Introduction to Data Acquisition (DAQ)

Reference reading in text: Meyer, Chapter 7, section 11 discusses some very simple converters (analog-to-digital and digital-to-analog).

Goals:

1. Become acquainted with two of the basic data acquisition hardware elements: digital-to-analog converters (DAC's) and analog-to-digital converters (ADC's), and with some of their properties.

2. Become familiar with the idea that data acquisition involves a cooperating combination of software and hardware.

3. Become acquainted with the LabVIEW graphical data acquisition programming environment, which provides the software component of DAQ.

4. Use a LabVIEW-controlled combination of DAC's and ADC's to perform a measurement of the analog output characteristics of the National Instruments USB-6215 data acquisition interface.

This lab is expected to take about 3 lab periods.

3.1 Introduction

As suggested in Goal 1 above, the basic elements of data acquisition (DAQ) systems include digital-to-analog converters (DAC's), and analog-to-digital converters (ADC's). (The fact that most people tend to pronounce DAQ and DAC the same way can lead to some confusion. So, pay attention).

3.1.1 Positionally encoded binary number notation and the digital-to-analog converter

The classic, simple DAC takes as it's input a discrete digital quantity, which usually (almost universally) is a so-called positionally-encoded binary number B, having a decimal value

$$B_{10} = b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + b_3 \cdot 2^3 + \cdots + b_{N-1} \cdot 2^{N-1} = \sum_{i=0}^{N-1} \mathrm{b_i} \cdot 2^{\mathrm{i}}$$

(The subscript 10 refers that we represent B in decimal system.) DAC produces an output which is usually a voltage, proportional to the numeric value of the positionally-encoded binary input:

$$V_{\text{out}} = kB$$

Where k is a coefficient specific to given DAC. The b_i coefficients of the powers 2 are binary values, i.e. each $b_i = 0$ or 1, and the individual b_i's are called the "bits" of B. Each bit b_i has a "weight" of 2^i in determining the value B – each successive value (or bit) has a weight or significance that is exactly twice that of the previous value. The quantity B in binary representation reads:

$$B_2 = b_{N-1}b_{N-2}b_{N-3} \cdots b_2b_1b_0.$$

In general, N-bit binary number can have 2^Nvalues; all of the integers between 0 and $2^N - 1$. Thus, the output of a DAC is proportional to B and assumes a discrete set of values between 0 and $k{\cdot}(2^N - 1)$. Although digital signal is converted to analog one in DAC, its output is thus not continuous. The more bits DAC has to cover for a given voltage range (say 5V), the smoother is its output. Typical DACs have 10, 12, 14 and 16 bits. DACs become more expensive the more bits they have. For example: if $V_{\text{out,}max}$=5 volts, and DAC has 10 bit input (2^{10}=1024) then the precision of the DAC will be 5/1024 volts $\cong 4.88$ mV; i.e. successive values of V_{out}will be spaced by about 5 mV.

Some further remarks on conversion between decimal and binary numbers. A binary number can be converted to a decimal using $\sum_{i=0}^{N-1} \mathrm{b_i} \cdot 2^{\mathrm{i}}$ as already stated above. To convert decimal to binary number, write down the decimal number and to continually divide-by-2 to give a result and a remainder.

The reminder can be either a “1” or a “0”. Continue this process until the final result equals zero. The reminders from these division give the binary representation of this number. The first reminder is the least significant bit b_0, the second b_1, etc.

Let’s go through these steps for number 5280. Dividing 5280 by two yields result 2640 and reminder is 0. This means that b_0=0. Dividing 2640 by two yields result 1320 and b_1=0. Then 660 and b_2=0; 330 and b_3=0; 165 and b_4=0; 82 and and b_5=1; 41 and b_6=0; 20 and b_7=1; 10 and b_8=0; 5 and b_9=0; 2 and b_{10}=1; 1 and b_{11}=0. Dividing 1 by 2 gives the last reminder b_{12}=1. Altogether we have 1010010100000_2 (subscript 2 indicates that this number is binary number).

While the above approach is good to represent positive integers, some convention is needed to represent binary negative integers. The convention used in digital computers is called the twos-complement binary notation. This is the most wide-spread notation and thus most relevant for us. It works as follows, using the example derived above for the decimal value of 5280. As the above example showed, 5280 can be represented by 13 bits. To adopt a twos-complement binary scheme, we add an additional, 14th bit, placed in the most significant position. The basic idea is that this extra bit allows specifying whether the value represented is positive or negative. The simplest representation were to assign value 1 to this bit if the output is negative; if it is a 0, the output is positive. Thus we would express our example value (5280) as 01010010100000, and the negative value,-5280 as 11010010100000. The problem of this approach is that it generates two values for zero, +0=00000000000000, and -0=10000000000000 and this complicates arithmetic because it creates a number system that produces a different value for adding 1 to -1 (produces -0) than for subtracting 1 from 1 (produces +0). The 2’s complement system avoids this problem. In 2’s complement system, we again will use a 14th bit in the most-significant position. Also again, 5280=01010010100000 – the 2’s complement representation of positive quantities is the same as the simple positionally-encoded binary. To generate the 2’s complement value for -5280, we complement all of the bits in +5280, to yield 10101101011111, then we add 1:

```
 10101101011111 = 5280 complemented
 +            1
_______________
 10101101100000 = 2’s complement representation of -5280.
```

Addition displayed in this “traditional” elementary-school manner works for binary just as it does for decimal: adding vertically, a 0 plus a 0 yields a 0; a zero plus a 1 yields a 1; a 1 plus a zero yields a 1, and a 1 plus a 1 yields a zero with a carry to the next more significant position. There is the possibility of a carry-in bit from the previous position; if the carry-in bit is 0 everything works as stated. If the carry-in is a 1, then the sum is 1 greater than when the carry-in bit is 0, which creates the need to deal with a 1 plus a 1 plus a carry bit of 1, which yields 11, i.e. a 1 for the sum plus a carry out to the next more significant position.

As an exercise, you might try computing the value $-(-5280)$, starting with the value from above, in 2’s complement representation, for (-5280), then complementing all the bits and adding 1 to compute $+5280 = -(-5280)$. In other words, applying the 2’s complement negation process twice produces the original number; the second operation ‘undoes’ the first.

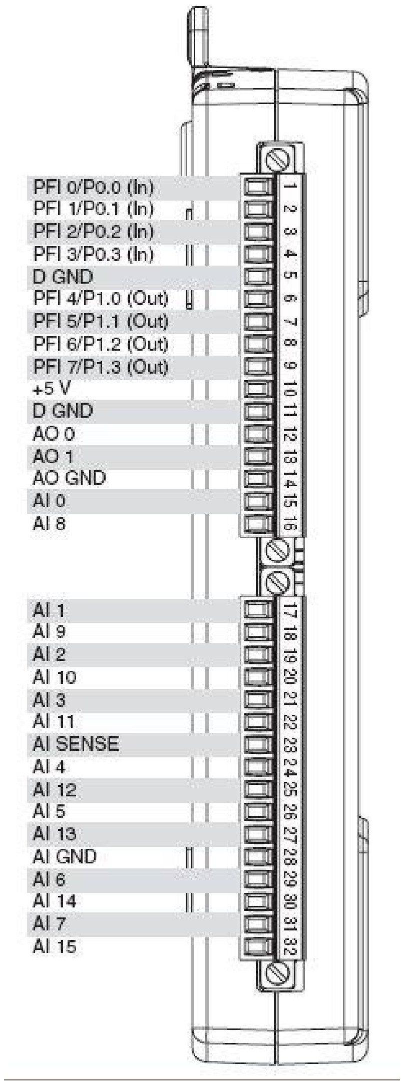
As a second exercise, you might show, using the 2’s complement representation, that (as should be expected) $+5280 + (-5280) = 0$.

3.1.2 The National Instruments USB-6215 digital acquisition system (DAQ)

What can the NI USB-6215 DAQ do? It has eight analog inputs and two analog outputs, 4 digital input lines and 4 digital output lines; the digital I/O lines also serve to connect to a counter/timer input/output. The analog inputs and outputs have 16-bit resolution (or precision) and sampling rates up to 250 kS/s (the voltages output by the DAC’s can be changed 250,000 times per second, and the ADC’s can sample ("measure") 250,000 voltages per second.)

A. Laboratory exploration

Examine the DAQ. Note that there are two connectors with screw terminals; one is labeled "Digital I/O & Analog Output", and the other is labeled "Analog Input"; the connector aprons, between the screws and the wire insertion openings, have individual terminal labels. Compare these terminals and labels with the diagram below to identify the eight analog inputs, two analog outputs, and 8 digital terminals (4 input and 4 output; these also double as connections to the counter/timer). Find the ground(s), and the 5-V output. Note how wires are to be connected to the terminals: if you look inside the opening for inserting the bare end of connecting wires, there is a metal clamping tab that is positioned by means of the screw above the opening. Turning the screw CW (viewed from above) raises the clamping tab in the opening and tightens on connecting wire ends.



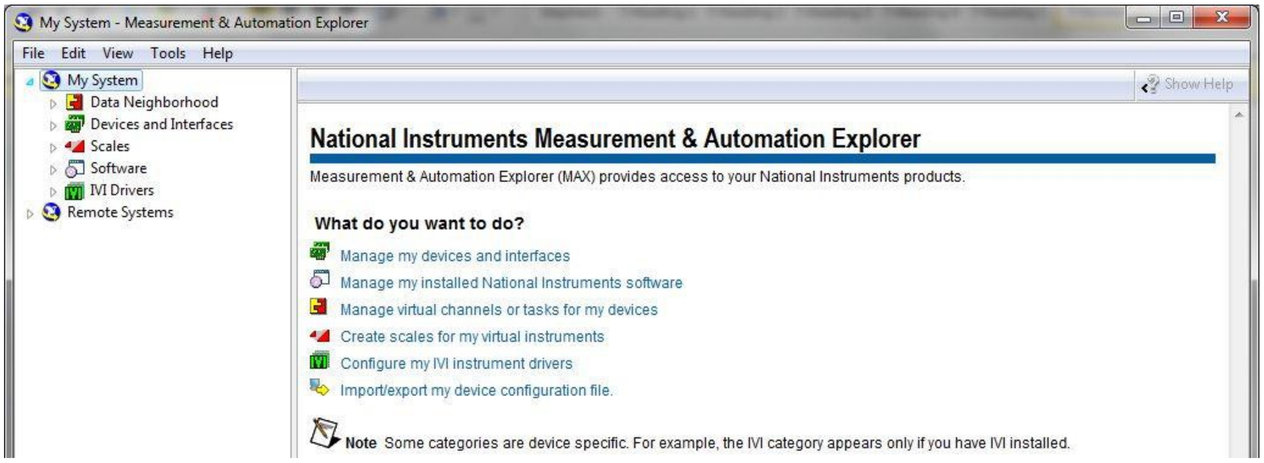
We will first use the simple software provided with the device to perform tests to verify that the software recognizes the USB-6216 DAQ interfaces.

Also, it should be pointed out that good practice would dictate that before connecting the device to the external world we should learn its voltage and current limitations. For this you could (and should) read the instruction manual, paying careful attention to the maximum voltage permitted for analog and digital inputs and the maximum current that the outputs can supply.

However, in the interest of brevity, we will skip this step and proceed (cautiously . . .).

Let’s explore the system. Plug the DAQ into a USB port on the computer (if it’s not already), turn on the computer, and look for and click on the “NI MAX” (National Instruments Measurement and Automation eXplorer shortcut icon:

When the program starts you will be presented with the “My System” screen below (the bottom has been cropped a bit to conserve space):



National Instruments Measurement & Automation Explorer

Measurement & Automation Explorer (MAX) provides access to your National Instruments products.

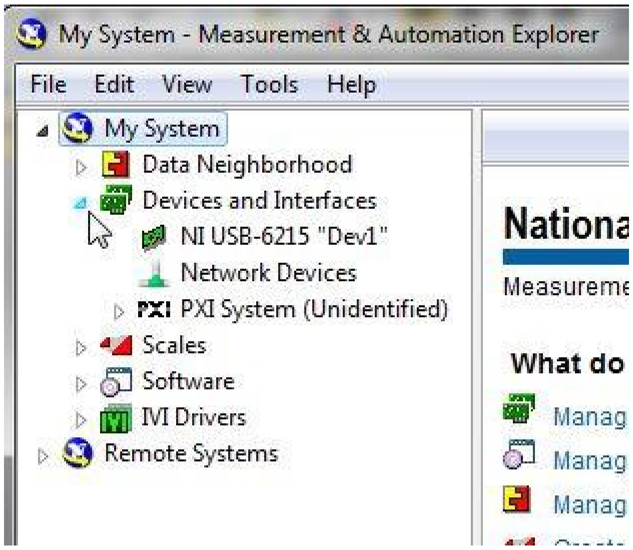
What do you want to do?

- Manage my devices and interfaces
- Manage my installed National Instruments software
- Manage virtual channels or tasks for my devices
- Create scales for my virtual instruments
- Configure my IVI instrument drivers
- Import/export my device configuration file.

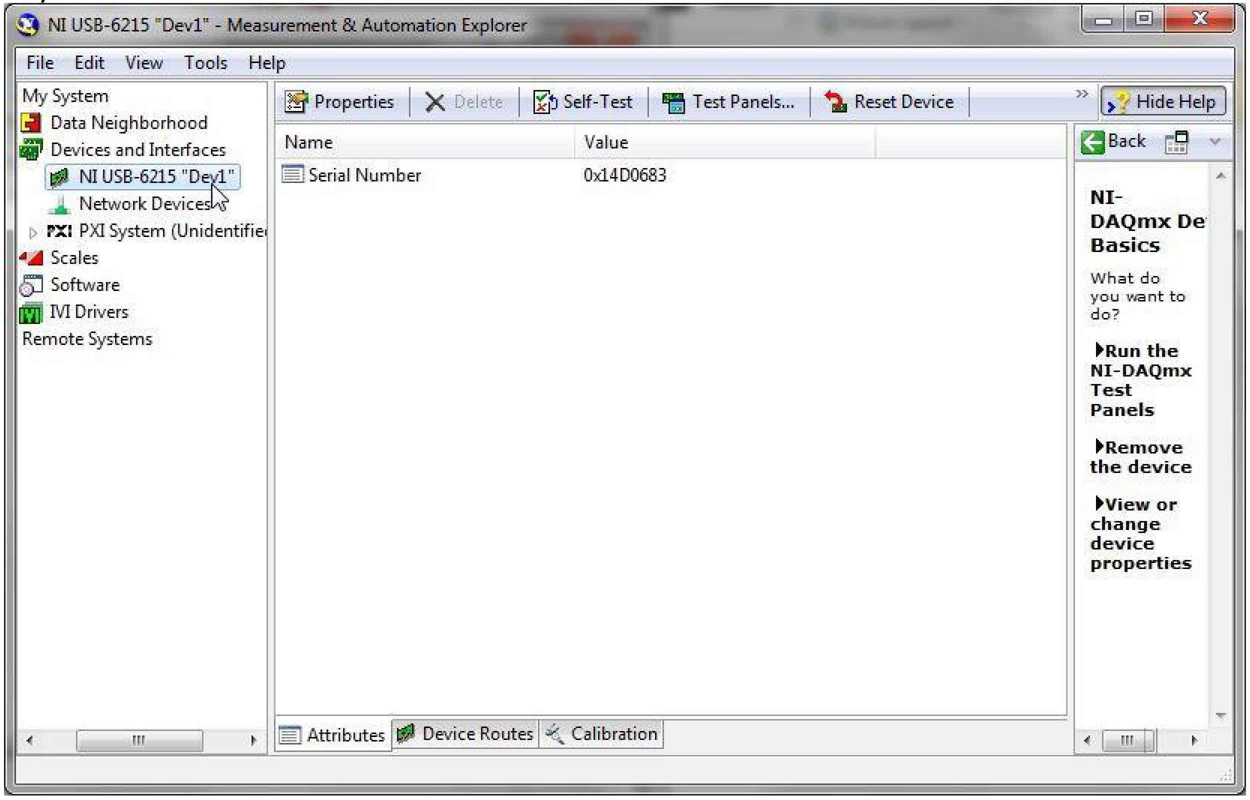
Note Some categories are device specific. For example, the IVI category appears only if you have IVI installed.

In the panel on the left of the “My System” screen is a hierarchical “tree” of aspects of the data acquisition system formed by the computer and attached data acquisition hardware and software. This panel is sometimes called a “navigation panel.” On this panel, expand the “Devices and Interfaces” entry (click on the + sign to its left), and you should see this (partial) screen:

The USB-6215 device should be listed. If it is not, check that the USB cable is connected properly.



Select / click on “NI USB-6215”, which should then alter the properties and settings panel in the center so that it provides some basic information about your USB-6215:



Name	Value
Serial Number	0x14D0683

The tabs at the bottom of the center panel allow you to view different property information about the selected device (in this case, the NI USB-6215 "Dev1". The name (Dev1) which is shown in quotes in the navigation panel is an "alias" for the device that can be used to identify it to, for example, LabVIEW. You need to know this name later. Your 6215 may have a different name from this one. You can use whatever you want, but I am going to suggest and use DAQ1. You can change it by double-clicking the name in the My System "tree" (where the cursor is shown in the above figure), editing the name, and hitting the Enter keyboard key. You may get an error message saying that there are existing resources already using the old name, and asking if you want to update those resources with the new name you've chosen. I recommend that you say, 'Yes", or "Cancel" to revert to the old name (and, of course, remember the name you leave it with).

The next step would ordinarily be to run the test panels (see the Test Panels... button) to see how they work, but again, in the interest of brevity we will assume that the DAQs will pass any tests (a risky assumption, in general).

B. Discussion: What's going on here?

The USB-6215 DAQ hardware device is a member of a family of general purpose data acquisition hardware produced by National Instruments, known as NI-DAQmx. There are a lot of these devices - there is a USB-6210, a USB6212, a USB-6216 and a USB-6218 as well as the USB 6215, and numerous non-USB devices that plug directly into the I/O Bus of the computer. These devices have a variety of capabilities – like the ability to produce one or more analog outputs, or to convert analog voltages at one or more inputs into a binary quantity compatible with a computer, etc. Each device has a different set of capabilities, but of these devices behave similarly with respect to particular capabilities – for example, all the ADC's across the family of devices produce a 2's complement binary number, but the specifics may vary from device to device – for example, some devices may have 14-bit AD converters and some may have 16-bit AD converters (the 6215 has 16-bit converters); others may have 20-bit converterts, and some may convert voltages over a range of -10 volts to +10 volts, but others may only digitize voltages in the range -5 volts to +5 volts.

All NI-DAQmx devices are supplied with “device driver” software, which is installed when the hardware is first installed (and it should already be installed on your workstation computer). The device driver software provides an interface between the hardware details of the specific device and the application software, and insulates the application software from the need to “know the details”. The application software runs under the computer’s operating system and treats the data acquisition devices in a device-independent manner. In our case, the application software is a combination of LabVIEW and specific application code, called a VI or Virtual Instrument, created by the user (using LabVIEW) to accomplish a specific task.

In practice, the VI (virtual instrument) code that a user produces may work with LabVIEW to specify that a measurement task will consist of generating a sequence of 100 voltages starting at -10 volts and increasing in steps of 0.15 volts to reach a final voltage of (about) +5 volts. At each of these voltage “steps”, two separate voltages that are presented at inputs to the interface are to be measured and recorded, with all the measurements recorded in an array, and ultimately in a spreadsheet. As this code is generated by the user, LabVIEW provides access to general purpose functions and to the “application programming interface” of the device driver code. The device driver code then translates (for example)-5.280 volts into whatever the specific digital-to analog converter (chosen by the user for this purpose) needs as a binary input to generate that voltage – perhaps `10101101100000` (from the example before for the 2’s complement representation of -5280) if the converter chosen is a 14-bit DAC, or `1010110110000000` if it’s a 16-bit DAC. The driver code also takes the binary code(s) produced by the analog-to-digital converters used for the measurement into a meaningful numbers for the application to store as a measured voltage. An ADC might get a result of 0101001010000000 and present it, via the device driver software interface the application program as +5.280 volts.

What’s going on here? So, the answer to the question, "What’s going on here?", raised 3 paragraphs ago, is that the NI MAX program is an application program, independent of LabVIEW, which “connects” to the hardware devices through the device driver software, and configures the devices (sometimes) and the driver software (always) to correctly manage the information needed to convert from the binary “machine” language of the device’s converters and other digital hardware to the more natural (to us) language used by LabVIEW and the application VI’s. NI MAX apparently communicates this information to LabVIEW by means of a database, but that aspect of operation is largely hidden from view – the user does not ordinarily need to worry about a lot of the details, unless the user wants to know what’s going on “under the hood.” What we have been doing is verifying that your USB-6215 is recognized by the device drivers and that there is configuration information stored in the database that LabVIEW will need to consult when it uses the device drivers to communicate with your USB-6215.

Close ✕ NI MAX when you’re done, before proceeding to the next section.

3.2. Using the USB-6215 with LabVIEW

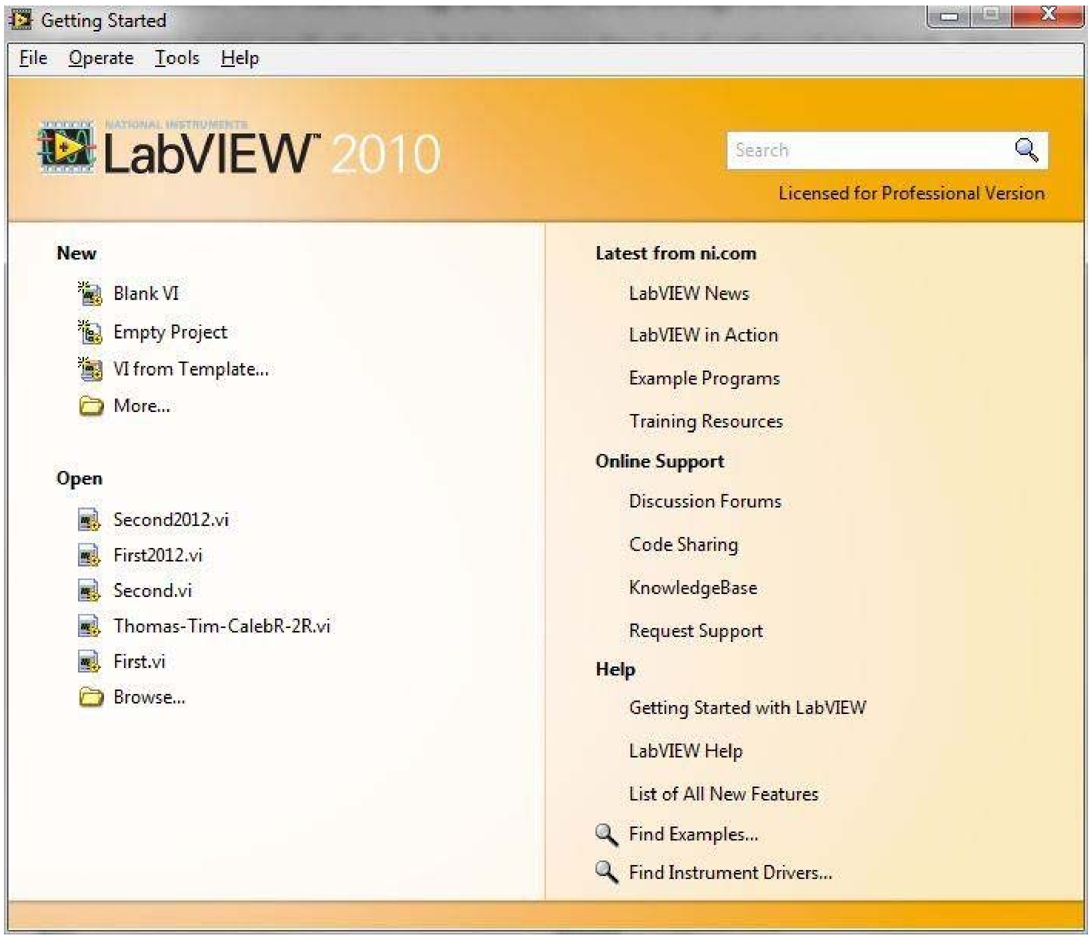
Using computer-controlled data acquisition techniques involves combining DAQ hardware with appropriate software. The LabVIEW programming environment permits the construction of the software component to complement the DAQ hardware (in this case, the USB-6215). LabVIEW is an acronym that stands for “Laboratory Virtual Instrument Engineering Workbench.”

3.2.1. (Guided) Laboratory exploration

In this section we will explore the use of a mixture of software, interfacing hardware (in the form of the USB-6215 DAQ) and a minimum of external circuitry to perform a specified measurement task. The software will consist of LabVIEW and a "VI", which stands for Virtual Instrument, and which is in effect the program code that, combined with the LabVIEW programming environment, controls the hardware in a manner that gets the data acquisition task accomplished.

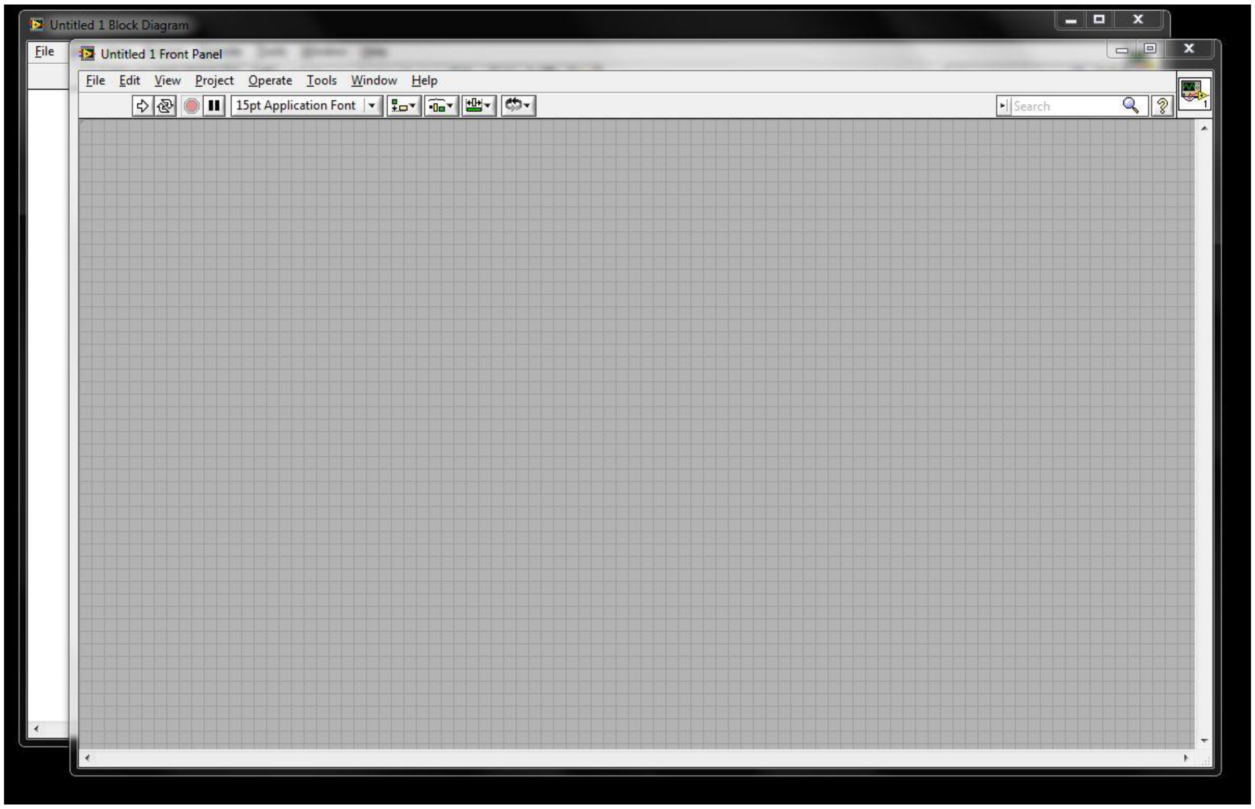
Begin by launching the LabVIEW programming environment. On the laboratory workstation computers, there should be a shortcut icon for this purpose (if not, click the Start button on the desktop toolbar, then click All Programs, then scroll through the list to find LabVIEW… :

LabVIEW is a huge application, so it takes some time (and patience) to launch. You'll see a "splash screen" for a while, then, when LabVIEW is up and ready for business, you will see the following window:



LabVIEW's opening screen/window.

To create your first LabVIEW application ("app"), which is called (in LabVIEW lingo) a "virtual instrument" or "vee eye", click on "Blank VI". You should then see the following on your desktop screen:



BlankVI Windows

Let's pause here to explain what you see after selecting "Blank VI" as the first step in creating a LabVIEW vi.

The programs created with LabVIEW are called virtual instruments, or "vi"s. The programming paradigm used by LabVIEW models a program (and sub-program objects) as functional blocks, represented by icons on a drawing canvas or screen, which are interconnected by lines (or virtual wires) that represent the flow of data between the functional objects. The flow of control from program object to program object is data-driven – in the lingo of computer science, LabVIEW employs "dataflow architecture" LabVIEW is also object-oriented, for what that's worth.

The creation process for a LabVIEW program (or virtual instrument) is graphical – you literally "draw" the program on two (coupled) drawing screens (or windows) or canvases. One canvas/window (the one hidden behind the quadrille-ruled window above) is the "block diagram" of the virtual instrument. This block diagram is the "guts" of the program – all program objects and their dataflow interconnections are shown in the block diagram.

Some LabVIEW objects (functional blocks represented by icons) are user-interface objects. These objects appear on the block diagram and also appear on a coupled drawing canvas or window called the "front panel." A well-designed virtual instrument (vi) is usually created in a "top-down" fashion. This means that one begins by considering the functional requirements of the vi and how the user will interact with it. One then designs the front panel by selecting the user interface objects and placing them on the front panel. The user interface objects are of two kinds – controls and indicators. Both control objects and indicator objects appear on both the front panel and on the block diagram, but they are initially placed on the front panel. As the name implies, control objects control the values of data objects in the block diagram and thereby control (through "dataflow") the operation of the vi program.

The indicator objects give the user feedback about the state of the vi. Control objects have built-in indicators that are an intrinsic part of their appearance and function. (The front panel design can also be enhanced with "decoration" objects that serve to make the front panel easier to understand, but which have no function or appearance in the block diagram).

In creating a vi, it is a good idea to frequently save your work. It is also a good idea to build your vi in stages, with a first stage being a relatively simple vi and with successive stages adding more and more functionality. Each stage can be (should be?) saved as a separate entity and then tested (and debugged, if needed, and if so, resaved), so that if the next stage goes "wrong" – perhaps something actually goes wrong, or you just decide that you want to do that stage a bit differently, with a better design approach, for example – then you can easily revert to the previous stage and start over from a point that you are happy with.

For now, save this blank vi, with a name that you'll be able to recall later. From the File menu, select "Save As", and give your empty vi a name in the "Save As" dialog box, for example, "myFirst" – which will create myFirst.vi.

We will design this vi in top-down fashion. Our goal for this vi is to ultimately produce a virtual instrument that generates an analog DC voltage (to ultimately be an output of the USB-6215) of specified value in the range -10V to +10V, and then (ultimately) to measure an analog voltage input, also assumed to be in the range from -10V to +10V. We will design this in several stages:

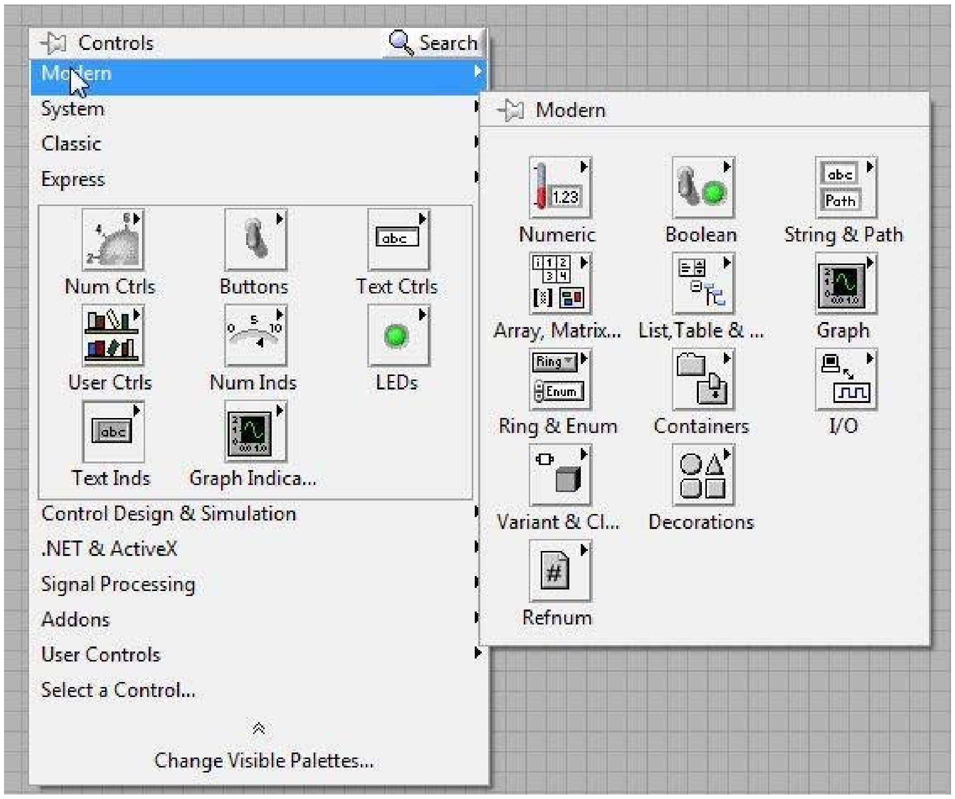
1. First, we will design and test the front panel functions. There will be one control that specifies the analog voltage to be (ultimately) output by the USB-6215 'AO0' terminal (refer to the "Pinout Diagram for USB-6215", the very first figure in this writeup), and one indicator that will (eventually) read the voltage at the USB-6215 terminals corresponding to analog input 'AI0' (see the pinout diagram). But for now, in the first stage, we will just connect the control directly to the indicator.

2. Second, we will connect controls and indicators to the USB-6215: the control to the digital-to-analog converter (DAC) (in the USB-6215) that actually generates a voltage between terminal 'AO0' and the analog "ground", or common, which is the terminal labeled 'AO GND'; we will also set up the USB-6215 analog-to-digital converter (ADC) to connect its reading to the front-panel indicator. In order to test this stage, we will need to physically (hey – this is a physics course, not a computer science course!) connect the 'AO0' output to the 'AI0' input.

Aside: In a real-world instrumentation situation, we would probably connect AO0 to an external circuit in order to control some parameter in a physical system that we wanted to control (a "stimulus"), and would connect further analog inputs (the USB-6215 has seven other AI's in addition to AI0 – AI1 through AI7) to various sensors in or on that physical system in order to measure the effect of the stimulus. To acquire the data corresponding to the response of the system, we would need to make our control program – our 'vi' – considerably more elaborate. For this first example of using LabVIEW, our physical system is simply a couple of wires attached externally to the USB-6215. (End of Aside.)

3.2.2 Doing it: Building your first VI.

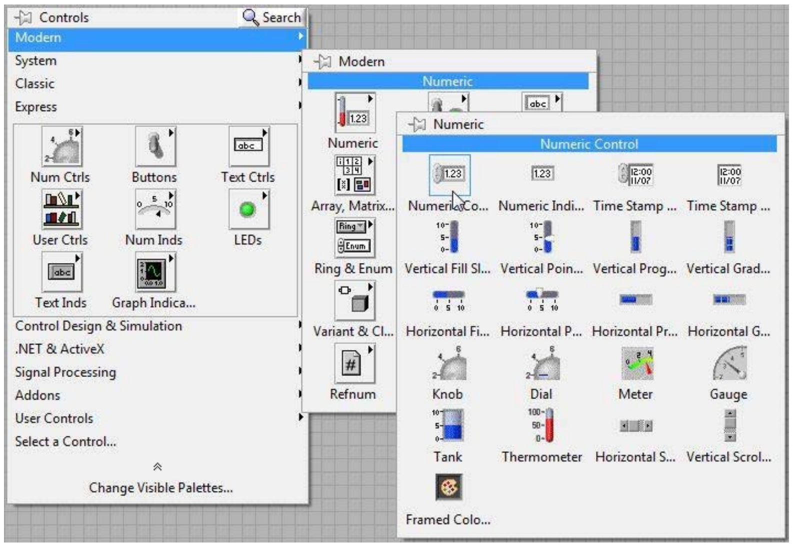
3.2.2a. Designing and Building the Front Panel (You have saved your blank VI as myFirst.vi, haven't you?). Right-click anywhere on the front panel to bring up the Controls palette:



Controls palette

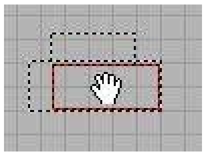
In the above, I also moused over and clicked the double arrow on the bottom to expand the Controls palette, then clicked the Modern entry on the palette to bring up the Modern control palette.

Now mouse-over the Numeric icon in the Controls palette and select (click on) Numeric Control from the panel that pops up:

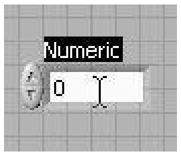


Selecting the Numeric Control

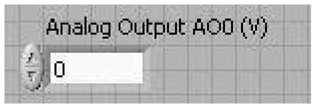
As you select the numeric control, the palettes disappear, to be replaced by an outline of the control, which you can then position on the front panel, and "drop" into place by a final click of the mouse. At this point, the legend above the control is highlighted (with a black background); you can enter an appropriate legend now. A reasonable suggestion for a label is shown in the figure below:



Selected numeric control ready for placement.

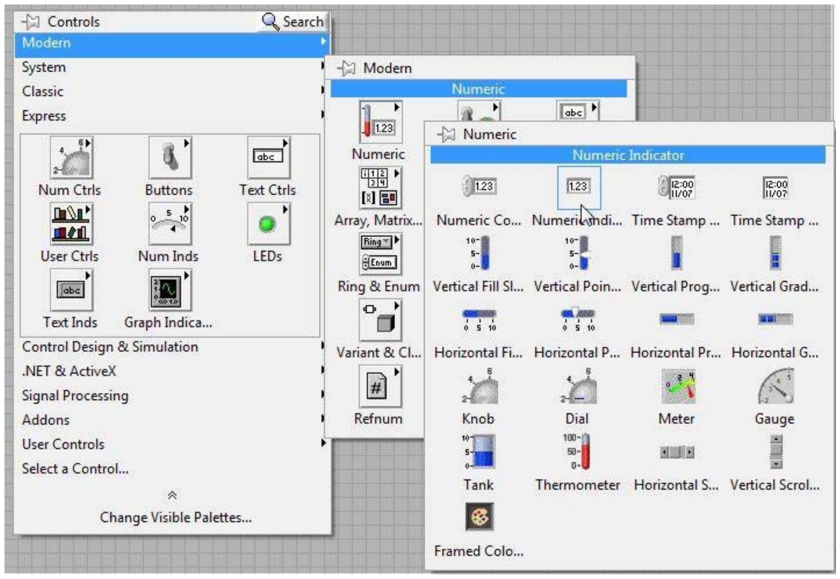


Numeric control after placement, ready for labeling.



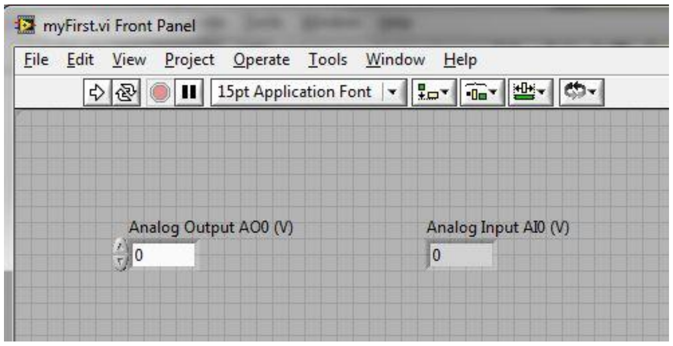
Numeric control with labeling; note proper use of units.

Now repeat the process just completed, but this time, use a numeric indicator from the controls palette:



Numeric Indicator Palette

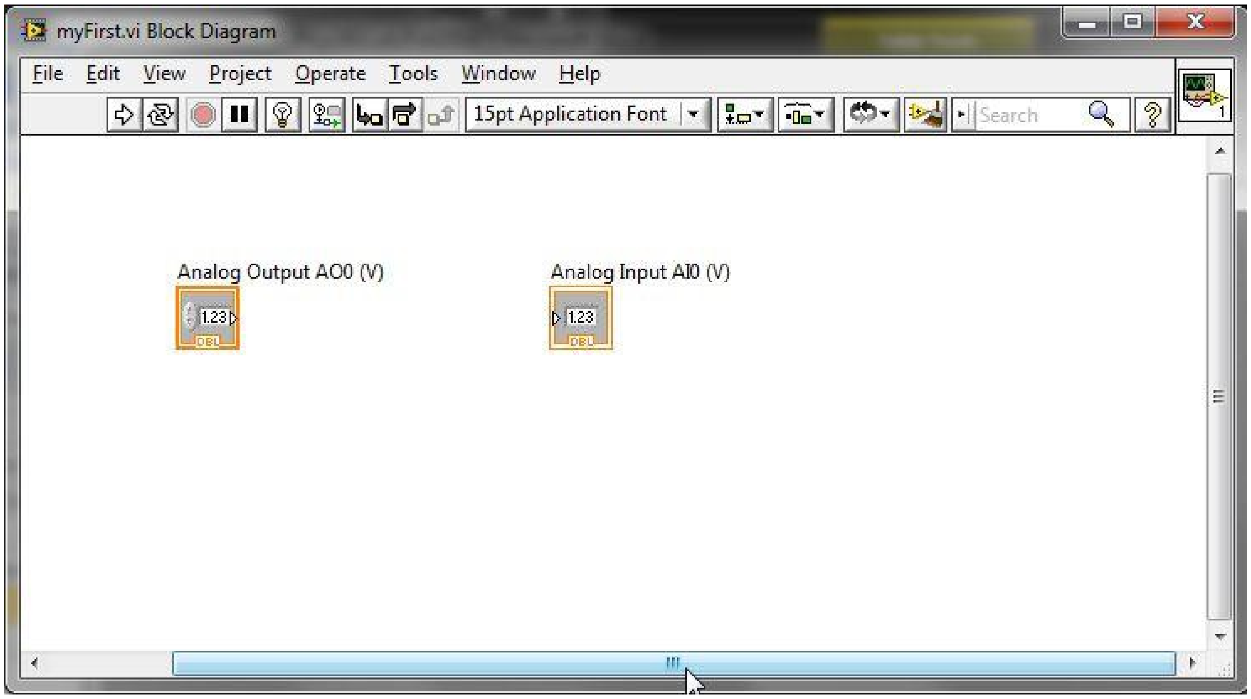
Place the indicator on the front panel to the right of the numeric control for AO0, and label it appropriately, perhaps like shown in the completed front panel below:



First.vi Front Panel (only upper left corner shown).

You can resize the front panel window so that it comfortably encompasses your control and indicator. This would be a good time to save your work.

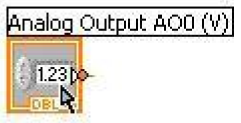
3.2.2b. Wiring the Block Diagram. Now we will see what the above operations did to the block diagram canvas/screen/window. Remember that user interface objects like controls and indicators have corresponding functional appearance or presence in the block diagram. Click on the title bar of the block diagram to bring it into focus and to the front. (If you do not see the block diagram, then on the front panel menu bar, select Window, and from the drop-down menu, select "Show Block Diagram." You should then see something like the following (I have resized the block diagram window):



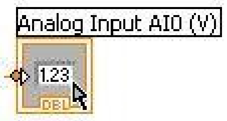
First.vi's Block Diagram

You may also see a palette of function icons, which you can ignore for now.

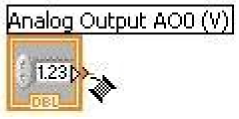
Now "wire" the Analog Output AO0 (V) control to the Analog Input AI0 (V) indicator using the following procedure. Notice that as you hover the mouse pointer over either the control icon or the indicator icon in the block diagram, an orange dot appears on the side of the icon. This dot is the "wiring terminal" – the virtual wiring terminal – of the control or indicator. Also notice that as the pointer hovers right over the terminal, the pointer turns into a "wiring tool", which looks a little like a spool of wire. This is shown below:



AO0's terminal (orange dot on right)

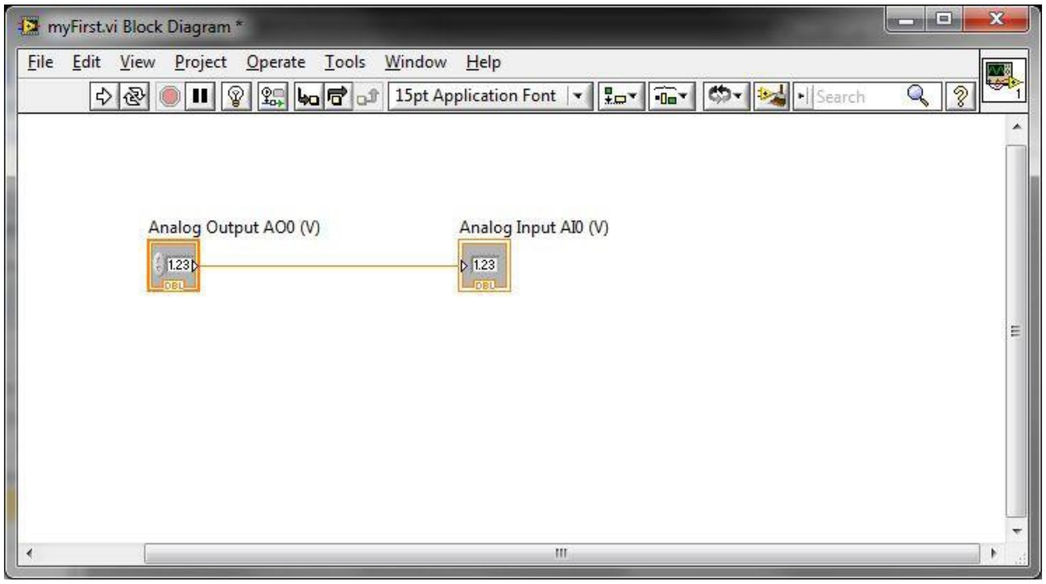


AI0's terminal (orange dot on left)



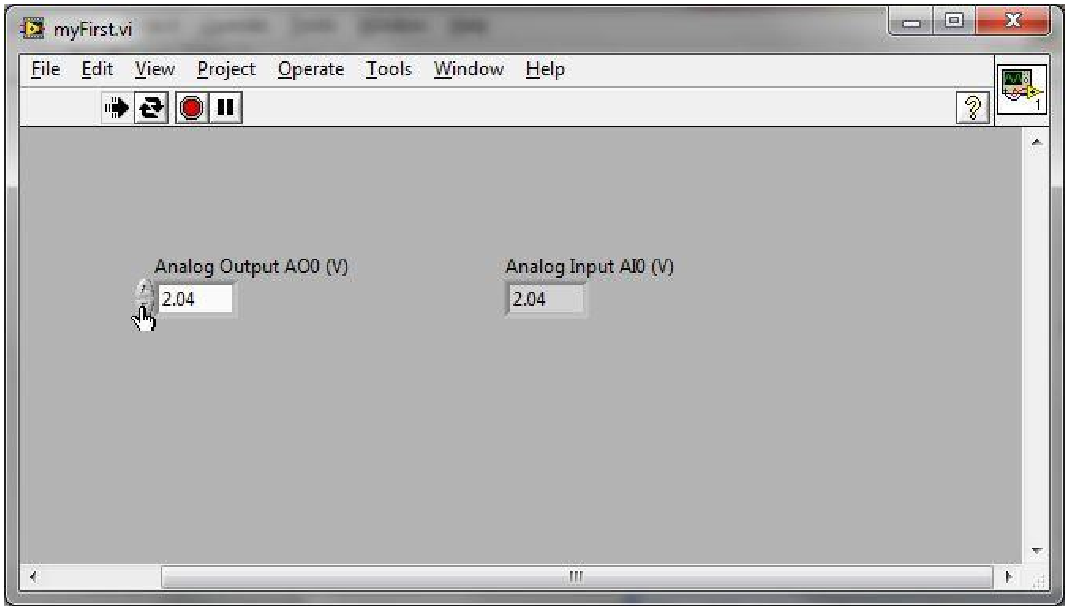
Wiring tool hovering over AO0's terminal

To wire AO0 to AI0, hover the mouse over AO0's terminal to activate the wiring tool, then click and drag (an orange wire) over to AI0's terminal, then release/drop the wire. You now have wired your VI, and it should look something like the following:



Wired block diagram for First.vi

3.2.2c. Now test your VI. Select the front panel by clicking on its title bar. Then click on the run continuously button (). You should be able to adjust the control on the front panel by clicking on the up/down arrows, or by clicking in the window and typing in a (reasonable) value. As you do so, the indicator window should reflect the changes you make.



First.vi in operation.

Confirm that the VI really works (that you aren't being swindled) by stopping execution (with the stop-sign button), selecting the block diagram, and deleting the wire that connects the control to the indicator. Rerun the VI and see what happens when you try to adjust the control.

Rewire the VI and verify that it now works again.

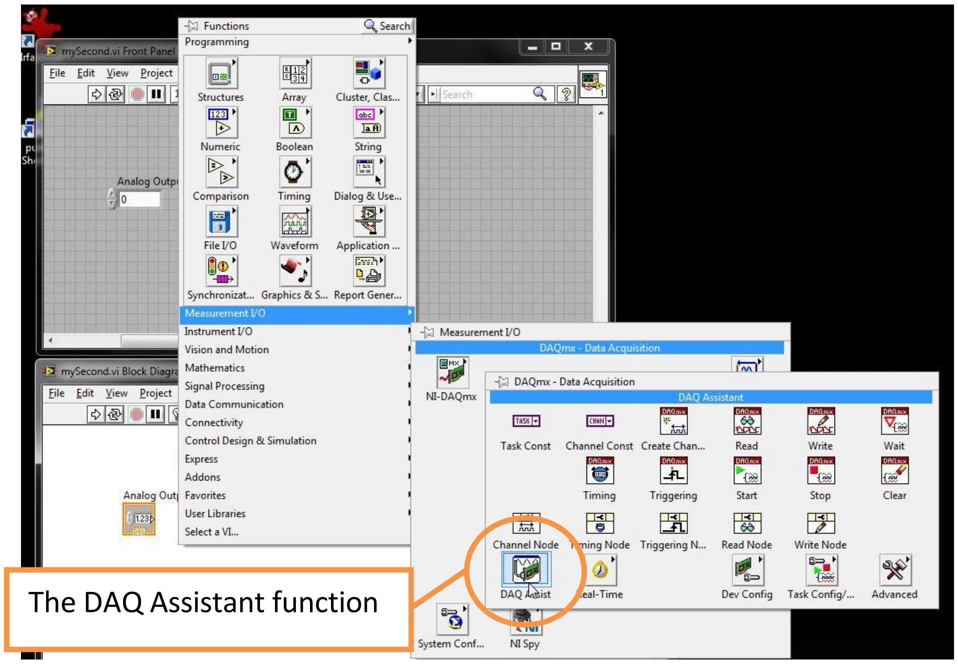
3.2.3 Making the Virtual Instrument "Real"

At this stage of development, our virtual instrument is truly virtual – it has no physical existence beyond the flipping and flopping of digital states in the computer. In stage 2, we will give it a physical presence by connecting the control object to one of the USB-6215 analog output channels (there are two of them, named "AO0" and "AO1") and by connecting the indicator object to one of the USB-6215 analog input channels (there are 8 input channels, named "AI0", "AI1"… "AI7"). The connections we make will be mediated or achieved by a class of LabVIEW software components known as "Express VI's". In more commonly used computer lingo, the ExpressVI's would be called "wizards". The particular Express VI we will use for this purpose is named "(the) DAQ Assistant." The DAQ Assistant gives us convenient access to the data acquisition hardware by programming the hardware drivers for us, based on information we supply to the DAQ Assistant. This programming is done automatically by the the DAQ Assistant – in effect, for each data acquisition task we need to accomplish, we tell the DAQ Assistant what we want to do, and the the DAQ Assistant writes the code needed to get the job done. Then all we have to do is pipe data to or from the code block (which shows on the Block Diagram as a functional block represented by an icon).

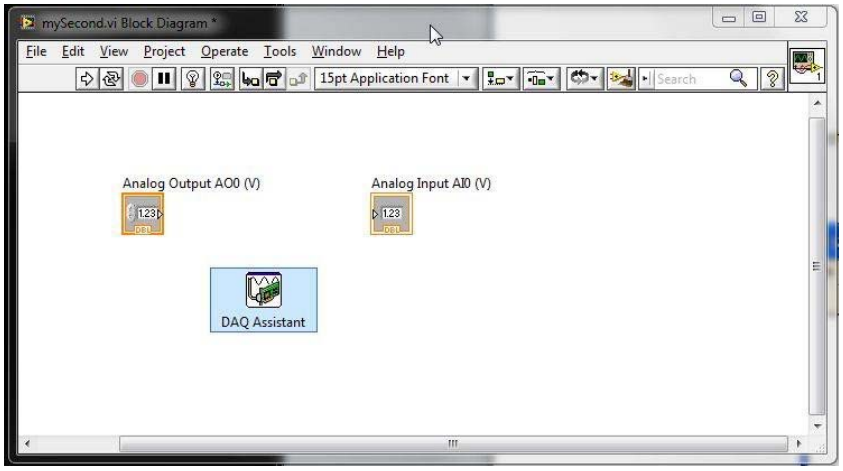
3.2.3a. Start by saving a fresh copy of myFirst.vi as mySecond.vi, using Save As; LabVIEW has a peculiar Save As dialog, which gives you 4 choices. Select Copy/Substitute copy for original, then click Continue; you'll then get a standard Save As dialog in which you get to rename myFirst as mySecond.

Go to the block diagram of mySecond.vi, and delete the wire that connects the control to the indicator. (Click on the wire to select it, then press the Delete key, or right-click with the cursor on the wire and select "Delete wire branch" from the pop-up menu.)

From the Functions palette (if the Functions palette is not showing, right-click on the Block Diagram), select the Measurement I/O line item; from the Measurement I/O panel that appears, select the NI-DAQmx icon (see figure below), which brings up yet another panel of functions; from the NI-DAQmx panel, select DAQ Assistant; this will give you a phantom of the DAQ Assistant which you can then place on your block diagram. Place the DAQ Assistant under the AO0 control on the block diagram.

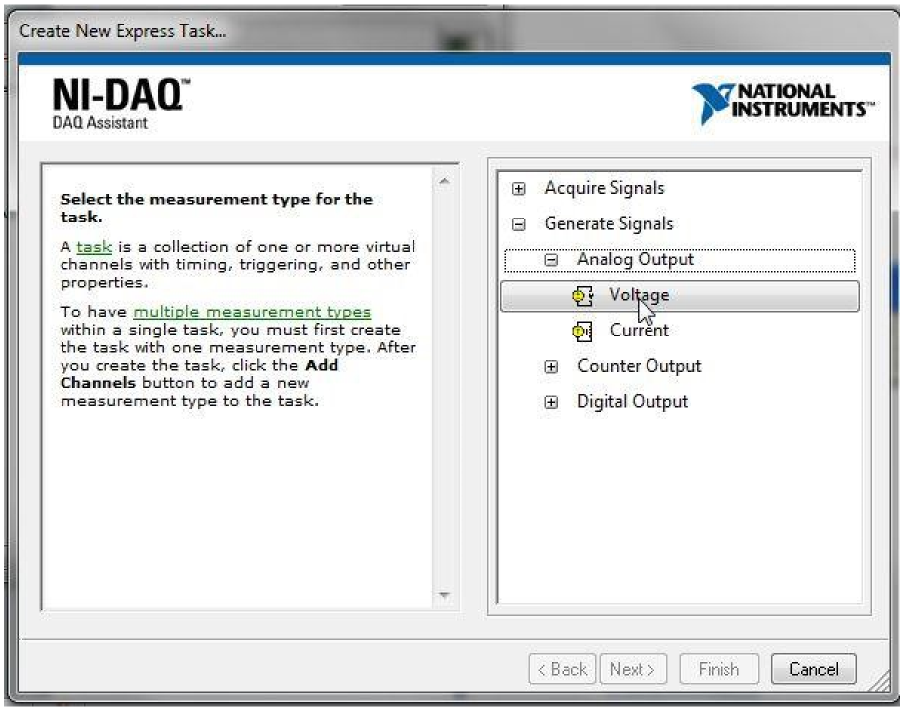


The Measurement I/O palette from which NI-DAQmx and DAQ Assist(ant) is selected.



The first DAQ Assistant in place, for connection to AO0

Once a DAQ Assistant Express VI icon has been placed on a block diagram, it automatically initializes and configures itself to perform the task needed. This causes a series of dialog screens to open; the first is shown below. On this dialog, select Analog Output and Voltage.



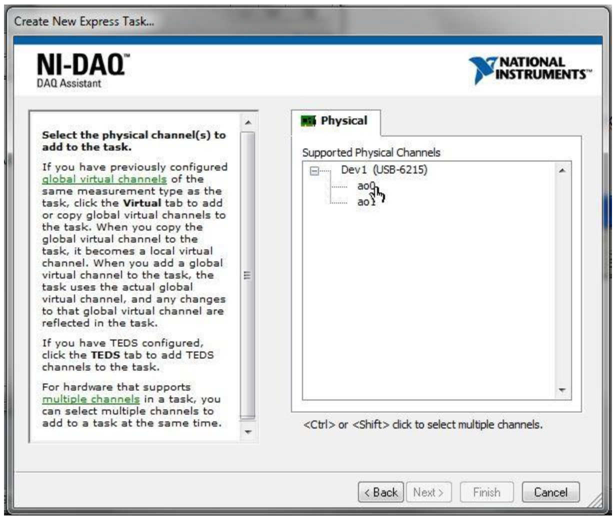
Select the measurement type for the task.

A task is a collection of one or more virtual channels with timing, triggering, and other properties.

To have multiple measurement types within a single task, you must first create the task with one measurement type. After you create the task, click the **Add Channels** button to add a new measurement type to the task.

Configuring the DAQ Assistant to produce an Analog Output Voltage

A succession of screens will then be presented for further configuration of the data acquisition task that will be constructed by the DAQ Assistant Express VI. These are shown, with appropriate choices for the VI being constructed here, in the figure below:



Select the physical channel(s) to add to the task.

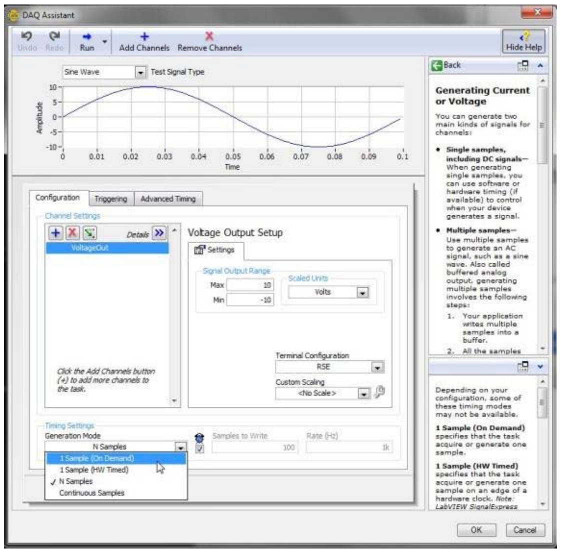
If you have previously configured global virtual channels of the same measurement type as the task, click the **Virtual** tab to add or copy global virtual channels to the task. When you copy the global virtual channel to the task, it becomes a local virtual channel. When you add a global virtual channel to the task, the task uses the actual global virtual channel, and any changes to that global virtual channel are reflected in the task.

If you have TEDS configured, click the **TEDS** tab to add TEDS channels to the task.

For hardware that supports multiple channels in a task, you can select multiple channels to add to a task at the same time.

<Ctrl> or <Shift> click to select multiple channels.

Selecting the analog output channel. Click Finish after selecting ao0.



Test Signal Type: Sine Wave

Voltage Output Setup

Signal Output Range: Max 10, Min -10

Scaled Units: Volts

Terminal Configuration: RSE

Custom Scaling: <No Scale>

Click the Add Channels button (+) to add more channels to the task.

Timing Settings

Generation Mode: N Samples

1 Sample (On Demand)

1 Sample (HW Timed)

✓ N Samples

Continuous Samples

Samples to Write: 100

Rate (Hz): 1k

Generating Current or Voltage

You can generate two main kinds of signals for channels:

- **Single samples, including DC signals**—When generating single samples, you can use software or hardware timing (if available) to control when your device generates a signal.
- **Multiple samples**—Use multiple samples to generate an AC signal, such as a sine wave. Also called buffered analog output, generating multiple samples involves the following steps:
  1. Your application writes multiple samples into a buffer.
  2. All the samples

Depending on your configuration, some of these timing modes may not be available.

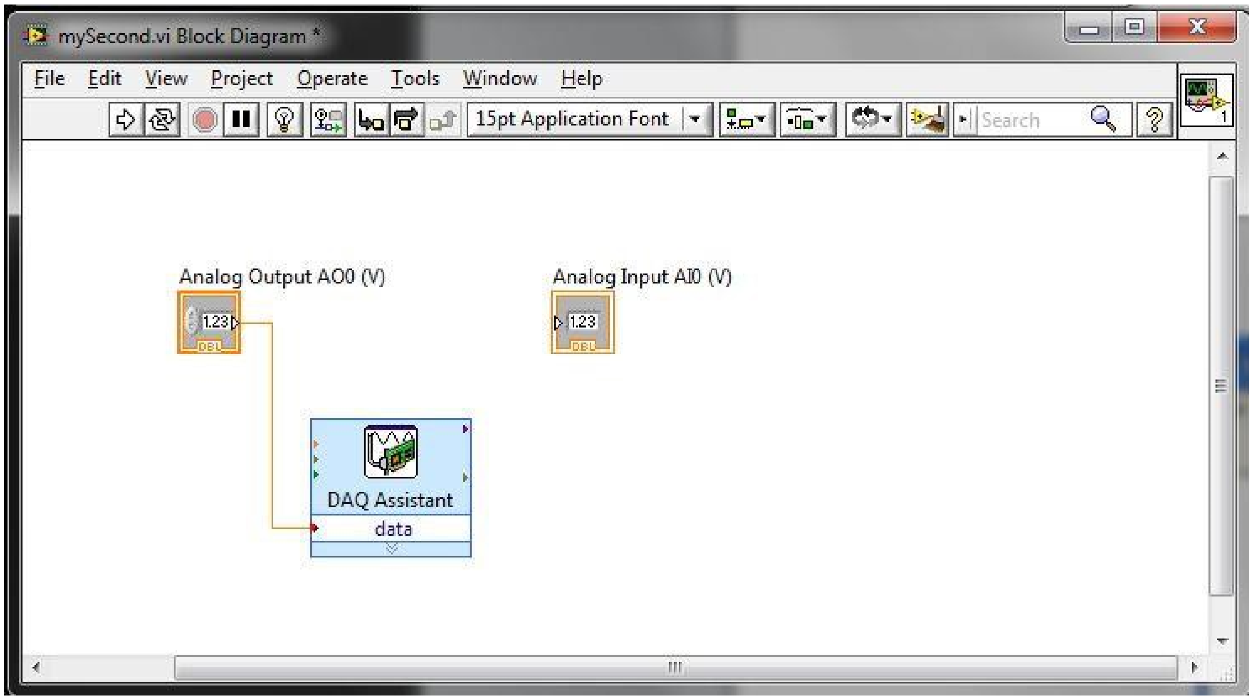
1 Sample (On Demand) specifies that the task acquire or generate one sample.

1 Sample (HW Timed) specifies that the task acquire or generate one sample on an edge of a hardware clock. *Note: LabVIEW SignalExpress*

Configuring the output channel. Note the selection of 1 Sample on Demand. Then click OK.

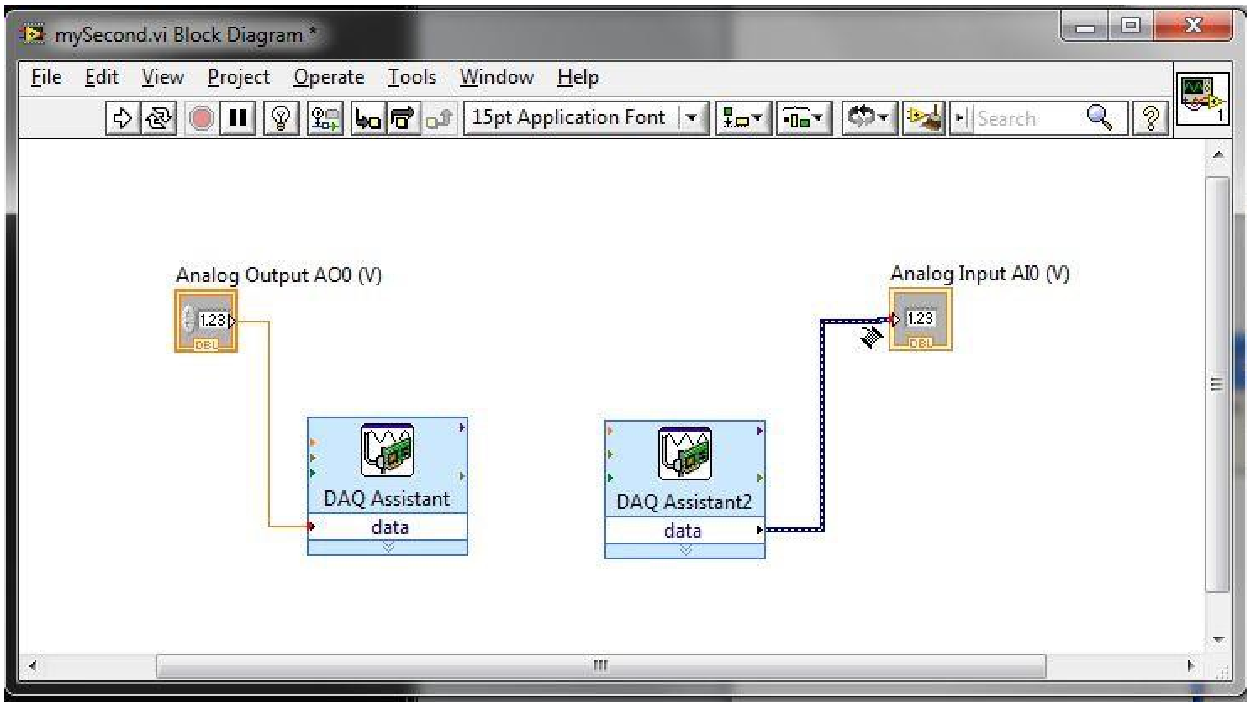
After this sequence of screens is completed, the DAQ Assistant generates the code needed to implement the data acquisition task. This transforms the appearance of the DAQ Assistant icon, to reveal terminals that permit it to be wired to the rest of the block diagram.

So, obviously, wire the data terminal of the DAQ Assistant to the Analog Output control:



Half of the Second.vi program is completed by wiring the DAQ Assistant to the Analog Out control.

Finally, repeat the DAQ Assistant selection, placement, and configuration process to connect an analog input function to the Analog Input indicator on the block diagram. As you go through this, you will need to select (in this order) Acquire Signals, Analog Input, Voltage, ai0, [Finish], 1 Sample on demand (under Acquisition Mode) – and make sure that the box labeled Terminal Configuration says Differential , [OK]. After clicking the OK button on the final DAQ Assistant configuration screen, code will be generated, and the DAQ Assistant icon will be transformed to reveal terminals. Wire this DAQ Assistant's data terminal to the Analog Input indicator. The final Second.vi block diagram should look like:



Completed block diagram for mySecond.vi.

I moved the "Analog Input AI0 (V)" indicator icon to the right to make space for the DAQ Assistant(2). To move any of the functional blocks, you just click on them with the pointer cursor to select them. Selection is indicated by "marching ants" – an animated dashed border. The cursor turns into the wiring tool when it gets too close to a terminal, so you have to pick a spot on the icon without a terminal (and hold your mouth just right . . .) to select by clicking. If you accidentally start to draw a wire because you got the wiring tool, you can make the wire and tool go away by hitting the Esc key. Once selected, the block can be either dragged with the mouse, or moved with the keyboard arrow keys. You can also move wires around without changing where they're connected, or you can select them with a right-click and select "Clean Up Wire Branch" to reorganize wiring. I often find I don't like the way the "Clean Up …" feature works. You can spend lots of time tidying up a block diagram.

Notice that the "wire" connecting AO0 to its DAQ Assistant is orange, but the wire between the AI0 indicator and its Assistant is a structured dark blue. The orange connection represents a double-precision floating point data type connection, whereas the structured dark blue represents a dynamically-typed variable, which the indicator is able to automatically convert into the double-precision floating point type it needs to display. Computer science geeks may recognize this as a reflection of the polymorphic nature of the object-oriented code routines that underlie the deceptively simple appearance of LabVIEW block diagrams.

Also notice that there is still no connection between the AO0 control and the AI0 indicator. This connection will be made (at first) with a real, physical pair of wires on the outside of the interface, connecting the AO0 and AO GND terminals to the AI0+ and AI0- terminals, respectively. This VI is no longer totally virtual, as it was when it was First.vi – it now has a real existence in our laboratory, and we can now measure AO0 with a DMM.

3.2.3b. Make the connections between AO0 and AI0 that are detailed in the last paragraph, and make a provision to physically measure the voltage produced by AO0 (and measured by AI0) with a DMM (set up as a voltmeter, of course). Record data to produce a graph of the DMM reading and AI0's reading versus the programmed value of AO0, for at least 10 values of the programmed value. This should give you a good idea of the precision and accuracy of the USB-6215 analog output and input systems. Chances are that graphs (using Excel, for example) of the DMM voltage reading vs. the programmed voltage, and AI0's reading (read from the VI front panel indicator) will be so linear that you won't be able to "eyeball" a difference. What does an Excel trendline tell you about the linearity of the result? Suppose you try a 3rd –order polynomial trendline – what does that tell you?

3.2.3c Characterize the analog output of AO0 by its I vs V dependence. For that end use your VI to program to output voltage of 10 V. Use a 10 kΩ pot as a variable load resistance. Measure the load curve varying the resistance of this variable load. Start with the pot adjusted to provide a 10 kΩ load resistance. Connect voltmeter and ammeter, and measure current and voltage as you did in Labs1 and Labs2. Plot your data. At this point you will be able to see that the load curve is not a straight line but a piecewise linear curve. This active source consists of non-linear elements and as such it cannot be represented by a simple Thevenin equivalent circuit. However, the source has two distinct operational regimes. Each of these regimes behaves in a good approximation as a simple Thevenin equivalent circuit but the Thevenin equivalent parameters (say $V_{th,1}$ and $R_{th,1}$, and $V_{th,2}$ and $R_{th,2}$) are different in these two regimes. Estimate these parameters in the two regimes as best as you can.

3.2.3d Controlling execution order.

There is a practical problem with the 'code' (= the block diagram) that we created for mySecond.vi. In spite of the fact that LabVIEW tends to execute the code represented by functional blocks from left to right and from top to bottom (sort of like spreadsheet cell dependencies suggest left to right and top to bottom evaluation order), this behavior is not guaranteed, and may vary depending on exactly how you arrange the functional blocks. What this means is that it is *conceivable* that the DAQ Assistant that reads the voltage produced by the analog output DAC may read that voltage *before* the DAQ Assistant that writes data to the DAC actually writes that data. This is probably not desirable behavior.

In this section, we will see one technique for controlling the execution order of the functional blocks in LabVIEW.

Begin by creating a new version of mySecond.vi, named myThird(.vi), using the File/Save As menu bar function that you used to make mySecond from myFirst.

In the block diagram of myThird, notice that there are terminals (shown by little colored triangles along the edges) on each DAQ Assistant block. You should be aware that it is common (and encouraged) practice for data input terminals to be on the left side of a block, and output data terminals to be on the right side. Exceptions to this are strongly discouraged, but they happen. Notice that the "data" input terminal of the DAQ Assistant that writes data to the analog output is on the left and that the "data" output terminal of the DAQ Assistant that reads data from the analog input is on the right, which is as it should be according to this convention.

Hover your mouse cursor over each terminal, and its function will pop up in a little window. Notice that each DAQ Assistant has a terminal named "error in" on its left edge and a terminal named "error out" on its right edge. If no data is actually wired to an input, there is a default value that is assumed by the code executed by the block (the DAQ Assistant in this case), and that default data is assumed to be automatically present. But if something *is* wired to an input, then the block is prevented from executing until that data is actually available from whatever the input is connected to (some output, somewhere). So – if you connect the “error out” output of the analog output DAQ Assistant to the “error in” input of the analog input DAQ Assistant, this “data flow architecture” feature will prevent the analog in DAQ Assistant from executing until after the analog output DAQ Assistant has done its thing (written its data to the analog output DAC). ***Before you blindly connect these two terminals, however*****, read the next paragraph.**

While we are getting fancy with execution sequence control techniques, we ought to anticipate another problem that may occur, a problem that follows from the simple physics of circuitry: to connect the analog output to the analog input takes two wires – one wire connects terminals labeled AO0 to AI0+ and the other connects AOGND to AI0-. There will probably be a voltage difference between these two wires, imposed by the analog output DAC. These two wires are insulated from each other (otherwise you have a short circuit and the system doesn’t work as you might expect). These two wires are therefore two conductors separated by an insulating space, and that is the prescription for a capacitor. In order to have a voltage difference across a capacitor requires a charge separation on the two conductors, +Q on one and –Q on the other, with Q=CV, where C is the capacitance of (or “between”) the two wires in question, and V the potential difference. So, in order to establish the potential difference V, the analog output DAC needs to physically move charge (probably electrons, which have mass and inertia) from the positive wire, through the rest of the circuitry and to the negative wire (if it’s electrons that are moved). This takes time – at a minimum, the charges can’t be moved faster than a tiny bit less than the speed of light (which is about a foot per nanosecond). But if there is resistance in the wires or the other circuitry, it will take much longer, with a characteristic time approximately equal to RC, where R is the total resistance between the voltage source and the capacitance (remember that the analog output DAC has some Thevenin resistance, which you measured earlier). The upshot of all this is that if you change the analog output from 0 volts to 10 volts, for example, there will be a time delay before the potential between the two wires at the end where the analog input is connected is 10 volts – the voltage seen by the analog input will rise from 0 to 10 volts in some time determined by the Thevenin resistance of the analog output source, the resistance of the wires, and the capacitance represented by the pair of wires. You can minimize the capacitance of the connecting wires, but it is physically impossible to make it zero.

So, while we are at it, we will take the precaution of inserting a controllable delay time between when the analog output is “written” and when the analog input is “read”. Between the two DAQ Assistants on your myThird.vi block diagram, place a time delay block: in the programming functions menu (brought up by right-clicking on the block diagram) is a page of functional blocks named “Timing” (it has a picture of a wristwatch on its icon), and on that sub-menu is a functional block named “Time Delay” (it has a picture of an hour-glass on its icon). Click the “Time Delay” block to select it so that you can place it on the block diagram between the two DAQ Assistants. You will probably need to move things around to make room. An easy way to move a bunch of stuff that’s wired together as a unit is to click on blank space near what you want to move, then drag open a selection box that either encompasses or touches what you want to move. When you release the drag-click, the items that are either inside the selection box, or intersected by it, will be selected, and you will be able to move everything selected by clicking and dragging. It is quite easy once you get the “hang” of it, but it does take a little practice.

The Time Delay block has an error input and an error output, and you can “wire” a “daisy chain”, from the analog output DAQ Assistant’s error out to the Time Delay’s error in, and from the Time Delay’s error out to the analog input DAQ Assistant’s error in. This daisy chaining of error outputs and inputs, plus the dataflow architecture of LabVIEW, guarantees that the analog out is set first, that a time delay occurs next, and that the analog input is read after the time delay is complete.

But wait – you need to add a numeric control to the front panel so that you can set the value of the time delay. Adding this control to the front panel causes its functional block counterpart to appear on the block diagram, where you can wire its output to the Delay Time (s) input. Note that this value is in seconds, so in practice, unless you really want to wait a number of seconds between the analog output and the analog input, you'll probably want to set the value on the control to something like a millisecond (0.001 s) or two. However, just to demonstrate that this execution timing control works as advertised, set the delay to something tangible like 2 or 5 seconds, run your myThird.vi, change the analog output control, and observe whether it takes 2 or 5 seconds for the analog input to register the change.

Chances are that you won't actually need a delay – you can set the delay time control to 0 – because the processing time or execution time required by LabVIEW to "get" from one block of code to the next is greater than the settling time of the external circuit, unless we deliberately add capacitance to the external connection between AO0 and AI0.

Aside – the "error in" and "error out" terminals on many functional blocks in LabVIEW are provided for error handing as well as for execution control; in fact their main purpose is for error handling and reporting, as the names imply, so that instruments and DAQ hardware capable of sensing anomalous behavior can report it to program code that can pass that information along to a user, or act on that information in some sensible way. The use of these terminals for execution sequence control is actually a (convenient) side effect, but a side effect that is widely used. In practice (i.e. in a professional LabVIEW VI that is intended for serious use - perhaps in a situation where a lot of money or life and limb are at stake, maybe because some analog output is being used to control a 100 kV power supply, for example) a lot of the block diagram will be devoted to handling anticipated and possibly dangerous error situations.

3.2.3 Acquire waveform data.

So far you acquired single readings form DAQ Analog to Digital Converter. It is possible to configure these inputs to acquire also time-varying signals. To do so you need to simply change in DAQ Assistant menu the Number Samples Acquired to Continuous. Also you need to set up display to show the waveform. Labview has two output graphs for waveform – WaveformChart and Waveform Graph. You can add them from Front panel view. Waveform Graph is more convenient of the two because it displays time axes in seconds rather than in year:month:day:hour:minute:second format. Once you have added the Graph to from panel you can wire it up to DAQ Assistant Output. Since you want to acquire samples continuously you need also a loop structure. DAQ Assistant offers you to create this structure automatically. Once you have finished the program connect function generator from your protoboard to AI0+ and ground from protoboard to AI0-. Record all three waveforms using using print screen function.

Did you notice that your waveforms drift from one acquisition to the next? This is because recording and wave generator on protoboard are not synchronized to each other. One way to synchronize them is to use TTL output from wave generator to start DAC recording. Unlike most wave generators, protoboard wave generator outputs TTL 5V pulse only for square wave pattern and not for sine or sawtooth waveforms. So, to synchronize the wave generator and DAC with TTL pulses you need to use square waveforms from the wave generator. Under Trigger menu in DAQ Wizard you can specify that you want to start recording only when trigger has been activated. Then wire TTL output to P0.0 input or any other digital input that you specify on the menu (P0.1-P0.3). Connect also protoboard ground to digital GND adjacent to digital inputs.

Instructions for lab write-up

Copy and paste screenshots of front panel and block diagrams to your report. Include in your report also measurements of source voltages (discuss accuracy of this measurement) and load curves (determine Thenevin) resistance.