

LabVIEW basics

BME MIT 2007.

© János Hainzmann, Károly Molnár, Balázs Scherer, Csaba Tóth

Table of contents

| | |
|---|----|
| REFERENCES..... | 1 |
| 1. INTRODUCTION..... | 2 |
| 1.1 VIRTUAL INSTRUMENTATION..... | 2 |
| 1.2 VISUAL PROGRAMMING..... | 2 |
| 2. GETTING STARTED..... | 4 |
| 2.1 STARTING LABVIEW, RUNNING A VI, BASIC NOTATION..... | 4 |
| 2.2 TOOLBARS..... | 5 |
| 2.3 CREATING A SIMPLE VI..... | 6 |
| 2.4 CONTROL STRUCTURES AND DATA TYPES..... | 9 |
| 2.4.1 Loops..... | 9 |
| 2.4.2 Case structure..... | 12 |
| 2.5 OTHER IMPORTANT LABVIEW ELEMENTS..... | 12 |

REFERENCES

1. **Virtual Instrumentation**; Tutorial, National Instruments, 2006,
<http://zone.ni.com/devzone/cda/tut/p/id/4752>
2. **Virtual Instrumentation**; Wikipedia,
http://en.wikipedia.org/wiki/Virtual_instrumentation

1. INTRODUCTION

1.1 Virtual instrumentation

| | |
|----------------------|---|
| Embedded System | The use of computers and microcontrollers in instrumentation and measurement started in the 1980s. In that time, there were two main approaches in the use of computers in measurement technology. The first approach kept using the traditional instrument form, but an embedded computer was placed inside the instrument. This special purpose computer controls the output signal generation, the signal processing and handles the traditional users interface (switches, knobs, displays) as peripherals. |
| Virtual instruments | The second approach is using standard computers (e.g. a PC) with standard peripherals, which is supplemented with special purpose analog input/output devices. The functionality of the instrument is therefore realized in software running on the computer. This approach is called virtual instrumentation . |
| Programming language | Several high level programming languages and development environments have been developed to build virtual instrumentation software. For example LabWindows/CVI is an ANSI C based development environment. The use of visual programming languages are also widespread, in order to make users with little programming experience capable of building virtual instrumentation software. Such visual programming languages for virtual instrumentation are VEE or LabView. In the following we present the National Instruments LabView programming language and development environment. |

1.2 Visual programming

| | |
|----------------------|--|
| Textual programming | Electronic and software engineer students typically learn to program computers by textual programming languages. Formerly, this was BASIC or Pascal, today it's C or Java. Therefore any other programming approach learned later is compared to textual programming. |
| Visual programming | LabVIEW is a visual programming language and development environment, therefore it requires a different way of thinking than the standard procedural or object oriented programming languages. Other visual programming languages are for example Simulink (MathWorks) or VEE (Agilent). |
| Dataflow programming | A typical virtual instrumentation software has to perform the following functionality: data input, data processing, result output. The main task of LabView is therefore to describe and execute such dataflows. The standard procedural programming principles cannot be used directly. |
| “Easy” usage | Dataflow based visual programming tools (such as LabView) are very appropriate for a wide variety of tasks. Of course there are also a couple of tasks for where this approach is not appropriate, and LabView is difficult to use. There are numerous examples when a certain programming task can be done in LabView by a few clicks, but a C program that solves the same task would take hours to write. |

Of course there are also counterexamples, some task can be very straightforward in C, but very elaborate in LabView. Both approaches have its own field of application. From this point of view, LabView is just another programming language with all of its special rules and conventions, so similar amount of effort is required to learn this language as it is required for any other.

2. GETTING STARTED

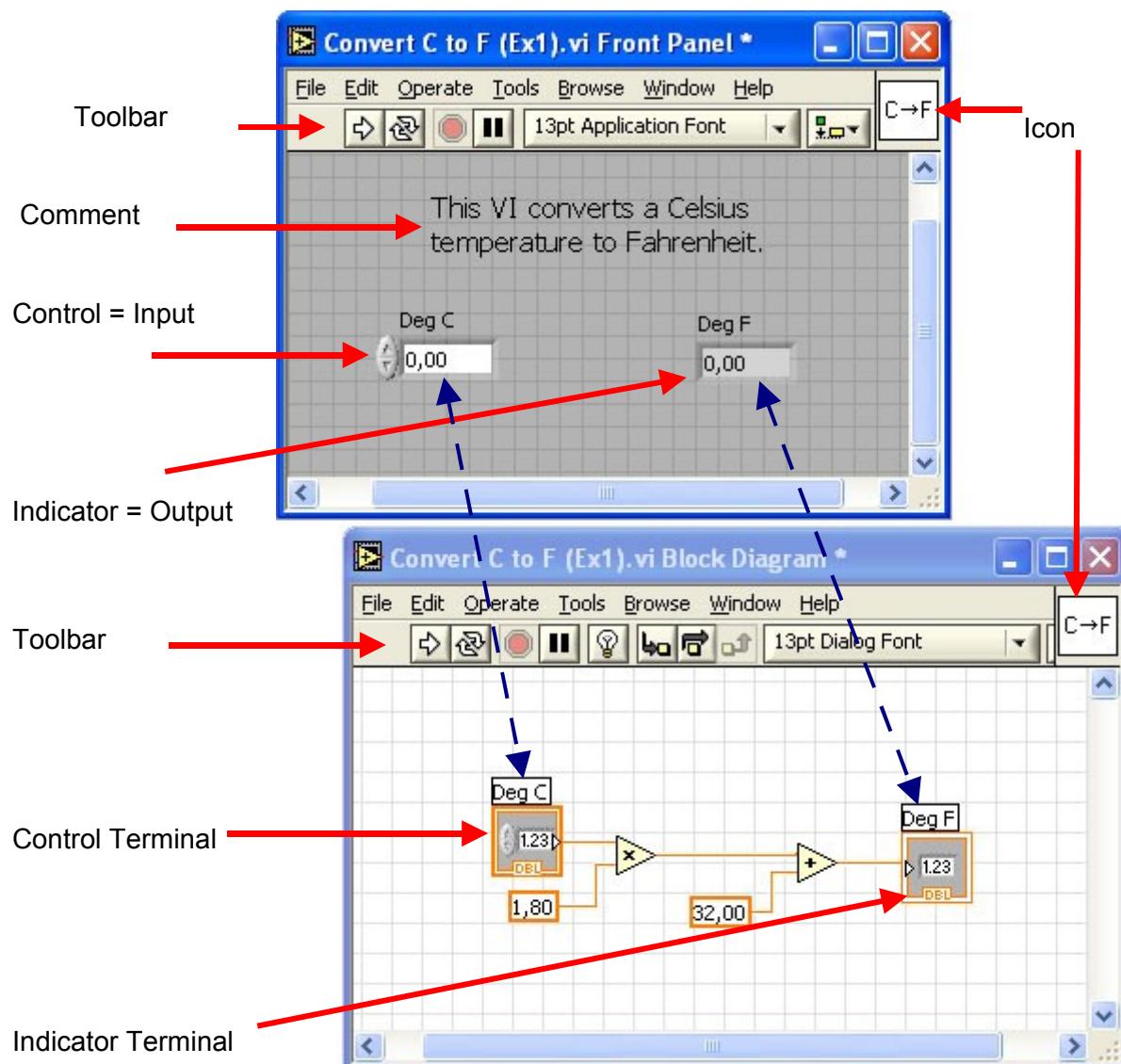
2.1 Starting LabVIEW, running a VI, basic notation

We present the basic notations in LabView by a simple example.

Program = VI

In LabVIEW, the programs are called Virtual Instruments (VI). If we open a virtual instrument source file (extension: .vi) in LabView, the Front Panel of the instrument pops up. By hitting the keystroke combination CTRL+E (or by selecting Show Block Diagram in the Window menu) the Block Diagram window is also shown.

The Front Panel and the Block Diagram of a simple VI is shown below:



Front Panel

The Front Panel of the Virtual Instrument is very simple in this case, it converts a temperature value in degree Celsius to Fahrenheit.

Control, Indicator

The example VI has one input and one output. In LabVIEW an input is called **Control**, an output is called **Indicator**. Besides this two

types there are **Constants**, which are values that don't change during the run of the program.

Block diagram

The input and output objects (Control, Indicator) and the functional blocks that make connection between them are connected in the Block Diagram. The connections are represented by "wires". The wiring describes the dataflow, i.e. the program itself. All Controls and Indicators have two different appearances, one in the Front Panel and another in the Block diagram. In the above figure we marked the related pairs by the blue dashed arrows.

Dataflow programming From every Control (input) there is a dataflow to all the Indicators (output) represented by the wiring. The execution of the Block Diagram is controlled by these dataflows. Generally, as a convention the dataflows are drawn from left to right, but the execution does not depend on the drawing, but only on the order of the objects in the diagram. One functional object executes whenever all of its input values are already known. After execution, the functional object puts out the results on all of its outputs.

Run



This button in the Toolbar starts the run of the VI. (During execution the arrow transforms to a broken black arrow.) The VI is run only once.

Continuous Run, Stop



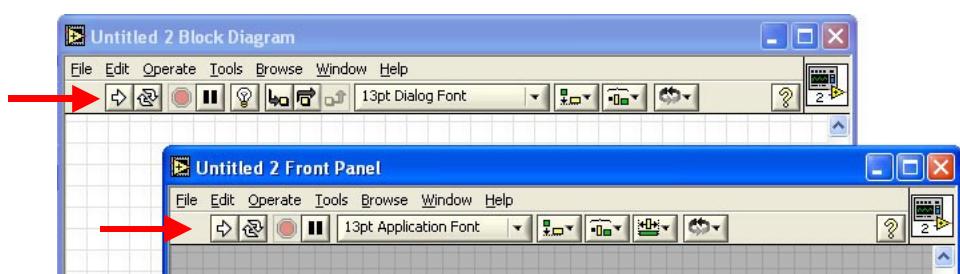
Clicking on the first icon will run the VI continuously, the second stops the running.

Execution Highlighting If this icon is selected, then the dataflow is visualized by little tokens moving on the wires, and the values are shown next to the wires. This is a handy tool to understand how a VI works, and also suitable for debugging purposes



2.2 Toolbars

The Block Diagram and the Front Panel has a very similar Toolbar.



The Toolbar of the Block Diagram consist of the following icons:



Run / Running / Syntax error



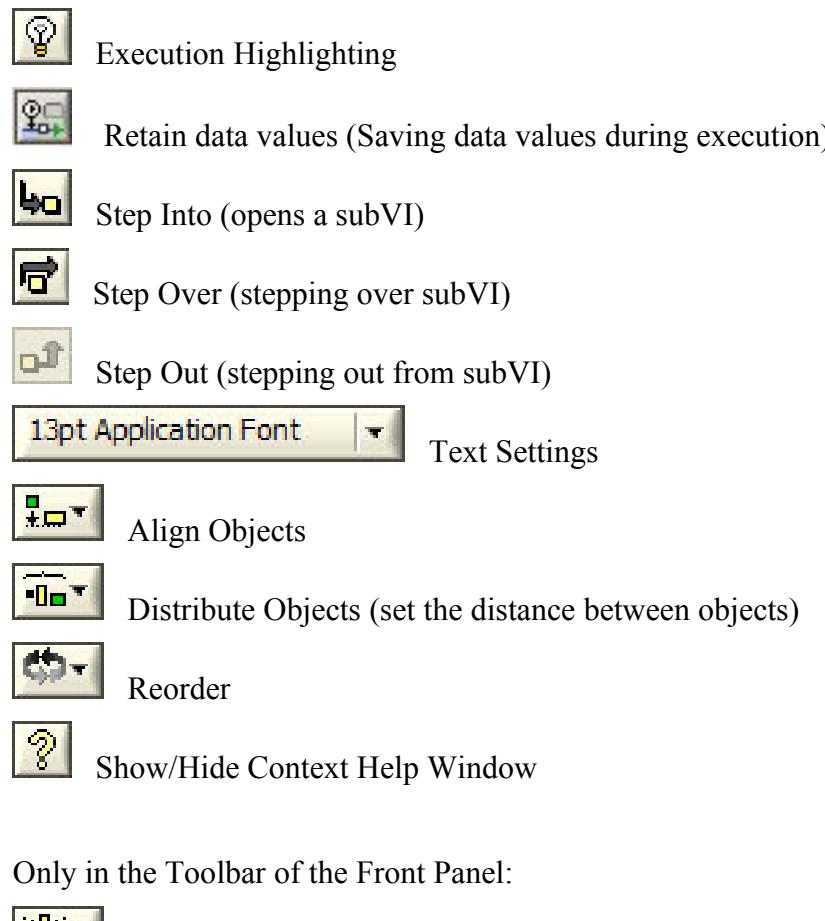
Continuous Run



Abort Execution



Pause/Continue



Only in the Toolbar of the Front Panel:



2.3 Creating a simple VI

After the introduction to basic LabView notation, we continue by writing a simple VI. We write the pair of the Celsius – Fahrenheit converter, the Fahrenheit – Celsius converter. The algorithm is very simple: 32 is subtracted from the value given in Fahrenheit, and then the result is divided by 1.8.

New VI

After starting LabVIEW, choose Blank VI from the New group. An empty Front Panel and Block Diagram pops up (if the Block Diagram is not shown, we can invoke it by hitting CTRL+E).

Palettes

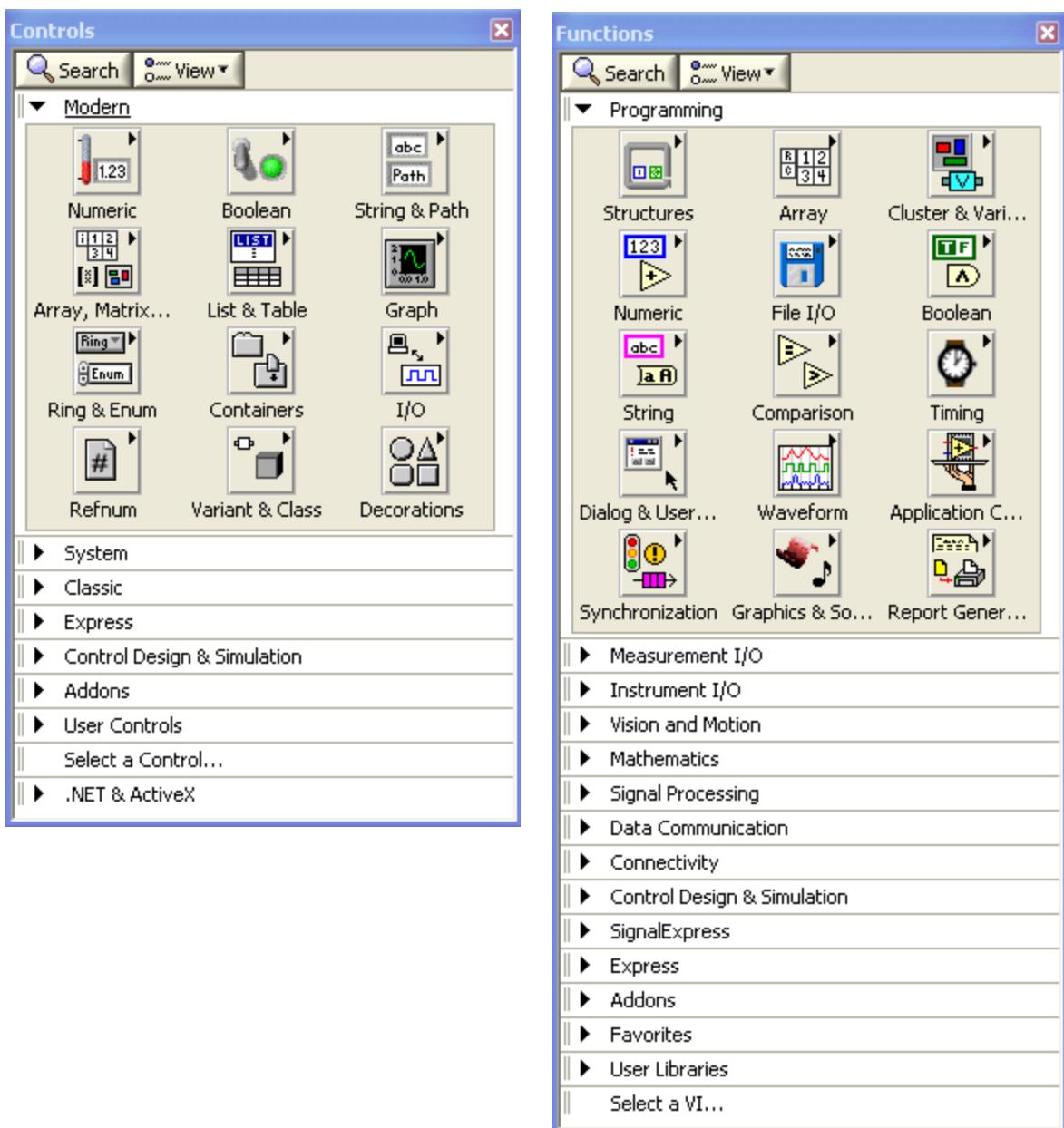
The tools for editing the VIs are organized in palettes.

Controls

The tools of the Front Panel are in the **Controls** Palette. To invoke the Controls Palette, simply right click on the Front Panel.

Functions

The tools of the Block Diagram are in the **Functions** Palette. To invoke the Functions Palette, simply right click on the Block Diagram.

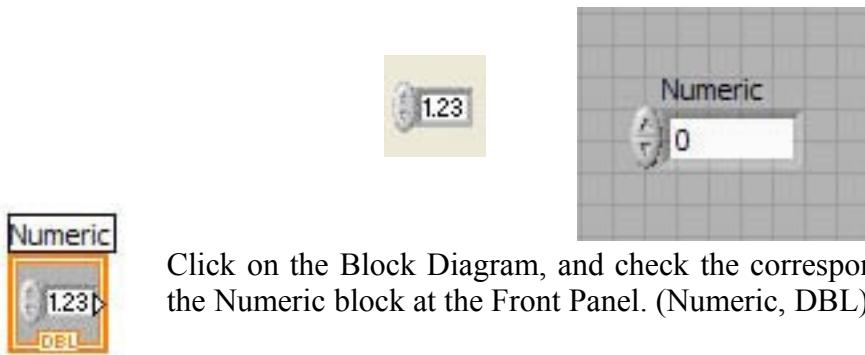


The Front Panel of our VI Let's build our Front Panel first! We have one Control (input: Fahrenheit degree) and one Indicator (output: Celsius degree).

Variables

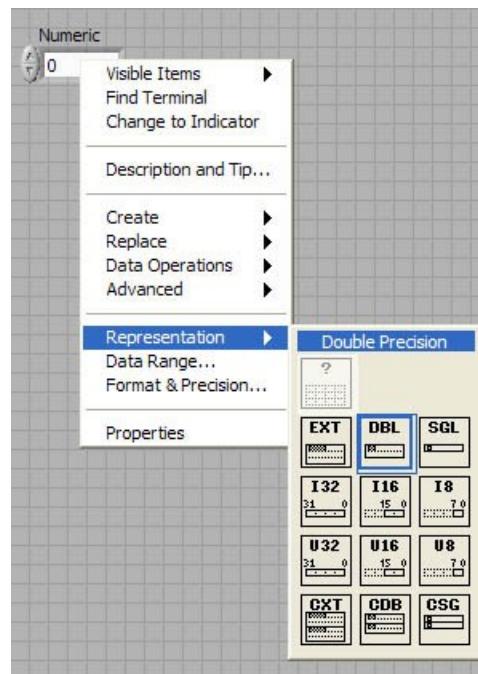
In case of a standard textual based programming language we would use two variables, one for the Celsius and another for the Fahrenheit value. So far we avoided the concept of variables on purpose, as LabView does not deal with variables, but with data sources (Controls) and data outputs (Indicators). Is it possible to use local and global variables also in LabView, but these are used only in case the given task can not be solved by the original dataflow model.

Creating a new Control Click on the Num Ctrl sub-palette (upper left icon) on the Controls Palette of the Front Panel, and choose the Num Ctrl icon!



Data Type

Change back to the Front Panel, right-click on the Numeric icon, and choose Representation from the pull-down menu! Here we can see the data type of our Control. The default is DBL, double precision. We can change the type if it is needed. (Don't change it for now.)



Creating a new Indicator Similarly to the Numeric Control, place an Indicator on the Front Panel! (Numeric Indic) We have the input and the output of our dataflow, now we have to make the connections between them.



Changing labels

We can change the default labels Numeric and Numeric 2 to more meaningful labels. Rename the former to Fahrenheit, the later to Celsius!

Subtract



In the algorithm, we need to subtract 32 from the input. We use a subtract object and a constant.

Choose the Subtract object from the Numeric sub-palette of the Functions palette, and place it between the input and the output!

Constants



From the same Numeric palette, choose Numeric Cons..., (lower left corner), and place it on the Block Diagram! Change the default 0 value to 32!

To do the same in a faster way, right-click on one of the inputs of the Subtract icon, and choose Create/Constants from the pull-down menu! This is an important feature, if we create Constants (or Controls or Indicators) this way, then the Constant will automatically be of the correct data type required by that certain input. This means we don't have to know what data type does the input require.

Division



Place a Division in the Block Diagram from the Numeric sub-palette!

Broken arrow



Before we start wiring, notice the broken arrow at the Toolbar! This means that the VI has a syntax error in it, and it is not ready to run. If all the wiring will be OK, and the program is ready to run, then the arrow will automatically change to an intact light colored arrow.

Run



After we have finished the wiring, click on the Run arrow! The converted value is written in the Indicator. (The program is executed only once.)

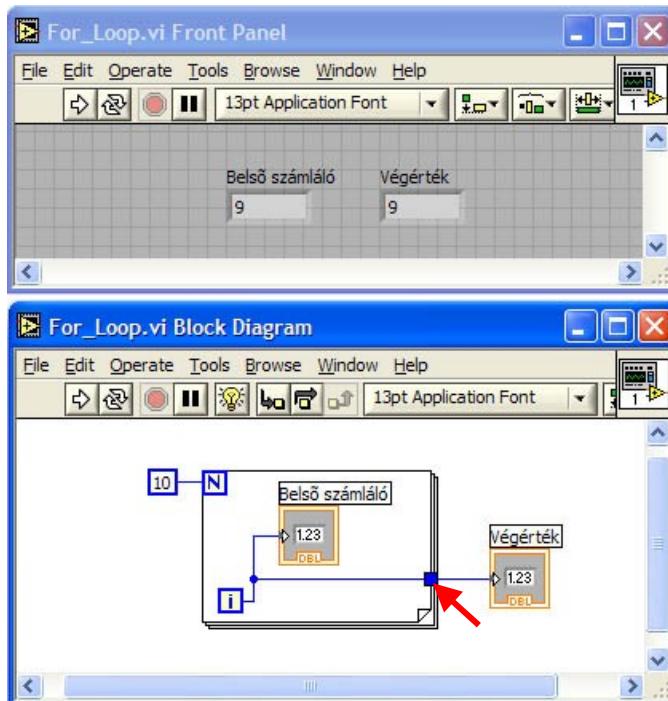
Save

Save our program named Convert_F_to_C.vi!

2.4 Control Structures and data types

2.4.1 Loops

Control structures can be found on the Structures sub-palette of the Functions palette. From these, we consider loops first.



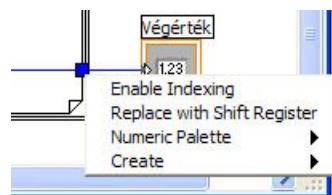
FOR loop

To use a For loop, we have to choose the **For loop** icon, and then we have to wrap around that part of the code, that we want to execute more than once. The For loop has two variables, the Iteration Terminal (i), and the Count Terminal (N). The Iteration Terminal is a counter that counts the number of loops during execution. The Count Terminal is a constant, which determines how many times the loop will be executed. The value of the Iteration Terminal goes from 0 to N-1.

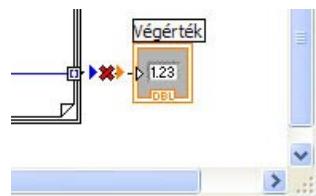
Please see the example above (FOR_Loop.vi)! „Belső számláló” means Inner Counter, „Végérték” means Final Value.

The role of the blue box marked by the red arrow is to lead out the value of the Iteration Terminal from the loop. If we right-click on the blue box, a pull-down menu is popped up. We can choose Enable indexing, which means that automatic indexing is now disabled.

Auto index disabled

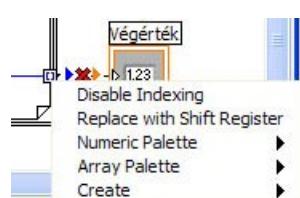


Click on the Enable Indexing line! We see the following:



The blue box is changed, and the wire to the Output is broken. Let's click on the box again!

Auto index enabled



Now it can be understood what Enable/Disable Indexing means. These are two different possibilities to route out the value of the Iteration Terminal form the loop. If the indexing is disabled, then only one value, the final numerical value of the Iteration Terminal will be routed out from the loop, while if indexing is enabled, then an array will be routed to the output, where the array contains all of the values which the Iteration Terminal had during the execution of the loop. In the example above the wire got broken because it is not allowed to connect a scalar value with an array. Once again, it is important to point out that a wire coming out from a loop in any case will have only one value. The Enable/Disable Indexing only determines if this

value is a single scalar value, or an array. This value or the array is sent out after the loop has finished execution.

The automatic indexing works similarly to other kind of loops as well.

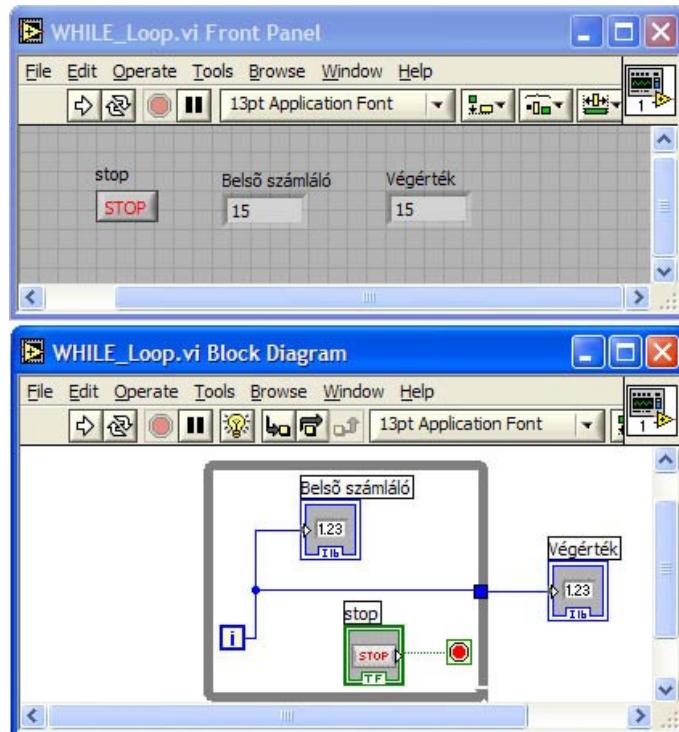
Data types

The FOR_Loop.vi example demonstrates also the use of data types in LabView. In LabView, colors have significant role. Blue color indicates an integer value (e.g. i, N and the constant 10), while brown indicates a double precision floating point number (DBL). Here we can see an example of automatic type conversion: we could connect the integer i with a DBL variable, and LabView does the required type conversion.

WHILE loop

The WHILE loop has an Iteration Terminal (i). The While Loop is executed at least once. The number of executions is determined by the Conditional Terminal. In the WHILE_Loop.vi example below, the loop is executed for infinite times until we click on the STOP button on the Front Panel. (Of course we can also stop every program with the Stop button in the Status Bar.)

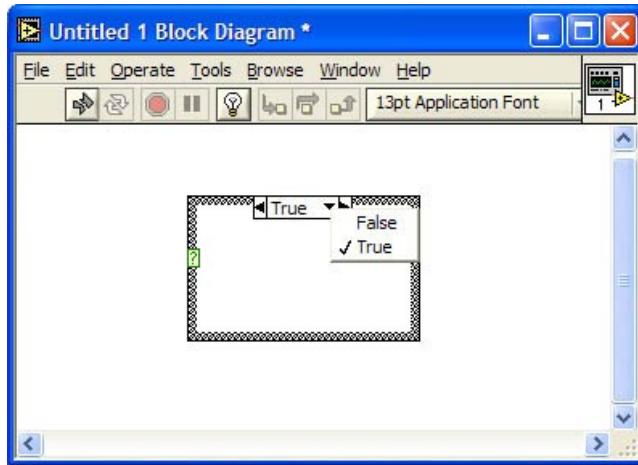
There are several possibilities to stop a While loop. We can choose the most suitable form the pull-down menu which appears if we right-click on the red Stop button.



2.4.2 Case structure

This is similar to the switch-case of C, which is visualized by a stack. The frames are behind each other like a deck of cards. At once, only one „card” is shown. It depends on the input condition, which one.

It can be found here: **Functions / Structures / Case Structure**.



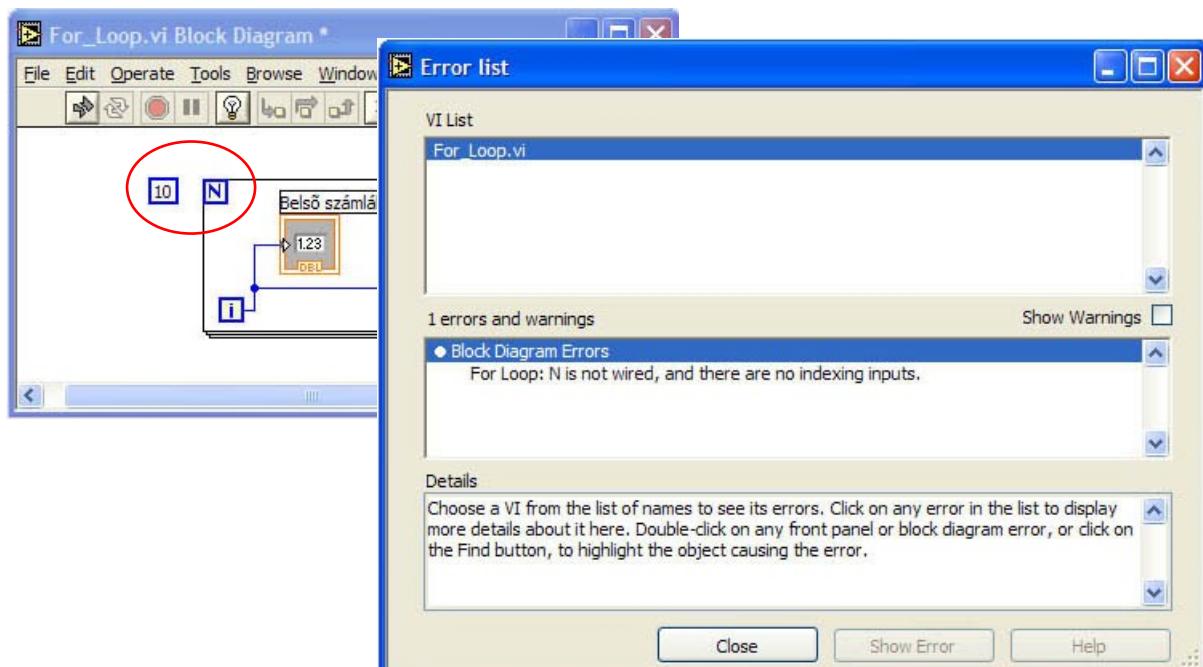
By right-clicking on the frame, the pull-down menu is invoked. We can add or remove new cases to the structure here. The shown cases depend on the data type of the input. By default, the Case structure has a logical input with two values, True and False, with two corresponding cases. If the input data type is changed to a different type (e.g. to string or integer), then the number of cases are also changed.

2.5 Other important LabVIEW elements

Debugging

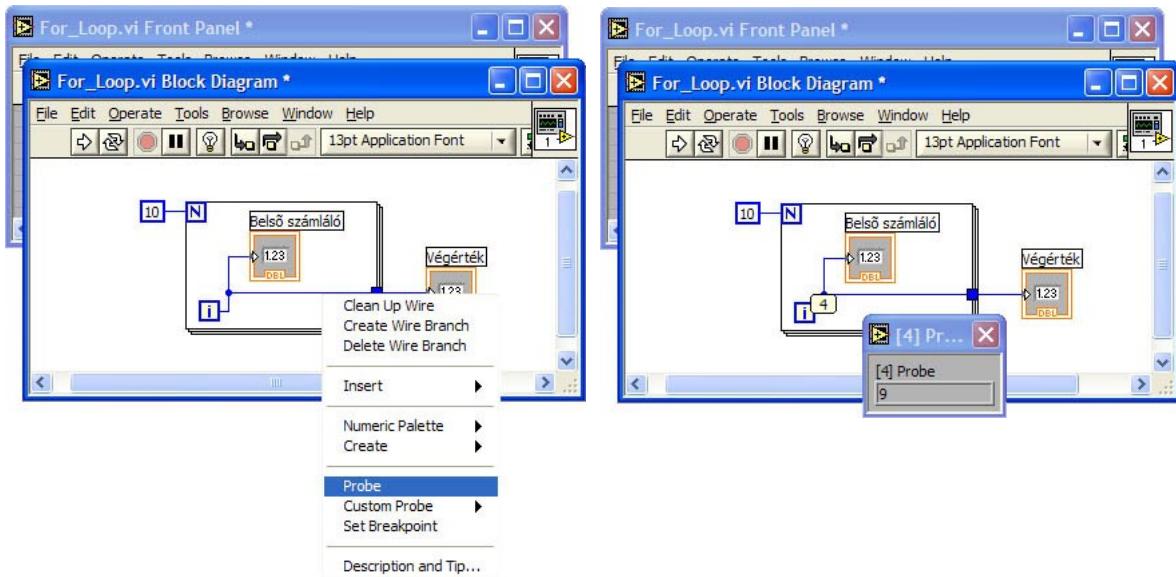


Click on the broken arrow icon! The appearing window lists the errors.



Probe

After right-clicking on a wire, a pull-down menu is invoked, where we can choose the Probe line. This Probe is showing the data flowing on the wire during execution. (We can do the same by choosing the Probe from the Tools palette, and click on the wire afterwards.)



Conrext Help



The context help is very useful. If it's turned on and we move the mouse pointer above any element, then the description of that element is shown in a separate window.

subVI

So far we were using VIs as separate, stand-alone programs. Besides this, in LabView, all the VIs can be used as single blocks in another VI. This means that every VI can be used as a subVI in other VIs, as a subroutine. A subVI does not have a Front Panel, but it has inputs and outputs instead. SubVIs can be made in the following way: we select the part in the Block Diagram that we wish to convert to a subVI, and then we choose the Create SubVI command from the Edit menu. SubVIs can be collected in libraries, for further reuse. The use of subVIs can make our programs simpler and clearer.