LabVIEW for Physicists

Daniel Janse van Rensburg

June 2022

Contents

Chapter 1

Introduction

1.1 Welcome

Welcome to the LabVIEW course, for 3rd year physics students, at UJ.
In your first semester of 3^{rd} year, you have been exposed to a programming language called `C++`, an *Object-Oriented* text based computer language. With this mighty tool in hand, you made a little microcontroller do your bidding, specifically an `ATmega328` found on the Arduino boards.

You may have noticed how terse statements in `C++` can get, sometimes obfuscating the programmers intent, see listing 1.1 on page 2 for an extreme example. In this course however, we will not be dealing with text based statements, instead you will make a computer bend to your will by means of pictures and threads!

Before we jump into the details, let us first compare two programs performing the same task, one written in `C`, and one assembled in `LabVIEW`. Suppose we want to capture two decimal numbers from a user in order to compute the sum of said numbers:

C Programming

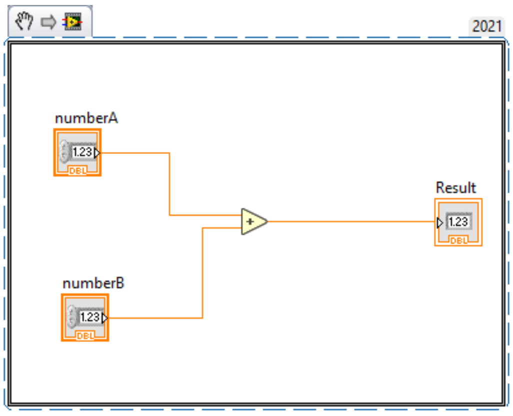
In listing 1.2, on page 3, the first line imports the required libraries for `C` to handle our input and output, or `IO`. We then have to declare what type of variables we are storing and give them names, in this case `numberA & numberB`. In this case, the values are initialised to 0, not required for this example, but good practice regardless. The numbers are then captured from the user on line 7 & 8. Finally, the results are printed so that the user may see that their system can indeed do basic arithmetic.

```c
char*l="ustvrtsuqqqqqqqqyyyyyyyy}{|~z|{}"
"   76Lsabcddcba .pknbrq  PKNBRQ ?A6J57IKJT576,+-48HLSU";
#define F getchar()&z
#define v X(0,0,0,21,
#define Z while(
#define _ ;if(
#define P return--G,y^=8,
B,i,y,u,b,I[411],*G=I,x=10,z=15,M=1e4;X(w,c,h,e,S,s){int t,o,L,E,d,O=e,N=-M*M,K
=78-h<<x,p,*g,n,*m,A,q,r,C,J,a=y?-x:x;y^=8;G++;d=w||s&&s>=h&&v 0,0)>M;do{_ o=I[
p=O]){q=o&z^y _ q<7){A=q--&2?8:4;C=o-9&z?q["& .$  "]:42;do{r=I[p+=C[l]-64]_!w|p
==w){g=q|p+a-S?0:I+S _!r&(q|A<3||g)||(r+1&z^y)>9&&q|A>2){_m=!(r-2&7))P G[1]=O,
K;J=n=o&z;E=I[p-a]&z;t=q|E-7?n:(n+=2,6^y);Z n<=t){L=r?l[r&7]*9-189-h-q:0 _ s)L
+=(1-q?l[p/x+5]-l[O/x+5]+l[p%x+6]*-~!q-l[O%x+6]+o/16*8:!!m*9)+(q?0:!(I[p-1]^n)+
!(I[p+1]^n)+l[n&7]*9-386+!!g*99+(A<2))+!(E^y^9)_ s>h||1<s&s==h&&L>z|d){p[I]=n,O
[I]=m?*g=*m,*m=0:g?*g=0:0;L-=X(s>h|d?0:p,L-N,h+1,G[1],J=q|A>1?0:p,s)_!(h||s-1|B
-O|i-n|p-b|L<-M))P y^=8,u=J;J=q-1|A<7||m||!s|d|r|o<z||v 0,0)>M;O[I]=o;p[I]=r;m?
*m=*g,*g=0:g?*g=9^y:0;}_ L>N){*G=O _ s>1){_ h&&c-L<0)P L _!h)i=n,B=O,b=p;}N=L;}
n+=J||(g=I+p,m=p<O?g-3:g+2,*m<z|m[O-p]||I[p+=p-O]);}}}}Z!r&q>2||(p=O,q|A>2|o>z&
!r&&++C*--A));}}}Z++O>98?O=20:e-O);P N+M*M&&N>-K+1924|d?N:0;}main(){Z++B<121)*G
++=B/x%x<2|B%x<2?7:B/x&4?0:*l++&31;Z B=19){Z B++<99)putchar(B%x?l[B[I]|16]:x)_
x-(B=F)){i=I[B+=(x-F)*x]&z;b=F;b+=(x-F)*x;Z x-(*G=F))i=*G^8^y;}else v u,5);v u,
1);}}
```

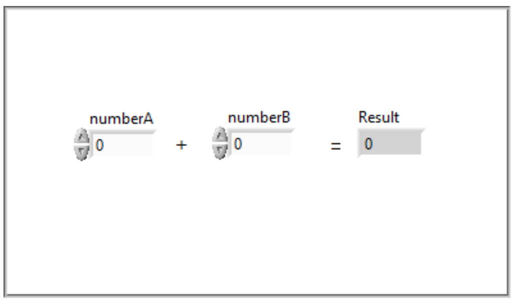
Listing 1.1: This listing is executable C code see http://www.nanochess.org

```c
#include<stdio.h>

int main() {

  double numberA = 0;
  double numberB = 0;
  scanf("%f", &numberA);
  scanf("%f", &numberB);

  printf("%d + %d = ",numberA, numberB, numberA + numberB);
  return 0;
}
```

Listing 1.2: A simple C program to add two user typed numbers



(a) A LabVIEW block diagram.



(b) A LabVIEW front panel.

Figure 1.1: A LabVIEW program to sum two numbers.

LabVIEW Programming

Without having any idea what LabVIEW is, I am sure you are able to follow what figure 1.1a, on page 3, is attempting to convey. In figure 1.1b you will find the *Graphical User Interface*, or GUI, for the summing program. The two figures are one program developed simultaneously, we will dive into the details throughout the course.

Before you even ask, yes it is really that simple. You will find that creating programs in LabVIEW is fun and rewarding, so relax and take it easy. By the end of this course, you will be able to create complex programs to help you solve problems in physics, as well as bridge the gap between the computer world and the real world.

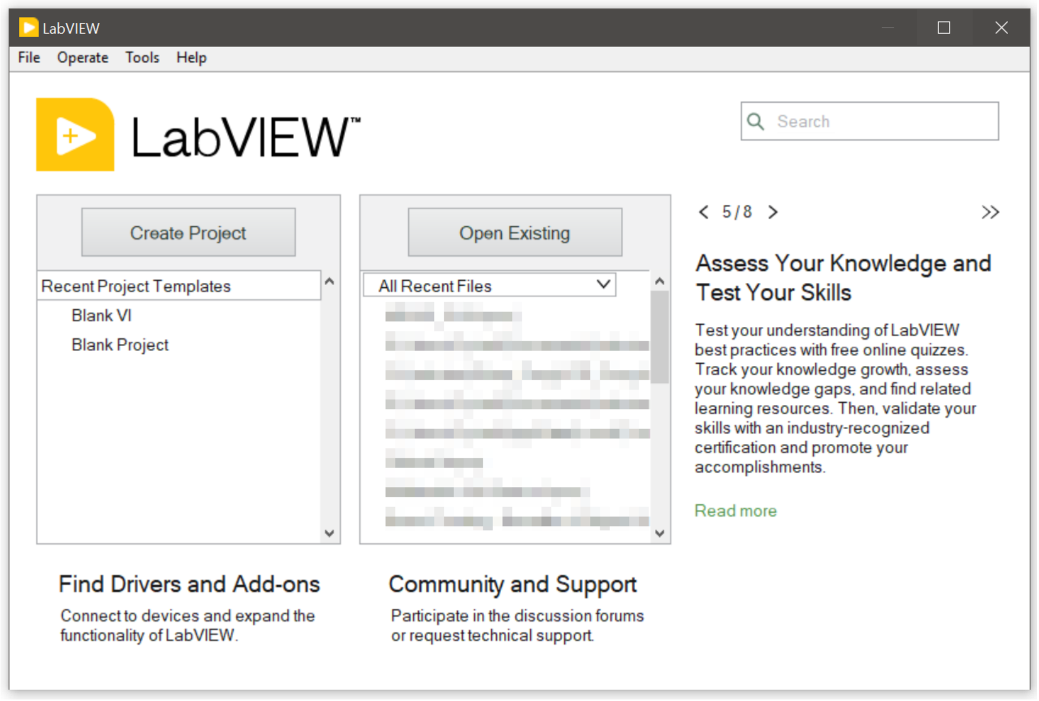

LabVIEW
File Operate Tools Help
LabVIEW™
Search
Create Project
Open Existing
Recent Project Templates
Blank VI
Blank Project
All Recent Files
< 5/8 >
Assess Your Knowledge and Test Your Skills
Test your understanding of LabVIEW best practices with free online quizzes. Track your knowledge growth, assess your knowledge gaps, and find related learning resources. Then, validate your skills with an industry-recognized certification and promote your accomplishments.
Read more
Find Drivers and Add-ons
Connect to devices and expand the functionality of LabVIEW.
Community and Support
Participate in the discussion forums or request technical support.

Figure 1.2: The window LabVIEW greats you with on startup.

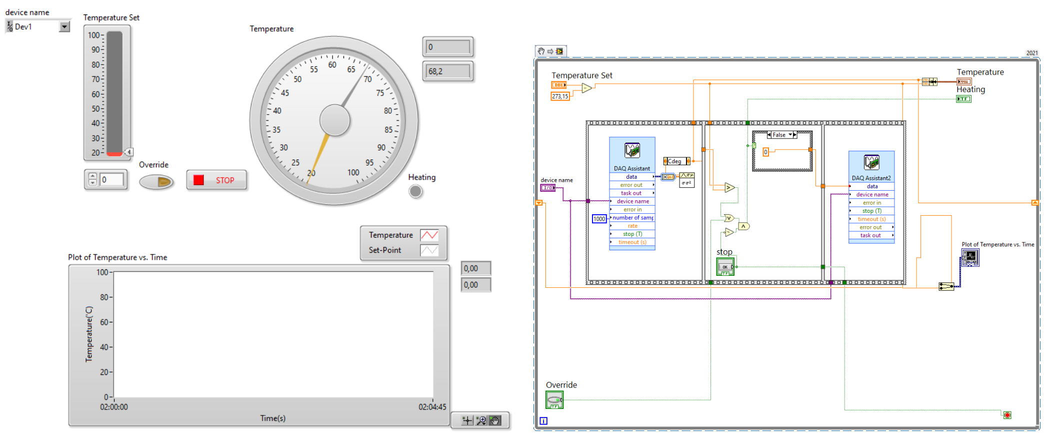
1.2 The LabVIEW Interface

1.2.1 Introduction

LabVIEW usually greats you with a launcher, figure 1.2. From here you may open your recent projects, start a fresh project, or open a virtual instrument, known as a VI. For now, from the “File” drop down menu, select “New VI” or press `Ctrl+N` on your keyboard.

Two windows will pop up, a “Front Panel” and a “Block Diagram”. You will need to familiarise yourself with the LabVIEW interface, this is best done by exploration, trail and error. Simply mousing over any button should give you some clue as to what the button does or what is contained in the menus. You probably would not break your computer or the LabVIEW installation by playing around with the interface.

In the next few sections we will go over what is meant by a VI, the difference between the block diagram and the front panel, how to cycle between these views, and how to place objects on these windows.



(a) A more useful front panel. (b) The logic behind the front panel.

Figure 1.3: A program which turns a kettle on and off to achieve a set temperature.

1.2.2 The VI

As stated before, VI is short for virtual instrument. The idea is that you create an interface which may be operated by a human, usually with a mouse and keyboard. This interface is called the "Front Panel" in LabVIEW. You then glue the elements of the front panel together in the "Block Diagram", this is where the LabVIEW magic happens, or the LabVIEW logic execution engine, if you do not believe in magic.

Figure 1.3a, on page 5, shows one of the first front panels I have ever created along with the block diagram in figure 1.3b. By the end of this course you will be able to understand exactly what is going on there so do not let it intimidate you. Assert dominance over your computer, lest it assert dominance over you.

The Front Panel

Figure 1.4 shows an empty front panel. `Right clicking` anywhere on the grey grid will open a menu containing controls, known as the "controls palette". Do not be overwhelmed, given some time you will get a feel for where the most important controls are. As mentioned, and will be repeated throughout this book, explore the interface on your own.

Here is a little secret, simply press `Ctrl+H`. This will open a floating window called "Context Help", your new life-line in LabVIEW. Hover over

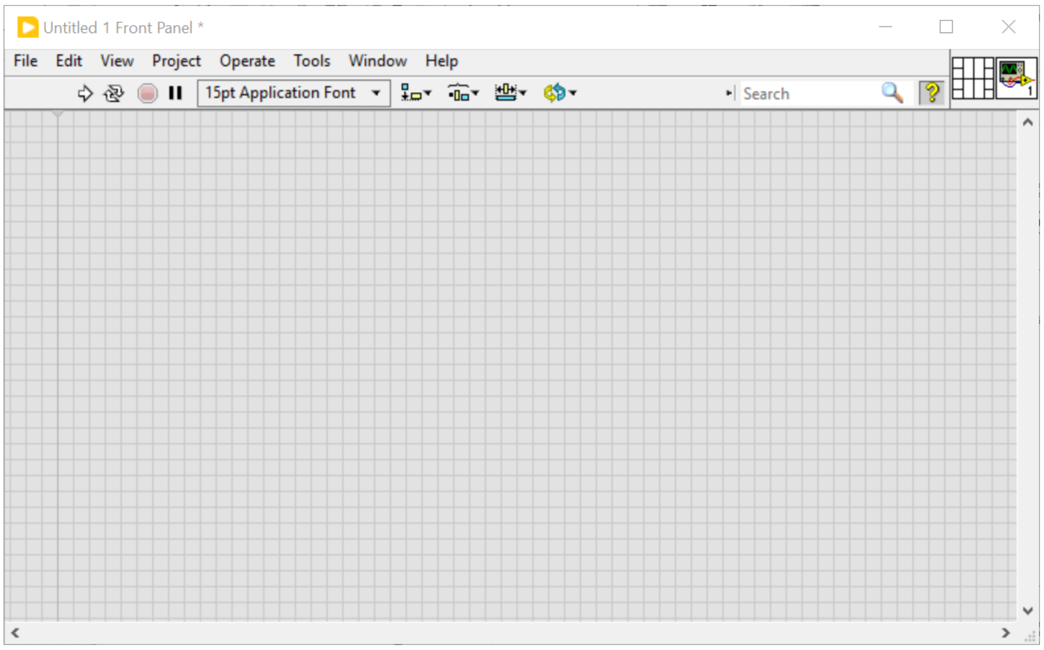


Figure 1.4: A barren wasteland of a front panel, not very useful.

anything and it will give you some information about what you are looking at. Pressing "Detailed help" takes you to the LabVIEW help files. You will spend a great deal of time reading these documents as you progress through your LabVIEW journey so you should know where to find it.

If you figured out how to place down controls in LabVIEW, that is great, do not let the interface intimidate you. If you have not done so yet, we will go through the details in section 1.4.2.

The Block Diagram

The sibling panel to figure 1.4 may be found in figure 1.5. As before, you may `Right click` on the white background to open the "functions palette". This is where you will spend most of your time in LabVIEW, other than the help files that is. You thread together small function blocks to build the logic of your program. If you are reading this and can't find the block diagram panel, simply press `Ctrl-E`, this will switch from the front panel to the block diagram and vice versa. This is probably the most important hot-key in LabVIEW so learn it, you will often have to flip between the two windows.

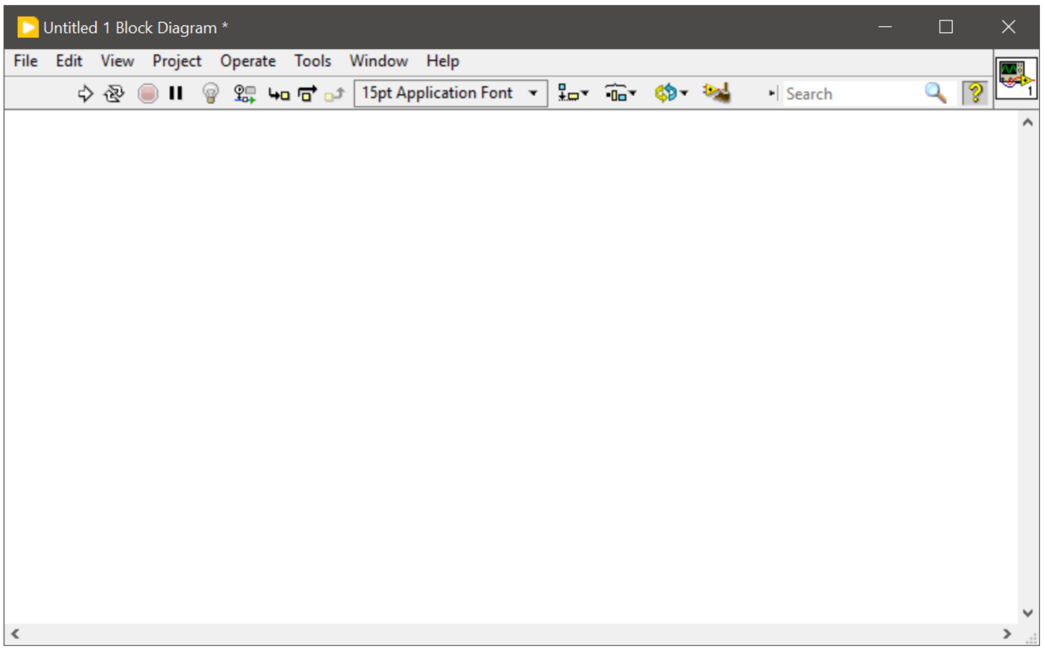

Untitled 1 Block Diagram *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search

Figure 1.5: The void, even interstellar space has more going on than this panel.

1.3 LabVIEW variables and you

1.3.1 Overview

From your knowlage of mathematics, you understand the concept of a variable. This concept is of fundamental importance to all fields of computer science so let us take a slight detour before we dig further into LabVIEW.

Variables are named containers in which we may store information. It is then possible to read from, or write to, this container using it's name. At any moment in time, there are several named variables in your head, for example: `cellphoneNumber`, `currentDate`, `amountOfCoffeeHadToday`, etc.

With these examples, it is clear that some variables are not of the same type, i.e. saying "I had 3 June 2022 coffees today" makes no sense. You should know by now, or if you do not you will learn very soon, that computers are incredibly stupid and they will happily add 3 June 2022 to 072 543 7711 and give you a result. (The answer is 29 May 2045 by the way).

In general, it is your job to tell the computer what type a variable is. This is not strictly true as many programming languages can infer the data type from your input, however for LabVIEW and programming languages such as `C` and `C++`, you are responsible.

1.3.2 Variable Types

Boolean

Perhaps the most fundamental variable type is the boolean, or bool for short. It consists of either yes or no, true or false, `1` or `0`, you get the picture. In LabVIEW, bools form the basis of nearly all your decision making code. It is represented as green icons, seen in figure 1.6a, on page 9.

Floating Point

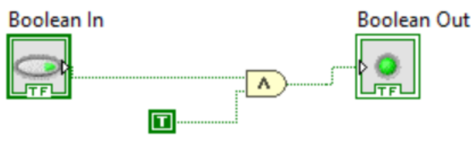
Floating point variables hold decimal numbers, in an application this may be the value of a magnetic field, the temperature of a thermocouple, basically any number that you can think of that would fit into 64bits of memory (more on that if you do computational physics in honours physics). All floating point values are represented in Labview as orange icons, seen in figure 1.6b, on page 9.

Integer

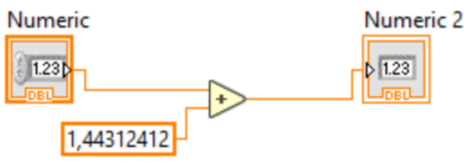
The integer type is self explanatory. This is the only type of number which may be represented in LabVIEW as exact numbers and is ideal for comparison and counting. It is worth mentioning briefly that integers exist as two types namely signed and unsigned. Negative numbers are contained in signed integers, unsigned integers are strictly positive. The nuances of signedness are best left for the computational physics coarse in honours. Integer types are represented as blue icons in LabVIEW, see figure 1.6c on page 9.

Strings

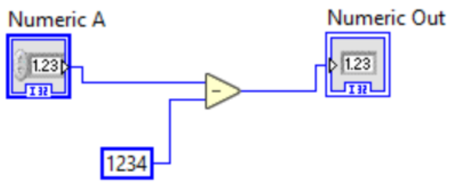
This line that you are reading is a string. Strings are made up of characters, itself a type of variable. Characters are not as important in LabVIEW as in other programming languages so they do not deserve their own subsection. Strings may be used to convey information to users, it may also be used to store information on your hard-disk. Strings are pink in LabVIEW, reference 1.6d on page 9.



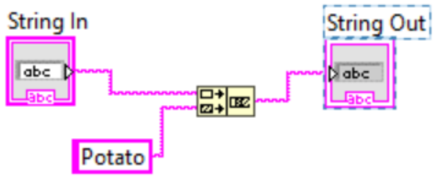
(a) Boolean controls and constants.



(b) Floating point controls and constants.



(c) Integer data types.



(d) String data types

Figure 1.6: Fundamental LabVIEW data types.

Arrays

Arrays are homogenous sections of sequential memory. Any of the above data types may be formed into an array. This variable type allows you to reference a collection of data points as a single item. Unless you derive joy from naming thousands of individual variables, you should use arrays when working with more than a handful of data points. Arrays are indexed from 0 in LabVIEW, important if you have programmed in Fortran before. Arrays follow the colour of their contents in LabVIEW, i.e. a blue array contains only integers.

Aggregate Types

Variable types may be grouped together to form a new variable type, usually with some relationship among one another. In LabVIEW, these types are known as clusters, similar to structs from the C family of languages. We will leave clusters for now, they are only really useful once your programs start getting big.

1.4 Your First LabVIEW program

1.4.1 Problem statement

We must create a program in which a user gives three integer values, `numberA`, `numberB`, and `NumberC`. The program needs to compute and output the sum of `numberA` and `numberB` as well as indicate if this sum is larger than `numberC`.

The rest of this section will show you in detail how such a program is made in LabVIEW, however it will not explain the details of the program in depth. This is left for later sections in chapter 2.

1.4.2 Implimentation

Front Panel

You may think of the front panel as your scratch pad. What information does the user need to provide for our program and how may the computer display the outcome of a process?

Anywhere on the front panel, `right click` to show the control palette. Mouse over the top left folder called "Numeric", a new window will pop up. Drag your mouse into this window and select "Numeric Control" by clicking once on the icon. You will notice that your mouse cursor now changed to a little grab hand with an outline of the proposed control. Simply click where you would like this control to live.

The name "Numeric" is not helpful to us. If the name is highlighted, white text in a black box, you may type a new name and it would replace the old one. Call this control "numberA". If the name is not highlighted, simply `double click` on the name until you see the black box with white text and then type "numberA". Once you have typed the name, `left click` anywhere on the background to make the change permanent.

Figure 1.7 shows how selected text looks like and what you should have on your front panel after you have renamed the control. Repeat what you have just done two more times to create "numberB" and "numberC".

For our resulting number, open the controls palette, go to "Numeric" and select the "Numeric Indicator" icon. Rename this to "Result".

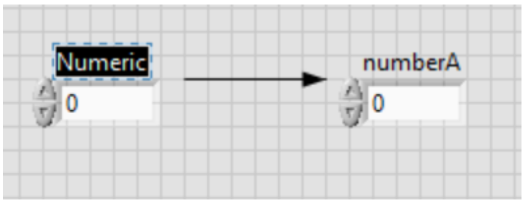


Figure 1.7: What you should have after renaming the control.

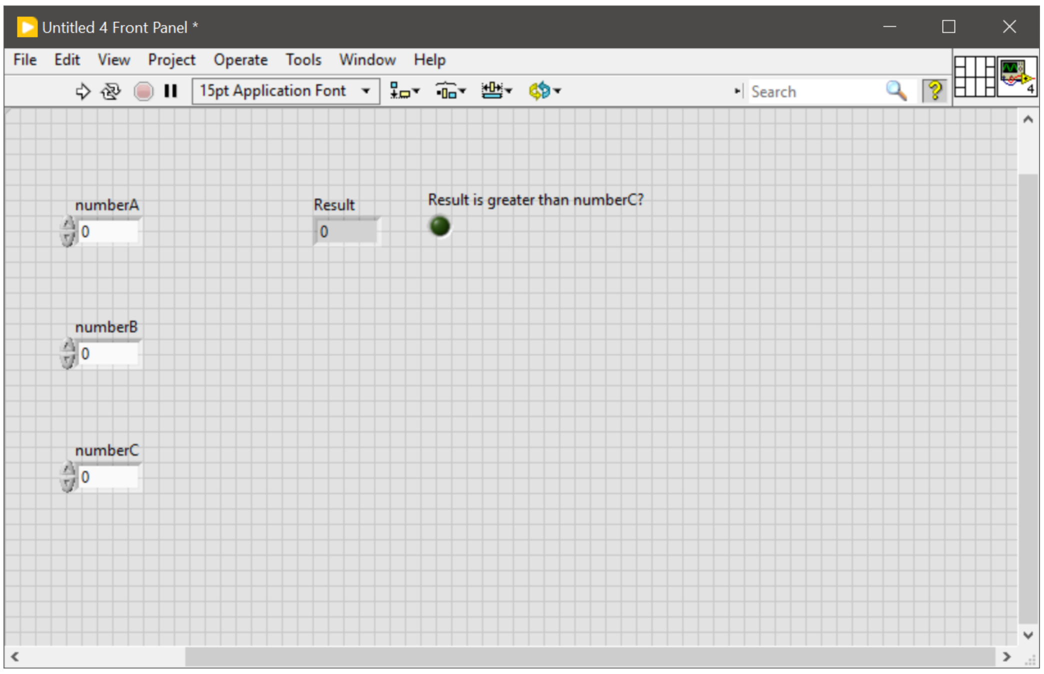


Figure 1.8: Roughly what your front panel should look like..

Finally, open the control palette again, this time go to the "Boolean" folder and select the "Round LED". You should place this close to your result indicator. You will start to notice that grouping controls together with regards to input and output makes it easier to understand the intent of your program. Lastly, rename this indicator to "Result is greater than numberC?".

Your front panel should now look like the panel in figure 1.8 on page 11.

If you are not satisfied with your layout, you may `left click` and hold on the edges of a control and drag it to where you would like to to be. This might take some getting used too. Your cursor will change from a cross to a pointer when you hover over a part of a control which is allowed to be moved.

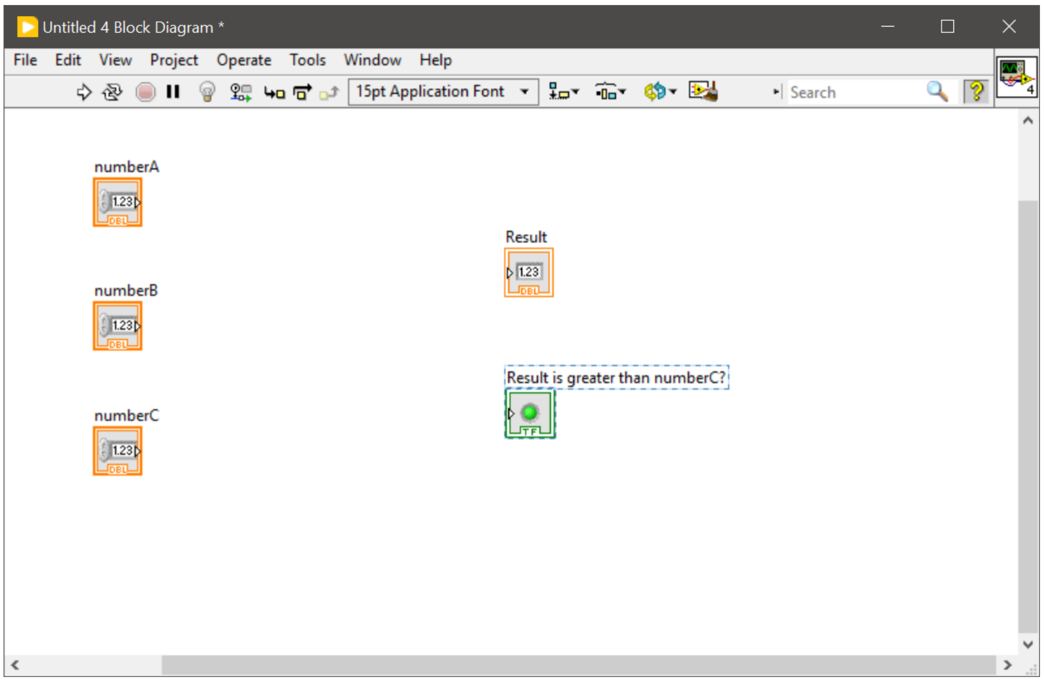

Untitled 4 Block Diagram *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
numberA
numberB
numberC
Result
Result is greater than numberC?

Figure 1.9: More or less what your block diagram should look like.

Block Diagram

While looking at your front panel, press `Ctrl-E` to flip to your block diagram. You would notice that what you have done on the front panel is reflected in the block diagram. Your inputs have little white arrows pointing out of the icon to the right. The outputs, also known as indicators in LabVIEW, have white arrows pointing into the icon on the left.

You should move these icons so that the interface flows from left to right, that is, inputs on the left and outputs on the right. You move the icons like you have moved the front panel elements, just click, hold, and drag the icons to where you think they should live. See figure 1.9 for inspiration.

You are now ready to build the logic of your program. From our problem statement, we need to sum numberA and numberB. Both are numerical values so we go to the “Numeric” folder of the function palette. The first icon you see is the add function. Place it somewhere in the middle of your inputs and outputs.

Hover your mouse cursor over the newly placed function, you will notice

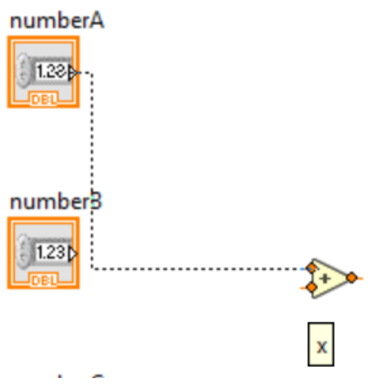


Figure 1.10: A ghostly wire follows your cursor to your desired terminal.

that little orange terminals are highlighted. If you hover over one of your number inputs, you will also see a little terminal, conveniently placed next to the white arrow. `Left click` on the terminal of numberA, moving your mouse around the screen you will see a black striped line following your cursor. Then `left click` on one of the terminals of the add function. You would see something like figure 1.10 before you click. This will create an orange line, also known as a wire, connecting your input to the function.

Connect numberB to the add function, then connect the output of the add function to the result indicator. You should now have a block diagram that looks like figure 1.11 on page 14.

From the function palette, select the "comparison" folder and look for the function that says "greater than". It is the triangle with the ¿ symbol on it. Place this somewhere below the add function, but still between the controls and the indicators.

You should now wire the output from this function to the terminal of your LED. The function compares two inputs and sends a true value to the LED if the one input is greater than the other, but how do we know which input is which? The "context help" window will show you which input is which, see figure 1.12. If you do not see this window, just press `Ctrl+H` on your keyboard.

Wire in the value from numberC into the comparison function and connect the other input to the output of your summing function. You may do this by starting the wire at the input of the comparison function and `Left clicking` on the wire leading from the summing function to your result in-

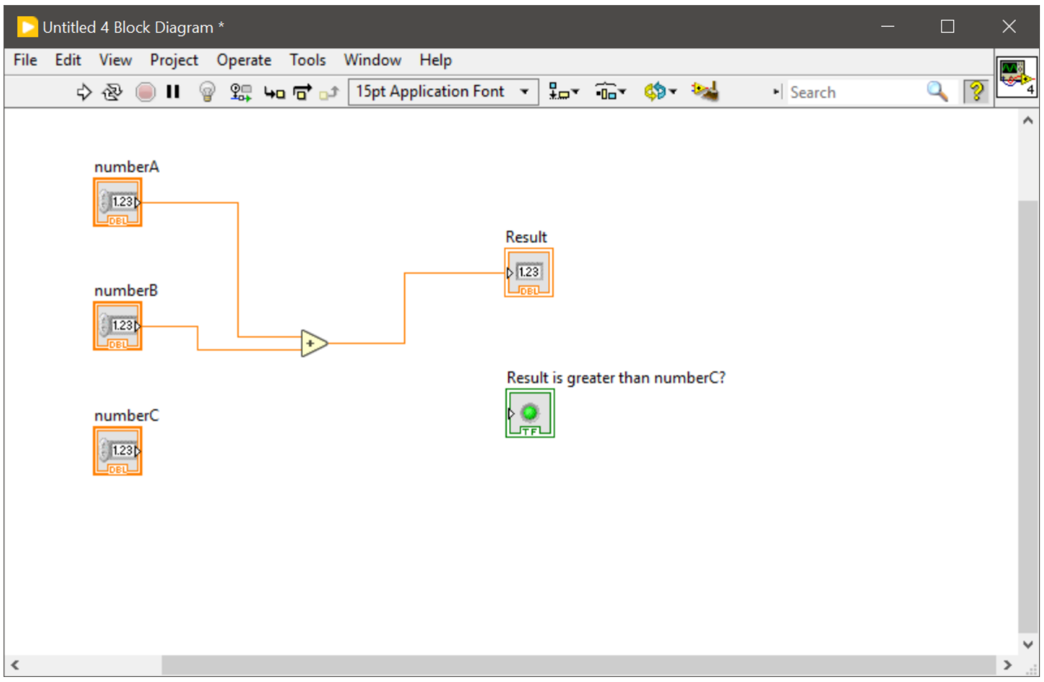
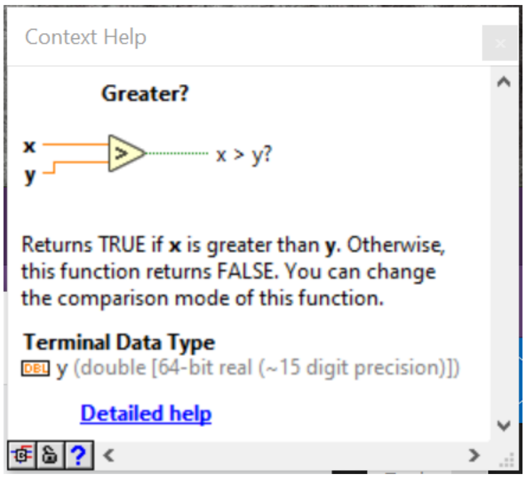

Untitled 4 Block Diagram *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
numberA
numberB
numberC
Result
Result is greater than numberC?

Figure 1.11: The summing logic wired correctly, your block diagram should look like this.


Context Help

Greater?

x > y?

Returns TRUE if **x** is greater than **y**. Otherwise, this function returns FALSE. You can change the comparison mode of this function.

Terminal Data Type

DBL y (double [64-bit real (~15 digit precision)])

Detailed help

Figure 1.12: The context help for the comparison function, the top terminal is x and the bottom one is y.

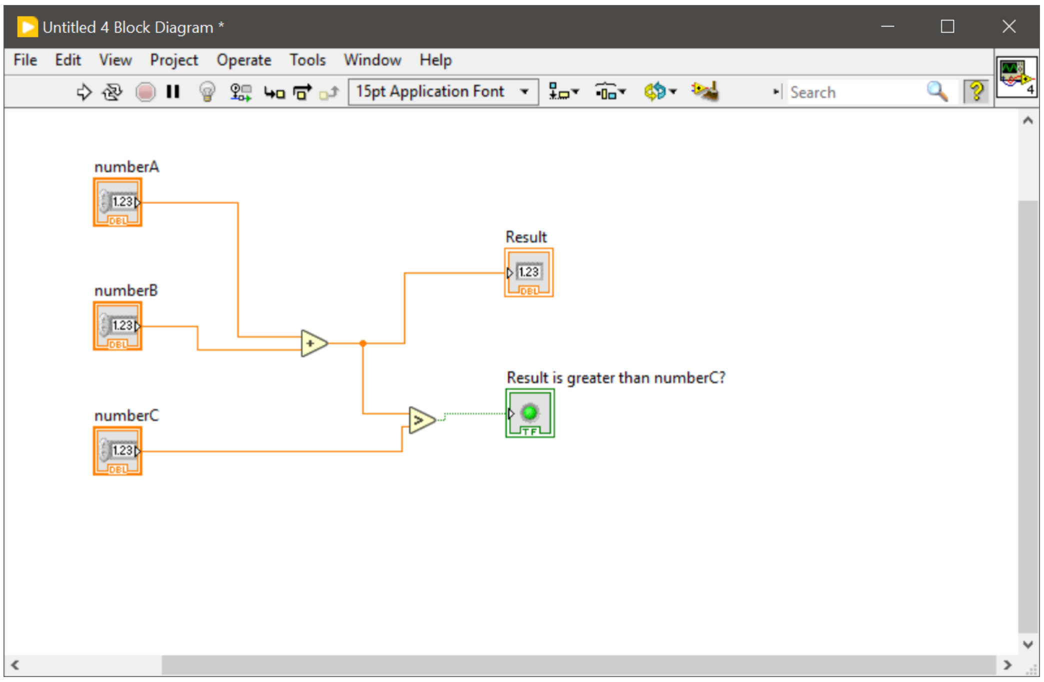

Untitled 4 Block Diagram *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
numberA
numberB
numberC
Result
Result is greater than numberC?

Figure 1.13: The final wired block diagram.

dicator. It is also possible to `Right click` on the wire leading from the summing function and selecting the “Create wire branch” option from the menu. You then wire this into the comparison function.

Finally, your block diagram should look like the one in figure 1.13. You may also move around the wire pieces by `left clicking` on them and dragging them about.

1.4.3 Testing your program

You may now go back to your front panel. You can click in the centre of the controls and type in a value. Do this for all three number controls and press the play button, the one found near the top left of your window.

Does your results make sense? Is the summing correct? Is the light on? Should it be on? These are all the questions that you should have when testing your program. Figure 1.14 shows what happens when the sum of numberA and numberB is greater than numberC. Figure 1.15 shows when numberC is smaller than the sum. If your program does the opposite of what

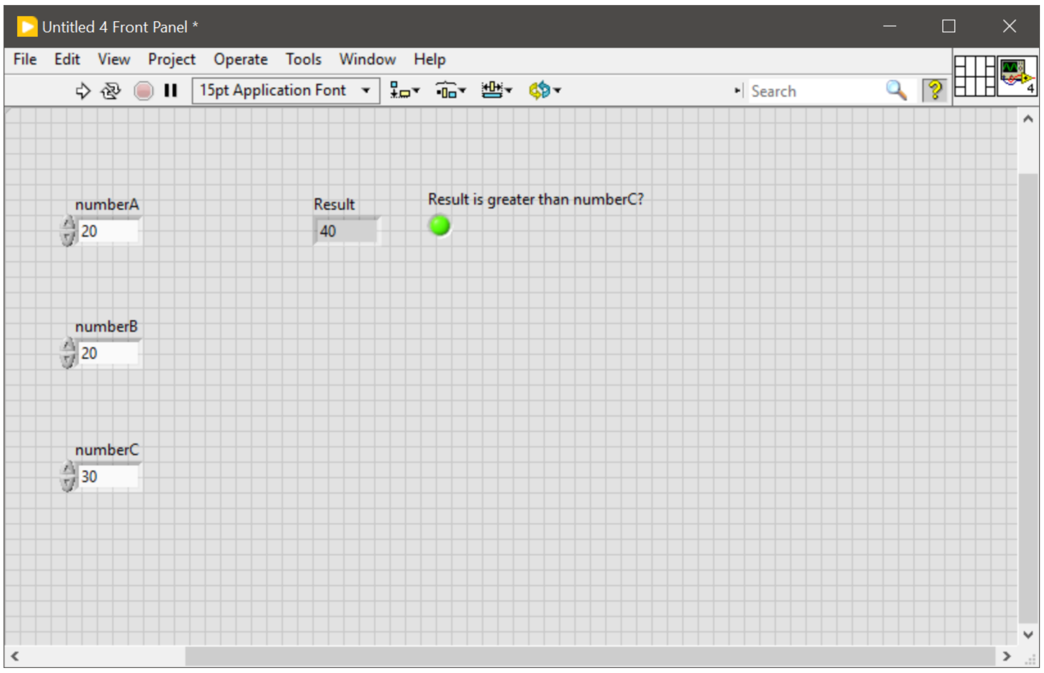


Figure 1.14: The sum is 40 which is greater than 30 so the LED is on.

it is supposed to do, make sure that you have wired in the correct terminals and have selected the correct comparison function. It is worth noting that the VI executes once and stops. After you have typed in new values, you will need to run the VI again.

You may now save your VI by clicking "File" on the task bar and selecting save from the menu. Name it something like "helloworld.vi".

1.4.4 Review

From this example, you have learned how to open LabVIEW and how to create a VI. You have also learned how programs are built using the front panel and the block diagram. This is enough for you to start experimenting with the available functions to create interesting programs.

1.4.5 A word on Accessibility

Colour Blindness

You may have noticed the unique colours attributed to different variable types, if you didn't, you may be colour blind. As of Q3 2022, LabVIEW has

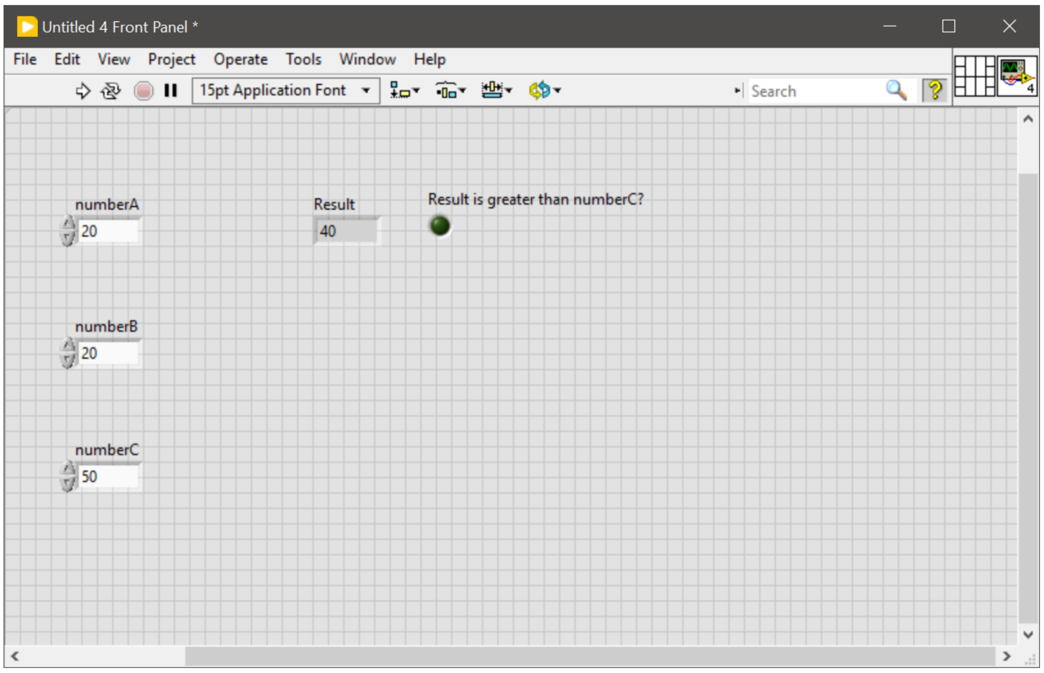


Figure 1.15: Changing numberC to 50 turns the LED off.

no colour-blind friendly mode so you need to study figure 1.6, on page 9, and make a note of the colours you struggle to distinguish. The icons also provide small abbreviations, i.e. TF for boolean and DBL for floating point types.

If this is not useful to you, you may change the colour filter on your computer if you are a windows user. Go to: Start>Settings>Ease of Access. Then select the Colour filters option. Here you will be able to change the colour profile of your monitor, do this while having an example VI open until you can distinguish the variable types with ease.

Vision Difficulty

Depending on various factors, you my struggle to build a block diagram in LabVIEW due to everything being small. As of Q3 2022, LabVIEW still does not offer a zoom feature for you to connect terminals easier. A workaround for this is to use the UI scaling feature built into windows.
Go to: Start>Settings>Ease of Access. Then select the display option. Here you may scale the UI using the "Make everything bigger" option. Adjust this value until you are comfortable connecting terminals in LabVIEW.

1.5 Mini Projects

1. You are building a safety interlock system for your laboratory.
   A powerful x-ray source is required to perform diffraction measurements on powdered samples. This x-ray source requires water to flow through its manifold so that it does not overheat and burnout. If this breaks, you owe the laboratory R100 000 in repair fees. Although the x-rays you receive at the dentist is safe, the intensity of the x-rays used for XRD would cause serious radiation burns which could lead to cancer, the x-ray source should not be allowed to run if the safety door is open.

   Build a interlock system in LabVIEW that would prevent damage to the XRD machine and prevent damage to you.

   To achieve this, you may use switches on your front panel to indicate whether water is flowing through the system and whether the safety door is closed or not. You need a switch that would be the power switch for the x-ray source. Your program needs to light up an LED indicator if the power is switched on and the conditions for operation are met. If the conditions are not met, the LED indicator should not light up.

2. You have a few friends from the United States who do not understand the Celsius temperature scale.
   They are confused that we are walking around on the beach when it is 30°outside.

   Build a simple program which would accept a numerical input as degrees Celsius and convert that to Fahrenheit. This should allow your friends to get a feel for the Celsius temperature scale.

   This program is rather boring, let us make it more interesting by adding indicator lights for the boiling point and freezing point for a couple of materials. For example, if the input temperature is 50°, neither the boiling nor the freezing light should be on. If the temperature is above 100°, the boiling light should be on. You should have lights for water and ethanol. Bonus marks for an extra substance.

Chapter 2

Computing Fundamentals in LabVIEW

In this chapter, we will go over some of the basic building blocks of a computer program in the context of LabVIEW. This chapter will be heavy on examples in LabVIEW, but will be light on the trivial details such as how to open a VI and how to place functions and controls. If you have not already done so, review the exercise in section 1.4.2 in Chapter 1.

If you get stuck, you may open the LabVIEW help files my pressing `Ctrl+?` and navigating to the "Fundamentals" section. If you would like to see more examples of how things are done in LabVIEW, on the taskbar select "Help" and navigate down the menu to find "Find Examples...", here you will find a library of examples. These examples range from trivial arithmetic, to programs which would make you a cup of coffee if you supply it with enough hardware.

2.1 Decision making structures

Without the ability to make decisions, computers would not be as useful as they are now. Try to think of a computer program that makes no decisions based on it's input, they do exist and some are even useful. We are not interested in such academic programs, we want our computer to do the heavy lifting for us, just imagine clicking `yes` or `no` for a data set of a billion numbers. My computer can do that in 5.76 seconds (I just checked using LabVIEW).

If you are familiar with programming, you know that the most fundamen-

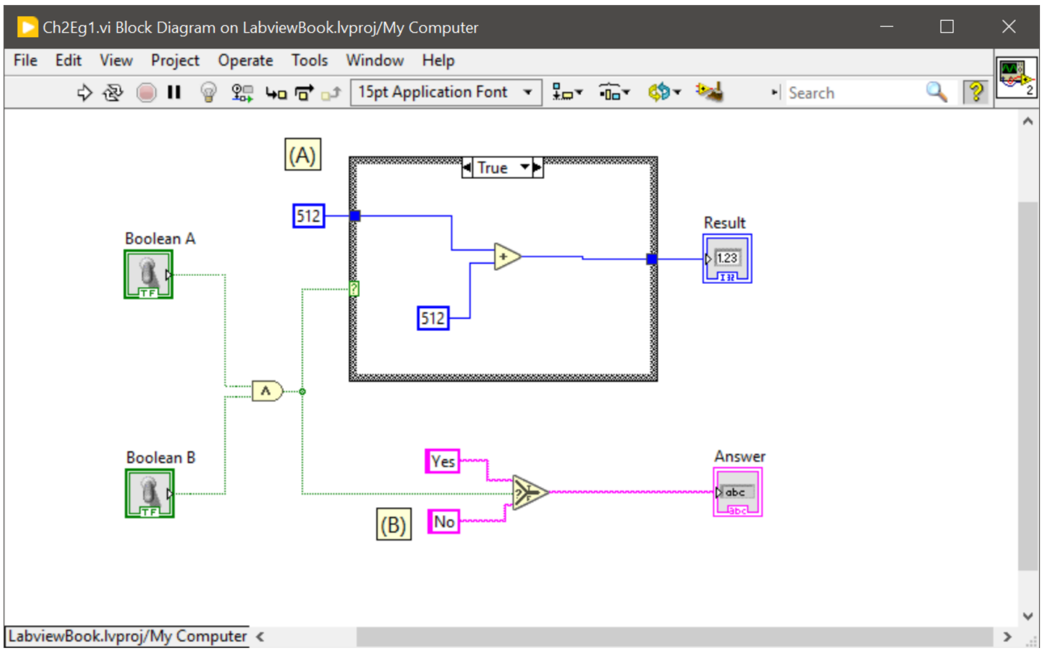

Ch2Eg1.vi Block Diagram on LabviewBook.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
(A)
True
512
512
Boolean A
Result
Boolean B
Yes
No
(B)
Answer
LabviewBook.lvproj/My Computer

Figure 2.1: A case structure showing its truth block (A), and a ternary operator (B).

tal decision making block is an “if-statement”. You supply the block with a condition, and the program executes statements based on that condition. Technically speaking, LabVIEW has no such thing as an if-statement, it has “case-structures” and “ternary-operators”.

2.1.1 Case Structures

The case structure executes a block of code according to its input condition. In LabVIEW, the input condition may be either a boolean value, an enumeration, or an error value. Figure 2.1 (A) shows how a case structure looks, a thick grey box with a green ? and a selection box stating “True”. What ever is inside this grey box will be executed if a boolean true value is wired into the conditional terminal, the green ?. Figure 2.2 shows what would be executed if the condition is false.

Comparison functions in LabVIEW produce only boolean values, this you have seen in the examples so far. Feeding these boolean values into case structures allow you to select different operations to be preformed. The example provided in figures 2.1 & 2.2 show how you would choose two different operations on an integer. If the condition is true, 512 is added to the input

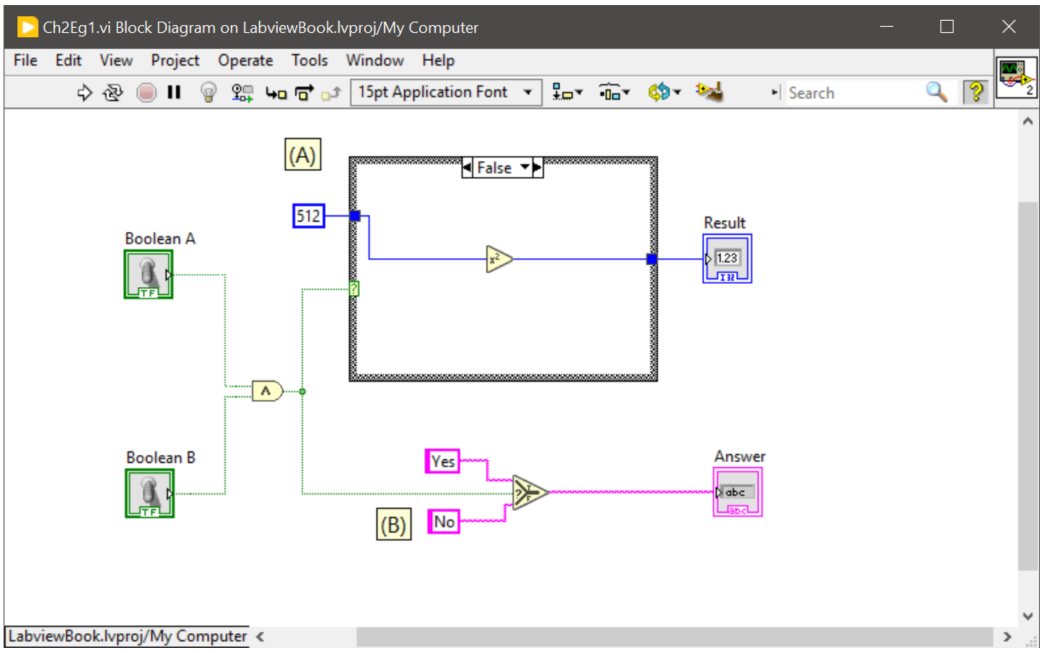

Ch2Eg1.vi Block Diagram on LabviewBook.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
(A)
False
512
Boolean A
Result
Boolean B
Yes
No
(B)
Answer
LabviewBook.lvproj/My Computer

Figure 2.2: A case structure showing its false block (A), and a ternary operator (B).

number and the result is given, the indicator outside the case structure. If the value is false, 512 is squared and the result given.

A case structure also accepts enumeration values. This is similar to a switch structure in `C` programming. In figure 2.3, a value named “Multiply” is placed into the case structure. This allows you to have many paths of execution in a single structure. In the example, a simple calculator is made which takes two input numbers and an enumeration condition, the block then executes the proper logic and hands you the result at the end. One of the mini projects at the end of this chapter will require you to create a 4-function calculator.

2.1.2 Ternary operators

The plural in the title is misleading, there is only one ternary operator in LabVIEW. Figure 2.1 (B), on page 20, is this little function in action. You supply it with a boolean value and outputs either its truth or false value. In this example, if the boolean value is true, then the ternary operator outputs the string “Yes”.

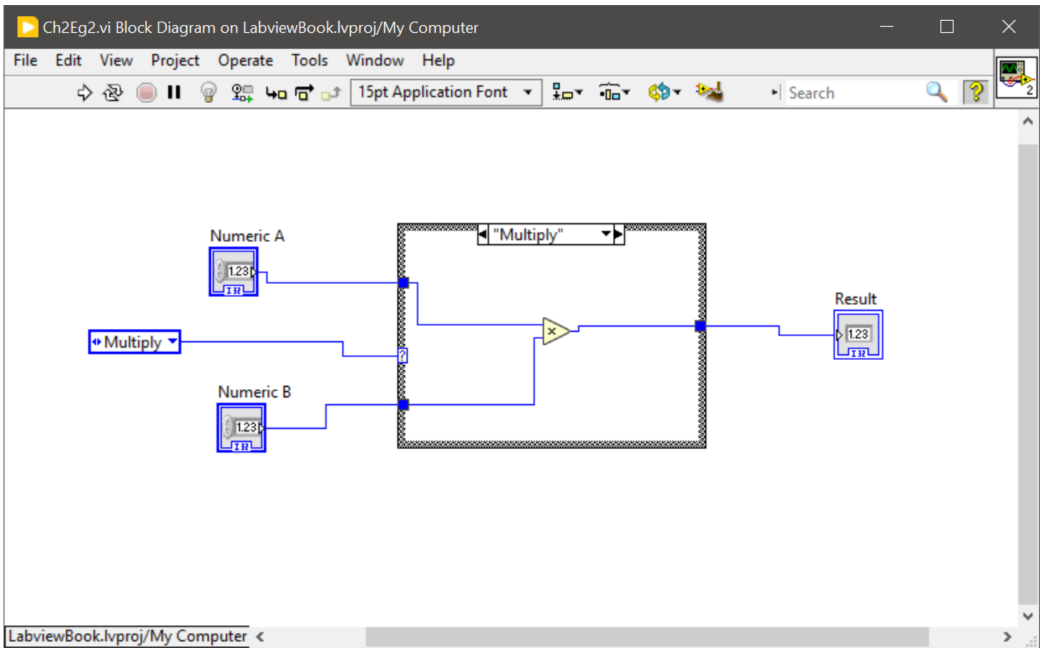

Ch2Eg2.vi Block Diagram on LabviewBook.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Numeric A
"Multiply"
Multiply
Numeric B
Result
LabviewBook.lvproj/My Computer

Figure 2.3: A simple calculator using an enumeration type and a case structure.

The only caveat is that you must use the same variable type for its true and false inputs, this type will also be its output type. There really isn't anything more special about the ternary operator. It is possible to implement a ternary operator using a case structure, an exercise left for the reader.

2.2 Code looping structures

In the previous section, I mentioned that my computer could make a billion decisions in 5.76 seconds. The program certainly did not contain a billion case structures, it only had one.

It is quite rare for a program to only make one decision before execution stops, most probably you have a few decisions which would need to be repeated thousands of times per execution. One may even say that you would like to have a program perform a "loop" of decisions. This is were loop structures come into play.

The two loop structures available to us in LabVIEW is the "For Loop" and the "While Loop". The for loop executes a set number of times while the while loop executes until some specified condition is met. These two structures are closely related however since the for loop can pretend to be a while loop. It is also worth mentioning in passing, to those of you with previous programming experience, that both loops act as 'do-while" loops since they execute at least once before evaluating their conditional terminals.

The flow of information in LabVIEW is rather unconventional, this will be elaborated upon in a future section, but for now there will be references to arrays, time functions, and shift registers without going into depth for either topics.

2.2.1 For Loops

A for loop executes a predetermined amount of times. Figure 2.4 is perhaps the most simple, yet interesting, loop possible in LabVIEW. You set the number of times the loop should run in the "Number of times" control and the loop shows you the iteration step number along with a random floating point number between 0 & 1. The little watch function in the bottom right corner is to slow the loop execution speed to twice every second so you are able to observe the number generated.

The example is heavily constrained however, there is no way for the information generated to be stored in memory so there is no knowlage for what the random number was in the previous iteration let alone what the series of generated numbers were.

Just by introducing a concept known as a shift register, we can actually start making useful programs, or what ever useful means in this case. A shift register allows you to store a variable from a current loop iteration so that it is useable in the next iteration. To illustrate this, let us create a loop that prints out, in sequence, the first 10 numbers in the Fibonacci sequence.

The Fibonacci sequence, defined by the recurrence relation:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

An implementation of this may be found in figure 2.5. On the right hand side of the loop structure is a little blue arrow in a box, this is the terminal where you value the number you want available to the next loop iteration. To add a shift register to a loop, `right click` on any side of the loop frame

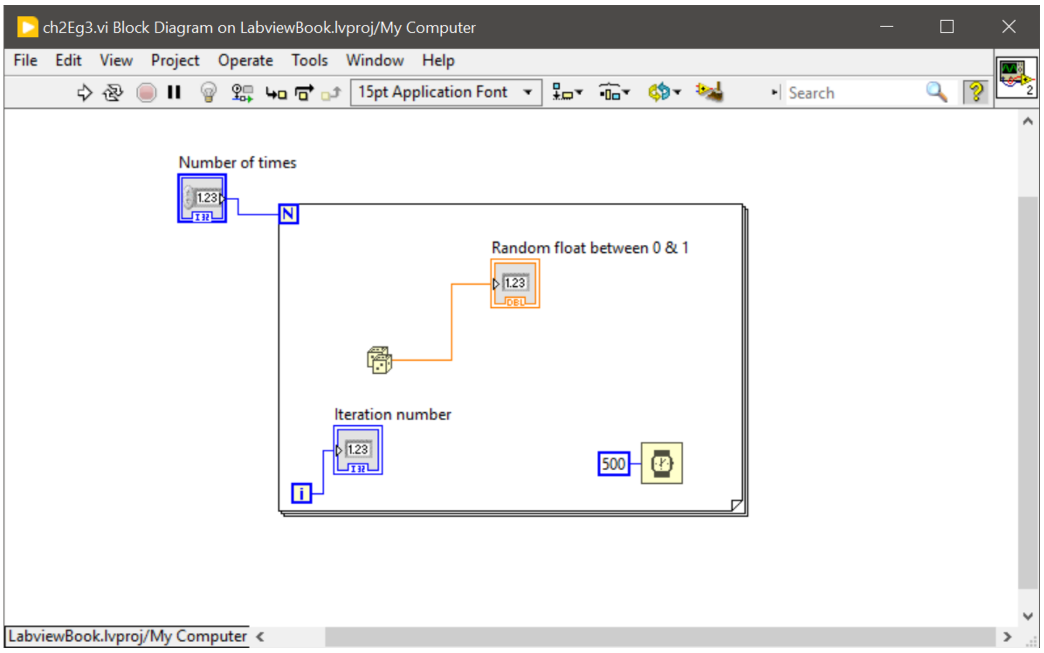

ch2Eg3.vi Block Diagram on LabviewBook.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Number of times
N
Random float between 0 & 1
Iteration number
i
500
LabviewBook.lvproj/My Computer

Figure 2.4: A simple loop that ejects random numbers, two every second.

and select "Add Shift Register" from the drop-down menu.

You may notice in that example that the left hand side has two little arrows stacked on top of each other, this gives you access to the previous iteration values. In this case the top arrow is the value of iteration n-2 and the bottom arrow is the value of iteration n-1. You can extend or reduce the number of iterations available by clicking and dragging the bottom edges of the shift register.

The values feeding into the left of the shift registers initialise the two first numbers in the sequence. If you do not do this, the two initial numbers would be 0. Do not rely on such implicit details, if you want to have the first numbers to be 0, wire in a zero constant explicitly. This will also show your intent in the code and not leave people reading the code guessing.

The indicator wired to the little blue `i` gives you access to the current iteration, starting from 0 end ending in $n-1$. You don't need to memorise this, just hover your mouse over the little blue `i` and it will tell you. Although not exactly useful in this example, it makes working with arrays significantly easier so you should look forward to that.

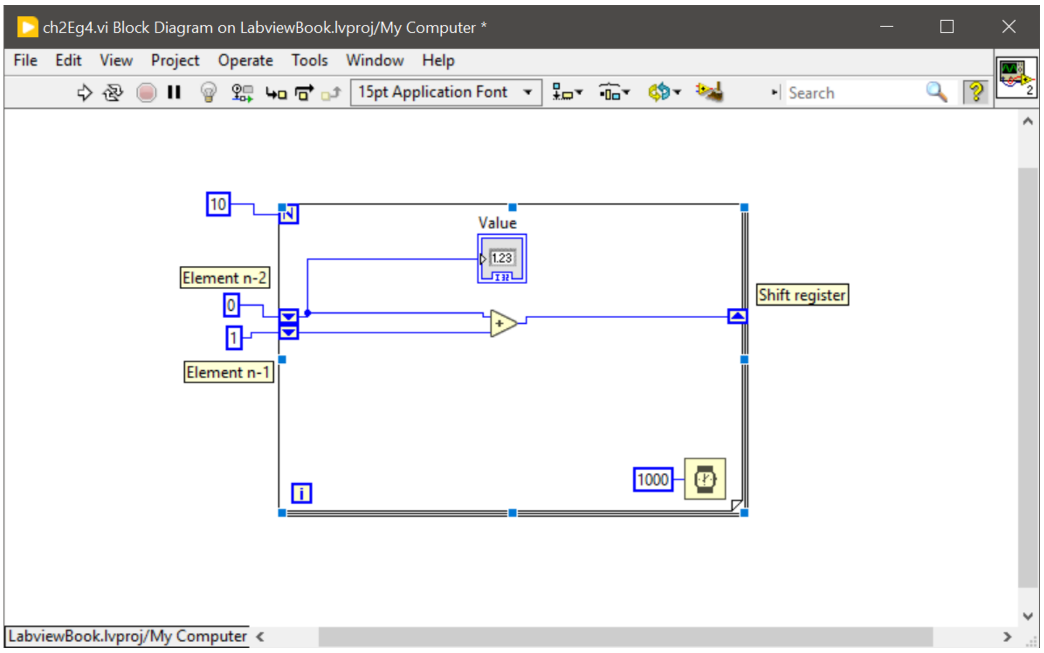

ch2Eg4.vi Block Diagram on LabviewBook.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
10
N
Value
1.23
Element n-2
0
1
Element n-1
Shift register
1000
i
LabviewBook.lvproj/My Computer

Figure 2.5: A loop that displays the Fibonacci numbers, updating every second.

2.2.2 While Loops

Suppose your program needs to monitor a temperature sensor and it requires the temperature to be above some set point before it turns on a fan, say. You can not know when this temperature threshold is reached, if it is ever ever reached. In such a case, a while loop allows us to execute some code only leaving the loop once some condition is met. If the condition is never met, then the loop never ends.

Figure 2.6 is the previous Fibonacci sequence program modified to use a while loop instead. Gone is the little blue `N` in the top left corner of the loop. Instead, a little red octagon is present at the bottom right corner of the loop. This is the conditional terminal

In this example, if a number in the sequence is greater than 100, the comparison function sends a “true” value to the condition terminal which causes the loop to terminate. If this is slightly confusing, just remember “Red means stop!.

If you `right click` on the condition terminal, you may change it to a

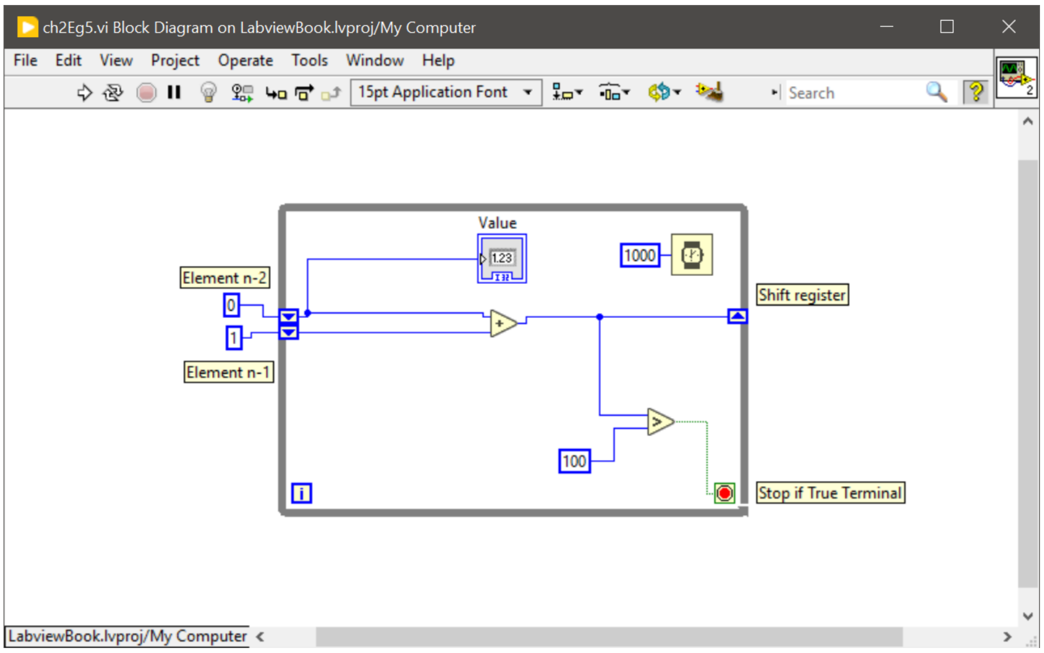

ch2Eg5.vi Block Diagram on LabviewBook.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Value
1000
Element n-2
0
1
Element n-1
Shift register
100
Stop if True Terminal
LabviewBook.lvproj/My Computer

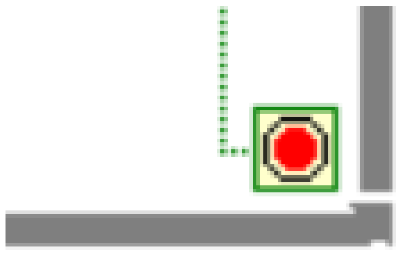
Figure 2.6: A loop that displays the Fibonacci numbers less than 100, updating every second.

"Continue if true" condition. In this case, the while loop will only terminate when it receives a "false" value. Conveniently, the terminal changes to a green circular arrow, in other words "Green means go!. Figure 2.7 shows the two available condition types side by side. Yes that is a little octagon stop sign, it took me almost 5 years to notice that.

Much like in other programming languages, infinite loops are considered unwanted behaviour. While loops must have their conditional terminals wired, even if your program would run for days or weeks on end, it should have a button to safely exit its execution. This will become more apparent in Chapter 3 where we will be programming hardware, other than your computer that is.

2.3 Aggregate data types

From here on out, the training wheels are off. The concept of an array is simple, the explanation in section 1.3.2 is as detailed as needed and will not be repeated here. Working with arrays in LabVIEW is tricky however, in the


(a) The "stop if true" while loop condition terminal.

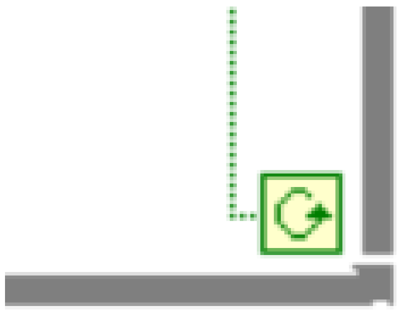

(b) The "continue if true" while loop condition terminal

Figure 2.7: The two available while loop conditions, also notice how the edge of the frame makes a little arrow.

next few pages we will dissect this creature and reveal some of the more ugly sides to this programming language.

2.3.1 Arrays

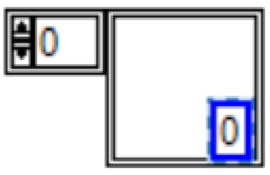
Creation

Technically an array is also a type of structure in LabVIEW. In the block diagram view, a placed array constant contains nothing, not even a type. You provide the type by placing a constant inside the array box. This is seen preformed in figure 2.8a. In this particular case, an integer constant creates an empty integer array, seen in figure 2.8b.

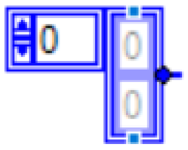
The two little blue terminals in figure 2.8b are drag points to change the amount of visible elements in the array. Even if only one element is shown, you may cycle through the array using the arrows next to the 0 seen in the figures. This 0 corresponds to the index of the top visible element in the array. Array controls are made in a similar fashion on the front panel. Once you have made an array, you can start adding values. You do this by clicking on the grey 0 and typing in a number.

Manipulation

Figure 2.9 shows how the numeric functions may be used to operate on arrays. These operations are preformed on an element by element basis. Figure 2.10 shows the result of these two operations. The result is truncated to the size



(a) An empty array constant, the integer constant has not been placed yet.



(b) An empty integer array constant, expanded to show two elements.

Figure 2.8: Creating an array constant.

of the smallest array.

There are also a few array specific functions in the numeric palette which yield the sum of all the array elements or the product all the elements. You should experiment with these on your own.

Exploitation

The real power, and complexity, of arrays emerge when operating on the elements themselves. Figure 2.11 shows some of the most important operations able to be performed on arrays.

The array size function is rather self explanatory, it just gives you the number of elements in the array.

The index array function picks out an element of an array at the index you specified. Index 0 being the first element in an array. Getting the last element of an array requires you to know its size, since indexing starts at 0, you need to decrement the size before sending it to the index function.

Unlike in `C` programming, indexing an array out of bounds in LabVIEW will not destroy your computer. For numeric arrays, the index function will return 0 upon indexing an out of range value and will continue with your program as if nothing has happened. Soon enough you will discover on your own what an "off by one" error is.

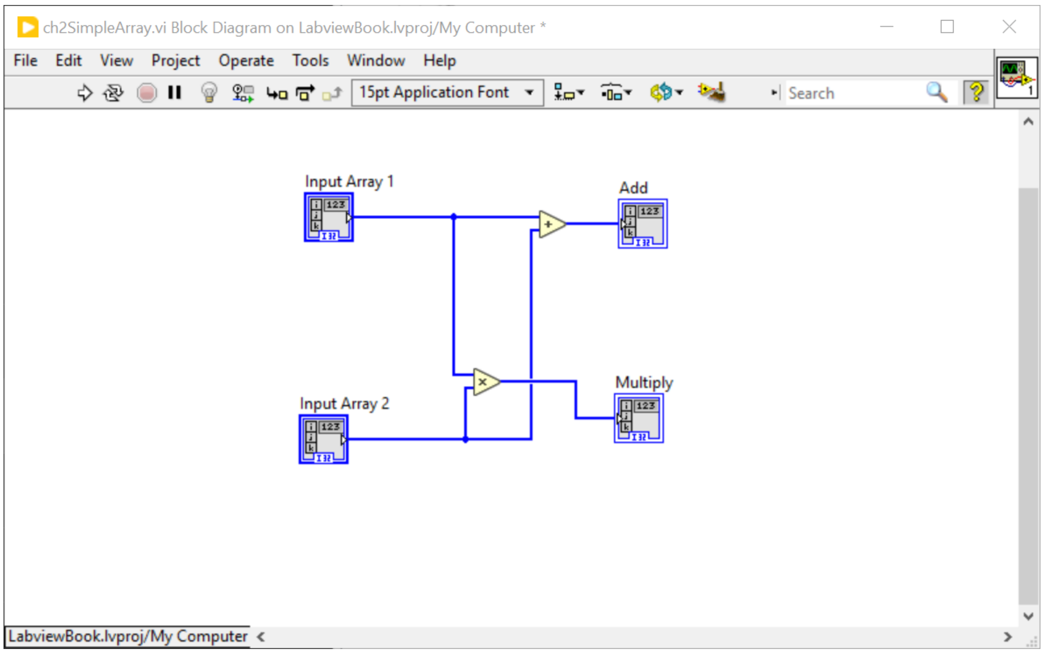

ch2SimpleArray.vi Block Diagram on LabviewBook.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Input Array 1
Add
Multiply
Input Array 2
LabviewBook.lvproj/My Computer

Figure 2.9: Two operations on array variables, these are the normal functions found in the “Numeric” folder in the functions palette.

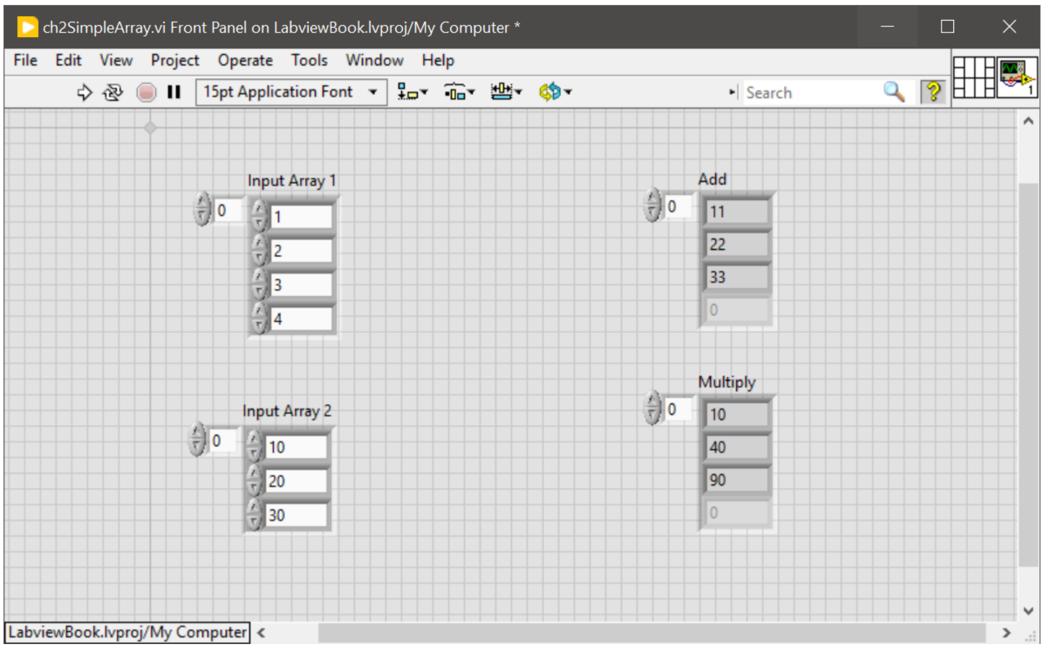

ch2SimpleArray.vi Front Panel on LabviewBook.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Input Array 1
0
1
2
3
4
Add
0
11
22
33
0
Input Array 2
0
10
20
30
Multiply
0
10
40
90
0
LabviewBook.lvproj/My Computer

Figure 2.10: The results of elementary operations on arrays.

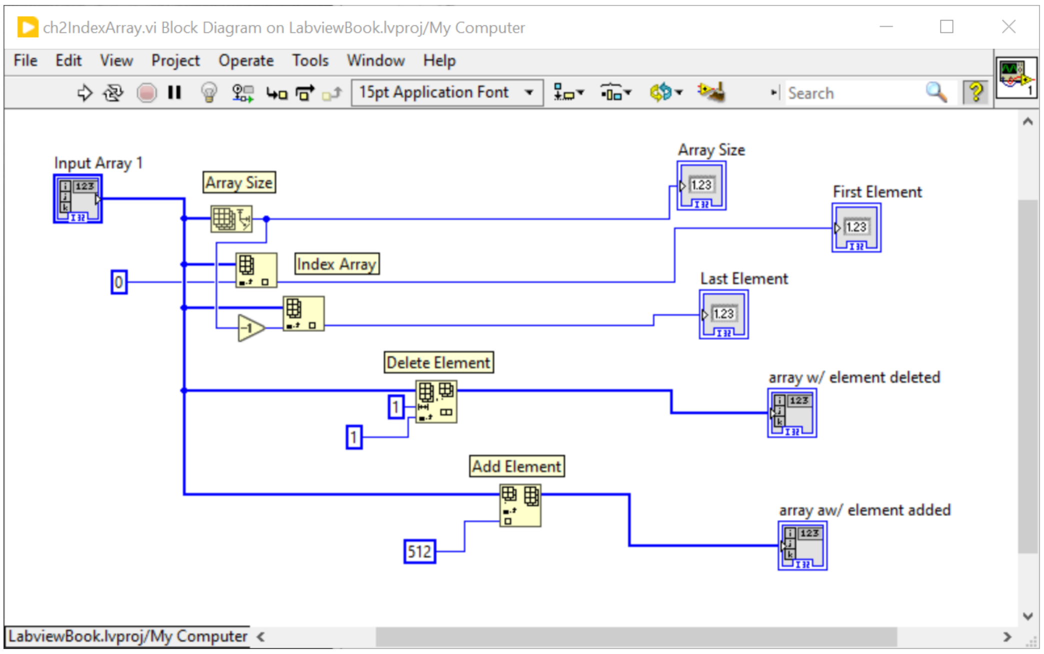


Figure 2.11: Fundamental operations on arrays.

The delete array element function takes two arguments, an index value and length. This deletes the value at the selected index. If the length is more than one, the selected index and subsequent indexes are deleted. What do you think happens when you wire in a negative length value? Sadly nothing, no element is deleted if a number less than 1 is given.

The add array element function adds an element into the index you specify, think of it like cutting in line at the bank. If you cut in line at position three from the front desk, the person behind you is now fourth and you are now third. I do not condone this behaviour, please exercise this in a computer program and not in real life.

Figure 2.12 shows the results of these operations. The next step is to put these functions into context so you can see how they are used in more useful programs.

Using arrays with loops

In the previous section, we used loops to create a sequence of numbers. There was no way to store the numbers so the value was just printed to the screen

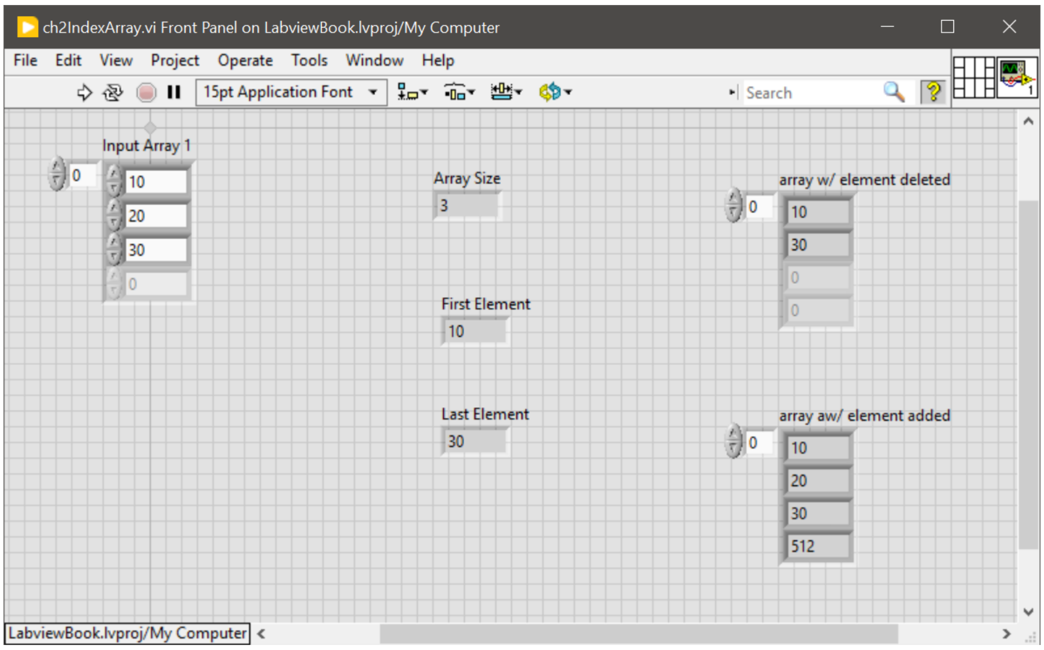


Figure 2.12: The results of fundamental operations on arrays.

on every loop.

Figure 2.13 shows a very simple way to extract a sequence from a loop. By branching the wire going into the shift register and connecting it to the while loop, LabVIEW creates what is known as an "auto indexing tunnel". The terminal has what looks like a blue `[]` inside of it. If yours is just a solid blue colour, `right click` on the terminal and select the indexing option. A solid blue box in this case means that the while loop will supply only the last value to that output.

Figure 2.14 shows the result of this computation, it is missing the first 0 and there is another small mistake. We will fix both these mistakes in a moment.

Auto indexing of tunnels also work as inputs. Feeding an array into a for loop allows you to iterate over all its contents. In figure 2.15, a previous example of element wise addition is recreated. The loop indexes input arrays, you may find the exact index by using the blue `i`, saving you the effort of manually indexing and inserting elements.

This tool becomes exceptionally powerful when combined with Arduino

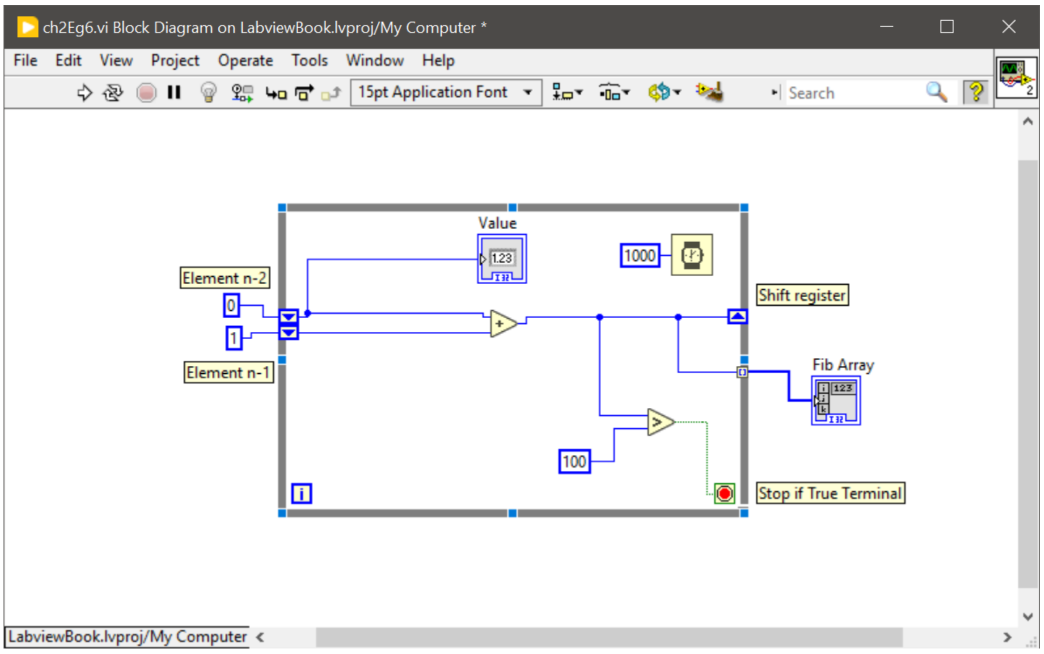

ch2Eg6.vi Block Diagram on LabviewBook.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Value
1000
Element n-2
0
1
Element n-1
Shift register
Fib Array
100
Stop if True Terminal
LabviewBook.lvproj/My Computer

Figure 2.13: Using auto indexing tunnels to create an array.

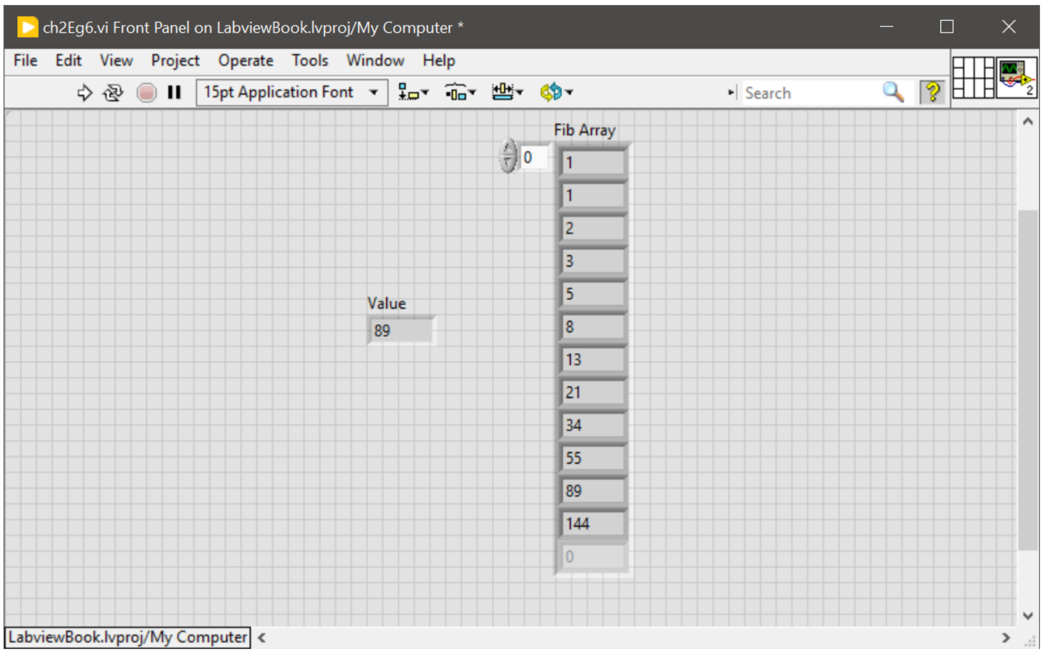

ch2Eg6.vi Front Panel on LabviewBook.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Fib Array
0
1
1
2
3
5
8
13
21
34
55
89
144
0
Value
89
LabviewBook.lvproj/My Computer

Figure 2.14: The output of an auto indexing tunnel, do you notice what is wrong here?

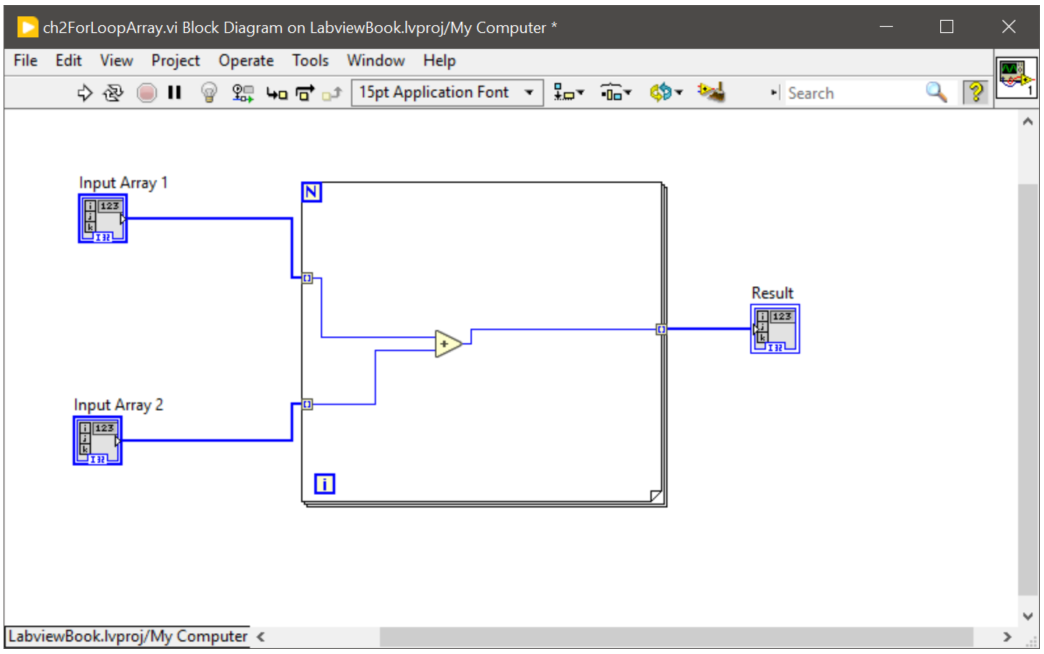

ch2ForLoopArray.vi Block Diagram on LabviewBook.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Input Array 1
Input Array 2
N
i
Result
LabviewBook.lvproj/My Computer

Figure 2.15: Using auto indexing tunnels to create an array.

hardware, or any instrument really. Suppose you are controlling a LED, you can build a function that turns this LED on and off. You can then create an array with ones and zeros and feed it into a for loop. The for loop will pick off the elements one by one and send it to your LED function. If this sounds stupid to you, how did this document get to your computer screen, or into the printer if you are reading a hardcopy? The answer; bit by bit.

Let us now return to the Fibonacci sequence problem, in the previous example, the first 0 was missing and the value 144 is found hanging on the end of the array, this is larger than 100, so it seems our condition value is ignored.

This occurs because, even though the value fed into our greater than function, the rest of the iteration executes. Thus the number 144 comes along for a ride and is not cast into the depths of oblivion.

Figure 2.16 is, perhaps, an extravagant example on how to fix the two small mistakes in the previous example. There is a simpler fix which just requires the relocation of a single wire, can you find that fix?

You should walk through the example in figure 2.16, going step by step following the wires. Draw up this program on your own and see how it works.

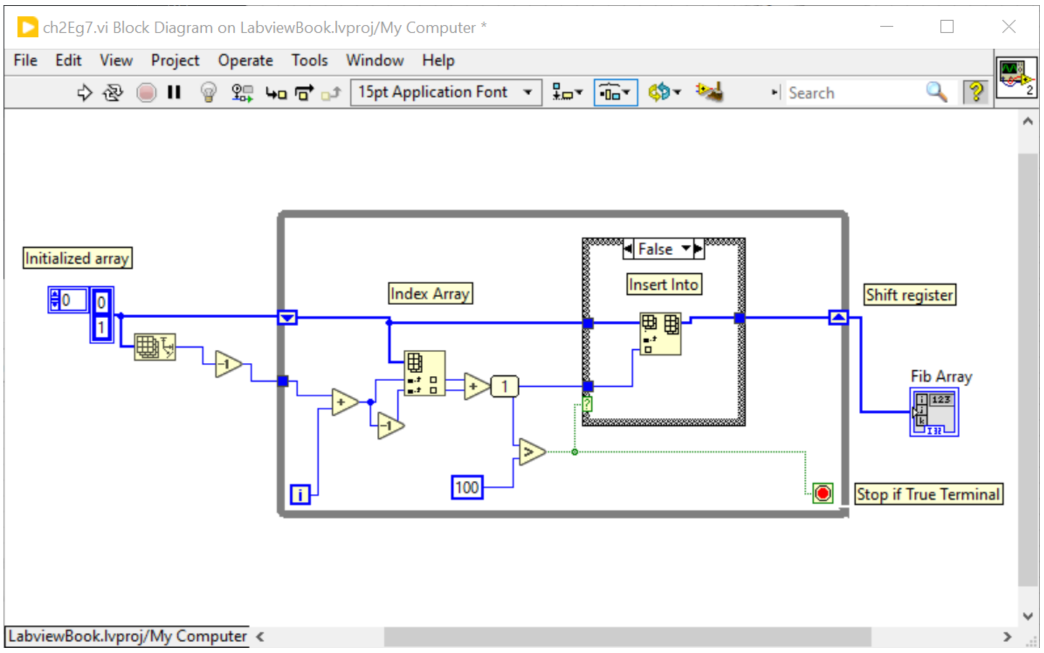

ch2Eg7.vi Block Diagram on LabviewBook.lvproj/My Computer *
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Initialized array
Index Array
False
Insert Into
Shift register
Fib Array
100
1
Stop if True Terminal
LabviewBook.lvproj/My Computer

Figure 2.16: A rewrite of the program that gives the expected results.

You can follow how LabVIEW executes this program by clicking on the light bulb in the top bar. This is the "highlight execution" button and is a useful tool to debug a program. If you grow tired of seeing the little signals passing through the wires, press the red button to stop the program.

2.3.2 Clusters

It is possible to group variables into bundles called clusters, this allows you to have one wire acting as a bus, caring around variables across your program. Figure 2.17 shows how this is done using the "Bundle" function from the function palette.

You can read an element from a cluster using either the "Named Unbundle" command or the "Unbundle" command. The named version uses the name of the wire being bundled, in this case the name is inherited from the control. This allows you to see at a glance what value you would like to choose. You can click and drag the blue terminals on the top or bottom of the bundle functions to increase or decrease the number of terminals. You can change the selected variables by clicking on their names and selecting a different one.

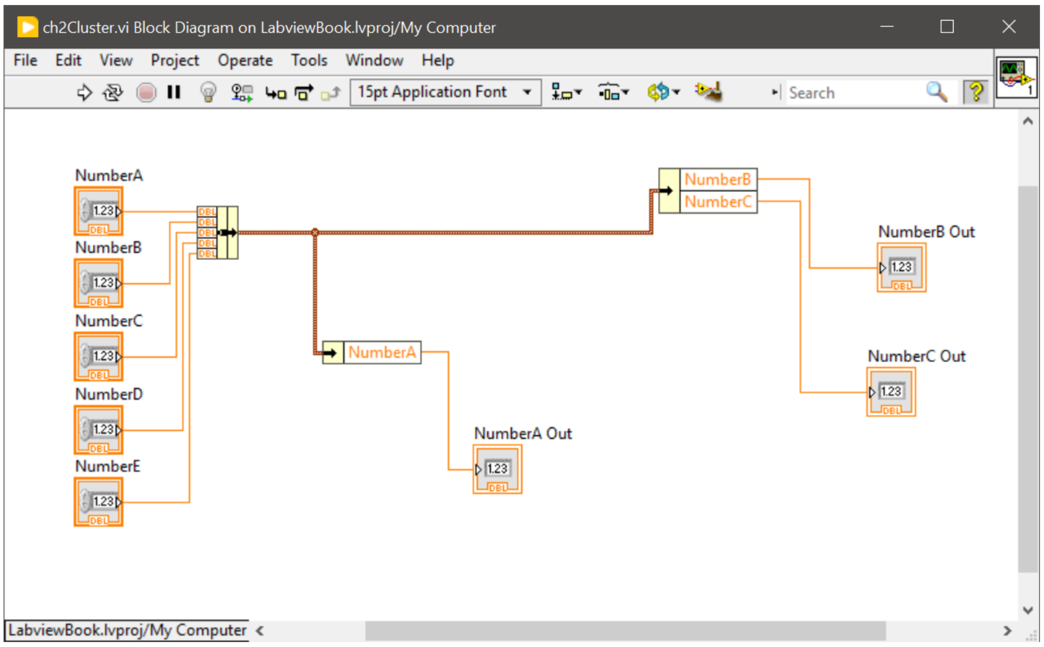

ch2Cluster.vi Block Diagram on LabviewBook.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
NumberA
NumberB
NumberC
NumberD
NumberE
NumberA
NumberB
NumberC
NumberA Out
NumberB Out
NumberC Out
LabviewBook.lvproj/My Computer

Figure 2.17: A group of numbers being bundled and unbundled.

In the simple example, figure 2.17, you can see that it is possible to unbundle a variable at random, you do not need to unbundle everything at once. The name is slightly misleading however, when you unbundle a variable from a bus, you are not actually removing it, you are just making a copy of it. To change that variable, you have to either unbundle all the elements, change one, and wire everything back into a new bundle. This is tedious and error prone. A better way is to wire in an old bundle, into a “Bundle” function, and only selecting the variables you want to overwrite. The other values will stay unchanged.

This is all that you need for now when it comes to clusters. They are exceptionally useful for functions and keeping track of variables which are related to one another. There is also the topic of “Type-definitions” which properly exploits the utility of clusters. You can look forward to that in a later chapter.

2.4 Creating functions

You have no doubt seen so far how cluttered your block diagram can get. Figure 2.16 is a program that does one thing, generate an array of the Fibonacci numbers, how busy would the diagram get if we build upon this example? Very busy, so busy that you would wish you learned `C` instead.

You may clean up your code by grouping together functions in a little block, which is also known as a function. Before we get to that however, let us first take a detour into project management in LabVIEW. Having a project groups your functions together and makes it easier to build upon your previous work.

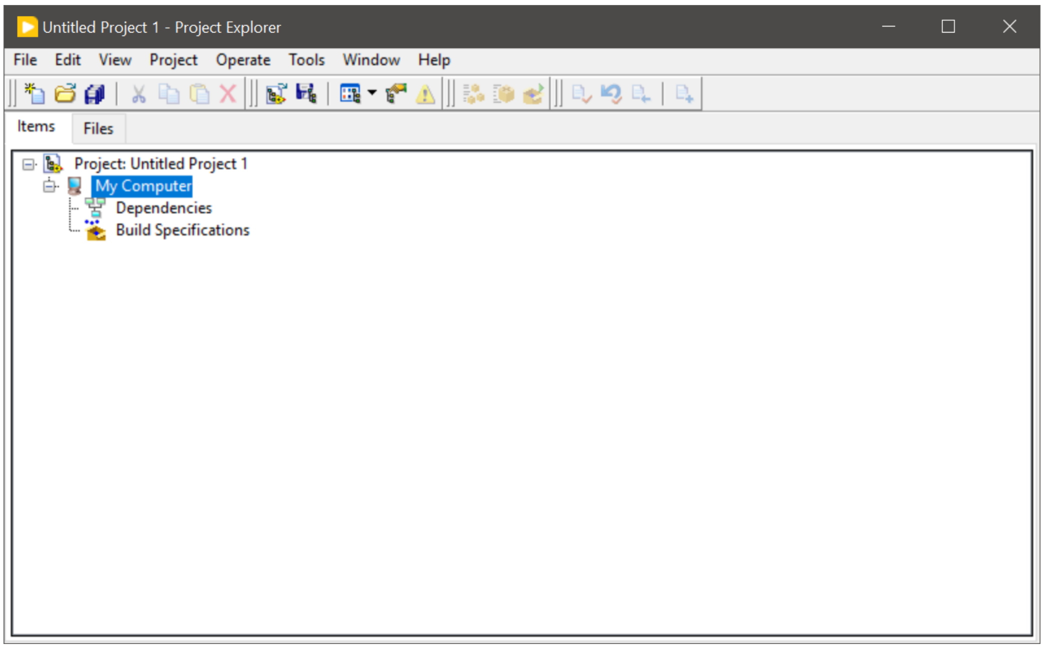
2.4.1 Project Management

To create a project, on the startup screen, press the button that says "Create Project", see 1.2 on page 4. This will ask you what type of project you would like, there are quite a few templates to choose from, but for now just click on "empty project". A window will open up with your new empty untitled project like in figure 2.18.

From here you can create a new VI by going to "File" and pressing "New...". You can also create a virtual folder in the project by `right clicking` and pressing "Create virtual folder". This folder does not exist in your file system, instead it allows you to organise your project in meaningful sections. The examples in this book is also contained in a project, see figure 2.19.

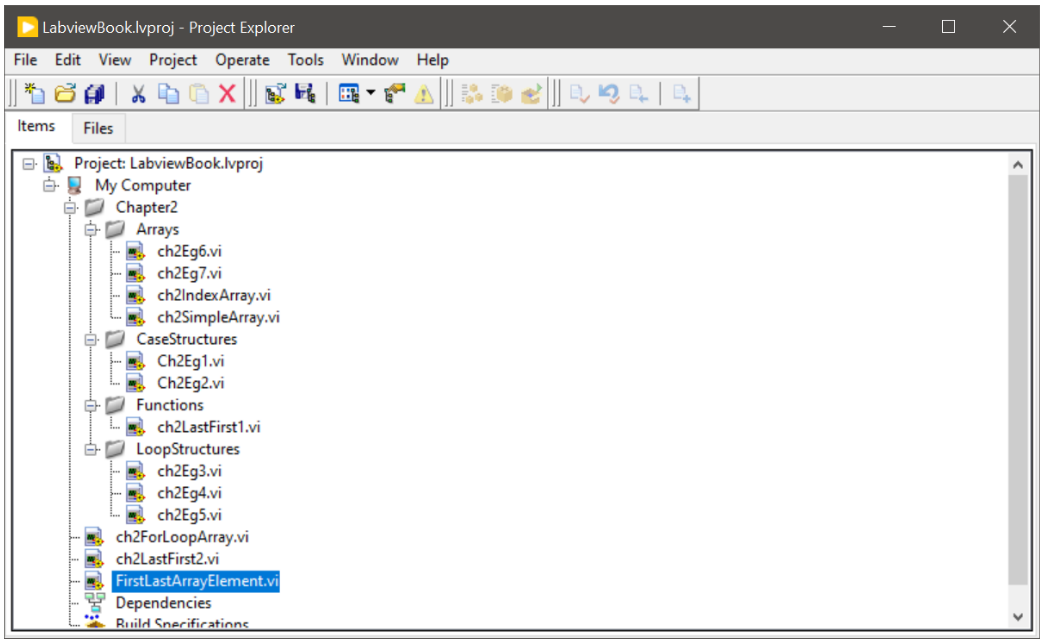
After you have created a VI in your project, it should open up by itself so you can start programming. You may also `double click` on a VI in the project manager to open up a particular VI. After you are done creating a program for launching ballistic missiles, you can save everything you are working on by going to file and then selecting "Save All". After that you can close the project manager and it will close all the other open VIs that are part of that particular project.

This is by no means a complete overview of the project manager, but it should be enough for you to get started. Once you have built some programs as functions, you can simply drag them into another VI to use them. We will get to that very soon.


Untitled Project 1 - Project Explorer
File Edit View Project Operate Tools Window Help
Items Files

Project: Untitled Project 1
My Computer
Dependencies
Build Specifications

Figure 2.18: A freshly baked project, it has nothing in it yet, you provide the rest.


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

```
Project: LabviewBook.lvproj
  My Computer
    Chapter2
      Arrays
        ch2Eg6.vi
        ch2Eg7.vi
        ch2IndexArray.vi
        ch2SimpleArray.vi
      CaseStructures
        Ch2Eg1.vi
        Ch2Eg2.vi
      Functions
        ch2LastFirst1.vi
      LoopStructures
        ch2Eg3.vi
        ch2Eg4.vi
        ch2Eg5.vi
    ch2ForLoopArray.vi
    ch2LastFirst2.vi
    FirstLastArrayElement.vi
    Dependencies
    Build Specifications
```

Figure 2.19: The project used to keep track of all the examples in this book.

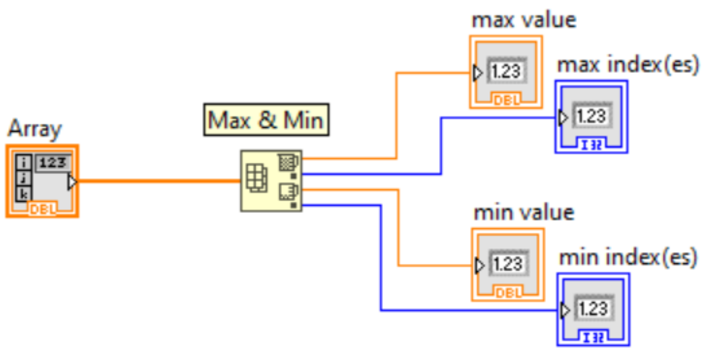


Figure 2.20: The inputs and outputs of the Max & Min function, we will use this as inspiration for our own function.

2.4.2 Creating a Function

Just as a reminder, every block you have placed down in the block diagram is known as a function. For example, let us look at the "Array Max & Min" function, in figure 2.20. I have wired in all the inputs and outputs to give you an idea of how it is used in an actual program. This is what we will be working toward.

Suppose we go back to figure 2.11, on page 30, we extract the first and last element of an array by grabbing index 0 and getting the last index by getting the array length and decrementing that by one. I have created a new VI to hold this program, seen in figure 2.21.

We can now go one step further and use the index values to extract the two elements from an array. This makes sense because you either need the index values themselves or the actual values. You could create a program to do each function separately, but it does not hurt to do both. Figure **??** shows how this is implemented. Note that every value not used is simply thrown away. If you are worried about program performance, just stop. Compiled LabVIEW programs are quite good at getting rid of parts of a program that is not used.

Now comes the fun part, drag a box and select the entire program excluding the control and indicator terminals, see figure 2.22a. While still having the contents selected, from the taskbar go to "Edit" and select "Create subVI

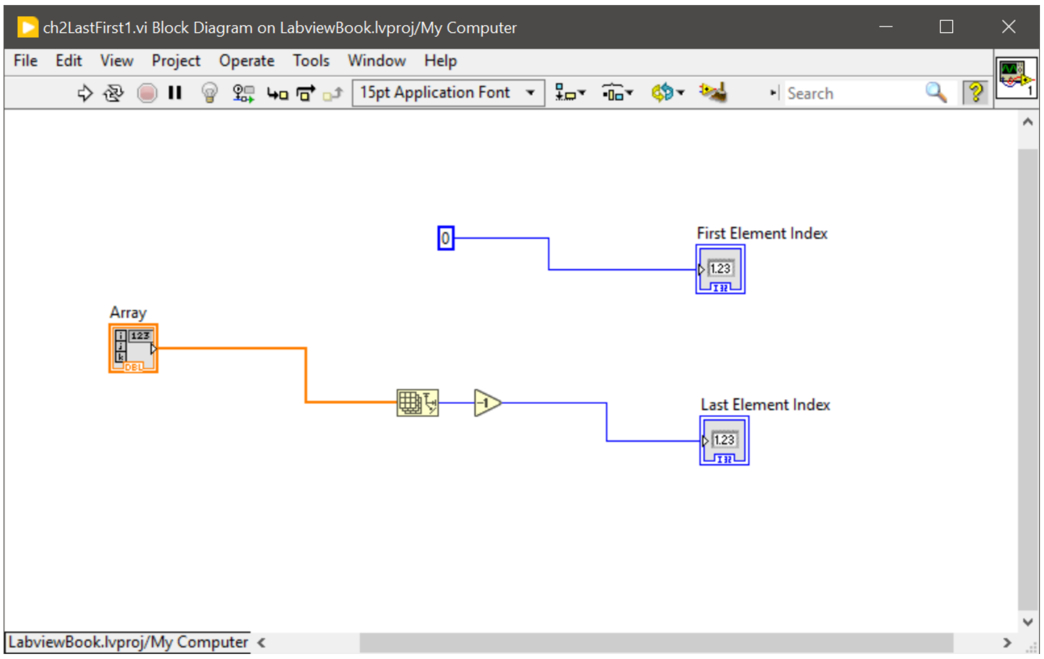

ch2LastFirst1.vi Block Diagram on LabviewBook.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
First Element Index
Array
Last Element Index
LabviewBook.lvproj/My Computer

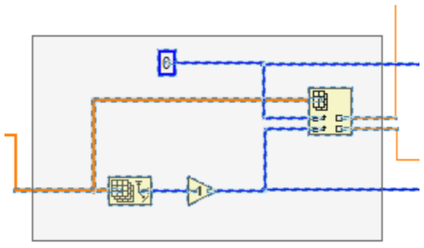
Figure 2.21: Prototype of a First and Last index function.

from selection". This wall squash everything into a little block, seen in figure 2.23.

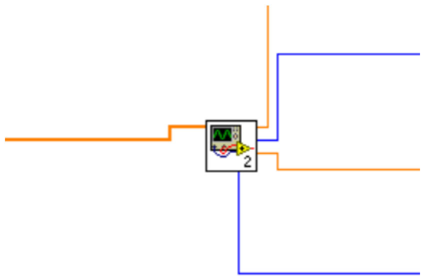
From a glance, it is not very clear what this function does, `double click` on this block to open up the VI. By a fluke, this example almost looks identical to the block diagram previously, bar the change in indicator names, see figure 2.23. Rename these so that they make some sense.

You can now double click on the icon in the top right corner of the window, figure 2.24 is the window that should open. This is the icon editor, it is like paint. Let your creative juices flow and create an icon that would make it clear what this function does among the hundreds of other functions in your programs. My attempt at an icon may be seen in figure 2.25.

With your new icon, you should focus on organising the wiring terminals. Figure 2.26a shows how your terminal grid might look like. Note that this is only seen in the front panel window, not the block diagram window. If you `left click` on any of the colour filled terminals, you will see a blue box appear around the control that is associated with that terminal. The current layout might not make sense so let us redo it completely. `Right click` on the grid and select "Disconnect All Terminals" from the menu. You can then



(a) Everything in this box will go into our function.



(b) A newly created subVI, it is rather ugly.

Figure 2.22: Creating a subVI, also known as a function.

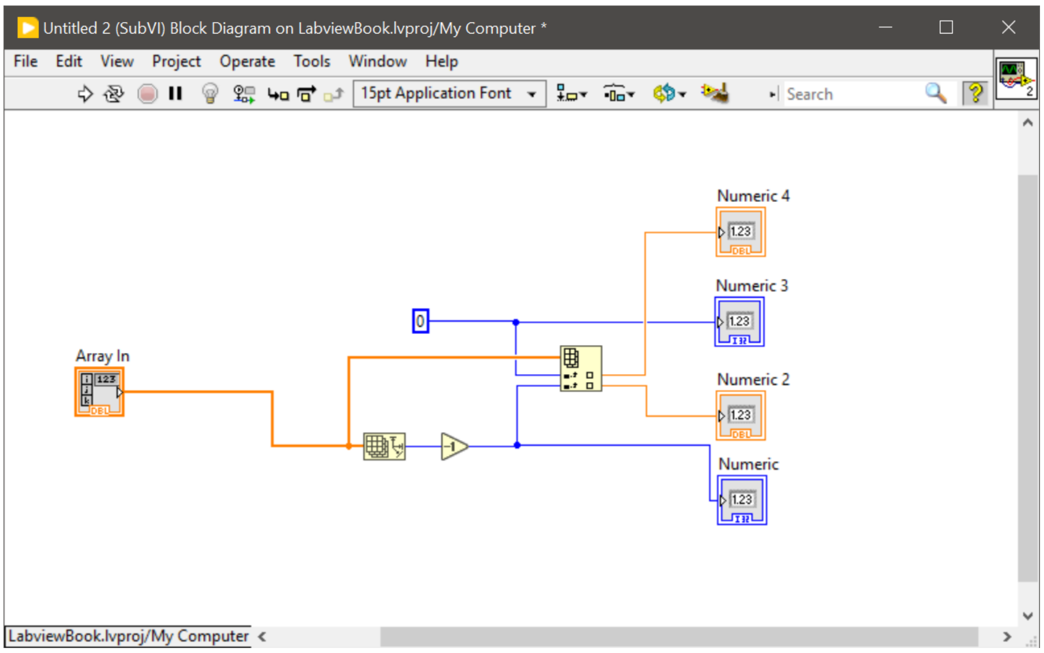


Figure 2.23: The block diagram of the auto subVI function.

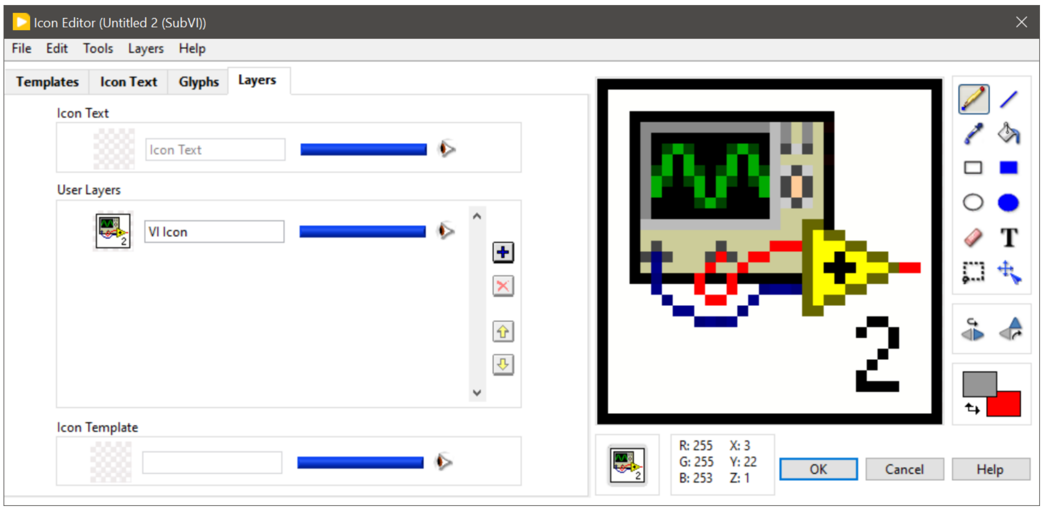

Icon Editor (Untitled 2 (SubVI))
File Edit Tools Layers Help
Templates Icon Text Glyphs Layers
Icon Text
User Layers
VI Icon
Icon Template
R: 255 X: 3
G: 255 Y: 22
B: 253 Z: 1
OK Cancel Help

Figure 2.24: The icon editor program.

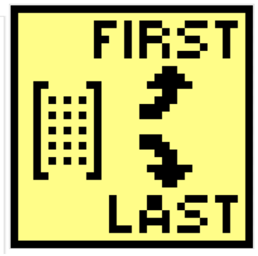
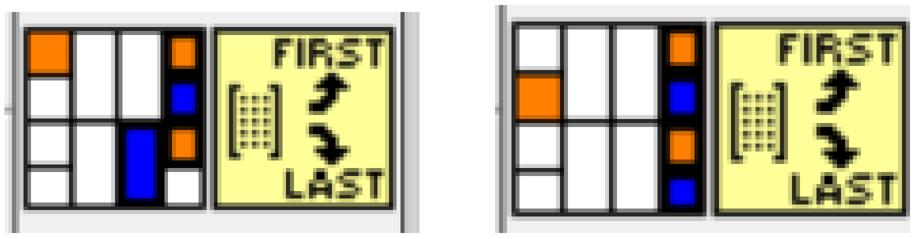


Figure 2.25: This icon conveys more or less what the function does.



(a) Automated creation of the terminal grid.

(b) A manually wired input and output grid.

Figure 2.26: The terminal grid for a function block.

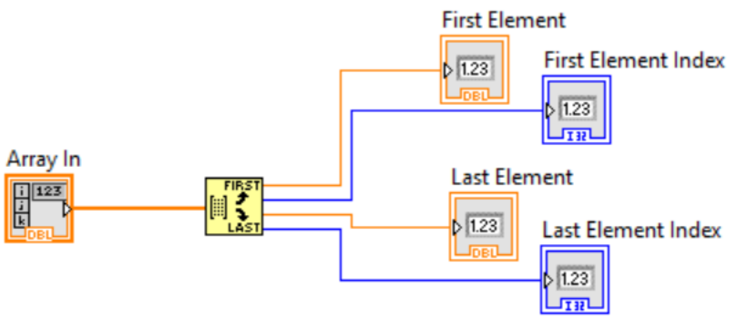
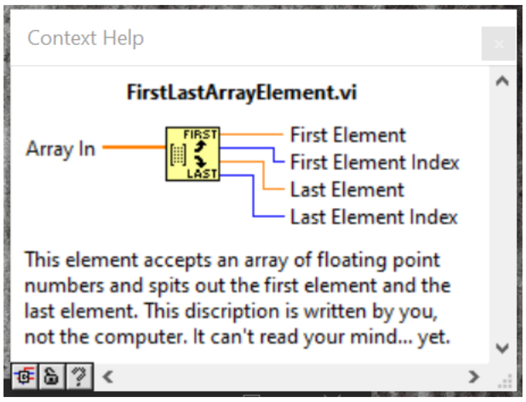


Figure 2.27: A fresh new function out of the oven, ready to be used in another program.

connect a control by clicking on an empty terminal (white) and clicking on a front panel element.

By convention we assume that inputs are on the left and outputs are on the right. Figure 2.26b shows what might be a good layout, the orange square on the left is the array input while the alternating orange and blue indicators are the element and index outputs respectively. Here the first element comes from the top and the last element comes from the bottom, just like the icon would suggest. We now have our function ready for use, see figure 2.27.

If you are wondering why we renamed the inputs and outputs to something useful, figure 2.28, is the context help message that appears when hovering over our function. The figure also shows a description of the function, this has to be supplied by you. You can do this by going to “File” in the taskbar and selecting “VI Properties”, in this window, using the list-box named “Category” select the “Documentation” panel. Here you can write a little paragraph on your function.



This element accepts an array of floating point numbers and spits out the first element and the last element. This discription is written by you, not the computer. It can't read your mind... yet.

Figure 2.28: The context help message for FirstLastArrayElement.vi.

2.4.3 Using a Function

We are now ready to clean up the example in figure 2.11, delete the part of the program that is being replaced with the function we just made. From the project manager window, click and drag the function over to the block diagram where you want the new function to live.

You can now wire in the function and you should have something that resembles figure 2.29. That is it, I am sure you appreciate just how much neater the example looks.

This function has a problem, it does not know what type of array is piped into it. The built in functions found in LabVIEW adapt to their inputs. They are what is known as "malleable" VIs. This topic is not really advanced, it is easy to create a malleable VI yourself. The difficulty is dealing with type-casting and how LabVIEW deals with types internally. This will be covered in a future chapter.

2.5 Slightly more advanced Mini Projects

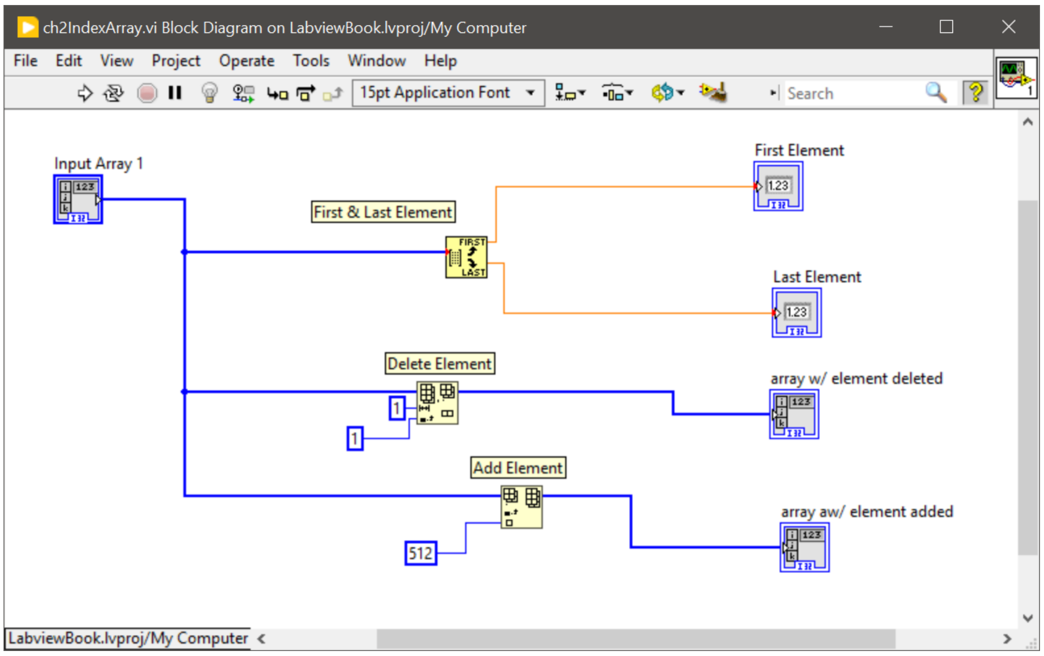


Figure 2.29: A much neater example using a function we made ourselves.

Chapter 3

LabVIEW and Arduino

3.1 Getting Started

3.2 Writing and Reading Ports

3.3 Quite Advanced Projects

Appendix A

Installing LabVIEW

Appendix B

Installing LiNX