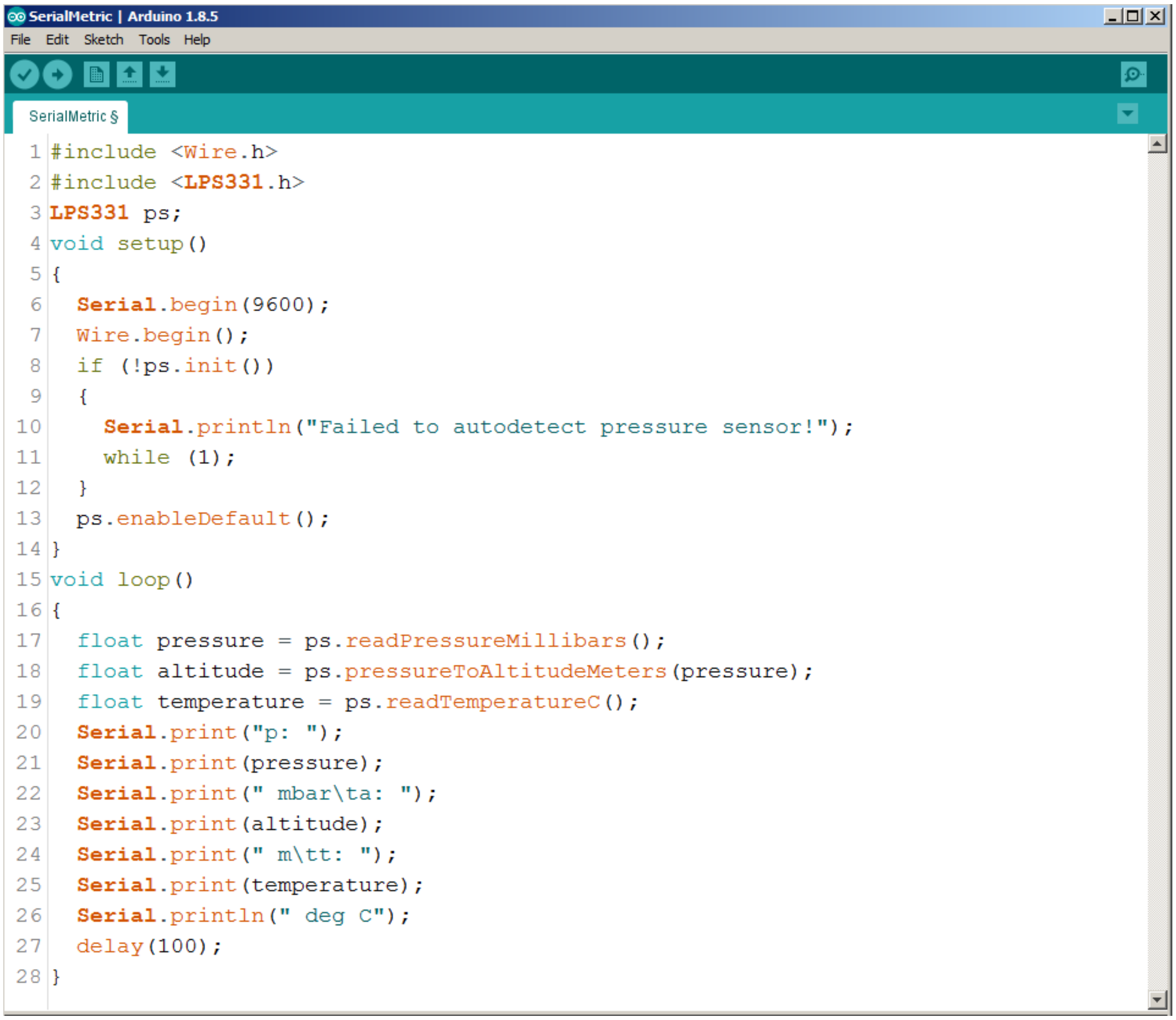
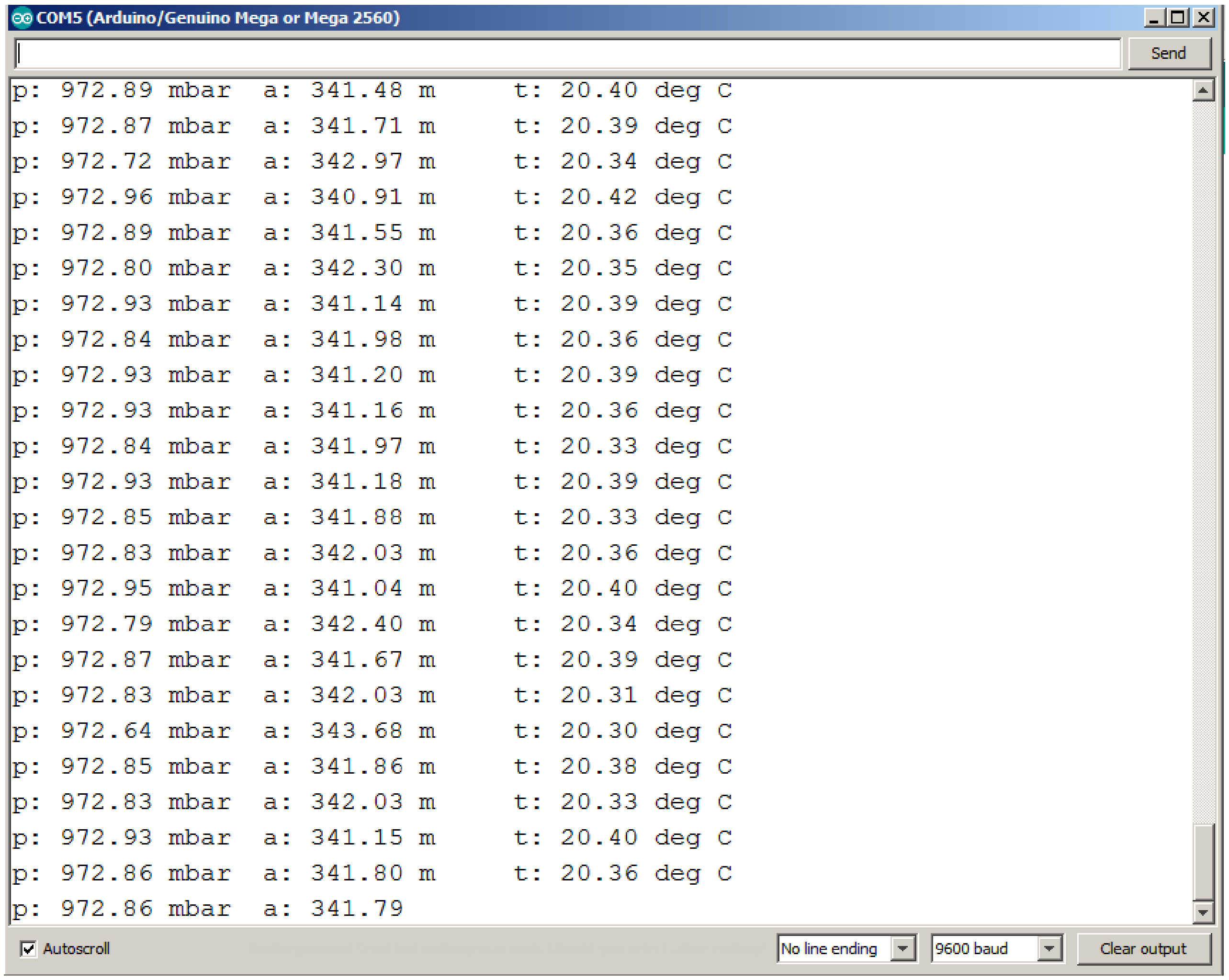


Figure 127: Results of program from figure 126 running presented in serial plotter.



```
#include <Wire.h>
#include <LPS331.h>
LPS331 ps;
void setup()
{
  Serial.begin(9600);
  Wire.begin();
  if (!ps.init())
  {
    Serial.println("Failed to autodetect pressure sensor!");
    while (1);
  }
  ps.enableDefault();
}
void loop()
{
  float pressure = ps.readPressureMillibars();
  float altitude = ps.pressureToAltitudeMeters(pressure);
  float temperature = ps.readTemperatureC();
  Serial.print("p: ");
  Serial.print(pressure);
  Serial.print(" mbar\ta: ");
  Serial.print(altitude);
  Serial.print(" m\tt: ");
  Serial.print(temperature);
  Serial.println(" deg C");
  delay(100);
}
```

Figure 128: Example program, that reads pressure and temperature from LPS331 sensor.



```
p: 972.89 mbar  a: 341.48 m     t: 20.40 deg C
p: 972.87 mbar  a: 341.71 m     t: 20.39 deg C
p: 972.72 mbar  a: 342.97 m     t: 20.34 deg C
p: 972.96 mbar  a: 340.91 m     t: 20.42 deg C
p: 972.89 mbar  a: 341.55 m     t: 20.36 deg C
p: 972.80 mbar  a: 342.30 m     t: 20.35 deg C
p: 972.93 mbar  a: 341.14 m     t: 20.39 deg C
p: 972.84 mbar  a: 341.98 m     t: 20.36 deg C
p: 972.93 mbar  a: 341.20 m     t: 20.39 deg C
p: 972.93 mbar  a: 341.16 m     t: 20.36 deg C
p: 972.84 mbar  a: 341.97 m     t: 20.33 deg C
p: 972.93 mbar  a: 341.18 m     t: 20.39 deg C
p: 972.85 mbar  a: 341.88 m     t: 20.33 deg C
p: 972.83 mbar  a: 342.03 m     t: 20.36 deg C
p: 972.95 mbar  a: 341.04 m     t: 20.40 deg C
p: 972.79 mbar  a: 342.40 m     t: 20.34 deg C
p: 972.87 mbar  a: 341.67 m     t: 20.39 deg C
p: 972.83 mbar  a: 342.03 m     t: 20.31 deg C
p: 972.64 mbar  a: 343.68 m     t: 20.30 deg C
p: 972.85 mbar  a: 341.86 m     t: 20.38 deg C
p: 972.83 mbar  a: 342.03 m     t: 20.33 deg C
p: 972.93 mbar  a: 341.15 m     t: 20.40 deg C
p: 972.86 mbar  a: 341.80 m     t: 20.36 deg C
p: 972.86 mbar  a: 341.79
```

Figure 129: Results from program run, which is presented in figure 128, observed in serial monitor

Raspberry PI

Raspberry PI platform is an example of successful introduction of System on Chip (SoC) board to community. Raspberry PI was founded in the UK. Its goal was Single Board Computer (SBC) for children and youth as an aid in learning programming and computer systems. Initially this system was prepared as Python Interpreter (PI) - which remains as part of the name. Python is a textual programming language, normally used as an interpreted language, easy to teach, with simple and clear syntax. Raspberry PI foundation main page is `https://www.raspberrypi.org/`. A great "MagPI" magazine was created next to Raspberry PI. Each issue is available online free on the site `https://magpi.raspberrypi.org`.

In the year 2019, the newest version is Raspberry PI 4, which comes with 3 different sizes of Random Access Memory (RAM): 1 GiB, 2 GiB and 4 GiB, 1.5 GHz, 64bit ARM processor, 2 USB3.0 ports and 2 USB2.0 ports, 2 HDMI outputs to connect 2 monitors. This model is presented in figure 130.

The first Raspberry PI model was introduced in 2012, and it is named Raspberry PI B.

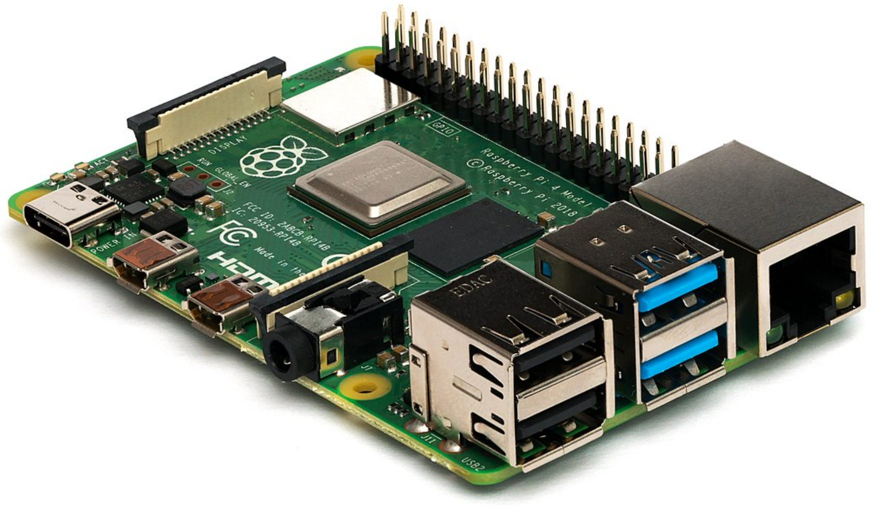


Figure 130: Raspberry Pi 4 Model B from the side. Michael Henzler / Wikimedia Commons / CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)

GPIO Header

SoC means, that most components of computer system are integrated in the single chip. In the case of Raspberry PI, SoC chips are manufactured by Broadcom. In Raspberry PI B, presented in figure 131, Broadcom BCM2835 was used, which is characterized by the ARMv6Z 32-bit instruction set. This was mainly the reason for derived operating system: Raspbian (to make the best use of this ARM architecture). The common graphics processing unit (GPU) is VideoCore IV

Description	Pin no.	Pin no.	Description
VDD_3v3	1	2	VDD_5v
I2C1_SDA	3	4	VDD_5v
I2C1_SCL	5	6	DGND
DIO_7	7	8	UART0_TX
DGND	9	10	UART0_RX
DIO_11	11	12	DIO_12
DIO_13	13	14	DGND
DIO_15	15	16	DIO_16
VDD_3v3	17	18	DIO_18
SPI0_MOSI	19	20	DGND
SPI0_MISO	21	22	DIO_22
SPI0_CLK	23	24	RESERVED_SPI0_CS0
DGND	25	26	RESERVED_SPI0_CS1
RESERVED_I2C0_SDA	27	28	RESERVED_I2C0_SCL
DIO_29	29	30	DGND
DIO_31	31	32	DIO_32
DIO_33	33	34	DGND
DIO_35	35	36	DIO_36
DIO_37	37	38	DIO_38
DGND	39	40	DIO_40

Table 7: GPIO header for Raspberry PI starting from 1B+/A+ is 40 pin. Raspberry PI A/B had only 26 pin connector.

at 400 MHz and for Raspberry PI 4B it is VideoCore VI at 500 MHz.

The family of Raspberry PI boards is an inexpensive solution for use in many projects, where more computing power is needed in contrast to Arduino boards. Because on SBC computers, operating systems are used, this allows new possibilities, like use of many computer languages, like Python, JavaScript, etc. In these boards, we have network Ethernet interface, which allows to prepare applications for Internet of Things (IoT). We can easily configure web servers, file servers, databases and experiment with many network remote control protocols and different client/server solutions. However beware in some versions of boards, there are no network capabilities, instead they offer low energy consumption. On top of that we can run Docker, for easy deployment and maintenance of complex services. Normally on Raspberry PI SBCs system is started from micro SD card (Raspberry PI model B used standard size SD card). New models starting from Raspberry PI 3 can boot also from other storage devices, like pendrives, hard disks or solid state disks (no need for SD cards).

In this book, an example using Raspberry PI 2B is presented, because already prepared solution for interfacing with National Instruments LabVIEW exists already. This board is presented in figure 132.

The use of SBCs has many advantages, but it is not always the best solution. There are also disadvantages, like power consumption is

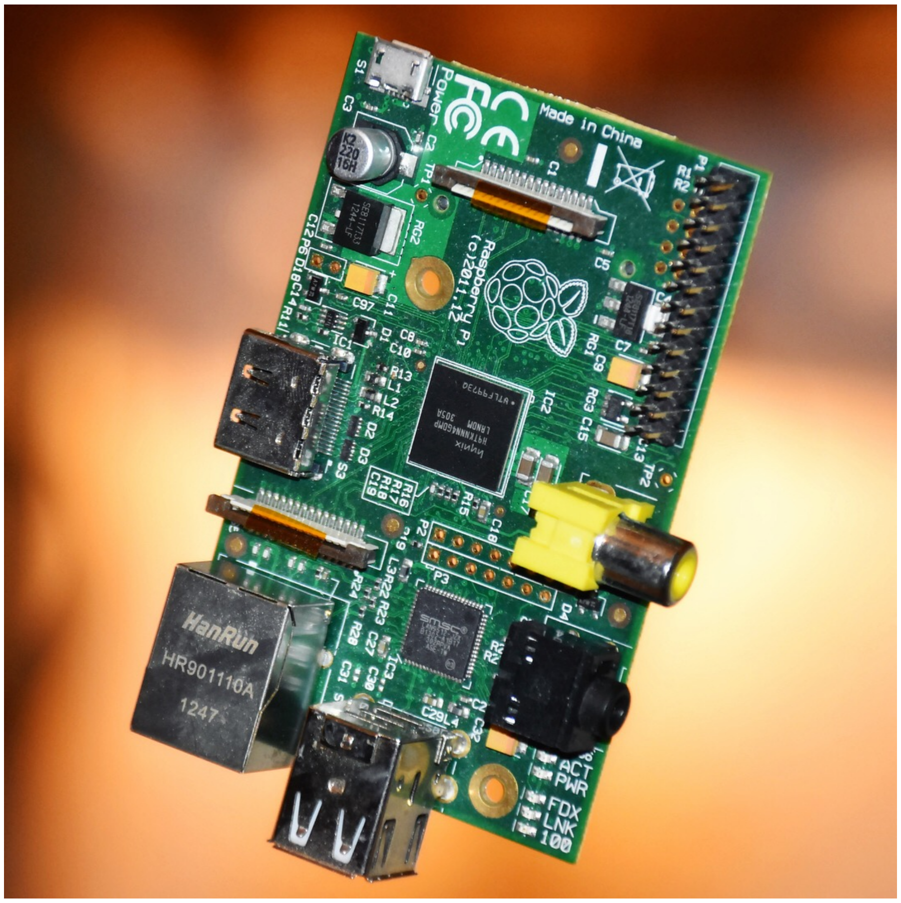


Figure 131: Raspberry PI B, Simon Waldherr / Wikimedia Commons / CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)

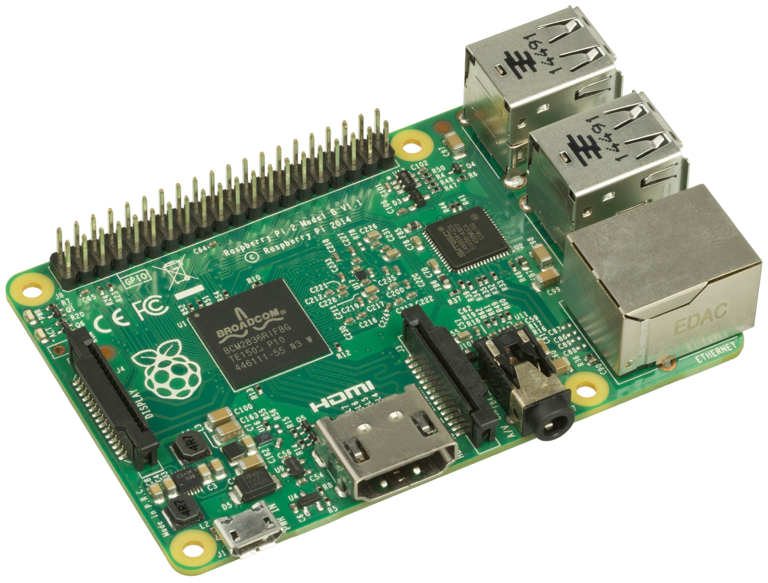


Figure 132: Raspberry Pi 2B, Evan Amos / Wikimedia Commons / Public Domain

clearly higher, then data corruption and system corruption may occur - sometimes user intervention is needed. SD cards may wear out if used improperly, etc.

It is important to choose the right tools for the specific problem. Excessive system complications are not recommended, and in many cases can be harmful for reliability.

GNU/Linux operating system

The core system used on this SBC is Raspbian (`https://www.raspbian.org/`). This is a GNU/Linux distribution, based on Debian GNU/Linux, but specially prepared form ARM processors used on Raspberry PI. Nowadays many different operating system can be used on Raspberry PI platform both GNU/Linux distributions (Raspbian, Ubuntu, Fedora, etc.) and others like RISC OS, Windows IoT Core, RaspBSD, etc. However Raspbian is very well suited for this hardware and it is well-known around the world. Components of GNU/Linux systems like Rasbian are licensed with open source licenses.

Usage of GNU/Linux system is quite easy. However it is recommended for users to get to know the Command Line Interface (CLI), as it allows to operate and configure many tools. The most popular shell is Bourne Again Shell (Bash). You can find many tutorials online, and also there is a freely available book from Raspberry foundation:

https://magpi.raspberrypi.org/books/essentials-bash-vol1.

There are also many other books concerning usage of Raspberry PI. Free books are available on https://magpi.raspberrypi.org/books.

There are many possibilities to interface LabVIEW with Raspberry PI. A server responding to commands and sending data can be used as an example. In LabVIEW there are network communication blocks for tcp and udp sockets. If Ethernet is not possible, then serial port can be used. In this book a ready solution with LINX library will be shown.

Raspberry PI SBCs have general purposes input / output (GPIO) exposed on GPIO pin connector. This can be used with many projects to control many sensors and actuators. First Raspberry PI has 26-pin connector, but starting from Raspberry PI 1B+ 40-pin connector is used. To prepare connections with external devices, a pin-out is used, which is presented in table 7.

LabVIEW Packages

Software distribution in form of packages proved very well on open source systems like GNU/Linux systems. This convenient method has gained recognition, and was adopted in LabVIEW also. Package manager can list available packages, show description of these packages and can help in choosing the right package for specific LabVIEW version.

LabVIEW Package Manager

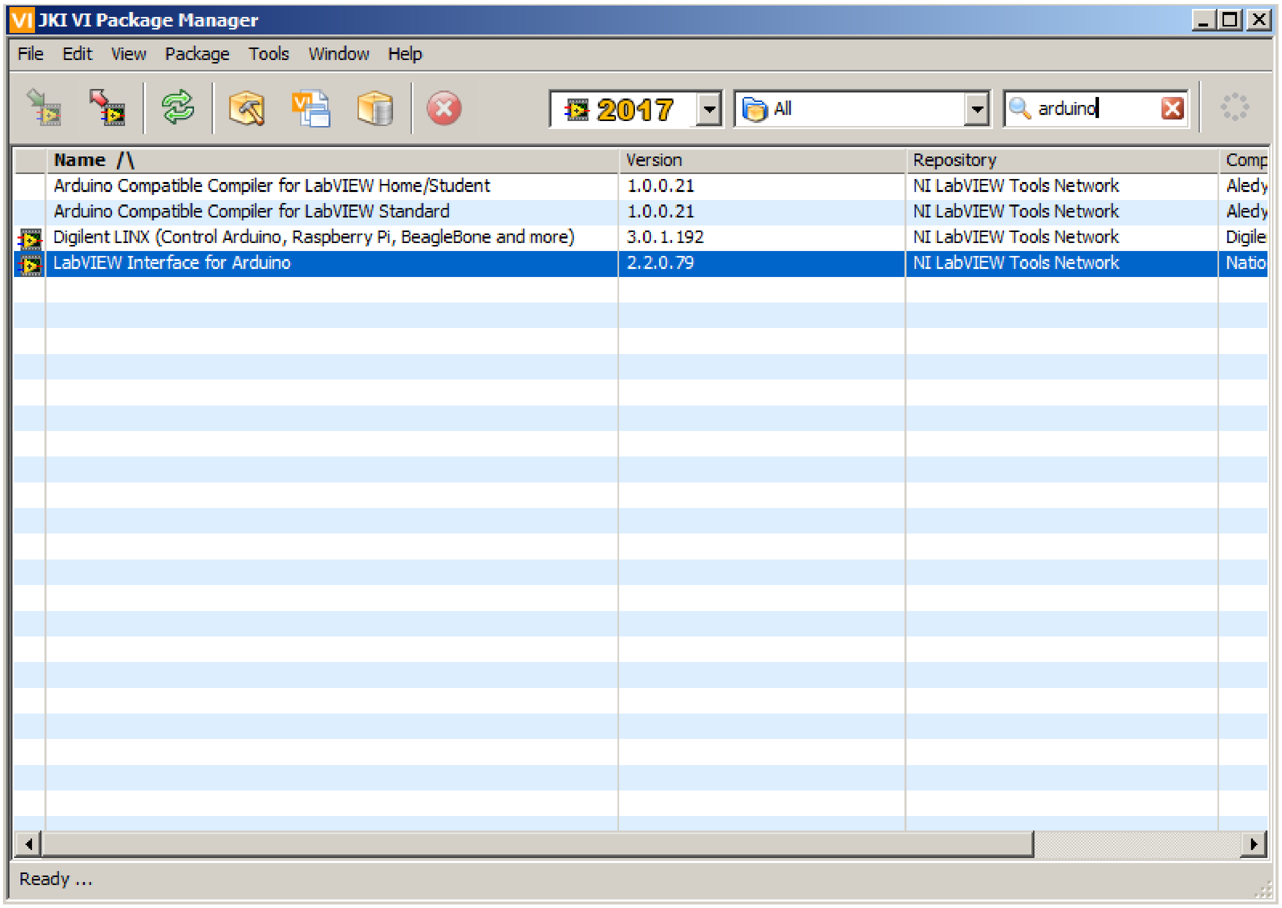
A very useful software which is used to distribute libraries in LabVIEW is *LabVIEW Package Manager* (VIPM). This piece of software allows to search and install selected library for chosen LabVIEW version. After installation of such library, functions are automatically added to *functions palette* of chosen LabVIEW version (useful if different versions of LabVIEW coexist on your PC). The big advantage of VIPM is cooperation with most popular operating systems like Windows, Mac or Linux. Example usage of VIPM for LIFA library installation is presented in figure 133.

LabVIEW Interface for Arduino (LIFA)

Arduino board can be programmed from LabVIEW level with use of libraries like: *LabVIEW Interface for Arduino (LIFA)* or *Digilent LINX*. First of them was prepared by National Instruments in 2009 year. To use LIFA library, one needs to:

1. Install NI VISA (`http://www.ni.com/visa`).
2. Install *LabVIEW Package Manager (VIPM)*.
3. Run VIPM, find LIFA library (presented in 133) and install this library.
4. Install Arduino IDE environment (freely available from `https://www.arduino.cc`).

5. Upload *LIFA_Base.ino* - dedicated sketch to Arduino board using Arduino IDE environment. This sketch is available in: */LabVIEW year/vi.lib/LabVIEW Interface for Arduino/Firmware/LIFA_Base*.



Name /\	Version	Repository	Comp
Arduino Compatible Compiler for LabVIEW Home/Student	1.0.0.21	NI LabVIEW Tools Network	Aledy
Arduino Compatible Compiler for LabVIEW Standard	1.0.0.21	NI LabVIEW Tools Network	Aledy
Digilent LINX (Control Arduino, Raspberry Pi, BeagleBone and more)	3.0.1.192	NI LabVIEW Tools Network	Digile
LabVIEW Interface for Arduino	2.2.0.79	NI LabVIEW Tools Network	Natio

Figure 133: LIFA library installation.

After a sketch was uploaded to Arduino, it is time to start LabVIEW. Functions from LIFA library are now be available in *Functions palette*, in fold named *Arduino* which is shown in figure 134.

LIFA Library usage

Every program using LIFA library, starts with *Init* function, that allows to initialize connection with Arduino board. On the input of the function, one need to provide serial port name, to which Arduino board is connected. The easiest way is to view the port name when sketch is being uploaded in Arduino IDE. Moreover Arduino board type must be provided. Supported types are: *Uno*, *Duemilanove w/ATmega 328*, or *Mega 2560*. recommended baudrate is 9600. With higher speeds, some communication problems with Arduino may occur.

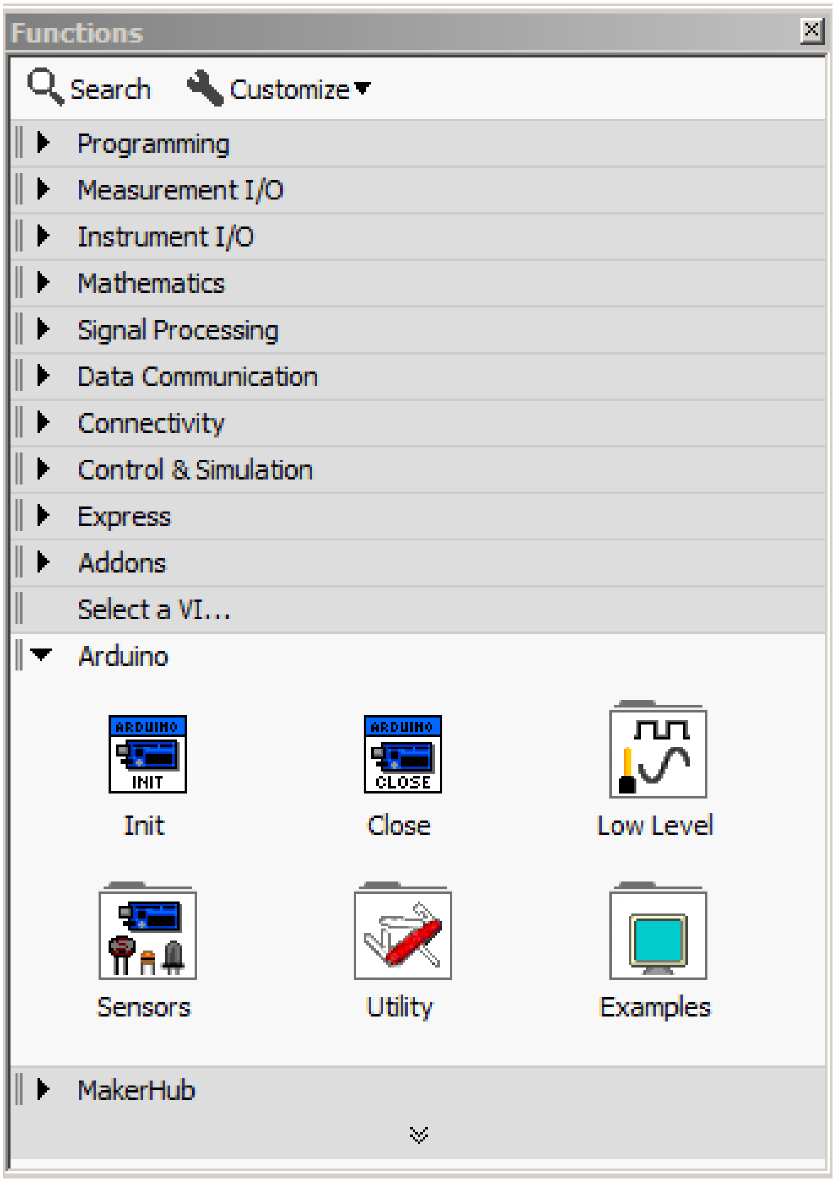


Figure 134: Fold *Arduino* in *Functions palette*.

Functions allowing analog pin, digital pin, PWM operations are placed in *Functions palette* → *Arduino* → *Low level* fold. In figure 136 a program is presented, whose task is turning LED on and off. Scheme of hardware setup is presented in figure 135.

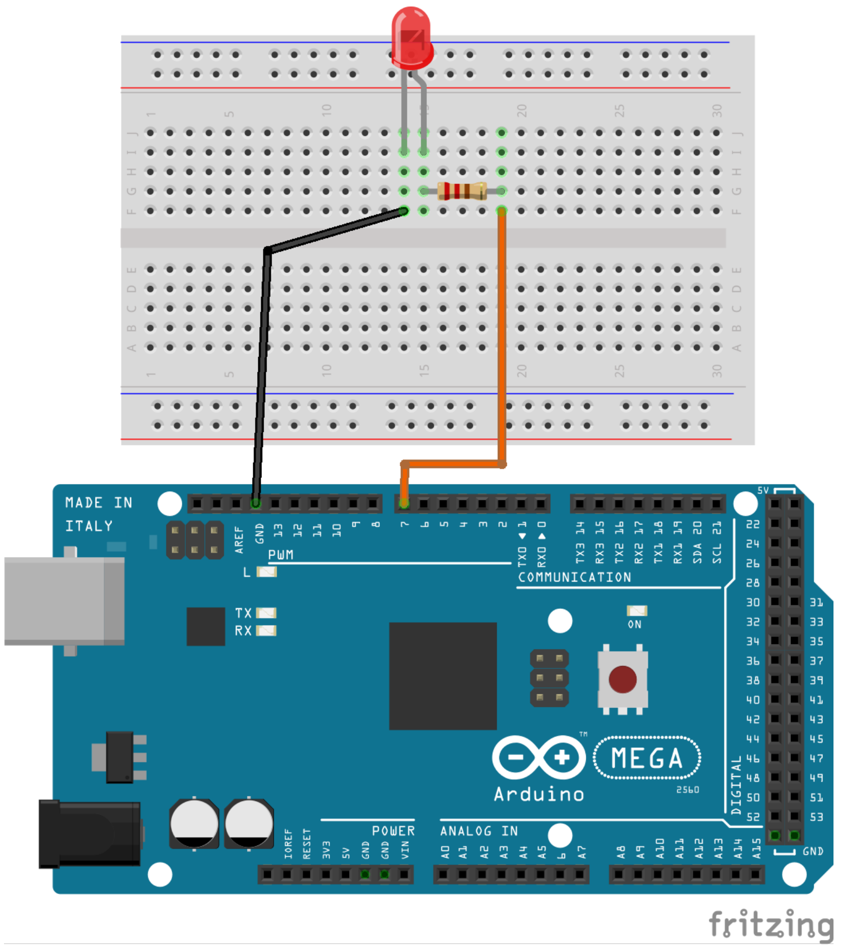


Figure 135: Scheme of hardware setup for program presented in figure 136.

Program begins from initialization of connection with Arduino Mega 2560 board, which is connected to COM5 port with help of *Init* function, which was described above. Next I/O port selected by user is configured as output with help of *Set Digital Pin Mode* function from *Low Level* fold. In while loop a *Digital Write Pin* function is placed, which sets the High (1) or Low (0) level on selected pin. This function accepts numeric value on input, so conversion of logic type to numeric value is used. Outside the while loop, a *Close* function was placed, that ends communication with Arduino board on selected serial port. In moment of pressing *Turn on/off* button, a LED diode connected to Arduino board lights up. After pressing the button again, LED diode turns off.

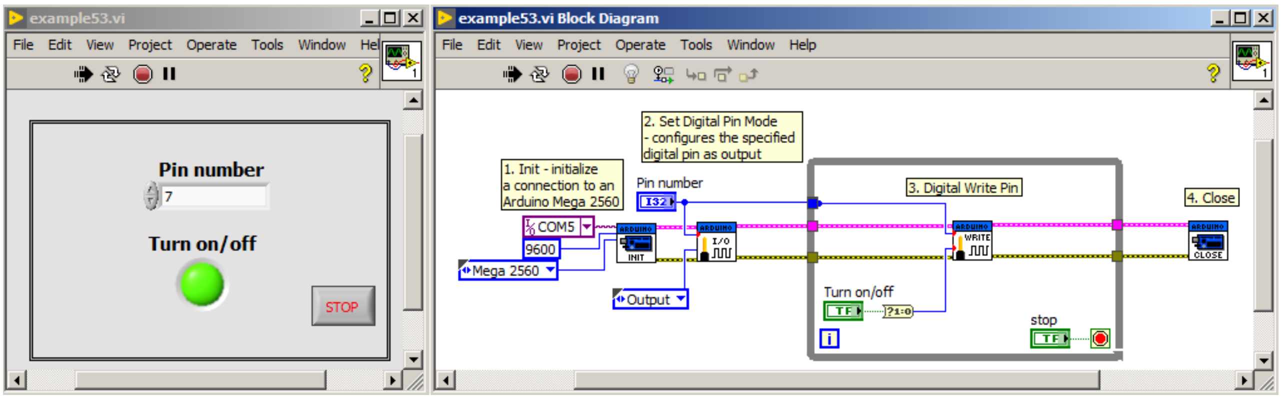

example53.vi
example53.vi Block Diagram
File Edit View Project Operate Tools Window Help
Pin number
7
Turn on/off
STOP
1. Init - initialize a connection to an Arduino Mega 2560
2. Set Digital Pin Mode - configures the specified digital pin as output
3. Digital Write Pin
4. Close
COM5
9600
Mega 2560
Output
stop

Figure 136: Program that turns LED on and off, created with help of LIFA library.

Example described above was modified by connecting PIR motion sensor to pin #6 (presented in figure 137). When PIR sensor detects movements, the LED diode lights up. Therefore pin #6 should be configured as input. We will read its state: 0 (no movement) or 1 (movement detected). Modified example is presented inf figure **??**. Another basic function is used here: *Digital Read Pin*, which returns read value from selected pin. As you can see reading and writing digital values to selected pins come to two activities: pin mode configuration (as input or output) and reading or writing values to/from the pin.

Arduino board allows also reading analog values from pins marked with "A" letter. In figure 140 an example program is presented, in which analog value is read from LM35 sensor. When read temperature is greater than 30^{o}c, the RGB (red-green-blue) LED diode glows red, otherwise LED diode glows green. Scheme of hardware setup is presented in figure 139. Program starts with connection to the Arduino board initialization with help of *Init* function. Next connection of RGB LED should be configured with help of *RGB LED Configure* function. This function returns pin numbers in the form of 1D array. In while loop analog value from temperature sensor is read using *Analog Read Pin* function. According to the LM35 datasheet, read voltage value should be multiplied by 100 to get the temperature value in Celsius centigrade. Calculated temperature is compared with value of 30, and if it is higher, then *Select* function returns red colour code. Otherwise *Select* function returns green color code. Output from *Select* function is connected to *RGB LED Write* function, which sets selected color on RGB LED connected to Arduino board. At the end, *Close* function is called, which ends communication with Arduino board. Summarizing, reading analog values comes down to using the *Analog Read Pin* function.

Many times it is convenient to show measured values on display,

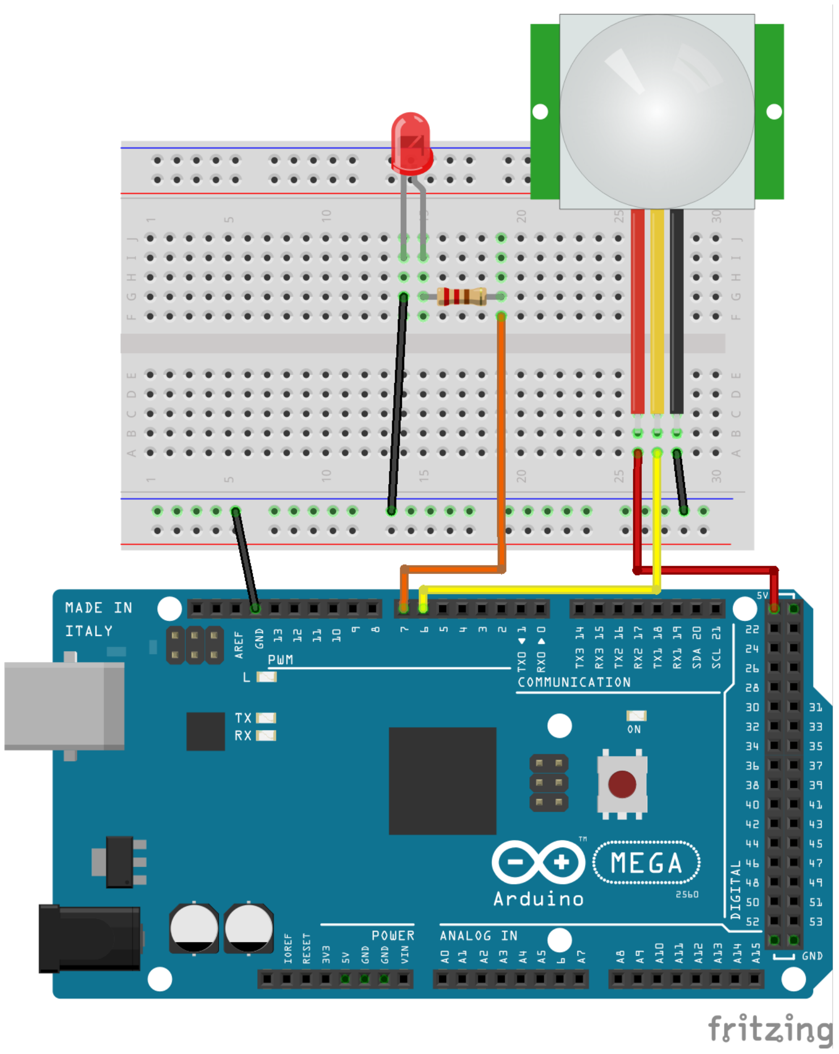


Figure 137: Scheme of hardware setup for program presented in figure 138.

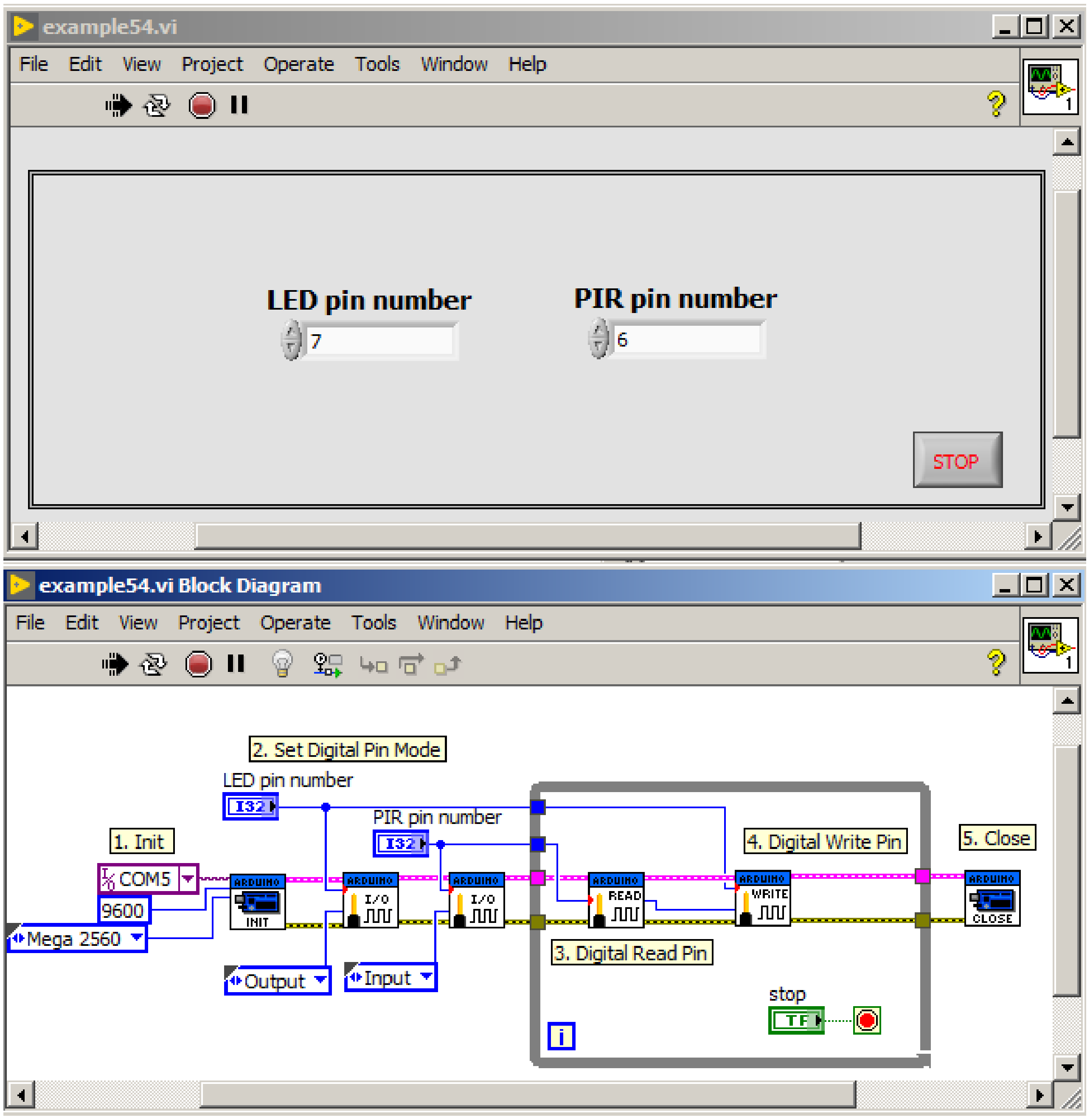


Figure 138: Program that lighten up LED diode after movement is detected by PIR motion sensor. Program is prepared with use of LIFA library.

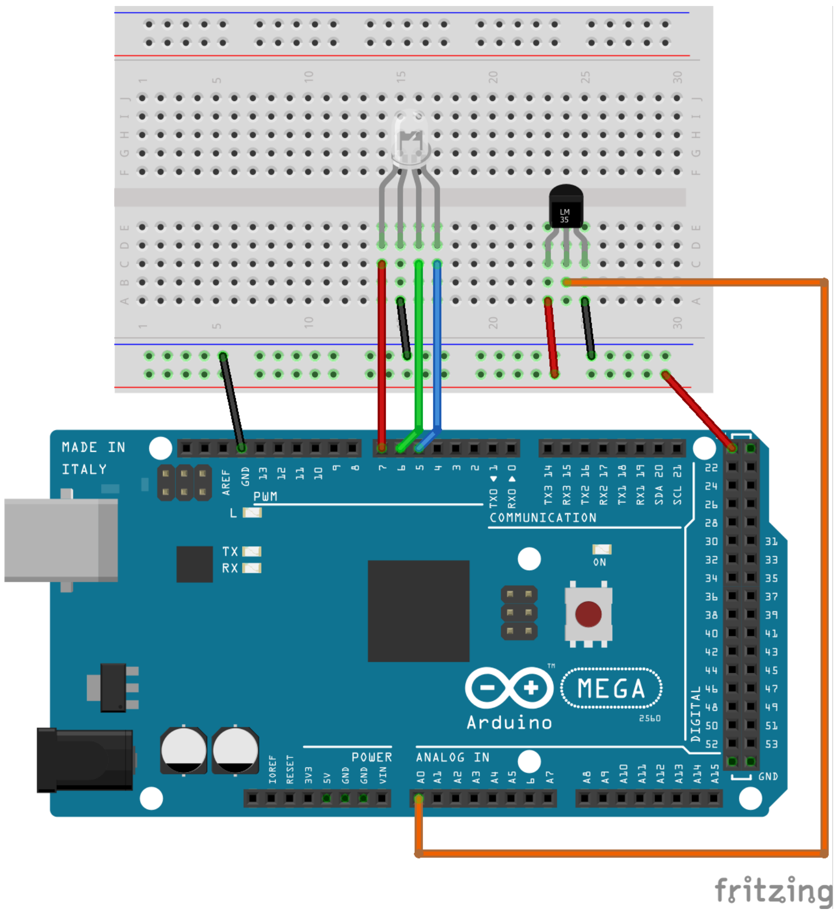


Figure 139: Scheme of hardware setup for program presented in figure 140.

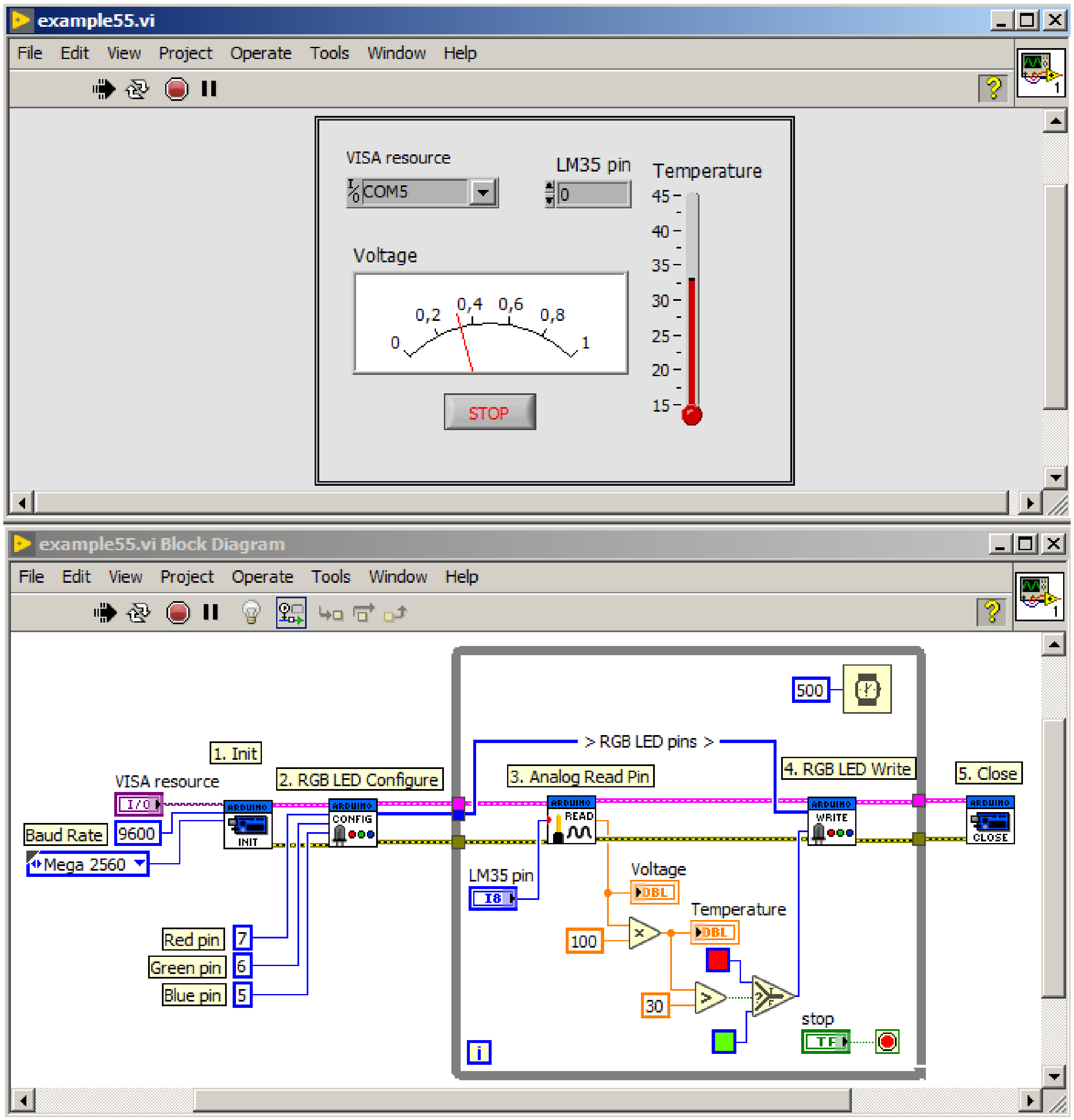


Figure 140: Program that measures temperature from LM35 sensor and changes RGB LED color accordingly.

e.g. LCD display. In figure 142 a modified example (from 140) is presented. A LCD display is added, which shows temperature value. Scheme of hardware setup is presented in figure 141.

At the beginning, a *LCD Configure 4-bit* function is used, where on input a cluster containing pin numbers connected to LCD display is passed. Next display size is configured with help of *LCD Set Size* function. In this case LCD with 16 columns and 2 rows is used. After reading temperature value from sensor, LCD cursor is set to the initial position. For this purpose a *LCD Set Cursor Position* function is used with input values of zero column and zero row. Temperature value is displayed on LCD display with help of *LCD Print* function, which accepts a string value on input. More useful functions for LCD operation can be found in *Functions palette* → *Arduino* → *Sensors* → *LCD*.

In example presented in figure 140, RGB LED usage was programmed with help of RGB LED dedicated functions. In figure 143 an example is presented, in which you can change the input of individual RGB colors to synthesize any color on RGB LED with use of low-level functions. First, three pins are configured as PWM outputs with help of *PWM Configure Port* function. Whereas in while loop, individual colors are set up on RGB LED diode with help of *PWM Write Port* function. After starting the program, on RGB LED a violet color is visible. Functions used in this example, operate three pins at once. To configure and change duty cycle of signal on single pin, one need to use *PWM Write Pin* function.

The LIFA library contains also functions to operate actuators like servos or DC motors. In figure 145 an example program is presented, which controls the servo. To do this, first a *Set Number of Servos* function is used, which accepts number of servos connected to Arduino board on its input. Next *Configure Servo* function is used, which accepts pin number on its input. In while loop a *Servo Write Angle* function is used, which sets selected angle in servo. All functions to operate servos are located in *Functions palette* → *Arduino* → *Sensors* → *Servo*. Scheme of hardware setup is presented in figure 144.

Examples presented above, demonstrate reading/writing data to digital pins, PWM signal generation, reading analog values and operation with basic electronics elements. More available functions from LIFA library for dealing with sensors are presented in figure 146.

LIFA controlled Arduino project

In the chapter above, it was shown hot to deal with simple sensors and actuators. In this chapter, a example project is presented, in which Arduino shield is used. This is KA-Nucleo-Weather shield (presented in figure 147). The shield consists of STLM75 temperature sensor, LPS331

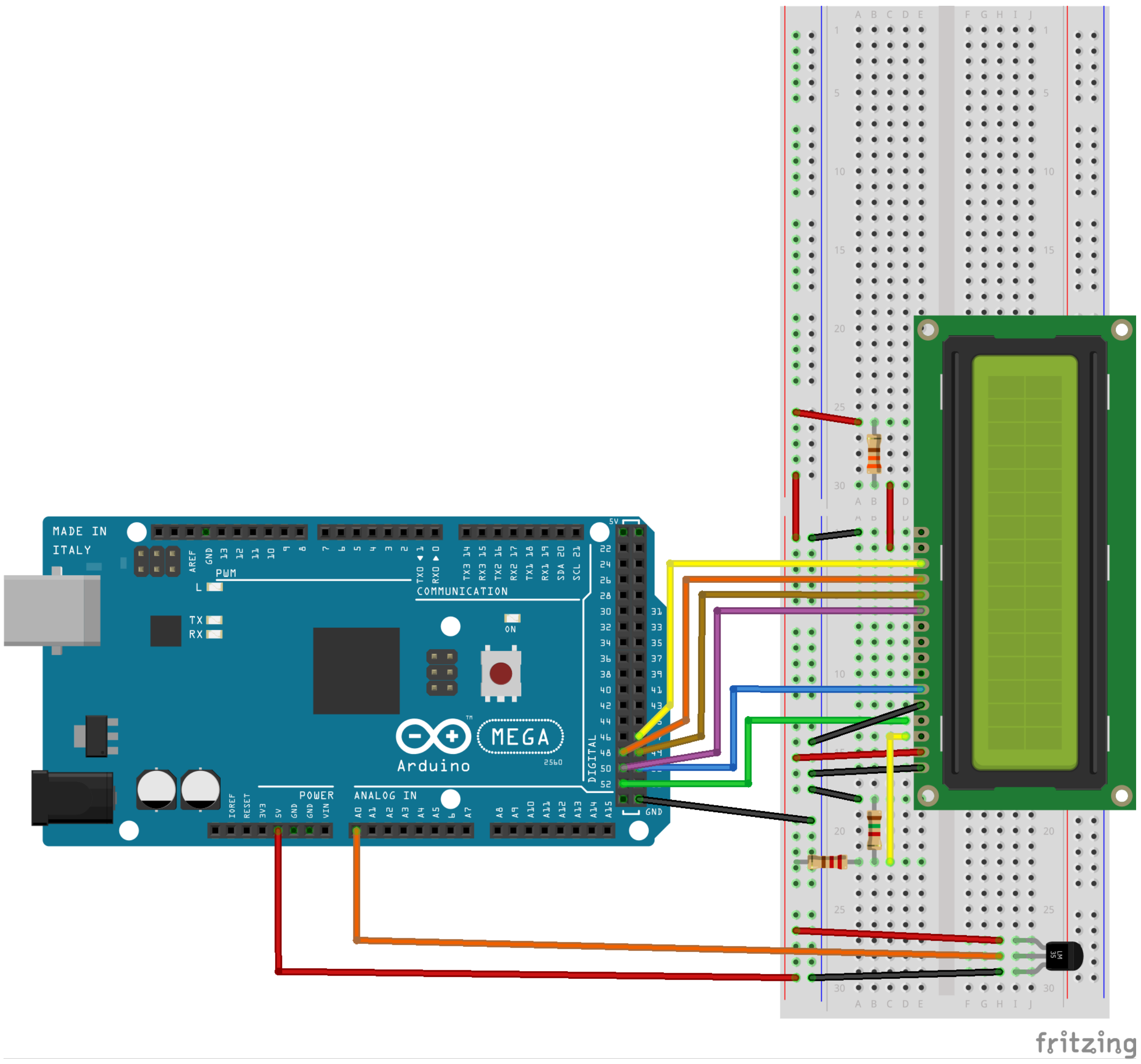


Figure 141: Scheme of hardware setup for program presented in figure 142.

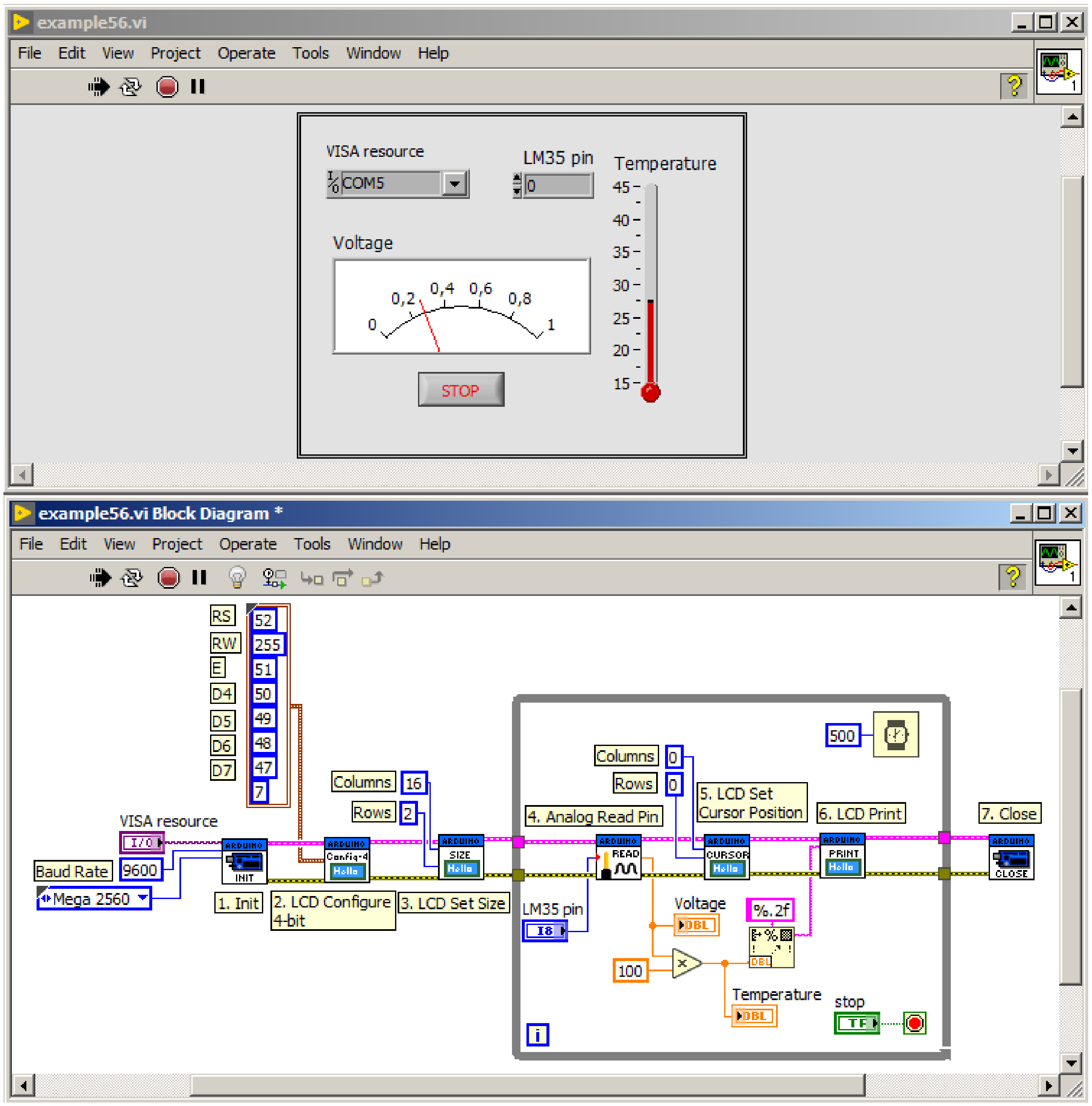


Figure 142: Program that measures temperature with use of LM35 sensor and displays temperature value on LCD display.

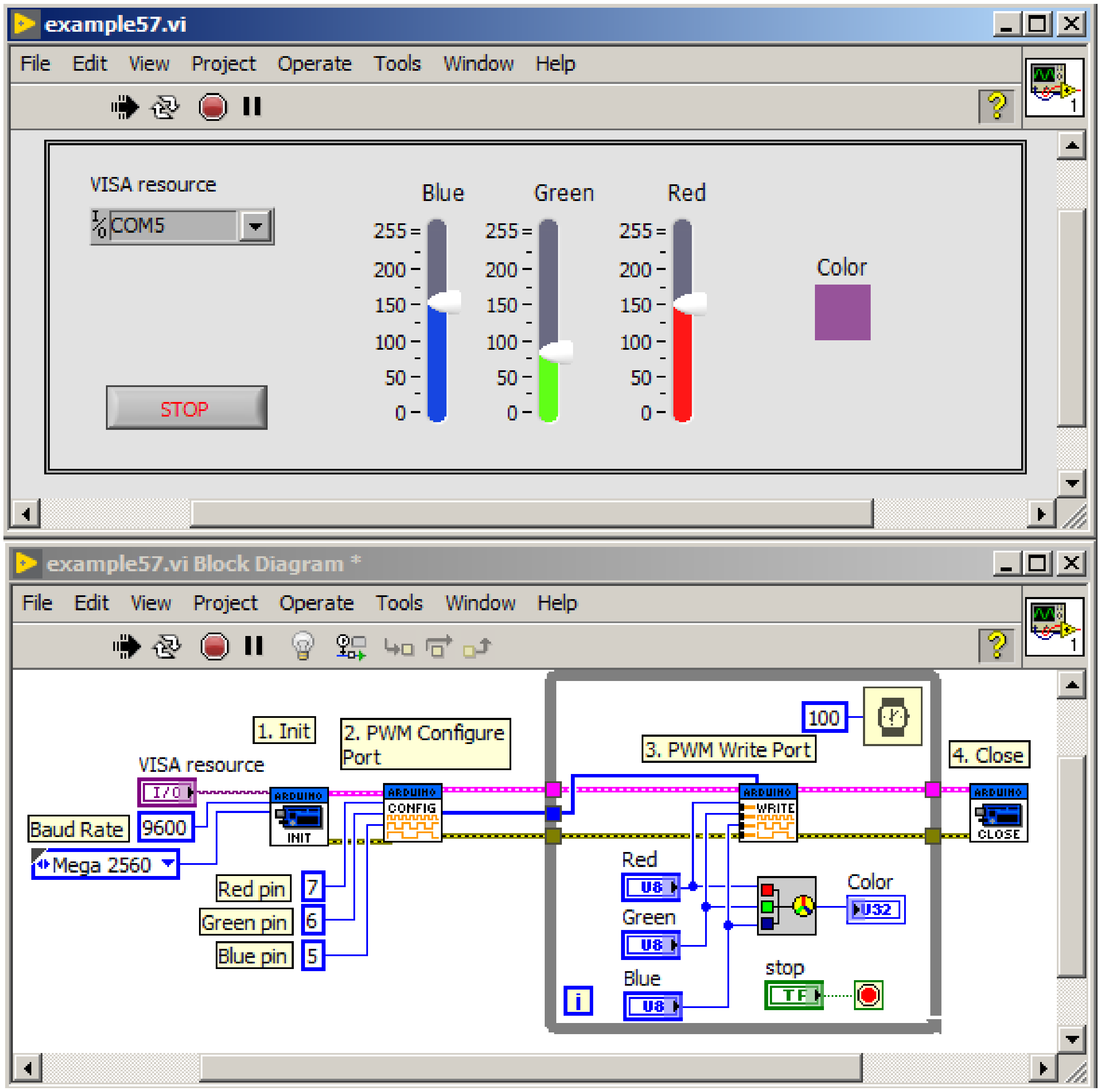


Figure 143: Program, that allows to dynamically change RGB LED color.

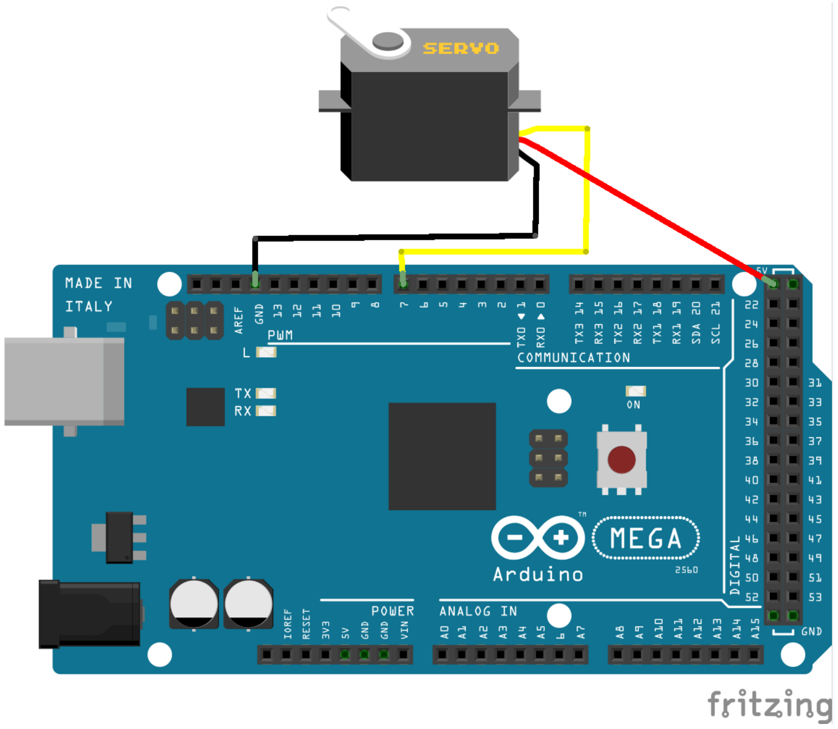


Figure 144: Scheme of hardware setup for program presented in figure 145.

pressure sensor, HTS221 humidity sensor and TSL25721 light intensity sensor.

A weather station reading temperature, pressure and humidity will be built as a project. All these sensors are connected through I^2C bus. In the project a design pattern called state machine is used, which block diagram is presented in figure 148.

First state is configuration of Arduino board connection with LabVIEW and set up of communication with pressure sensor located in shield. It is presented in figure 149. For that a *Init* function is used, which initializes connection with Arduino Mega 2560 board. Next *I2C Init* function is called, which initializes I^2C bus. Pressure sensor has address of 93. According to sensor datasheet, communication with sensor can be checked by reading one byte from its register from address 15. For this purpose a *I2C Write* function is used, which accepts on its input, an address of sensor (numerical value) and values, which should be sent to I^2C bus in form of 1D array of numerical values. In this case register address is sent (15), from which a value will be read. To read data from I^2C bus a *I2C read* function is used, which returns read values in the form of 1D array of numerical values. Read value is extracted from array with use of *Index Array* function and then it is compared with 187. According to sensor datasheet, if read value from register of address 15 is equal to 187, then communication with

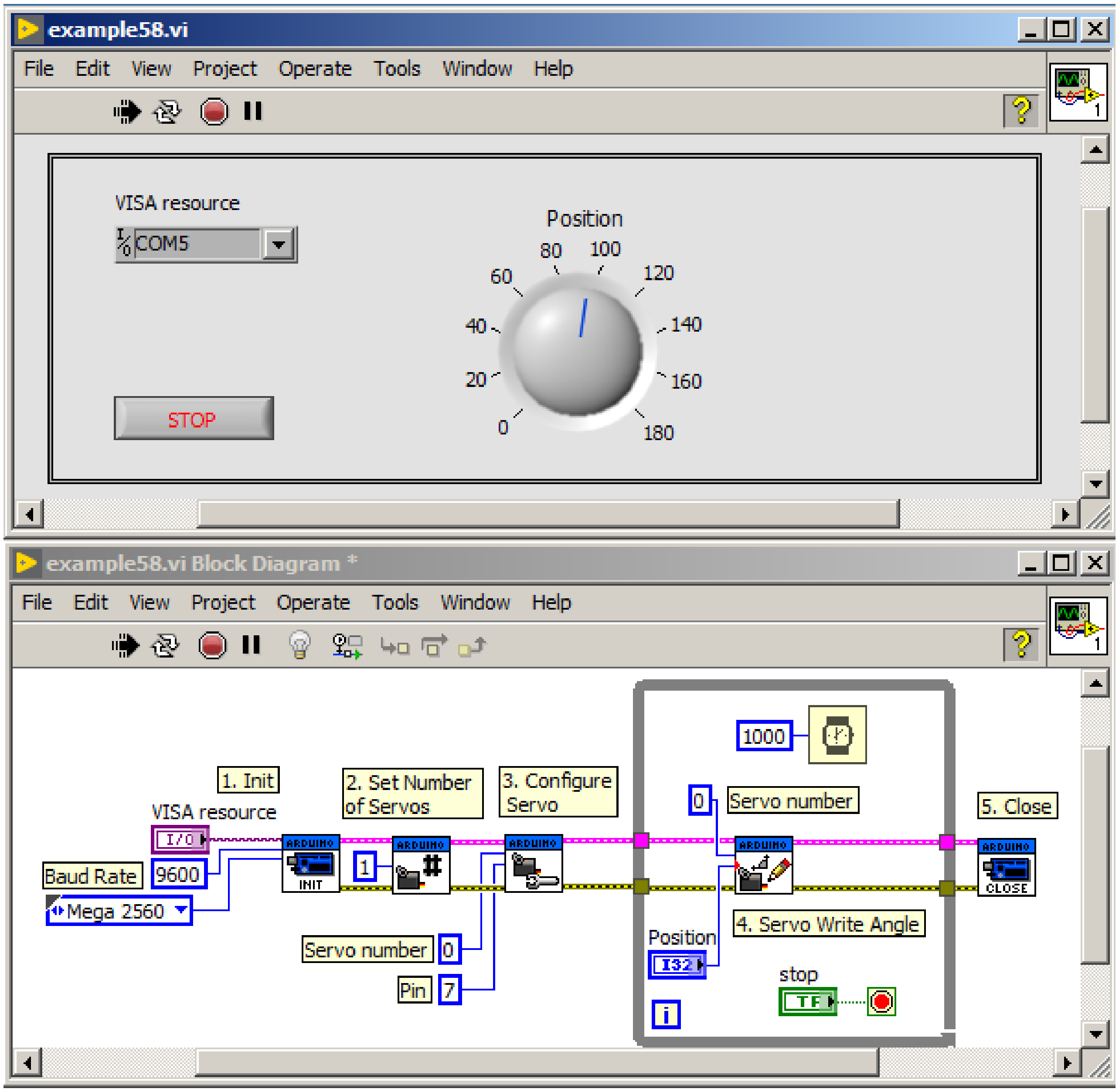


Figure 145: Program that sets selected angle on servo.

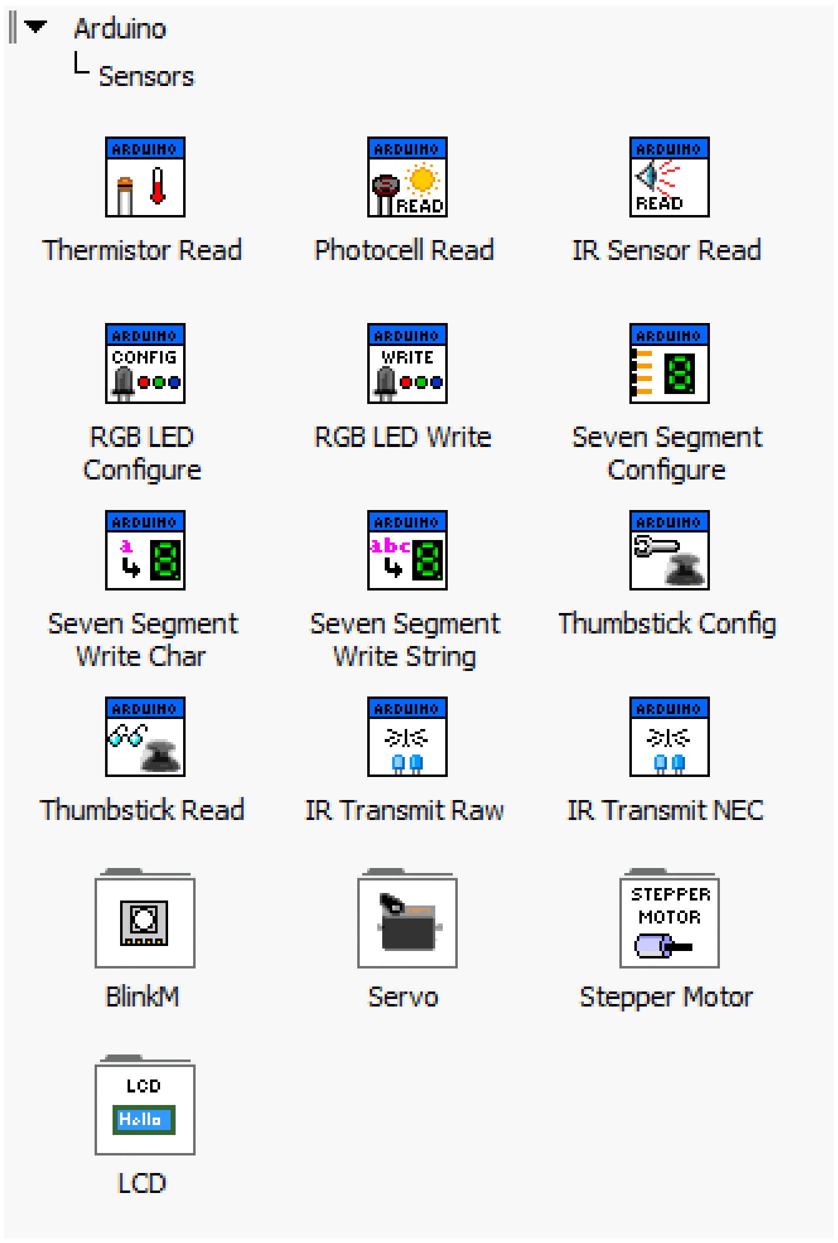


Figure 146: Available functions in *Sensors* fold of LIFA library.

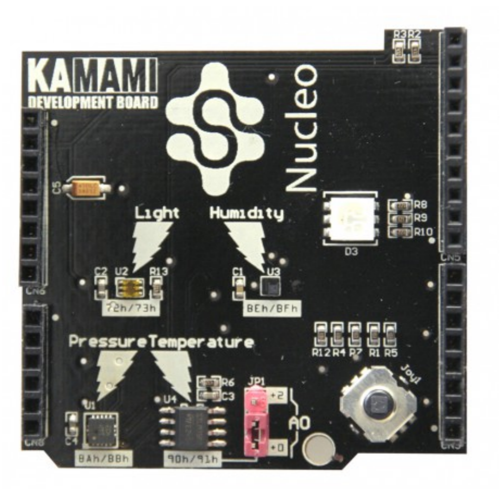


Figure 147: KA-Nucleo-Weather shield. Source: `https://kamami.pl`.

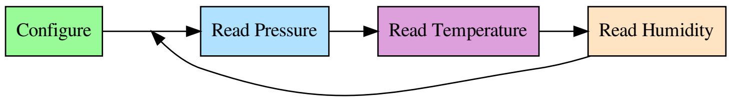


Figure 148: Block diagram of weather station project.

pressure sensor was successful.

In next step, pressure value is read from sensor. This is presented in figure 150. The value is stored as 3 bytes in 3 registers of addresses 168-170. At the beginning, address of first register is send - this is 168, with help of *I2C Write* function. Next 3 bytes are read with *I2C Read* function. The read data bytes must be stuck together according to figure 151.

Conversion of read 24-bit raw value to pressure value is done as follows:

$$p = \frac{p_{raw}}{4096} + 260 \quad (1)$$

In figure 152 a 3 bytes conversion to 24-bit value is presented, which is used to obtain pressure value. At the beginning byte values are extracted from 1D array with help of *Index Array* function. Next numerical values are converted to array with individual bits using *Number to Boolean Array* function. The resulting arrays are combined in one with help of *Insert into Array* function. Ready array is converted to numerical value with use of *Boolean Array to Number* function. Following steps in subVI are calculation of pressure with help of equation 1.

Next state is temperature measurement. It is presented in figure 153. To read temperature value from sensor a an appropriate register address is sent, which is 0 in this case. For this a *I2C Write* function is

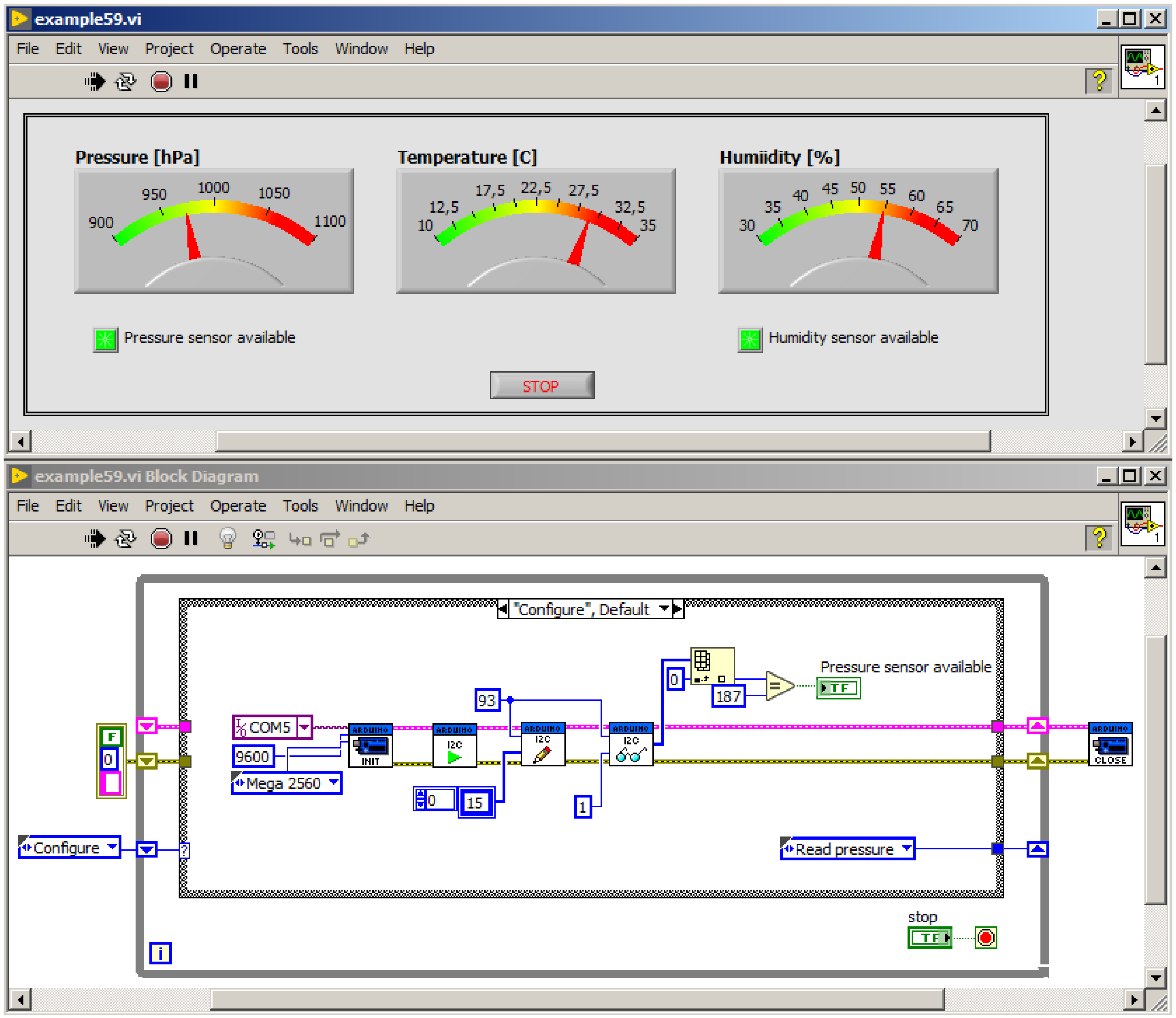


Figure 149: Program, that reads values of temperature, pressure and humidity. A *Configure* state of state-machine is presented here.

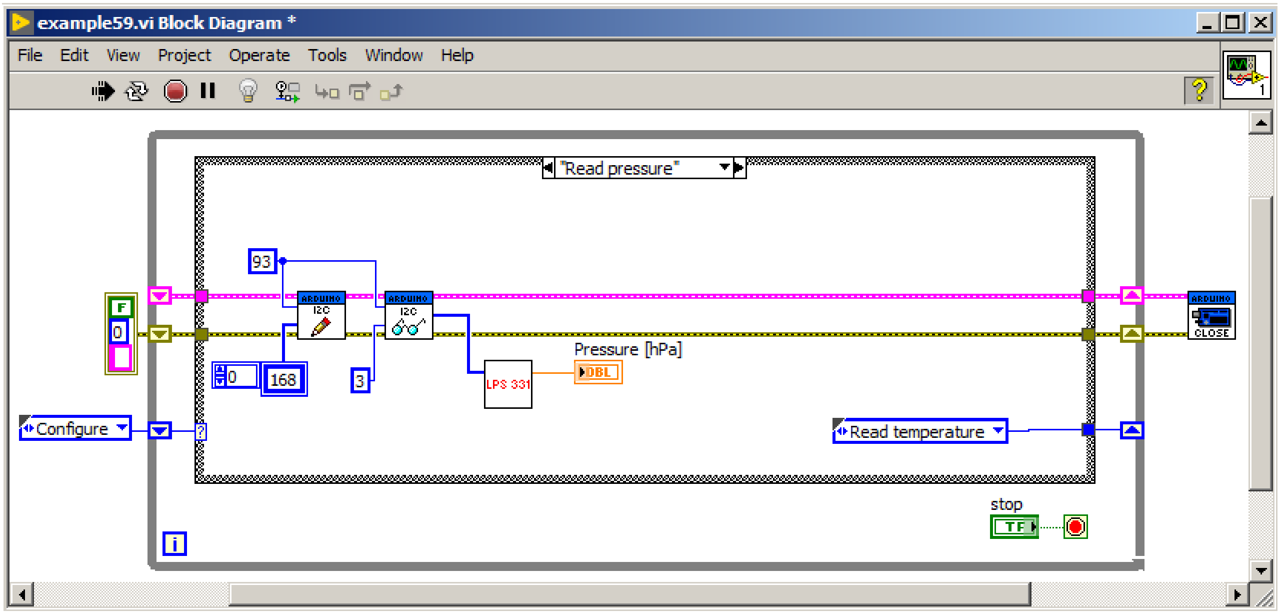


Figure 150: Program, that reads values of temperature, pressure and humidity. A *Read pressure* state of state-machine is presented here.

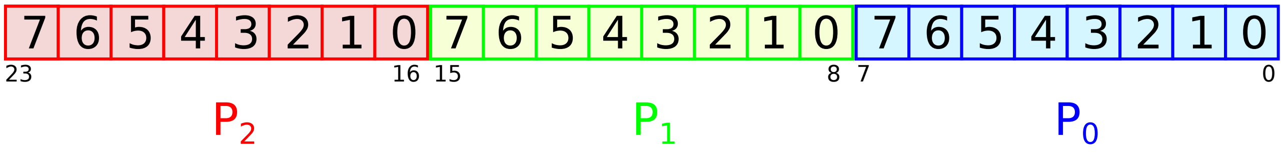


Figure 151: 24-bit value p_{raw} consists of P1, P2, P3 read bytes.

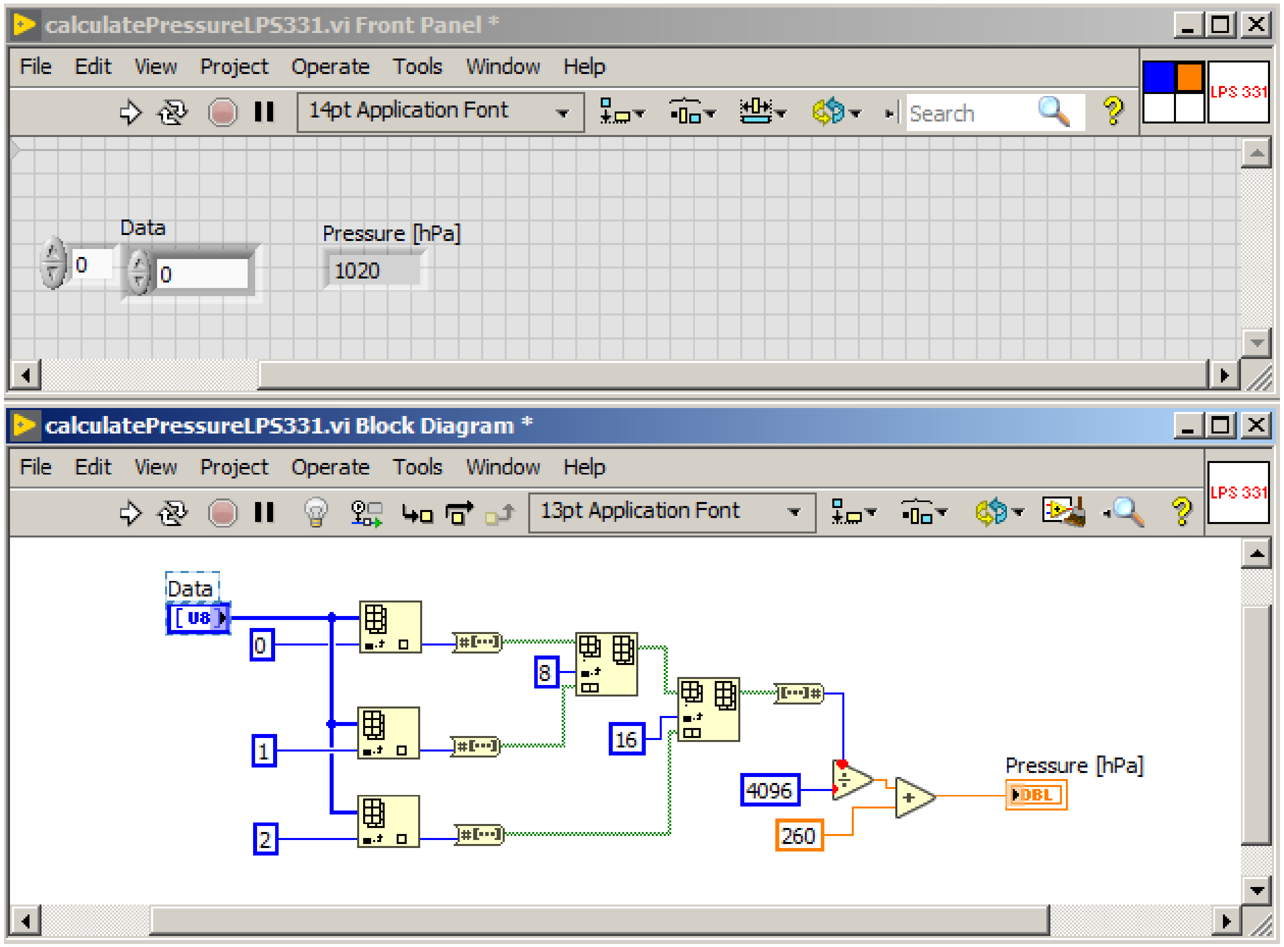

calculatePressureLPS331.vi Front Panel *
File Edit View Project Operate Tools Window Help
14pt Application Font
Search
LPS 331
Data
0
0
Pressure [hPa]
1020
calculatePressureLPS331.vi Block Diagram *
File Edit View Project Operate Tools Window Help
13pt Application Font
LPS 331
Data
U8
0
1
2
8
16
4096
260
Pressure [hPa]
DBL

Figure 152: SubVI, that converts 3 bytes to 24-bit value and calculate pressure value.

used, where sensor address is 72 and, and data value to sent is 0. Next temperature is read as 2 bytes (T1, T2) using *I2C Read* function. Temperature is determined according to equation 2, wherein the second byte is shifted by 7 bits to the right, as it only has single bit (most significant bit) for 0.5 temperature value.

$$T = T1 + 0.5 \cdot T2 \tag{2}$$

In figure 154 a subVI is presented, that from two read bytes calculates temperature. At the beginning byte values are extracted from 1D array using *Index Array* function. Next T2 byte is converted to bits array using *Number to Boolean Array* function. Operation of desired shifting by 7 bits right is analogous to rotation of of an array left by 1bit , because other bit values are 0, so *Rotate 1D Array* function is used with value 1 passed on input. The resulting array is converted to number using *Boolean Array to Number* function. Such prepared values are then passed to equation 2.

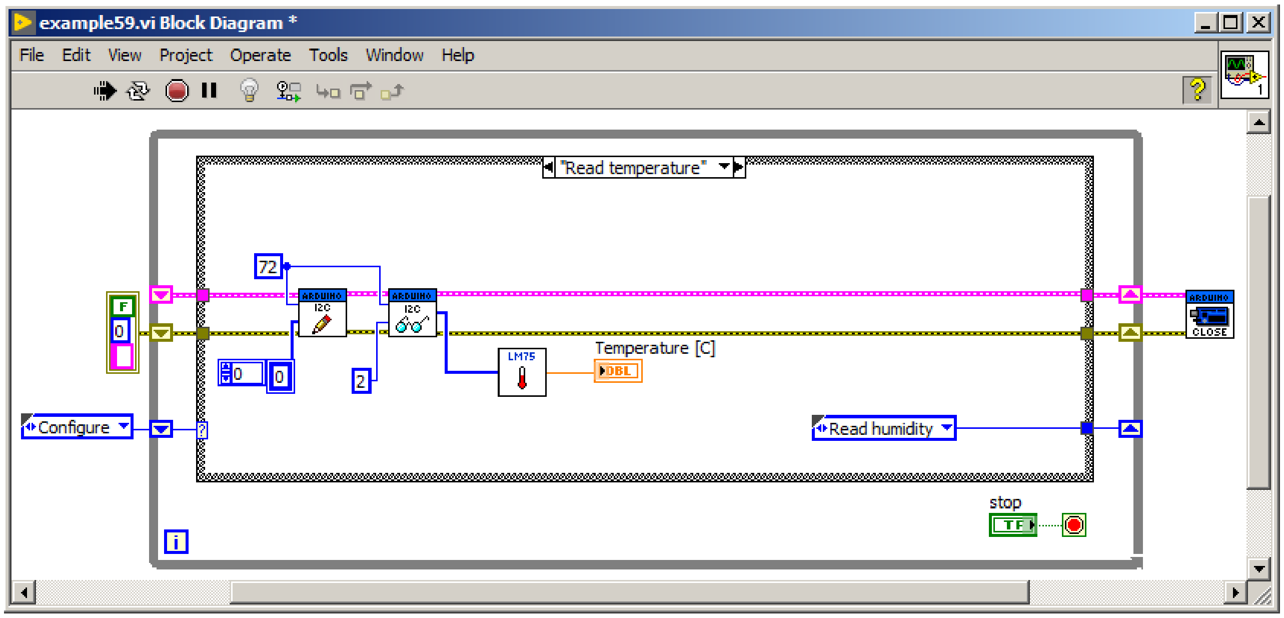


Figure 153: Program, that reads values of temperature pressure and humidity. A *Read temperature* state of state-machine is presented here.

The last state is reading humidity from sensor. This is presented in figure 155. At the beginning, a communication with sensor is established, by sending register address of 15 to device with address of 95 using *I2C Write* function. If value returned from *I2C Read* function is equal to 188, then according to sensor datasheet, communication from sensor was successful. Next a subVI is called, which block diagram is presented in figure 156. For configuration, first register address is sent, then values with help of *I2C Write* function, e.g. to power up

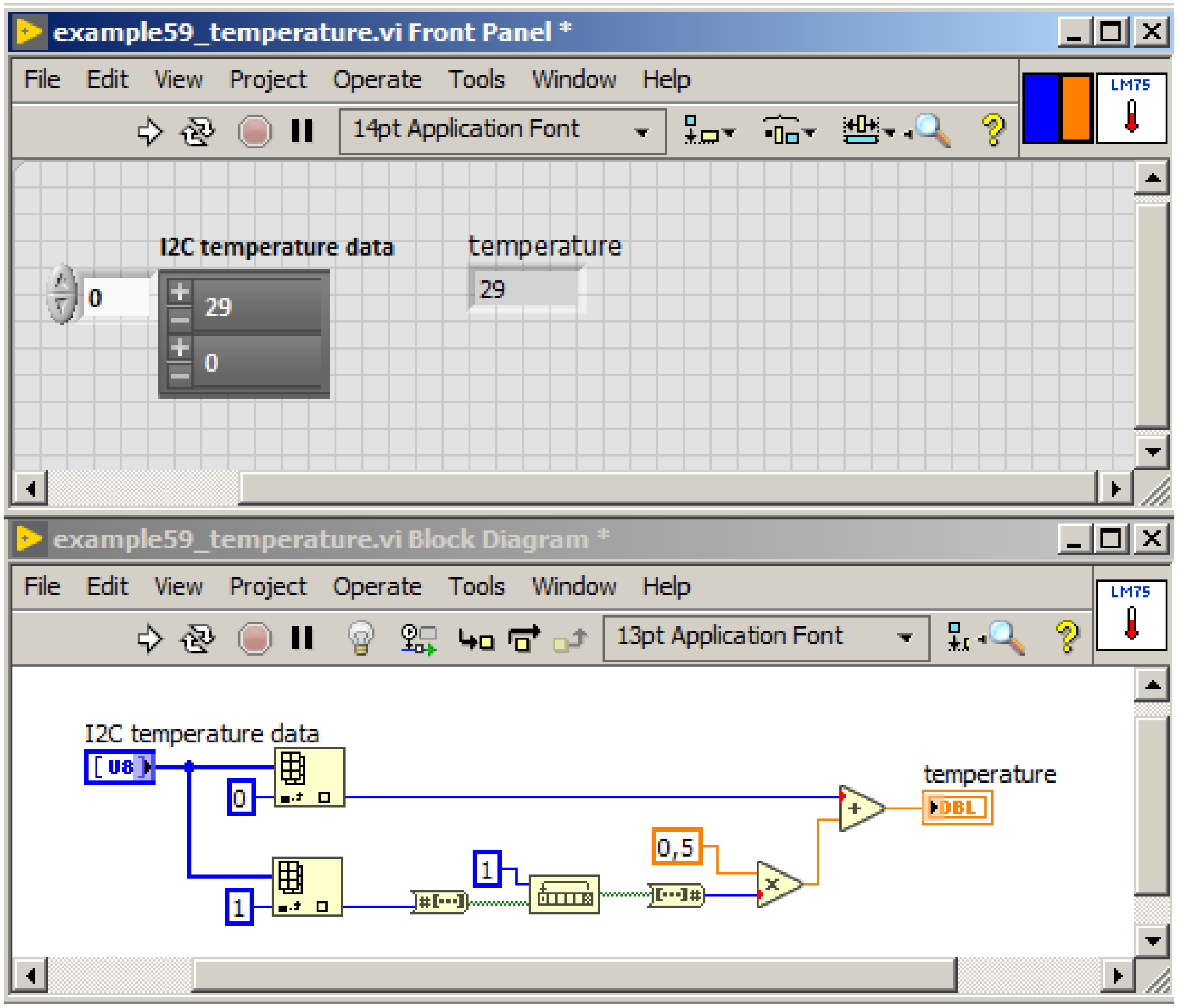

example59_temperature.vi Front Panel *
File Edit View Project Operate Tools Window Help
14pt Application Font
LM75
I2C temperature data
0
29
0
temperature
29
example59_temperature.vi Block Diagram *
13pt Application Font
I2C temperature data
U8
0
1
1
0,5
temperature
DBL

Figure 154: SubVI, that calculates temperature value taking two bytes on input.

sensor, a value 129 should be sent to register at address 32. Humidity sensor may return single value (one-shot) or can make measurements in selected frequency. In this project single shot value was chosen, so according to datasheet, a value of 96 should be sent to register at address 16.

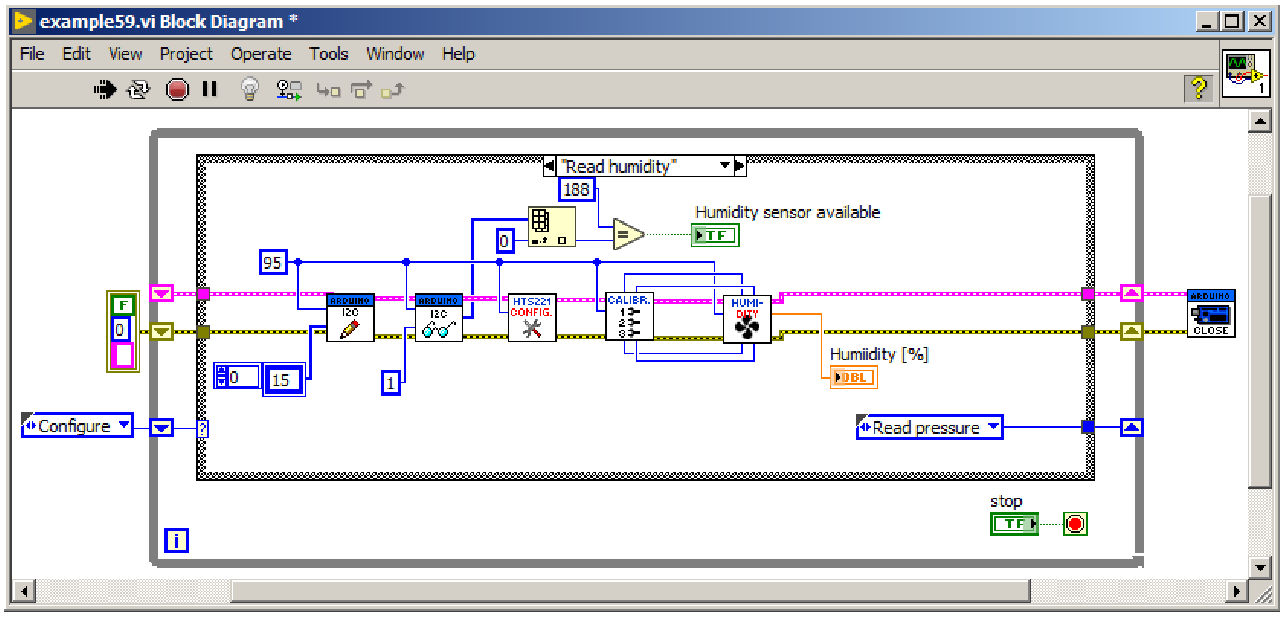


Figure 155: Program, that reads values of temperature, pressure and humidity. A *Read humidity* state of state-machine is presented here.

In datasheet for humidity sensor, there is a plot, presented in figure 157 showing how to obtain humidity from values obtained from registers and calibration registers. This is a linear calibration plot. On the basis of designations from plot, an equation for relative humidity (rH) can be written as equation:

$$rH = H0\_RH + \frac{[H\_(T)\_OUT - H0\_(T0)\_OUT] * [H1\_RH - H0\_RH]}{H1\_(T0)\_OUT - H0\_(T0)\_OUT} \tag{3}$$

Therefore to calculate relative humidity, one need to read following values from registers: $H0\_RH$, $H1\_RH$, $H1\_(T0)\_OUT$, $H0\_(T0)\_OUT$ and $H\_(T)\_OUT$. Register map from sensor datasheet is presented in figure 158.

In figure 159 there is a subVI, which purpose is to read necessary values from registers. At the beginning, a $H0\_RH$ value is read. According to register map, $H0\_RH$ multiplied by 2 (($H0\_RH\_x2$) is stored in register at address 0x30. Converting hexadecimal value 0x30 to decimal, we have 48. A *I2C Write* function is called, where on its input a value of 48 is connected. Address of humidity sensor is 95. Value from register is read by *I2C Read* function in form of array. To extract

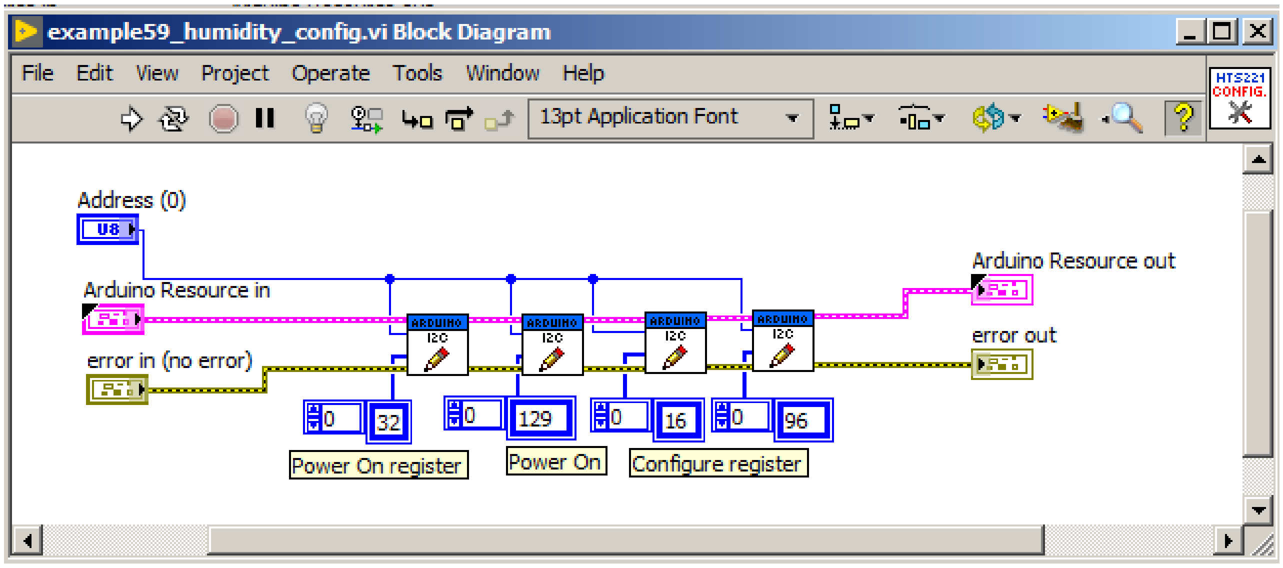


Figure 156: SubVI, that configures humidity sensor.

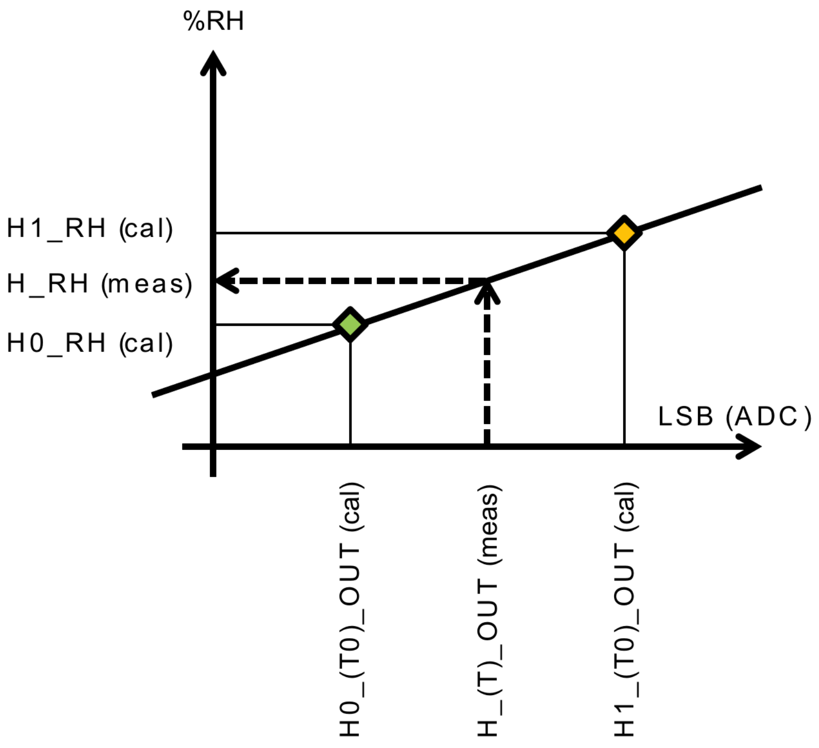
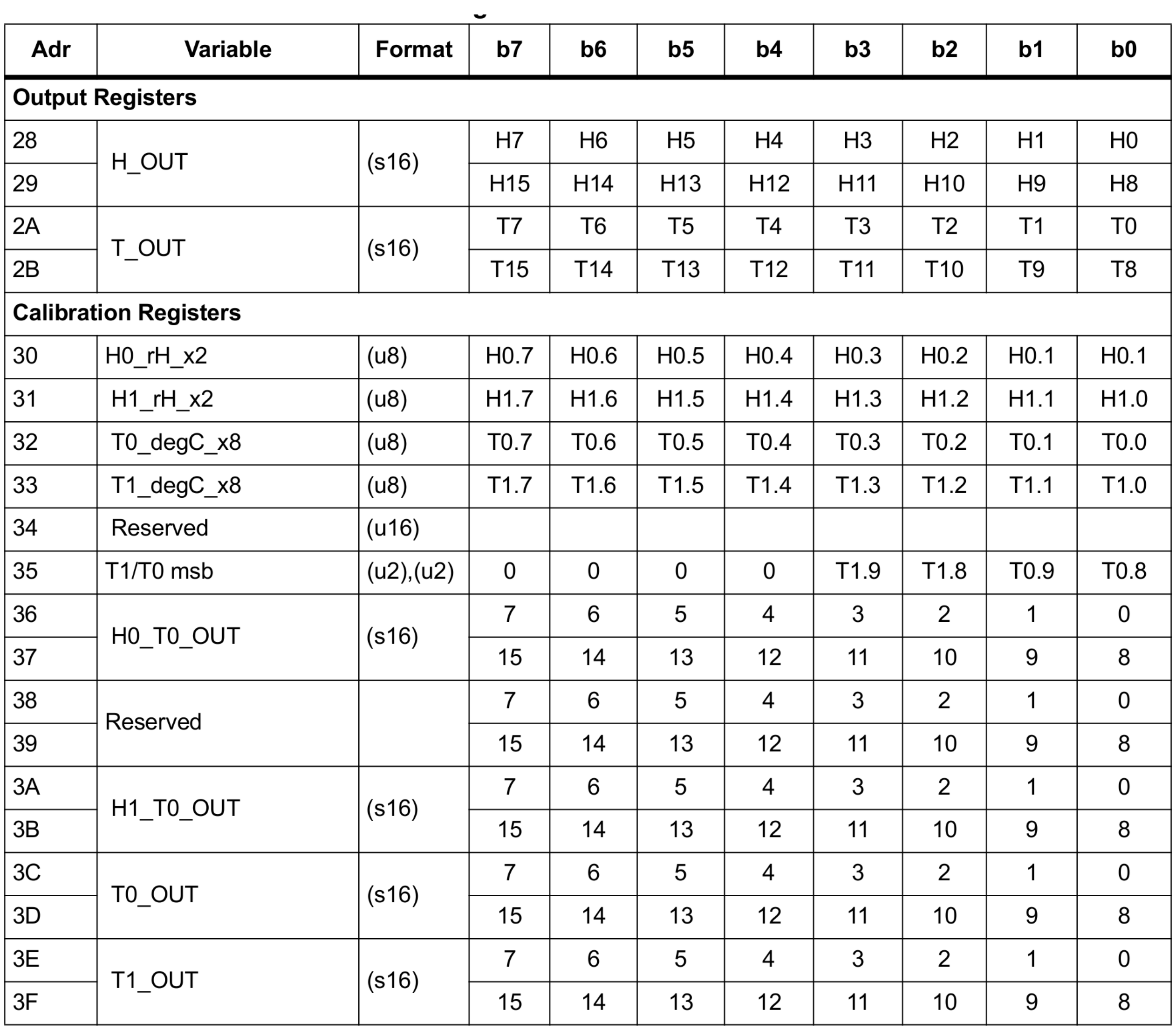


Figure 157: Idea how to obtain relative humidity value, using calibration points. Source: https://download.kamami.pl/p558410-hts221tr.pdf.



Adr	Variable	Format	b7	b6	b5	b4	b3	b2	b1	b0
Output Registers										
28	H_OUT	(s16)	H7	H6	H5	H4	H3	H2	H1	H0
29			H15	H14	H13	H12	H11	H10	H9	H8
2A	T_OUT	(s16)	T7	T6	T5	T4	T3	T2	T1	T0
2B			T15	T14	T13	T12	T11	T10	T9	T8
Calibration Registers										
30	H0_rH_x2	(u8)	H0.7	H0.6	H0.5	H0.4	H0.3	H0.2	H0.1	H0.1
31	H1_rH_x2	(u8)	H1.7	H1.6	H1.5	H1.4	H1.3	H1.2	H1.1	H1.0
32	T0_degC_x8	(u8)	T0.7	T0.6	T0.5	T0.4	T0.3	T0.2	T0.1	T0.0
33	T1_degC_x8	(u8)	T1.7	T1.6	T1.5	T1.4	T1.3	T1.2	T1.1	T1.0
34	Reserved	(u16)								
35	T1/T0 msb	(u2),(u2)	0	0	0	0	T1.9	T1.8	T0.9	T0.8
36	H0_T0_OUT	(s16)	7	6	5	4	3	2	1	0
37			15	14	13	12	11	10	9	8
38	Reserved		7	6	5	4	3	2	1	0
39			15	14	13	12	11	10	9	8
3A	H1_T0_OUT	(s16)	7	6	5	4	3	2	1	0
3B			15	14	13	12	11	10	9	8
3C	T0_OUT	(s16)	7	6	5	4	3	2	1	0
3D			15	14	13	12	11	10	9	8
3E	T1_OUT	(s16)	7	6	5	4	3	2	1	0
3F			15	14	13	12	11	10	9	8

Figure 158: Register map of humidity sensor. Source: `https://download.kamami.pl/p558410-hts221tr.pdf`.

value from array *Index Array* function is used. In this way $H0\_RH\_x2$ value is read. To convert it to $H0\_RH$ value, one need to just divide by 2. In similar way $H1\_RH$ value is read. $H1\_(T0)\_OUT$ value is stored in two registers at address 0x3A and 0x3B (in decimal that is 58 and 59). Therefore two values need to be read, and next they should be combined into 16-bit value in similar way as P1, P2 and P3 to P_{raw} when pressure was read. For this purpose a subVI is created, that converts 2 bytes to 16-bit value, which is presented in figure 160.

After reading necessary calibration data, the last subVI is called, which is presented in figure 161. Measured humidity value is placed in registers at addresses 0x28 and 0x29 (40 and 41 in decimal). Thus similar as in $H1\_(T0)\_OUT$ case, one need to convert two bytes read to 16-bit value. Following part of subVI is calculation of relative humidity from calibration curve, according to 3 equation.

After starting the program, all three values: pressure, temperature and humidity are read. As it is visible communication with sensors through I^{2}C bus comes to: sending internal address of register to sensor, providing its bus address with help of *I2C Write* function, reading value from register with help of *I2C Read* function and data processing like calibration, conversion, etc. Besides I^{2}C bus, in LabVIEW with LIFA library one can use also SPI BUS if SPI sensors are available. Functions available to operate SPI interface are presented in figure 162.

Digilent LINX library

The LINX library can be used for communication with Arduino, Raspberry PI, myRIO and chipKIT devices. This library is available from LabVIEW MakerHub, an organisation supporting *makers* environment with use of LabVIEW. More information can be obtained from website: https://www.labviewmakerhub.com/. LINX library can be installed from web page: http://sine.ni.com/nips/cds/view/p/lang/pl/nid/212478 and also using VIPM (VI Package Manager) tool.

The LINX library contains basic functions like:

- **Open Serial** - initial function, which start connection with device on selected serial port.

- **Close** - function, which ends connection with device.

and functions grouped in folders:

- *Peripherals* $\rightarrow$ *Analog* - supporting analog inputs.

- *Peripherals* $\rightarrow$ *Digital* - supporting digital input/output ports.

- *Peripherals* $\rightarrow$ *PWM* - supporting PWM signal generation.

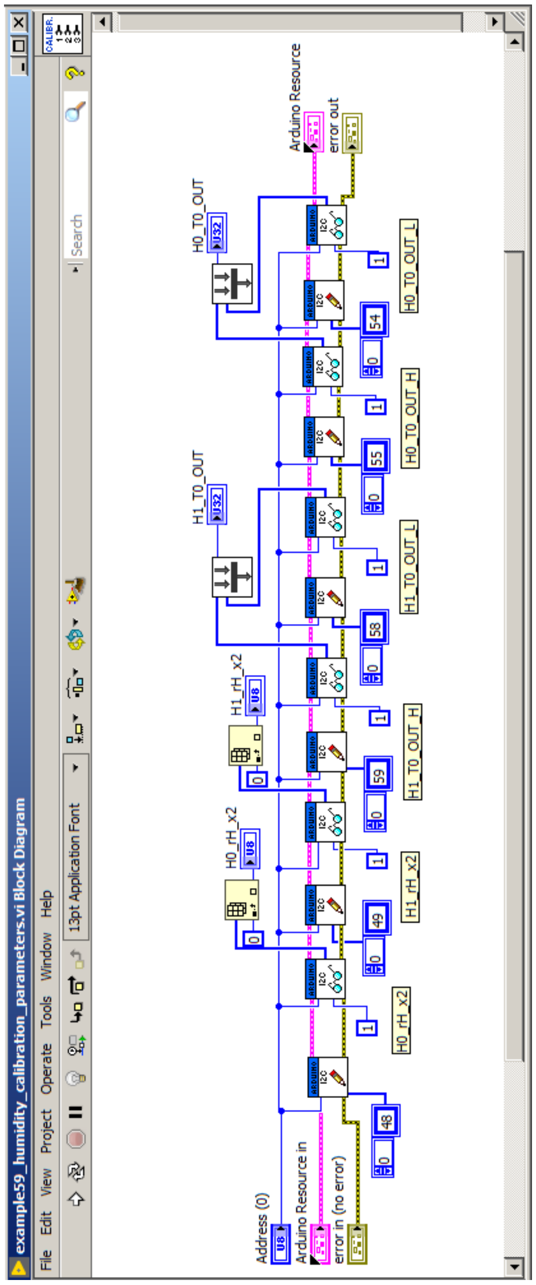


Figure 159: SubVI, that configures humidity sensor.

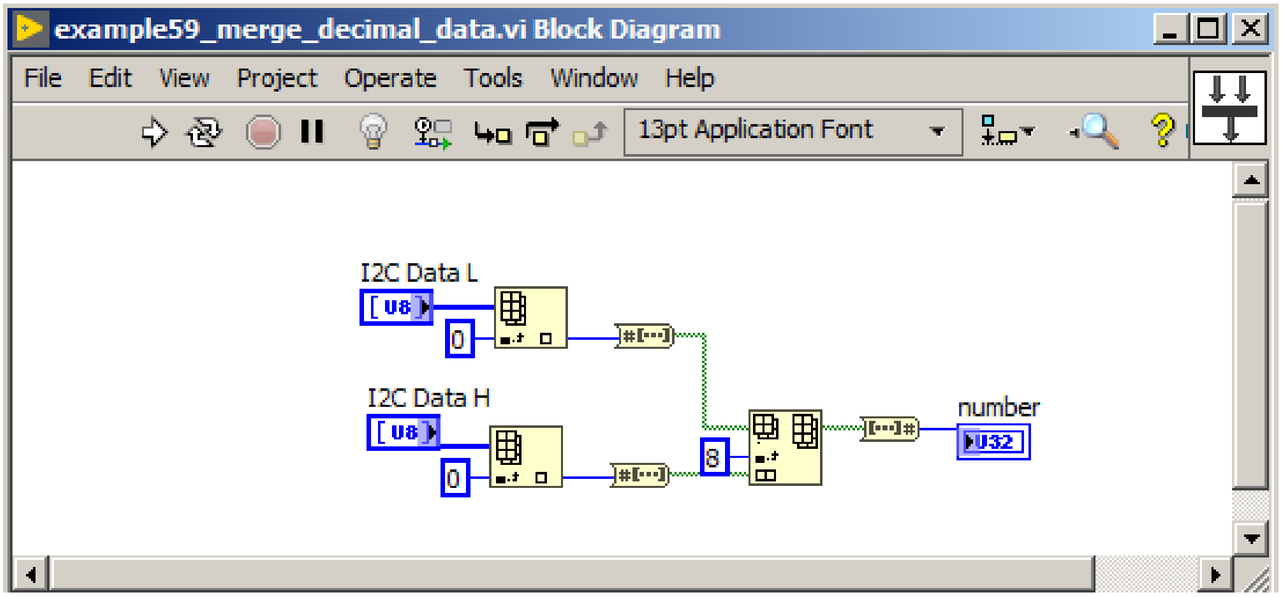

example59_merge_decimal_data.vi Block Diagram
File Edit View Project Operate Tools Window Help
13pt Application Font
I2C Data L
U8
0
I2C Data H
U8
0
8
number
U32

Figure 160: SubVI, that converts 2 bytes into 16-bit value.

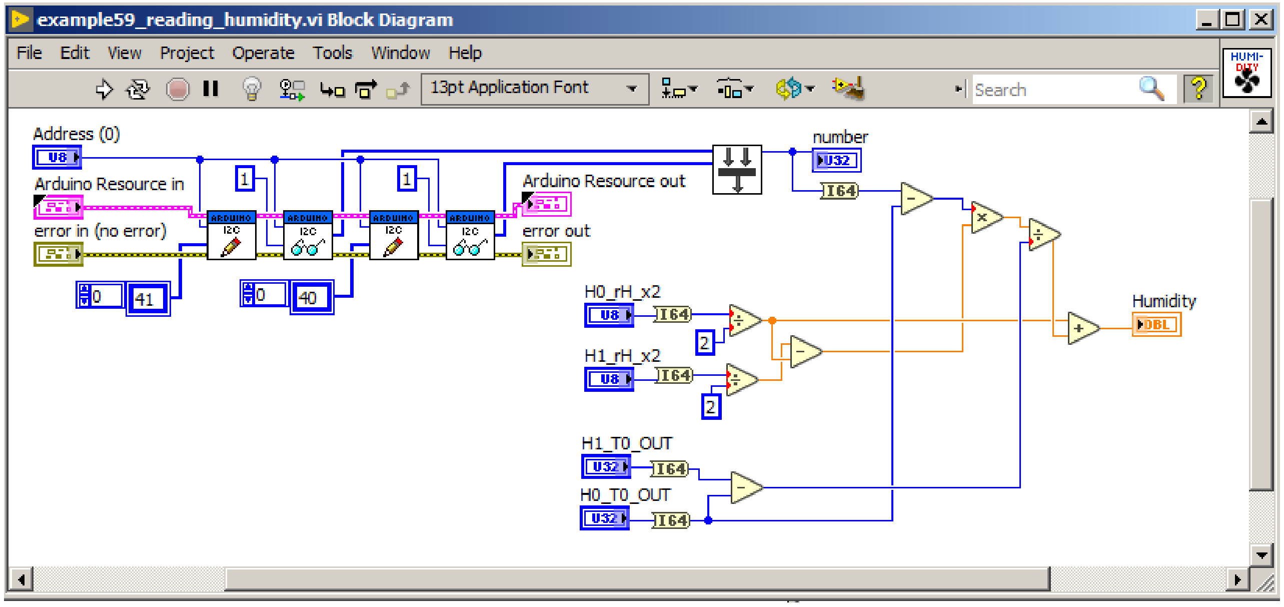


Figure 161: SubVI, that reads humidity values and calculates relative humidity using equation 3.

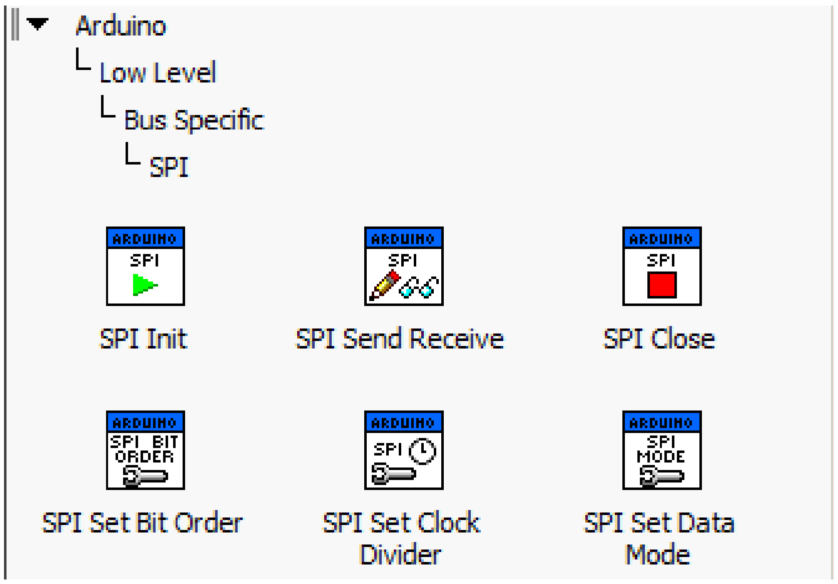


Figure 162: Functions for communication with devices over SPI bus from LIFA library.

- *Peripherals* → *I2C* - supporting I^{2}C bus operation.
- *Peripherals* → *SPI* - supporting SPI bus operation.

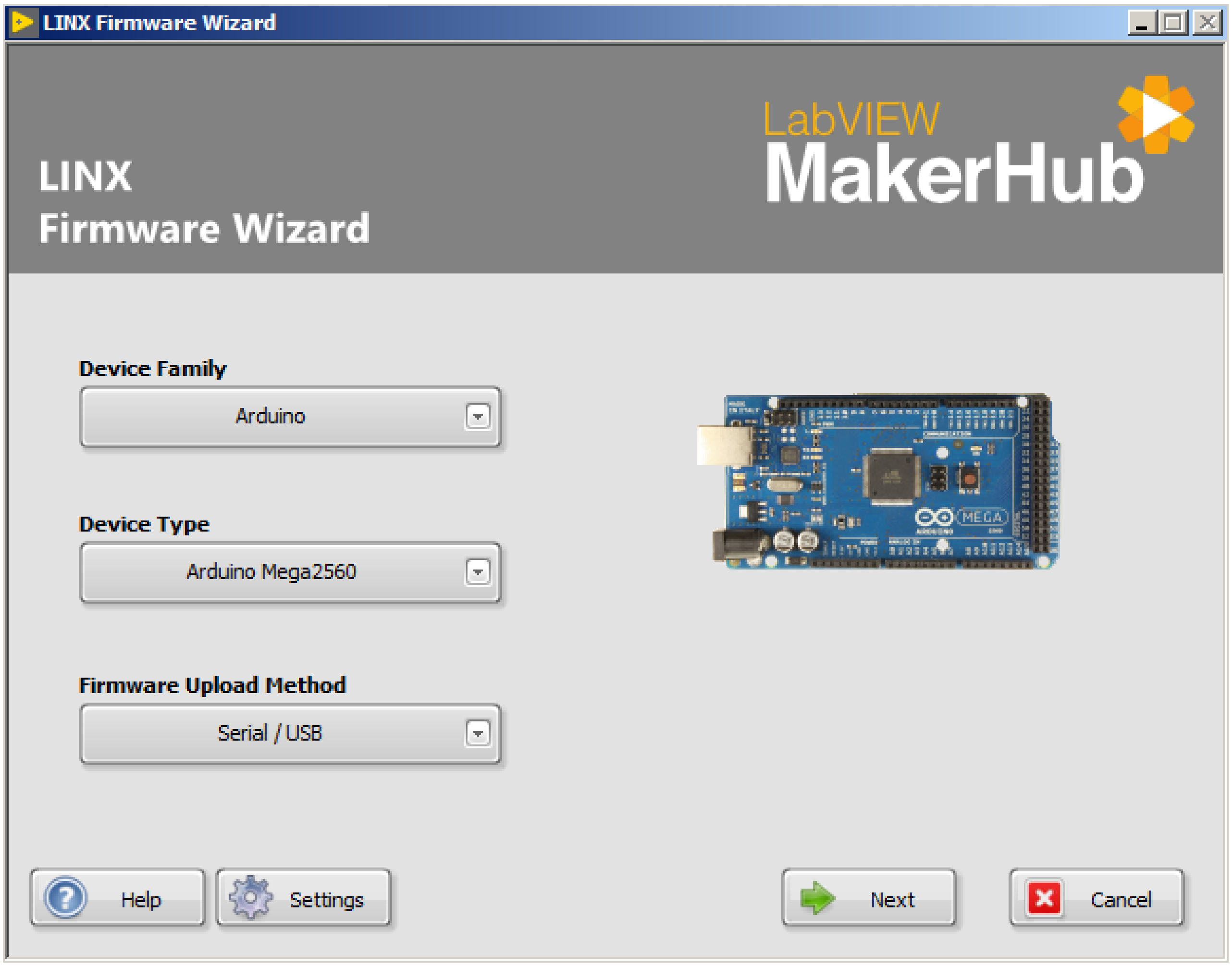
To upload firmware for Arduino, after starting LabVIEW, one need to select: **Tools** → **MakerHub** → **LINX** → **LINX firmware Wizard...** and next choose proper type of Arduino board and choose serial port. This is shown in figure 163.

LINX Library usage

The LINX library allows for reading and writing digital signals. First example program will turn on and off LED diode. Scheme of hardware setup is presented in figure 135. In figure 164 a ready program is presented, which starts with *Open Serial* function to begin connection with Arduino board on selected port. Next, inside while loop, a *True/False* Boolean value is sent to selected pin according to state of button. Function used to send digital data is *Digital Write*. Program ends with calling *Close* function, which stops connection with the Arduino board.

Previous example was modified adding PIR motion sensor, according to scheme presented in figure 137. In figure 165 there is a program, which turns LED diode on, when motion is detected by PIR sensor. This time on input of *Digital Write* function, value read from pin #6 is passed. Reading value from digital pin is realized by calling *Digital Read* function.

Above two examples used reading/writing digital signals. In fig-



Device Family: Arduino

Device Type: Arduino Mega2560

Firmware Upload Method: Serial / USB

Figure 163: Configuration of Arduino board type.

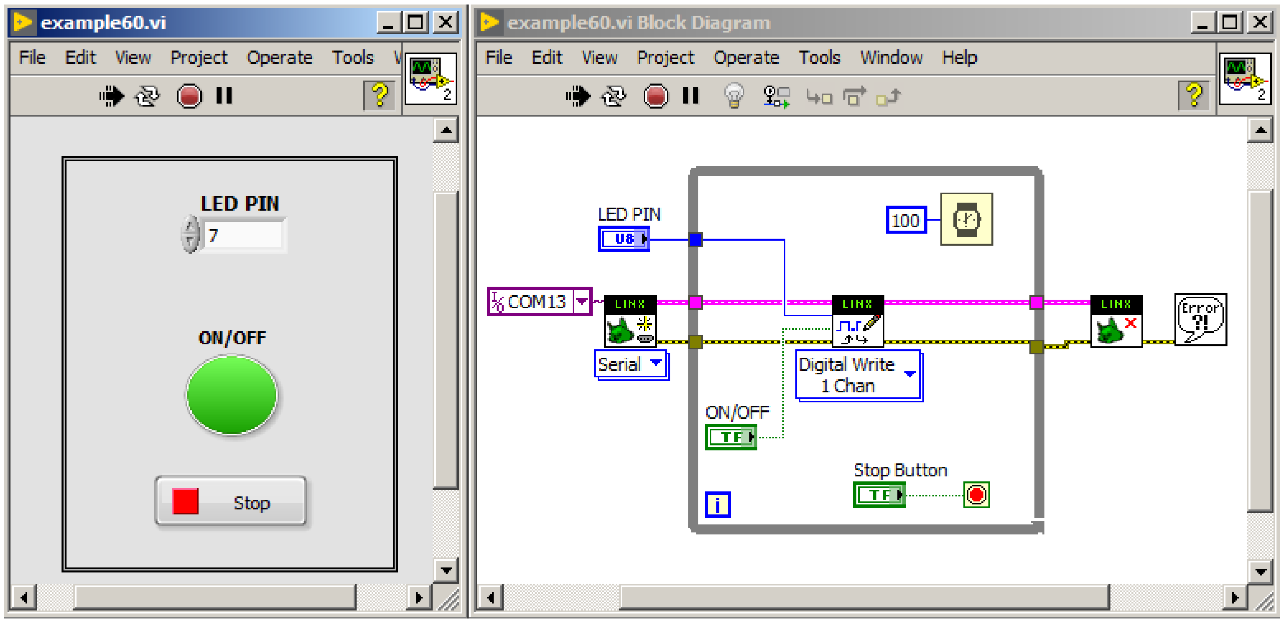

example60.vi
File Edit View Project Operate Tools
LED PIN
7
ON/OFF
Stop
example60.vi Block Diagram
File Edit View Project Operate Tools Window Help
LED PIN
U8
100
COM13
Serial
Digital Write 1 Chan
ON/OFF
TF
Stop Button
i

Figure 164: Program which turns on and off LED diode with help of LINX library.

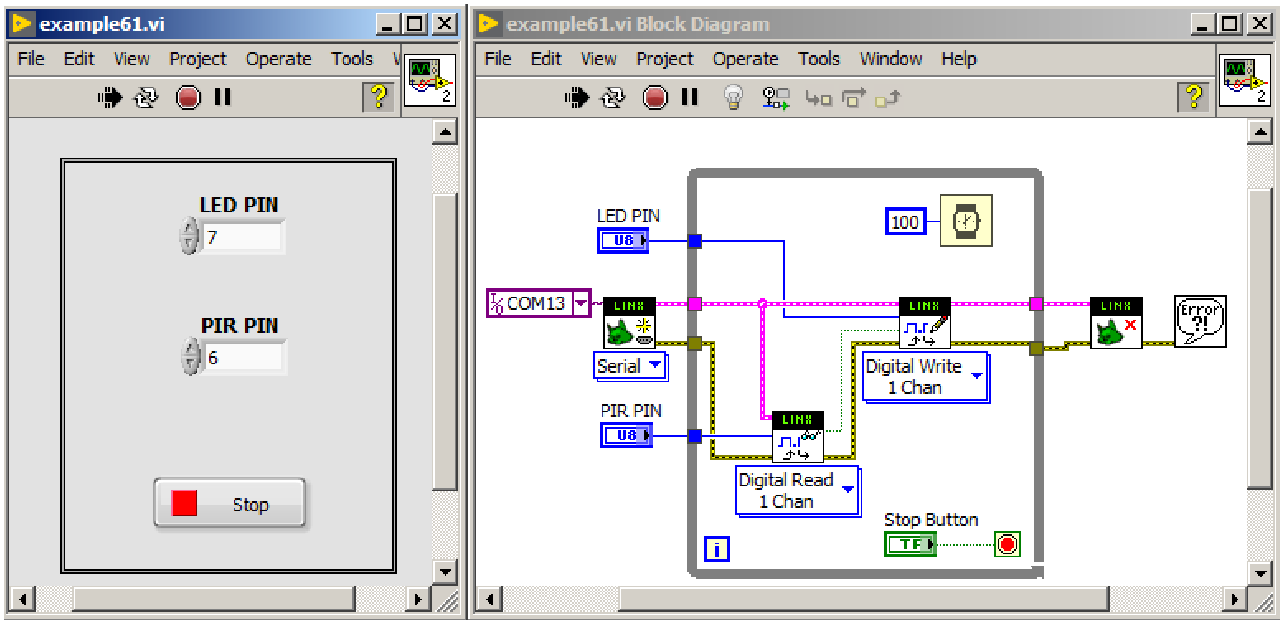


Figure 165: Program that turns LED diode on when motion is detected with help of LINX library.

ure 166, however there is a program, that reads voltage from LM35 temperature sensor. This read voltage is proportional to actual temperature of the sensor. Program starts with call of *Open Serial* function, which initializes connection with Arduino board. Next, inside while loop, voltage is read from pin #0 with use of *Analog Read* function. Obtained voltage value is multiplied by 100 to get temperature in Celsius centigrade. After while loop finish, a *Close* function is called, which ends communication with Arduino board. Scheme of hardware setup is presented in figure 167.

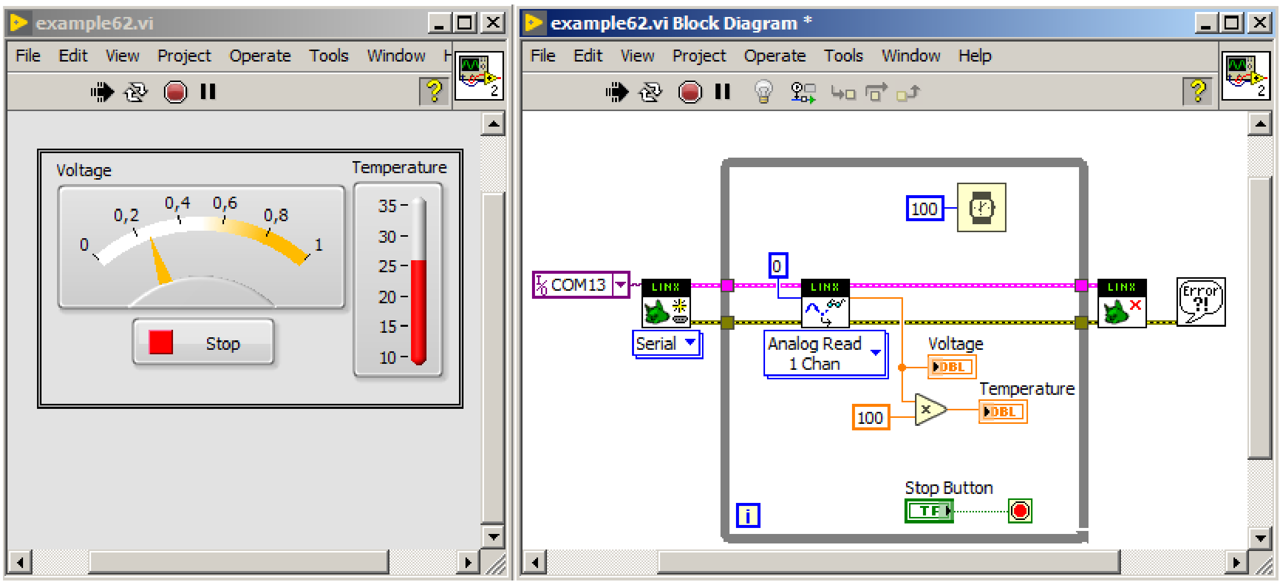


Figure 166: Program that reads voltage value from analog pin, where LM35 temperature sensor output is connected. Program prepared with use of LINX library.

The LINX library allows for PWM signal generation with help of *PWM Set Duty Cycle* function. This function accepts on its input: pin number, which will be used for PWM signal generation and duty cycle (filling - numerical value from 0..1 range, where 1 means 100%). In figure 168 a program is presented, in which duty cycle can be changed on RGB led diode colors. In loop, three functions *PWM Set Duty Cycle* are placed. To inputs of these functions, user controls and pin numbers are connected, which is presented in figure 169.

The LINX library also supports functions for controlling popular sensors and actuators, which is presented in figure 170. One of folds is useful for servo control. In figure 171, an example program is shown, which can set an angle of servomotor. To do so, three functions are used. One of them is *Servo Open One Channel*, which accepts pin number with connected servo. Second function is *Servo Set Pulse Width One Channel* which is placed inside while loop. It accepts pulse width in μs. Third function is *Servo Close Channel*. Scheme of hardware setup is pre-

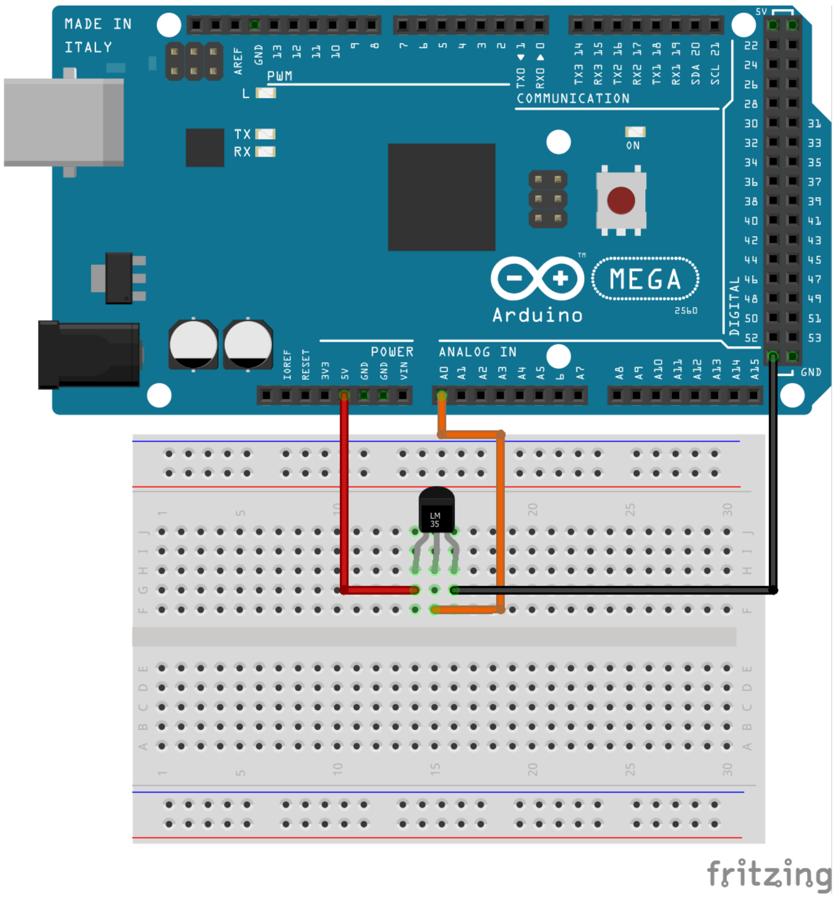


Figure 167: Scheme of hardware setup for program presented in figure 166.

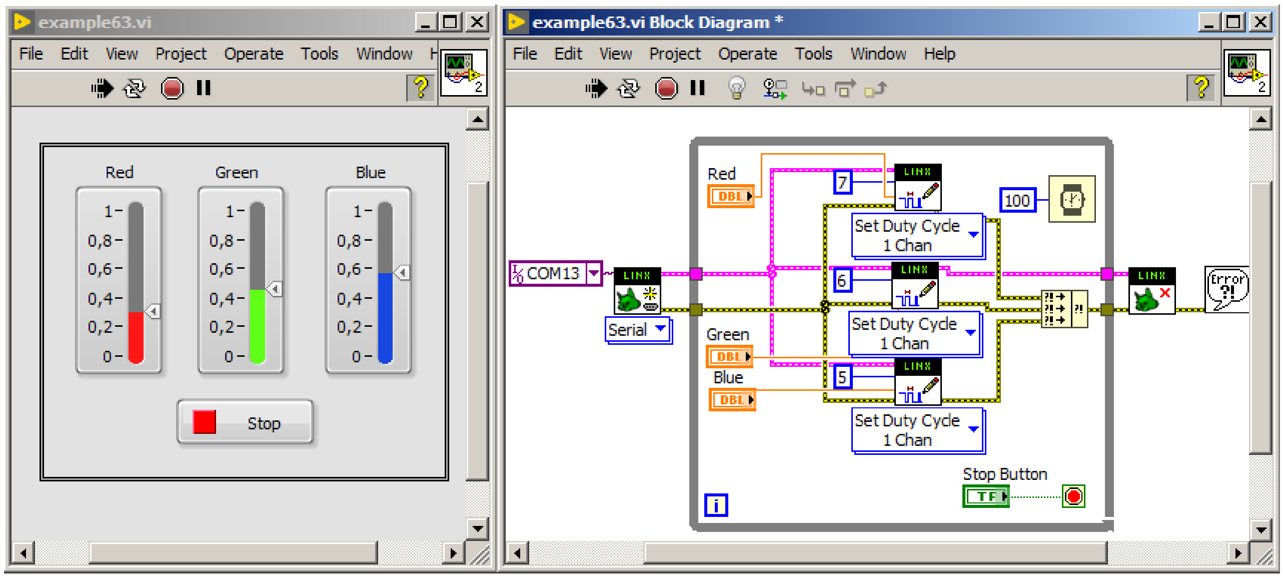


Figure 168: Program that allows duty cycle changing of RGB diode colors with help of LINX library.

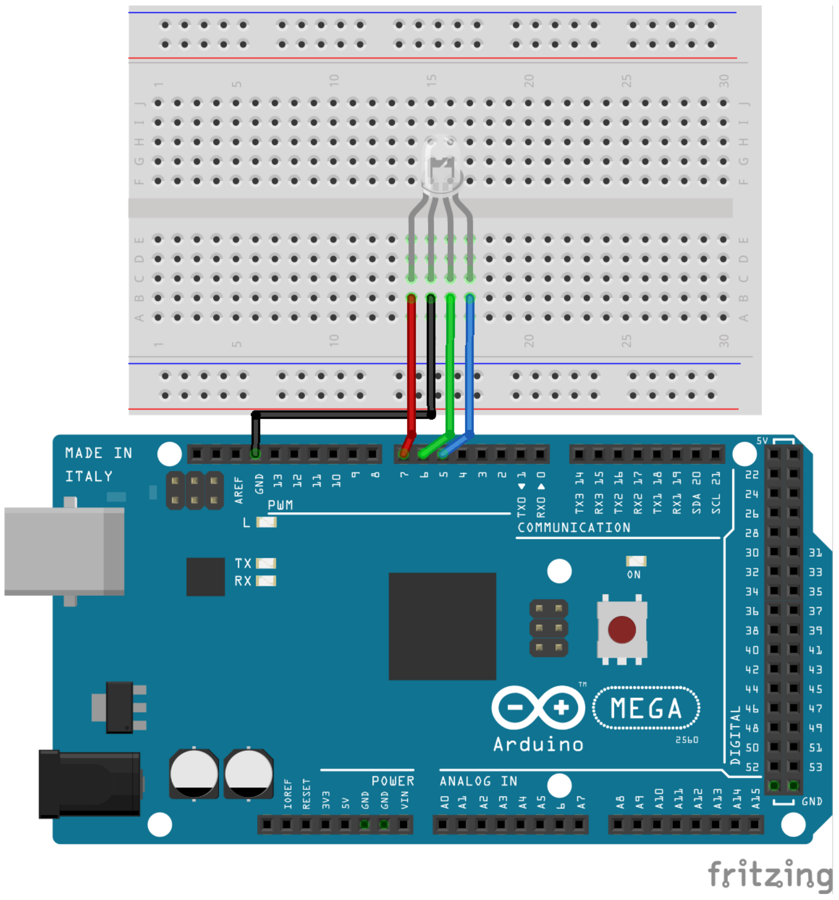


Figure 169: Scheme of hardware setup for program presented in figure 168.

sented in figure 144. After starting the program, servomotor changes its angle according to provided duty cycle value.

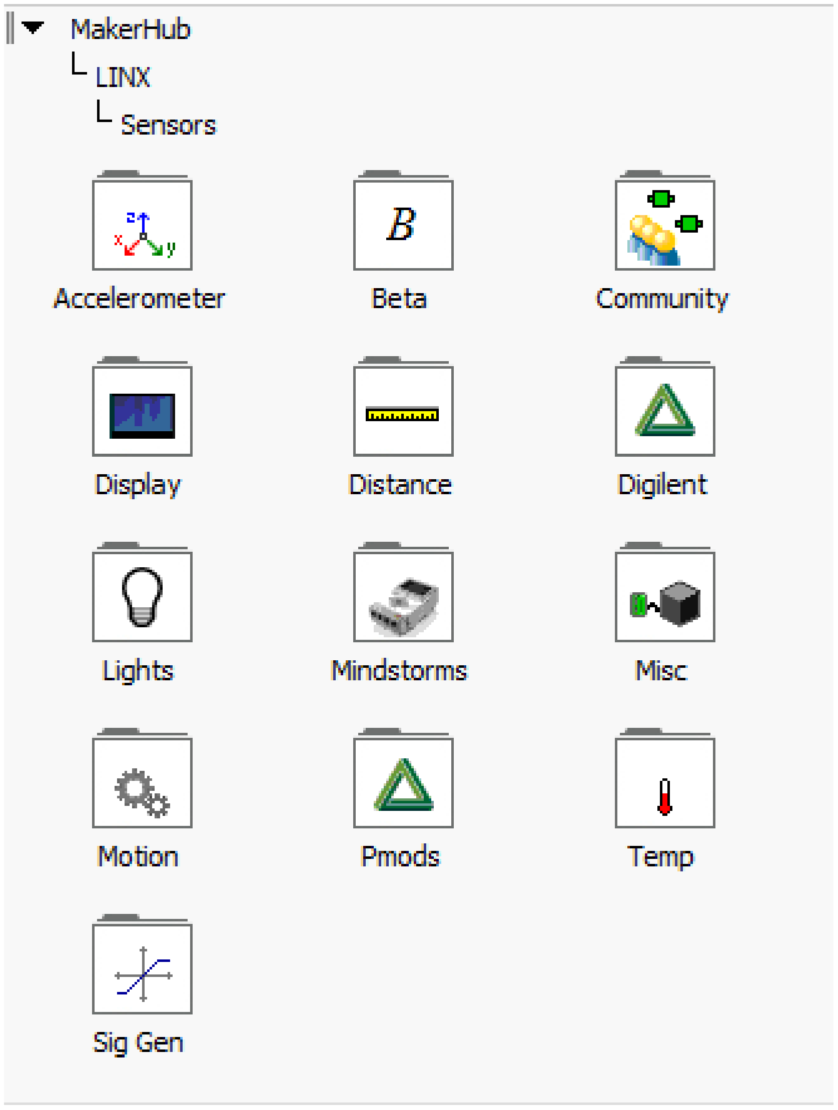


Figure 170: List of available folds with functions for sensors and actuators control in LINX library.

Using LINX library, it is easy also to control DC motor. The DC motor connection to Arduino board is presented in figure 173. Program to control DC motor is presented in figure 172. The DC motor rotation speed is controlled by PWM signal duty cycle with help of *PWM Set Duty Cycle* function. Rotation direction is controlled by digital pin, so we use 0 - False, and 1 - True, Boolean states.

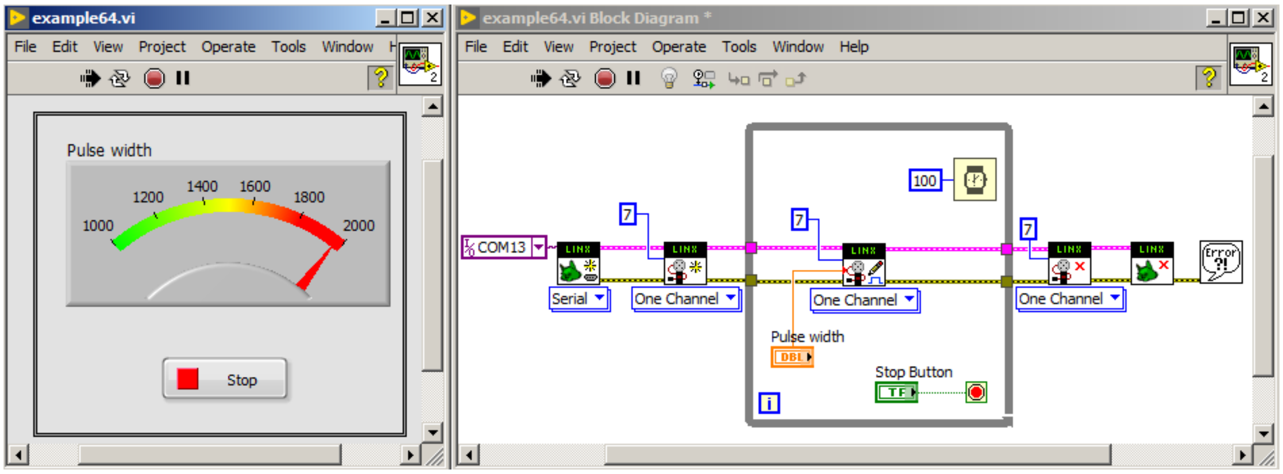

example64.vi
File Edit View Project Operate Tools Window
Pulse width
1000
1200
1400
1600
1800
2000
Stop
example64.vi Block Diagram *
File Edit View Project Operate Tools Window Help
COM13
7
100
Serial
One Channel
Pulse width
Stop Button
Error

Figure 171: Program that allows servo-motor angle changing with help of LINX library.

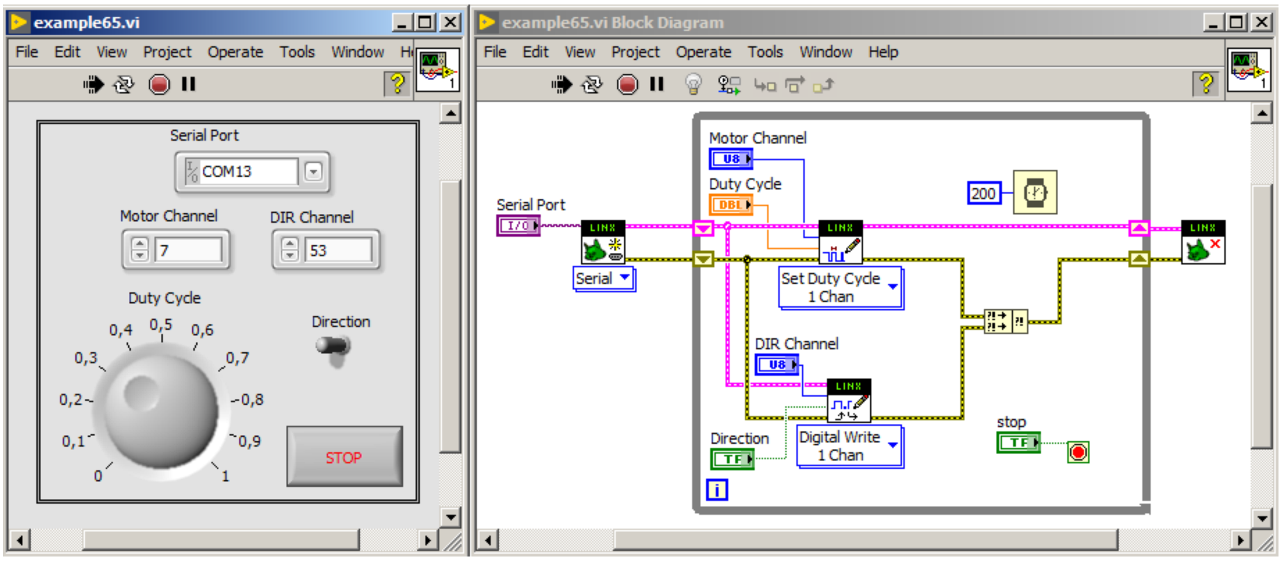


Figure 172: Program that controls speed of DC motor and controls direction of rotation with use of LINX library.

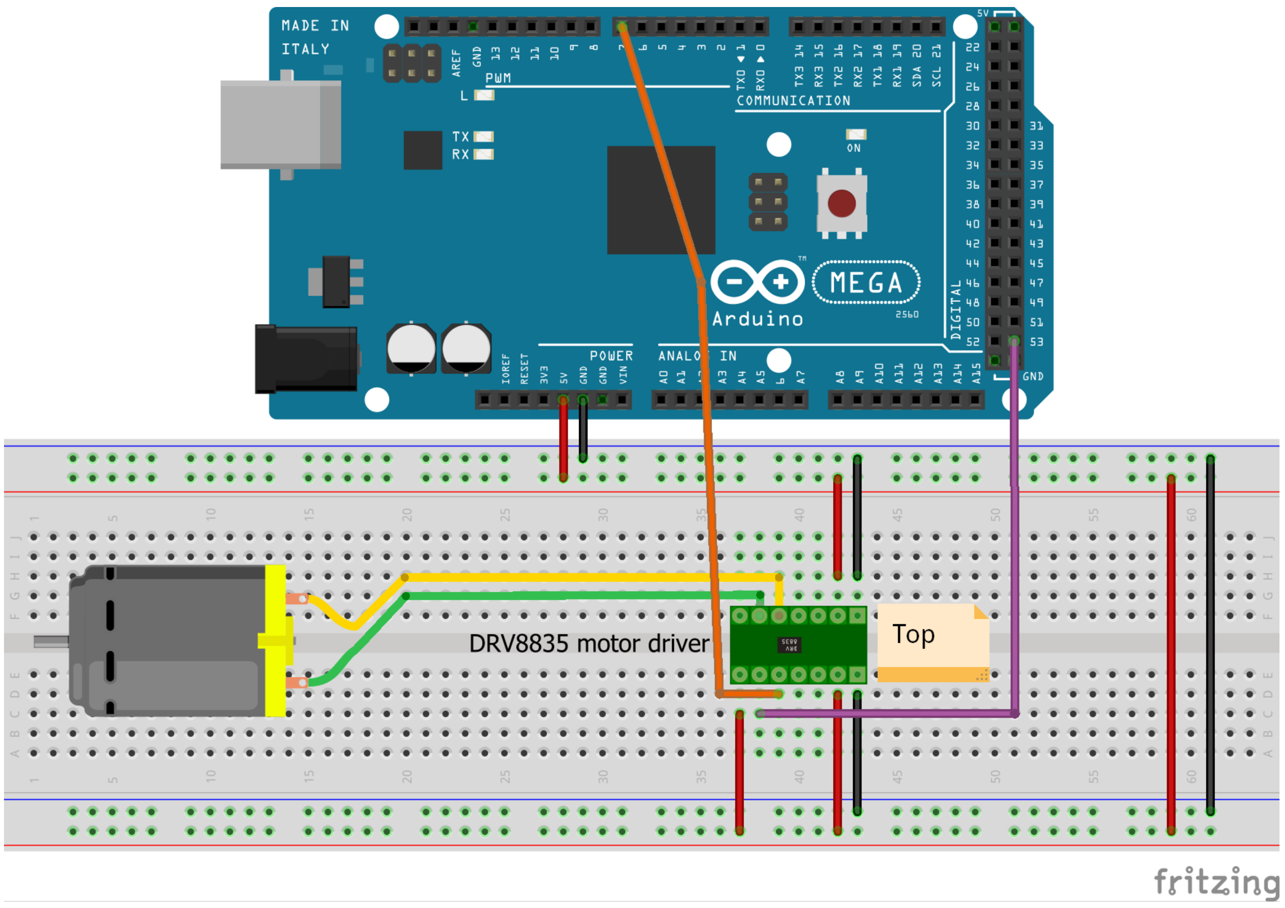


Figure 173: Scheme of hardware setup for program presented in figure 172.

LINX controlled Arduino project

In this chapter a project for controlling weather station is presented. Pressure, temperature and humidity is read from sensors. with use of LINX library. Previous attempt was prepared with use of LIFA library. Now we will look how to use the LINX library for this. In figures 174,175,176,177 new version of project is presented with use of LINX library. The difference between both versions of project lies initialization functions, read and write functions to I^2C bus. Functions *I2C Init*, *I2C Write* and *I2C Read* from LIFA library are replaced by functions *I2C Open*, *I2C Write* and *I2C Read* from LINX library. Moreover three subVIs presented in figure 177 were modified. First of them configures humidity sensor and it is presented in figure 178. Second subVI reads data necessary to perform interpolation of humidity results (calibrated read) and it is presented in figure 179. The last subVI reads measured humidity value and performs interpolation, which is presented in figure 180. Modifications of subVIs also consisted only on replacement of functions controlling I^2C bus.

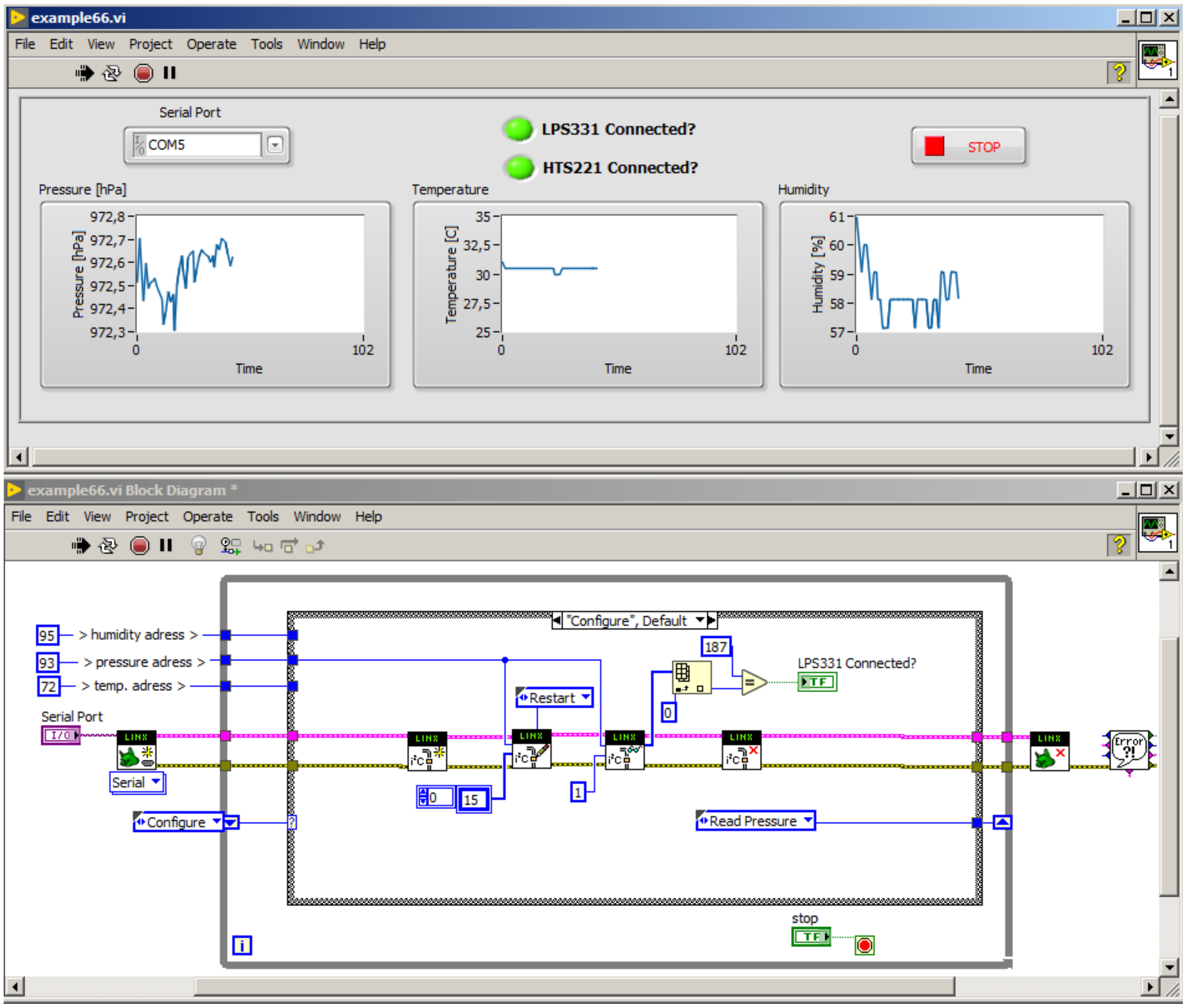


Figure 174: Project of weather station realized with use of LINX library. *Configure* state of state machine is presented here.

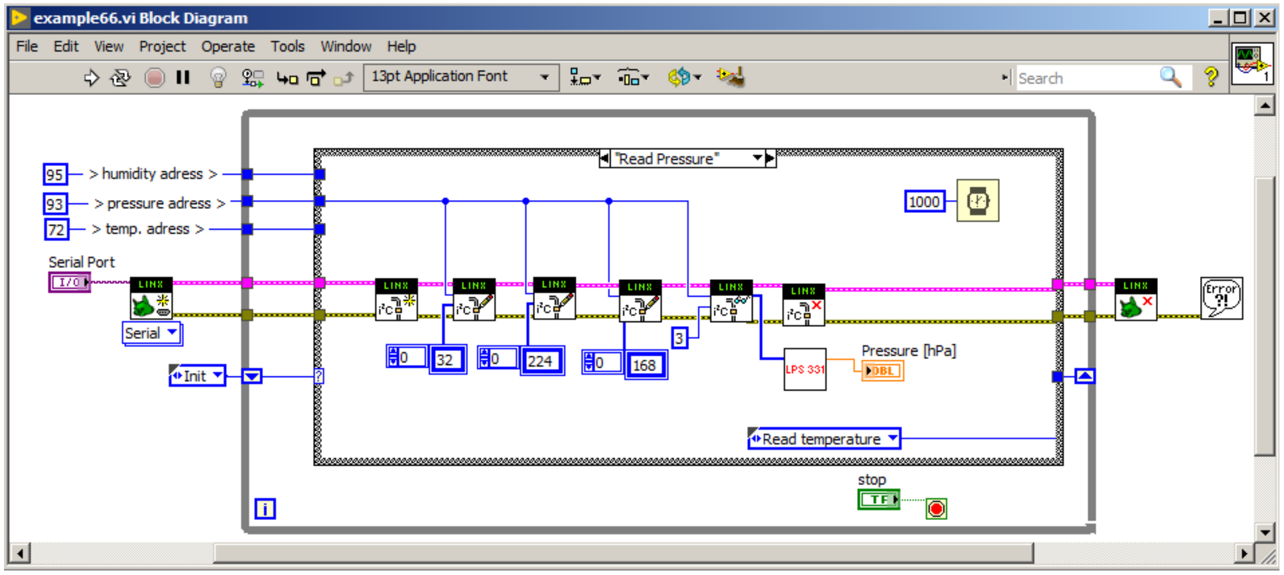


Figure 175: Project of weather station realized with use of LINX library. *Read pressure* state of state machine is presented here.

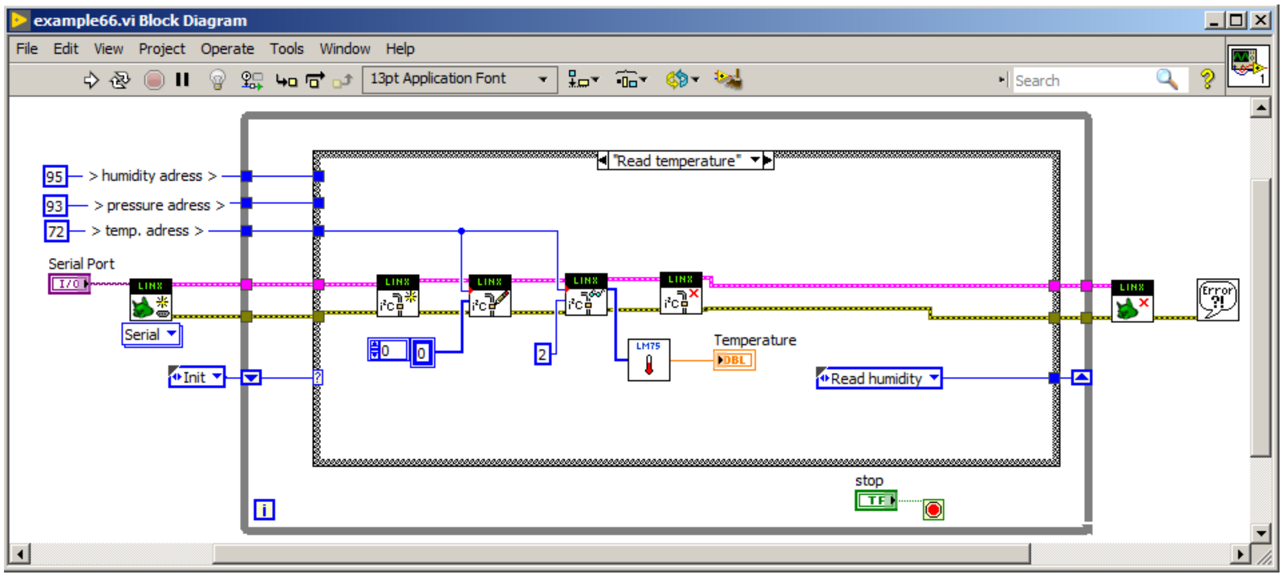


Figure 176: Project of weather station realized with use of LINX library. *Read temperature* state of state machine is presented here.

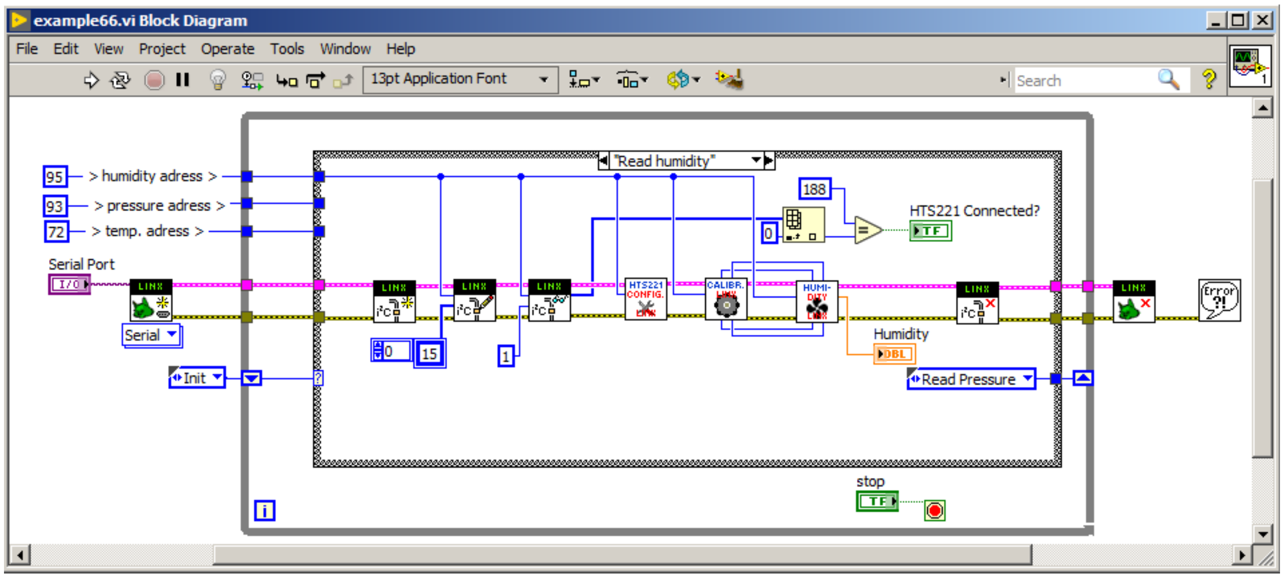


Figure 177: Project of weather station realized with use of LINX library. *Read humidity* state of state machine is presented here.

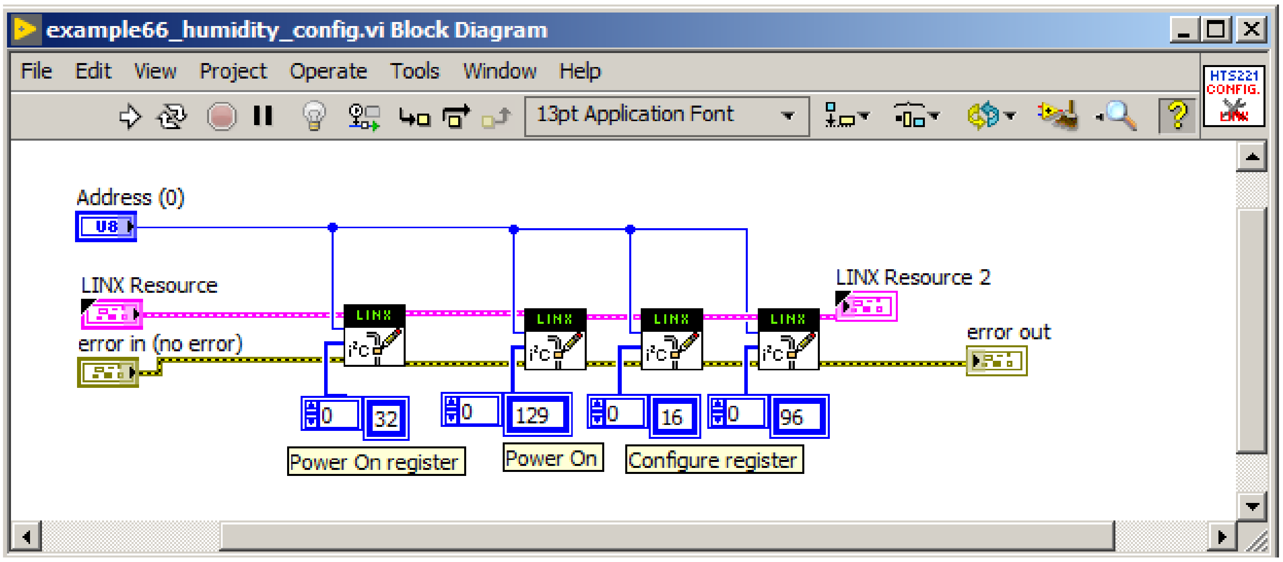


Figure 178: SubVI used to configure humidity sensor with use of LINX library.

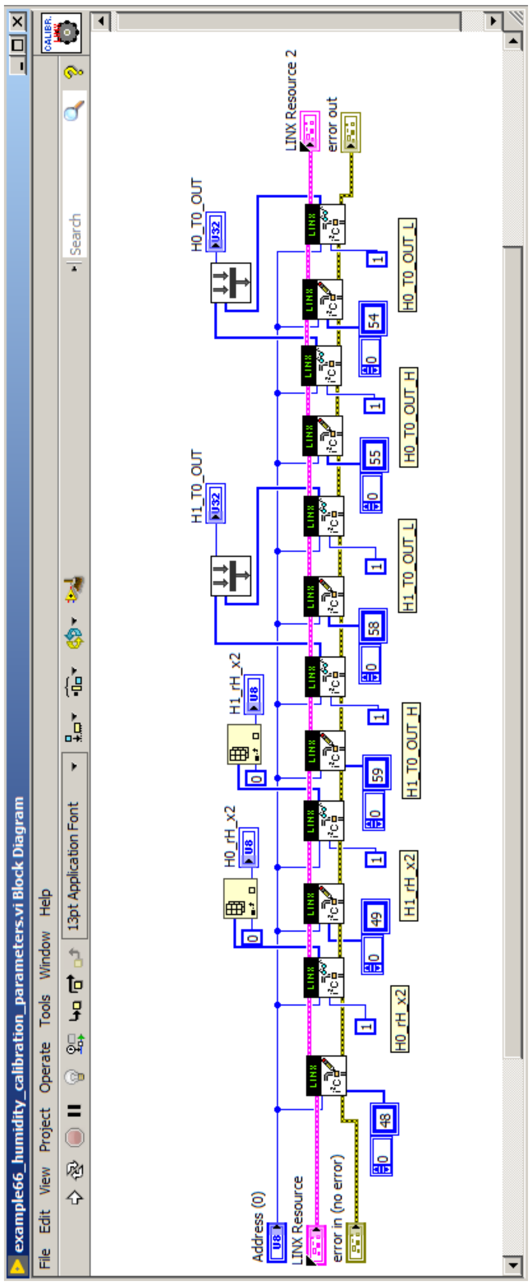


Figure 179: SubVI used to read data necessary for performing interpolation of humidity value (calibrated read) with use of LINX library.

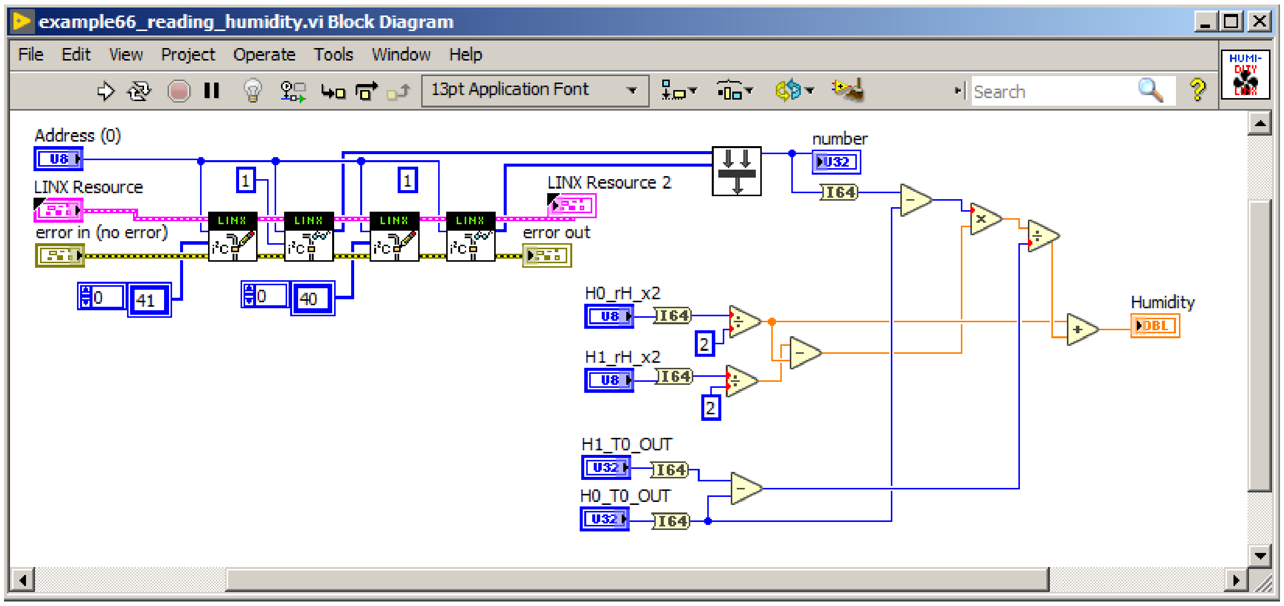


Figure 180: SubVI used for reading measured value of humidity and performing results interpolation (calibrated read) with use of LINX library.

LINX controlled Raspberry PI

LINX library can also be used to control Raspberry PI single board computer. This library will work flawlessly with Raspberry PI with **National Instruments LabVIEW 2014 32-bit version**. There is no official support for later versions. However some people managed to use also other versions, but this needs many tweaking. The same problem is for hardware part: **Raspberry PI 2 is supported**, but with later Raspberry PI models there are problems. People get around this by making modifications in library LabVIEW code, where processor type is checked (Raspberry PI 3 and 2 have different processors). If you think about going this way, be warned, that it is a bumpy road. Unfortunately, it feels that this library was only prepared for support of commercially available sets: LabVIEW Home 2014 + Raspberry PI 2B. This is visible on site: `https://www.labviewmakerhub.com/doku.php?id=learn:tutorials:libraries:linx:3-0`. The same is for another platform: Beagle Bone Black, which was also supported. Currently there is probably no interest in further development of this library. It is a pity, that this is fading out, as these libraries used interesting solution. Actually, on the Raspberry PI, LabVIEW for ARM is running and applications created in LabVIEW on PC are just deployed on this architecture. This is in contrast to Arduino, where LabVIEW program runs on PC platform and communicates with Arduino via USB-Serial connection. In case of Raspberry PI, PC could be detached, and program will still operate on Raspberry PI.

Simple Application on Raspberry PI

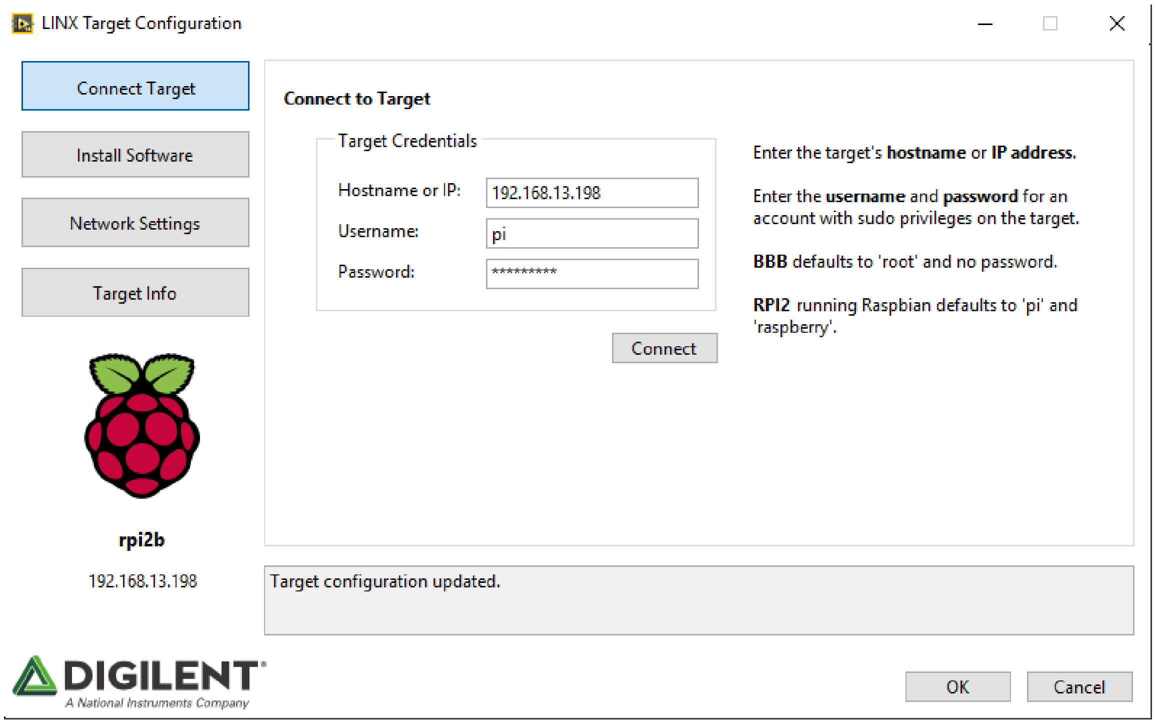
To test LINX library with Raspberry PI, LabVIEW and LINX library, we need first to make preparations. We need:

- National Instruments LabVIEW 2014
- Raspberry PI 2B
- micro SD card with installed Raspbian GNU/Linux operating system, version: 2017-04-10-raspbian-jessie.zip Later versions were reported to not cooperate.

As for Raspbian OS images, older versions can be downloaded from site: `http://downloads.raspberrypi.org/raspbian/images/`. So there is the file 2017-04-'0-raspbian-jessie.zip available to download. This file should be unzipped, then resulting image file should be written to SD card using special software which writes images. Such software is balenaEtcher `https://www.balena.io/etcher/`. This should work on Linux, Mac and Windows. There are many software with such capabilities. On Windows, e.g. Win32 Disk Imager can be used `https://sourceforge.net/projects/win32diskimager/files/latest/download`. On Linux, command line tool **dd** can also be used. However pay attention to which drive you are starting to write image, as in case of mistake you can overwrite your hard disk, which means that everything on disk, the whole system will be not usable any more.

Raspberry PI 2B should be connected to the Ethernet, where PC with LabVIEW is also connected and ssh service on Raspberry PI should be enabled. On the beginning if you can connect Raspberry PI to display via HDMI port and also connect USB keyboard to Raspberry PI, then enabling of ssh service is possible in graphical mode by launching *Raspberry Pi Configuration* from the *Preferences* menu, or by using command line (terminal) application: *raspi-config*. If you have no display available - working in "headless mode", then first read the SD card boot partition on PC computer. This small partition intended for system startup has FAT32 file system, so it is visible either on Windows or Linux systems. To enable ssh just create empty file named "ssh". Unmount SD card from PC and put it into Raspberry PI, SD card slot. When Raspberry PI starts, it looks for "ssh" file and if it founds it, the file is deleted and ssh is enabled. For more details how to enable *ssh*, look at `https://www.raspberrypi.org/documentation/remote-access/ssh/`. Next we should know the IP address of Raspberry PI. This could be found on your router, if DHCP service gave the address to Raspberry PI. If you don't have DHCP service, you should manually set IP address on Raspberry PI. This can be done via editing /etc/dhcpd.conf file on root partition of Raspberry PI. If you cannot use display and keyboard with Raspberry PI, then you rather need Linux system on PC, because this partition file system is EXT4 - typical file system for GNU/LINUX systems. See https://learn.sparkfun.com/tutorials/headless-raspberry-pi-setup/ethernet-with-static-ip-address.

If Raspberry PI was configured as described above, we need to perform some steps from LabVIEW side on the PC. First we need to install LINX library. This can be done via VIPM. Then we can start LabVIEW 2014 and then open the Target Configuration Wizard, going through menu: Tools → MakerHub → LINX → Target Configuration. Then we should provide IP address of Raspberry PI, username and password and finally click "connect". If nothing was changed, then default username in Raspberry PI is: **pi**. Default password is: **raspberry**. This is visible in figure 181.



Connect to Target

Target Credentials

Hostname or IP: 192.168.13.198

Username: pi

Password: *********

Enter the target's **hostname** or **IP address**.

Enter the **username** and **password** for an account with sudo privileges on the target.

BBB defaults to 'root' and no password.

RPI2 running Raspbian defaults to 'pi' and 'raspberry'.

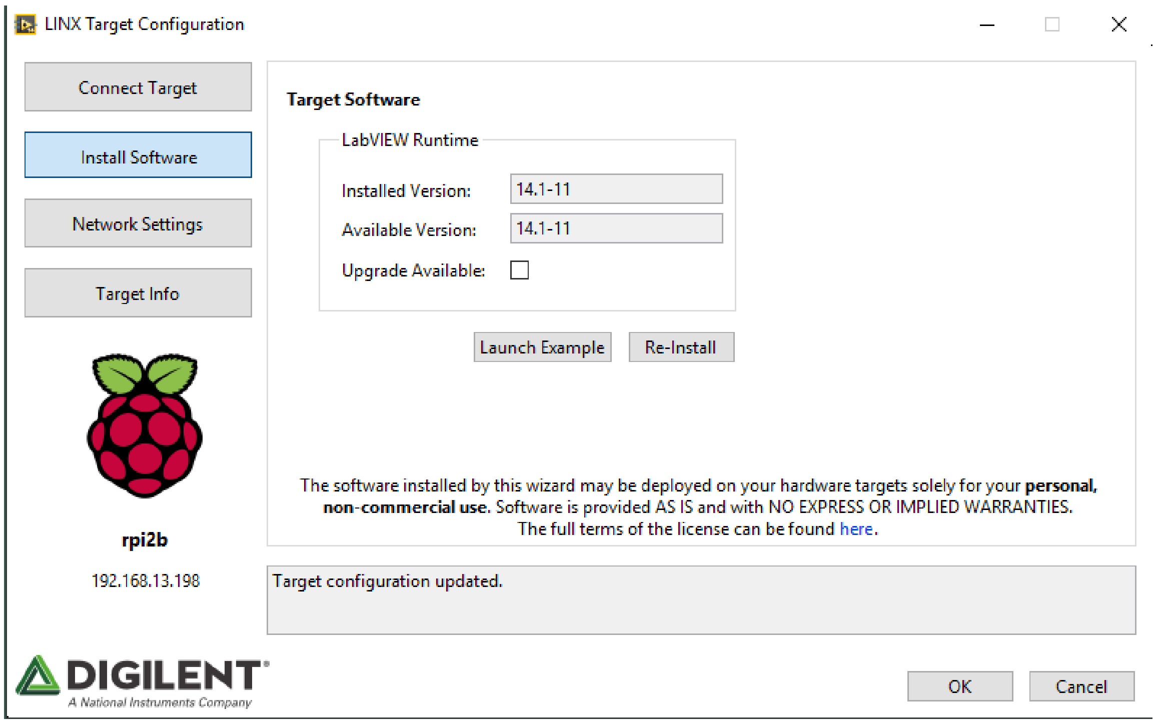
Target configuration updated.

Figure 181: Making a connection to target device inside LINX Target Configuration.

If connection was established, then we need to select *Install software* tab and click *Install* button. If software was installed already, the button will change name to *Re-Install*. This is presented in figure 182.

If everything finished without errors we are ready to prepare test application. First it will be simple blinking led "hello world" application, then we will improve it for some control from LabVIEW panel. Finally this could be a basic stroboscope control application.

In figure 183 a ready "hello world" application, which just blink led is presented. Here are three important parts: on the left - a project window, on top right - the front panel, on bottom right, the block



Target Software

LabVIEW Runtime

Installed Version: 14.1-11

Available Version: 14.1-11

Upgrade Available: ☐

The software installed by this wizard may be deployed on your hardware targets solely for your **personal, non-commercial use.** Software is provided AS IS and with NO EXPRESS OR IMPLIED WARRANTIES. The full terms of the license can be found here.

rpi2b

192.168.13.198

Target configuration updated.

Figure 182: Install Software tab in Linx Target Configuration window.

diagram.

To create and deploy application to Raspberry PI, an empty project needs to be created (File → New → Empty Project) or just at start LabVIEW window, which is presented in figure 184.

Then new VI can be created, as showed in figure 183 with use of LINX functions: open, write and close. We also need to define LINX target in the project. For that, you need to click the right mouse button the top of the project item in Project Explorer window, and select New → Targets and Devices. You can then choose *New Target or Device*, and then select Raspberry PI and put the right IP address of Raspberry PI. This is presented in figure 185.

Then you should connect to the newly created Device by clicking the right mouse button on this device (with Raspberry icon) and select *Connect*. This is depicted in figure 186. If necessary you can change the IP address of Target Device using *Properties* menu item in right click menu.

If connection was successful, you shouldn't get any error window, and bright green dot should be present in bottom right of raspberry icon, which is visible in figure 187.

Now, it is important that vi in Project Explorer is located under target device in this project tree. You may need to drag&drop the vi to the target device. Then you can open the vi and run it. The you should see deployment window, which shows the progress of deployment to

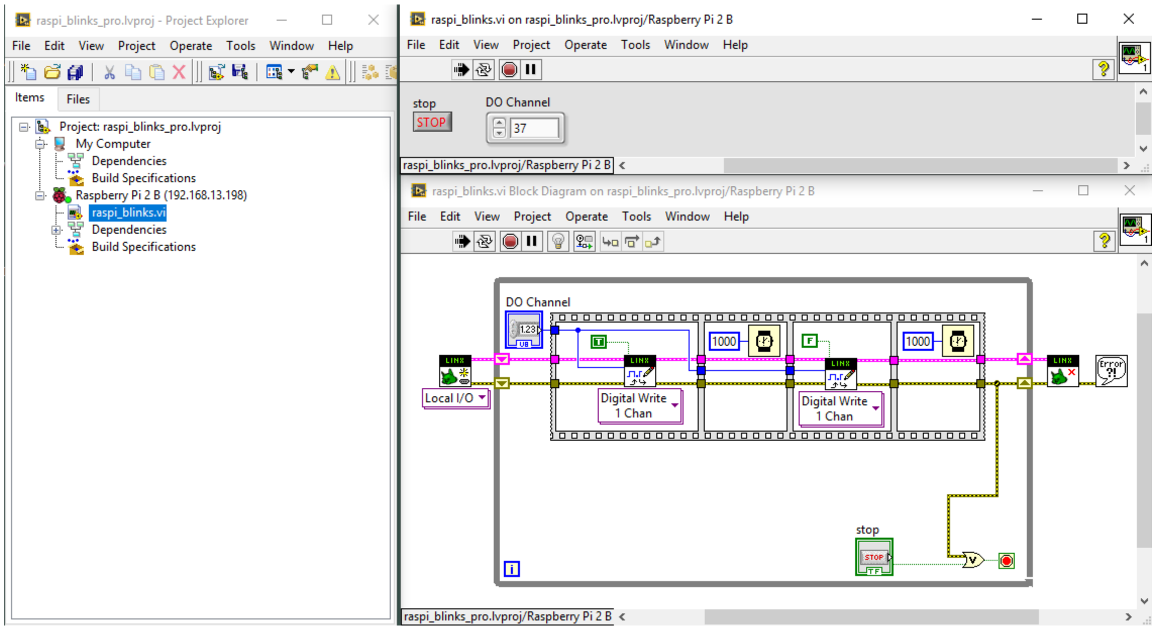
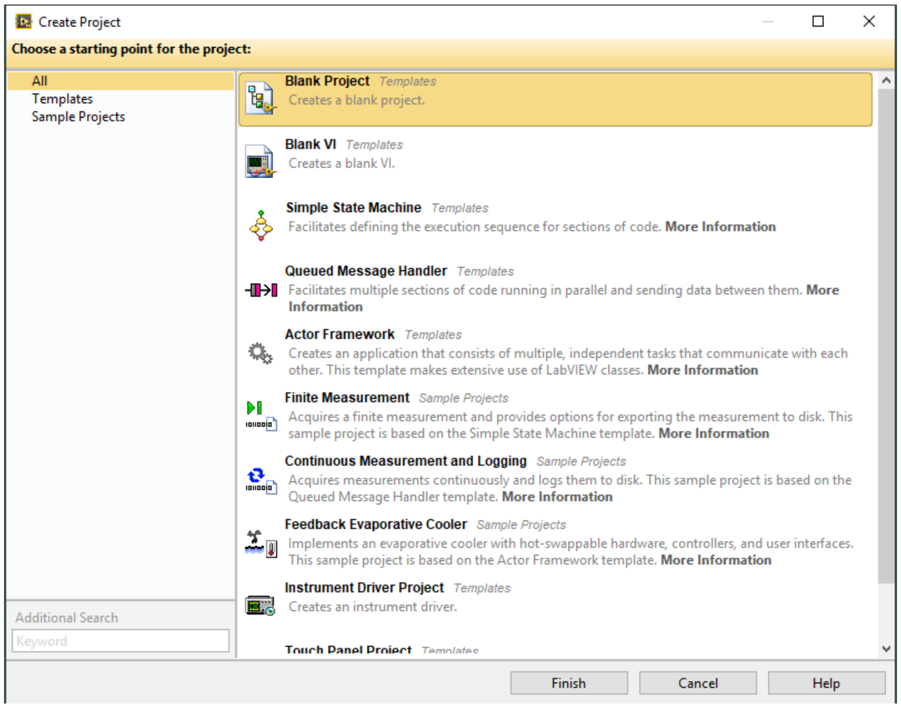

raspi_blinks_pro.lvproj - Project Explorer
File Edit View Project Operate Tools Window Help
Items
Files
Project: raspi_blinks_pro.lvproj
My Computer
Dependencies
Build Specifications
Raspberry Pi 2 B (192.168.13.198)
raspi_blinks.vi
Dependencies
Build Specifications
raspi_blinks.vi on raspi_blinks_pro.lvproj/Raspberry Pi 2 B
stop
STOP
DO Channel
37
raspi_blinks_pro.lvproj/Raspberry Pi 2 B
raspi_blinks.vi Block Diagram on raspi_blinks_pro.lvproj/Raspberry Pi 2 B
DO Channel
Local I/O
Digital Write 1 Chan
1000
Digital Write 1 Chan
1000
stop
raspi_blinks_pro.lvproj/Raspberry Pi 2 B

Figure 183: An example of LED blinking application created in LabVIEW using LINX library.


Create Project

Choose a starting point for the project:

All
Templates
Sample Projects

Blank Project *Templates*
Creates a blank project.

Blank VI *Templates*
Creates a blank VI.

Simple State Machine *Templates*
Facilitates defining the execution sequence for sections of code. **More Information**

Queued Message Handler *Templates*
Facilitates multiple sections of code running in parallel and sending data between them. **More Information**

Actor Framework *Templates*
Creates an application that consists of multiple, independent tasks that communicate with each other. This template makes extensive use of LabVIEW classes. **More Information**

Finite Measurement *Sample Projects*
Acquires a finite measurement and provides options for exporting the measurement to disk. This sample project is based on the Simple State Machine template. **More Information**

Continuous Measurement and Logging *Sample Projects*
Acquires measurements continuously and logs them to disk. This sample project is based on the Queued Message Handler template. **More Information**

Feedback Evaporative Cooler *Sample Projects*
Implements an evaporative cooler with hot-swappable hardware, controllers, and user interfaces. This sample project is based on the Actor Framework template. **More Information**

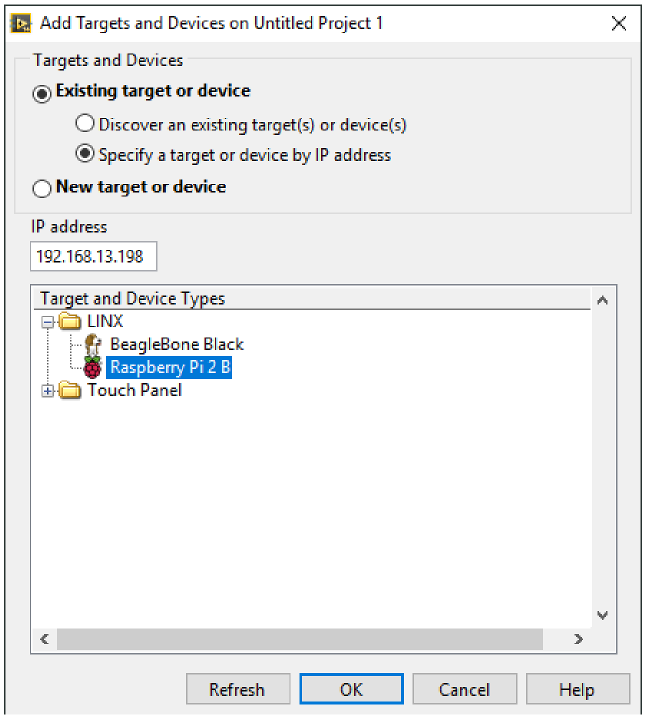
Instrument Driver Project *Templates*
Creates an instrument driver.

Touch Panel Project *Templates*

Additional Search
Keyword

Finish Cancel Help

Figure 184: Blank project creation.


Add Targets and Devices on Untitled Project 1
Targets and Devices
Existing target or device
Discover an existing target(s) or device(s)
Specify a target or device by IP address
New target or device
Target and Device Types
LINX
BeagleBone Black
Raspberry Pi 2 B
Touch Panel
Refresh
OK
Cancel
Help

IP address
192.168.13.198

Figure 185: Creation of Raspberry PI Target device.

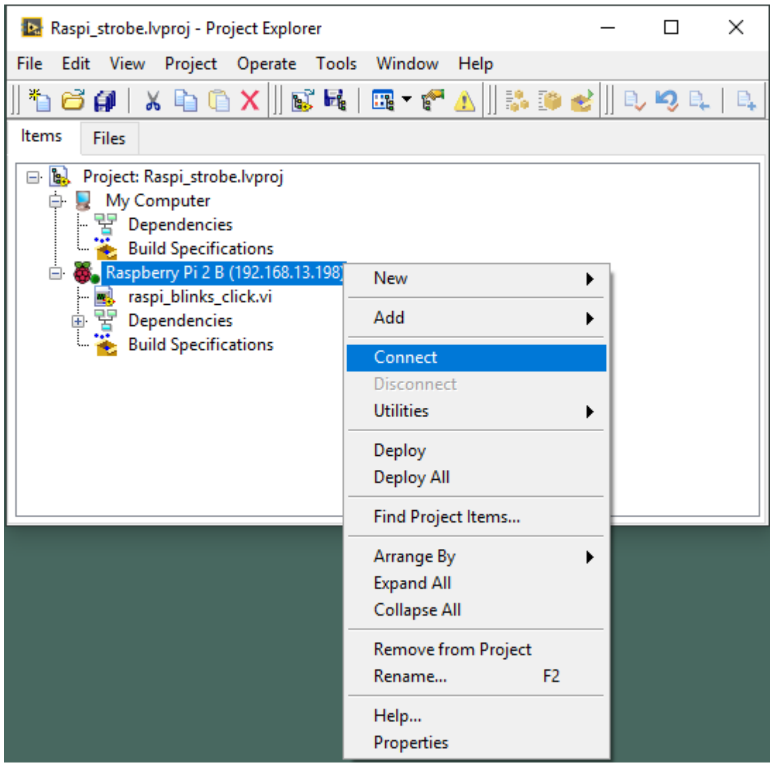
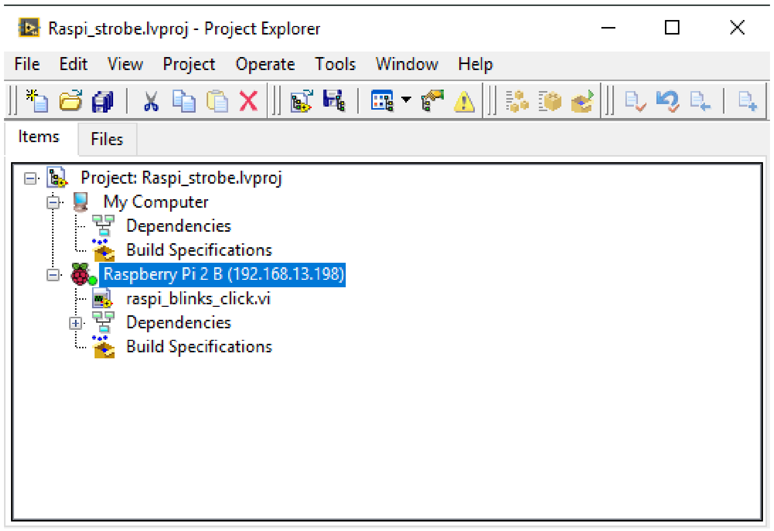


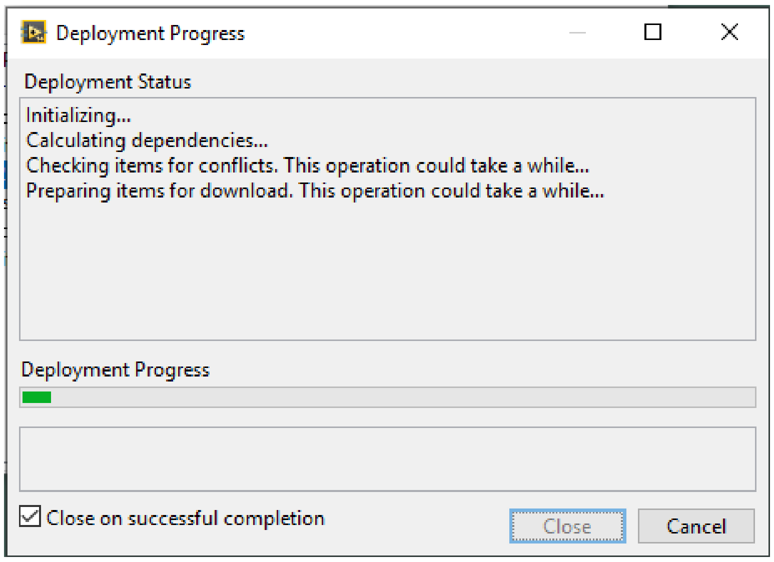
Figure 186: Make a connection to the target device in Project Explorer window.


Raspi_strobe.lvproj - Project Explorer
File Edit View Project Operate Tools Window Help
Items Files

Project: Raspi_strobe.lvproj
My Computer
Dependencies
Build Specifications
Raspberry Pi 2 B (192.168.13.198)
raspi_blinks_click.vi
Dependencies
Build Specifications

Figure 187: Status of the connection to the target device. IP number and bright green dot in bottom right part of raspberry icon are visible.

the target device. This is shown in figure 188. If everything is OK, then application should run, and if you connected LED with resistor to the 37 pin of 40-pin I/O connector on Raspberry PI, then this LED should blink in 50% duty cycle of 2 seconds period, according to the block diagram in figure 183.



Initializing...
Calculating dependencies...
Checking items for conflicts. This operation could take a while...
Preparing items for download. This operation could take a while...

Figure 188: Deploying an application to the target device.

Stroboscope controller application

Next let simply modify the blinking led application to be more professional tool and make it a stroboscope controller. In this modification we just add controls to front panel, just to allow user to change the time of ON and OFF states. For this purpose two dials were added, one which sets LED ON state duration and second which sets LED OFF state duration. Front panel is visible in figure 189.

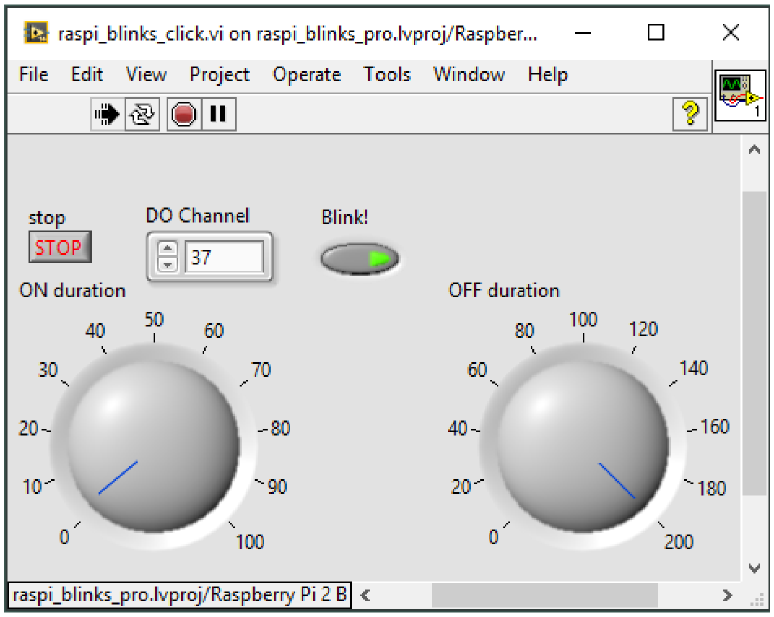

raspi_blinks_click.vi on raspi_blinks_pro.lvproj/Raspber...
File Edit View Project Operate Tools Window Help
stop
STOP
DO Channel
37
Blink!
ON duration
0 10 20 30 40 50 60 70 80 90 100
OFF duration
0 20 40 60 80 100 120 140 160 180 200
raspi_blinks_pro.lvproj/Raspberry Pi 2 B

Figure 189: Front panel of stroboscope controller application.

Modified block diagram is presented in figure 190. Controls, added to the front panel are connected to the wait functions, which are located between LINX write functions on flat sequence.

After running this application, by pressing right arrow button in menu, again we will see application deployment, presented in figure 191.

When application is running after deployment, user can interactively change parameters of duration of light and dark parts in stroboscope cycle by rotating the dials on the front panel.

From this point, creation of any application with use of LINX library should be possible for the reader, whether he will use Arduino Boards or Raspberry PI. LINX library offers to easy use of different sensors and actuators. Just review the tools palette.

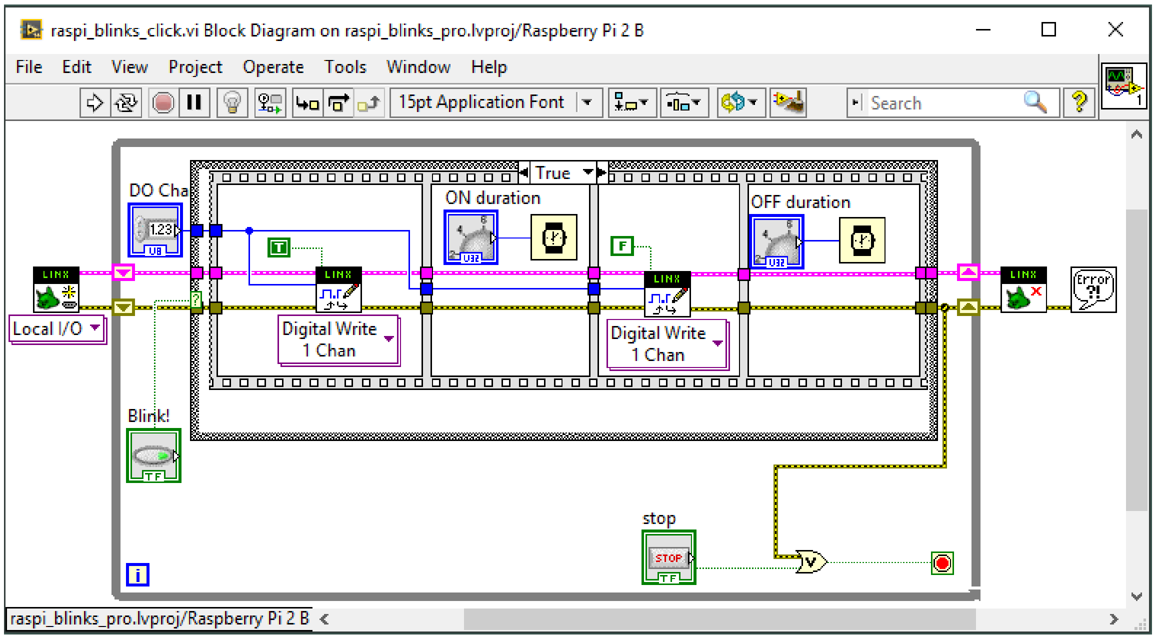
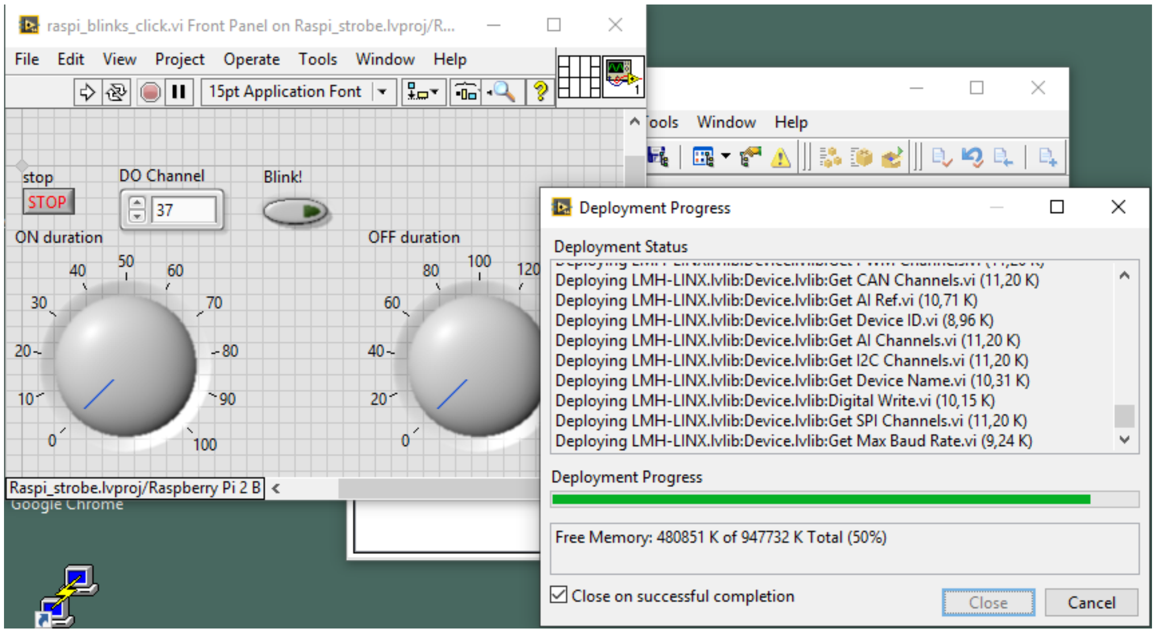

raspi_blinks_click.vi Block Diagram on raspi_blinks_pro.lvproj/Raspberry Pi 2 B
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
DO Cha
True
ON duration
OFF duration
LINX
Local I/O
Digital Write 1 Chan
Digital Write 1 Chan
Error
Blink!
stop
STOP
raspi_blinks_pro.lvproj/Raspberry Pi 2 B

Figure 190: Block diagram of stroboscope controller application.


raspi_blinks_click.vi Front Panel on Raspi_strobe.lvproj/R...
File Edit View Project Operate Tools Window Help
15pt Application Font
stop
STOP
DO Channel
37
Blink!
ON duration
OFF duration
Raspi_strobe.lvproj/Raspberry Pi 2 B
Google Chrome
Tools Window Help
Deployment Progress

Deployment Status

```
Deploying LMH-LINX.lvlib:Device.lvlib:Get CAN Channels.vi (11,20 K)
Deploying LMH-LINX.lvlib:Device.lvlib:Get AI Ref.vi (10,71 K)
Deploying LMH-LINX.lvlib:Device.lvlib:Get Device ID.vi (8,96 K)
Deploying LMH-LINX.lvlib:Device.lvlib:Get AI Channels.vi (11,20 K)
Deploying LMH-LINX.lvlib:Device.lvlib:Get I2C Channels.vi (11,20 K)
Deploying LMH-LINX.lvlib:Device.lvlib:Get Device Name.vi (10,31 K)
Deploying LMH-LINX.lvlib:Device.lvlib:Digital Write.vi (10,15 K)
Deploying LMH-LINX.lvlib:Device.lvlib:Get SPI Channels.vi (11,20 K)
Deploying LMH-LINX.lvlib:Device.lvlib:Get Max Baud Rate.vi (9,24 K)
```

Deployment Progress

Free Memory: 480851 K of 947732 K Total (50%)

☑ Close on successful completion

Close Cancel

Figure 191: Deployment of stroboscope controller application.