BE/EE/MedE 189a: Design and construction of biodevices

Justin Bois

Caltech

Winter, 2017

© 2017 Justin Bois and Changhuei Yang.
This work is licensed under a Creative Commons Attribution License CC-BY 4.0.

2 Data types, cases, and subVIs

In this lesson, I will present a hodgepodge of topics to help you gain more capabilities with LabVIEW.

2.1 Data types

Whether you are programming in LabVIEW or pretty much any other language, you will be working with variables. The following can be properties of a variable:

1. The type of variable. E.g., is it an integer, like 2, or a string, like 'Hello, world.'?
2. The value of the variable.

LabVIEW has many data types. For now, we will focus on three data types, **numeric**, **string**, and **Boolean**.

1. **Numeric** data are integers and floats. For controls (remember, "control" is LabVIEW speak for "input", which you put in the front panel), the numeric data type may be specified by right clicking the object, selecting Representation, and then choosing among the data types. There are many integer types, e.g., U16 is an unsigned 16-bit integer. Floats can be double (DBL, the default), or single (SGL). You may also specify extended precision (EXT), which depends on your hardware and operating system. LabVIEW also has capabilities for complex numbers, such as CDB for a complex double.
2. **Strings** are an array of characters. Importantly string comparison in LabVIEW is done character-by-character comparing the ASCII value of the characters. E.g., abc < abcd < e. Identical strings are considered equal.
3. **Booleans** take on values of TRUE or FALSE. Internally, they are stored as 8-bit values. Anything that is nonzero and cast as Boolean evaluates TRUE.

These basic data types can be organized into higher order structures, such as **arrays** and **clusters**, which we will talk about in coming lessons.

To convert form one data type to another in the block diagram of a VI, you can use the operators in the Programming $\rightarrow$ Numeric $\rightarrow$ Conversion pane of the Functions palette.

2.2 Useful quick-keys

As a graphical language, you will be pointing and clicking with your mouse a *lot*. It is nice to know a few quick keys.

Ctrl-S	Save a VI
Ctrl-R	Run a VI
Ctrl-E	Toggle between the front panel and block diagram
Ctrl-H	Toggle the Context Help window
Ctrl-B	Remove all bad wires
Ctrl-W	Close the active window
Ctrl-F	Find objects or text
Ctrl-Tab	Cycle through LabVIEW windows

These are just a few. You should check out this reference guide from the National Instruments website.

2.3 Debugging tools

As with any code, you are likely to have bugs in your LabVIEW VIs. Fortunately, LabVIEW has some debugging tools to help you find and fix your bugs. If there is the graphical programming equivalent to a syntax error, that is an error that prevents LabVIEW from compiling your VI, the normally white, intact Run arrow will be gray and broken, as shown in Fig. 10. If you click on it, you will get an Error list, which will show descriptions of the errors in your front panel and block diagram. If you click the Show Warnings checkbox, you will also see warnings that do not necessarily prevent compilation, but are bad programming practice and can lead to bugs.

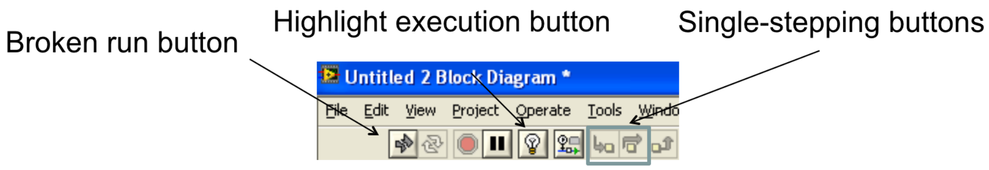

Broken run button
Highlight execution button
Single-stepping buttons
Untitled 2 Block Diagram *
File Edit View Project Operate Tools Windo

Figure 10: A block diagram toolbar with debugging tools highlighted.

You click the Highlight execution button (a lightbulb) to visually show the VI execution. This is much like a standard debugger and reduces performance. Finally, you can select single-step modes to step into or step over a node in a black diagram.

You can insert breakpoints into your code using the Breakpoint button on the Tools Palette (Fig. 11). You can also insert a probe, allowing you to observe variable values while your VI is running.

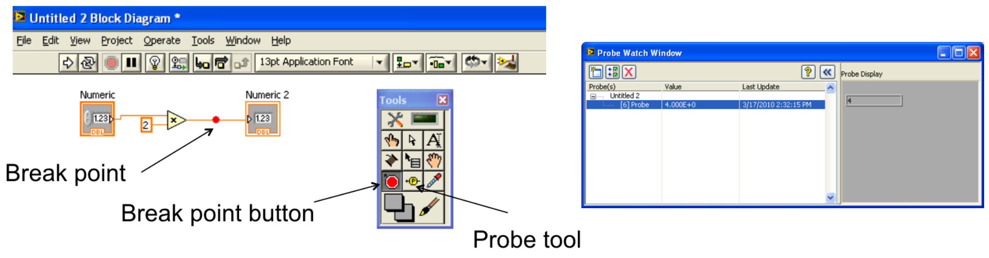


Figure 11: Left, illustration of break points and probe tools. Right, example probe watch window.

2.4 Case structures (LabVIEW's *if else if*)

Case structures allow you to implement *if else if*-type statements in LabVIEW. They behave like *switch case*s in C and Java. Their graphical representation is illustrated in Fig. 12.

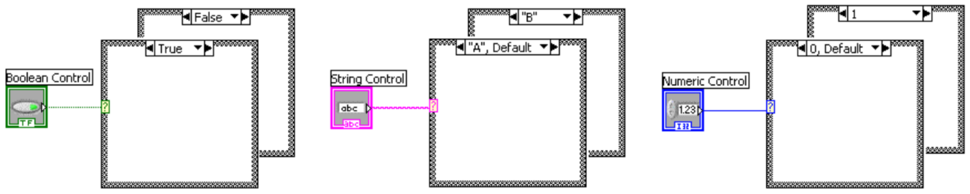


Figure 12: Case structures with different types of input.

Let us consider first the case structure that takes a Boolean as input. There are then only two possible cases, TRUE or FALSE. To enter objects into the case structure, you need to place objects and wires in the box. To toggle between what operations happen in the TRUE case and the FALSE case, you click on the menu at the top of the case structure box.

We can also input a string or Enum control (a ring-style control—remember "control" is LabVIEW speak for "input"). If this type of control comes into the case structure, you are unrestricted in the options for the cases, i.e., you are no longer restricted to either TRUE or FALSE. Therefore, you must specify a default case (akin to the else clause of an if else if statement).

Finally, you can have numeric control. LabVIEW has strange behavior for numeric control and, in my opinion, you are asking for bugs if you directly input numeric control. I will therefore not comment any further on this, and just advise you not to do it.

2.5 Modular programming: SubVIs

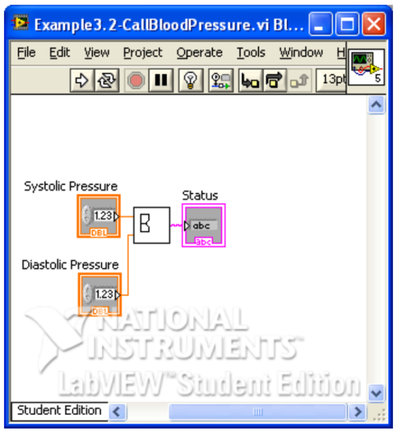
When you write text-based code, you often write functions that do specific, small tasks. They have a prototype; they takin input, perform operations on it, and then return output. You larger scale program or project calls these functions in succession, perhaps with logic and looping. This is generally good practice, to do **modular programming**.

The same is true for LabVIEW. *Any VI you write can be incorporated into another VI.* When a VI is used in another, it is called a **subVI**. When building your LabVIEW programs, you should create many small VIs that perform well-defined takes that are simply and clearly connected in your main VI.

To have a concrete example in mind, consider a VI that evaluates blood pressure measurements to give a categorical characterization. The category of blood pressure is given in the table below, where the more severe category is chosen.

Systolic		Diastolic	Category
< 120	AND	< 80	Normal
$120 - 139$	OR	$80 - 89$	Prehypertension
$140 - 159$	OR	$90 - 99$	Stage 1 high blood pressure
≥ 160	OR	≥ 100	Stage 2 high blood pressure

The block diagram for the main VI for this program and that for the subVI are shown in Fig. 13. This example is somewhat trivial in that the subVI is the entirety of the routine, but serves to show how subVIs can be situated in a VI.



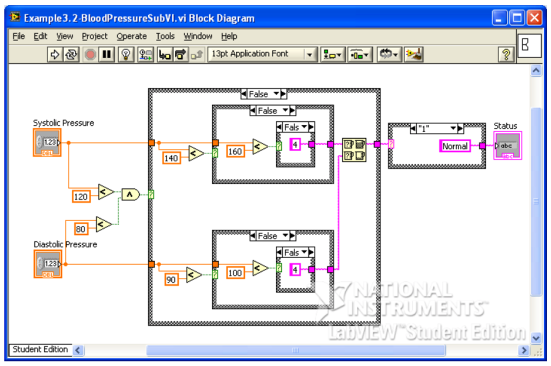


Figure 13: Left, block diagram of top level VI for a categorial indicator of blood pressure. Right, the block diagram for the subVI at the heart of the program.

To make your existing VI capable of being inserted as a subVI, you need to take a few steps working on the front panel of your VI. In the upper right corner of the

VI is an icon that looks like an an oscilloscope with an addition operator. This will be the icon representing your subVI when you insert it in another VI. To edit this, right click on the icon an select Edit Icon.... You can then edit the icon with rather primitive and annoying tools. I usualy just make the icon a box with text in it.

After choosing your icon, you need to specify how your subVI connects to controls and indicators (inputs and outputs). To do this, right click on the icon and select Show Connector (step 1 in Fig. 14). Click on the connector, which not replaces the icon. (On newer versions of LabVIEW, the connector is always shown next to the icon.) Right click the connector and select Patterns, and a palette of connector patterns appears. The default pattern is a $4 \times 2 \times 2 \times 4$ This allows for four inputs to the left and four outputs to the right. Some LabVIEW users advocate for always using the default, even if you do not need that many inputs and outputs. I advocate against this because this is poor programming practice: you should prototype your subVI so that you know what inputs and outputs to expect. You should pick the input/output pattern that is appropriate for your function. Next, use the wiring tool to select which controls and indicators in your front panel correspond to which inputs and outputs in your connector pattern (steps 3 and 4 for Fig. 14). After saving, you will have a subVI that you can plop into any other VI you are making. The inputs and outputs behave like the controls and indicators of the front panel of the original VI.

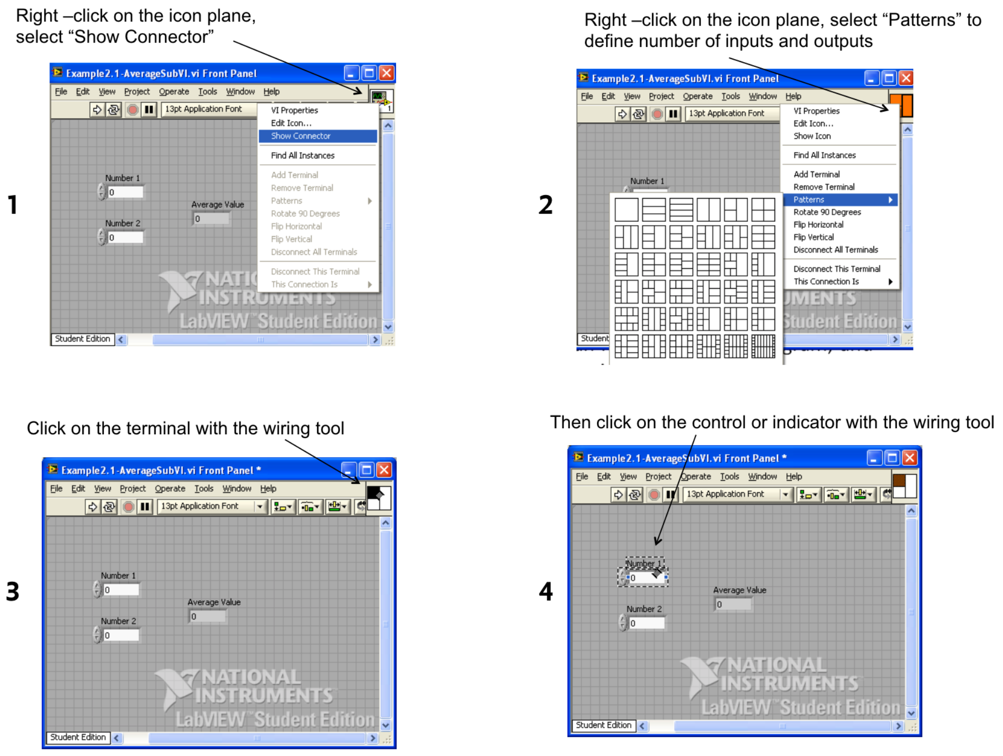

Right –click on the icon plane, select "Show Connector"
1
Right –click on the icon plane, select "Patterns" to define number of inputs and outputs
2
Click on the terminal with the wiring tool
3
Then click on the control or indicator with the wiring tool
4
Example2.1-AverageSubVI.vi Front Panel
VI Properties
Edit Icon...
Show Connector
Show Icon
Find All Instances
Add Terminal
Remove Terminal
Patterns
Rotate 90 Degrees
Flip Horizontal
Flip Vertical
Disconnect All Terminals
Disconnect This Terminal
This Connection Is
Number 1
Number 2
Average Value
Student Edition

Figure 14: The steps to enabling a VI to be a subVI.

To add a subVI to a block diagram you are working on, scroll to the bottom of the Functions Palette and click on Select a VI.... You will then be able to wire it up according to your specification of the connector pattern.

To get an overview of how all of the subVIs are connected and depend on each other in a top level VI, click on View → VI Hierarchy.