LabVIEW

LABVIEW BASICS I

Eng. Salvatore La Malfa

Eng. Salvatore La Malfa
PhD Student
D.I.E.E.S. - Department of Electrical, Electronic and Systems Engineering
University of Catania - Faculty of Engineering
V.le. A. Doria, 6 - 95125 Catania, Italy
Tel: +39-095-7382341
Fax: +39-095-330793
e-mail: salvatore.lamalfa@diees.unict.it

NATIONAL INSTRUMENTS

Introduction to VI

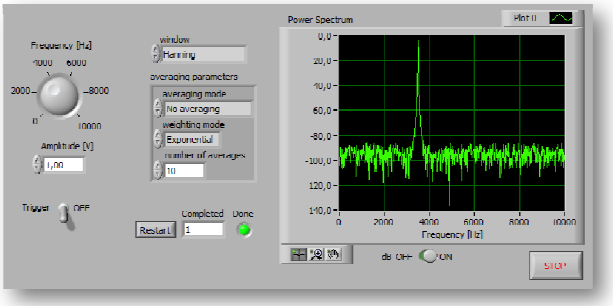
▪ Introduction to VI
▪ LabVIEW data types
▪ Block diagram nodes
▪ Built-in Functions
▪ Array & Clusters
▪ Waveforms
▪ Express VI & Dynamic Data
▪ Loops & Structures
▪ Tunnels & Indexing
▪ Shift Register & Feedback N.

*LabVIEW programs are called virtual instruments (VI) because **their appearance and operation imitate physical instruments**, such as oscilloscopes and multimeters. Every VI uses functions that manipulate input from the user interface or other sources and display that infromation or move to other file or other computers.*

LabVIEW User Manual

A VI contains the following three components:

- **Front panel** – Serves as the user interface
- **Block diagram** – Contains the graphical source code (G language) that defines the functionality of the VI
- **Icon and connector pane** – Identifies the VI so that you can use the I in another VI. A VI within another VI is called subVI. A subVI corresponds to a subroutine in text-based programming languages.



You build the front panel with **controls and indicators**, which are the interactive input and output terminals of the VI, respectively. Controls are knobs, push buttons, dials, and other input devices. Indicators are graphs, LEDs, and other displays. **Controls simulate instruments input devices** and supply data to the block diagram of the VI. **Indicators simulate instrument output devices** and display data the block diagram acquires or generates.

Introduction to VI

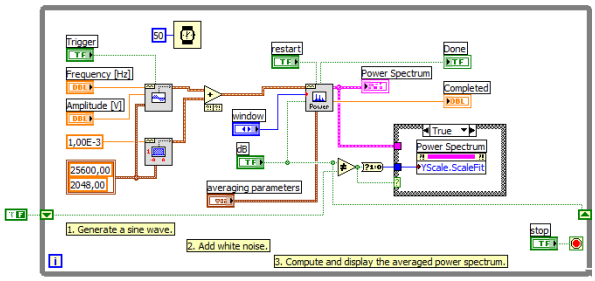
Ing. Salvatore La Malfa, Sistemi Integrati di Misura, 3.4.2009

*LabVIEW programs are called virtual instruments (VI) because **their appearance and operation imitate physical instruments**, such as oscilloscopes and multimeters. Every VI uses functions that manipulate input from the user interface or other sources and display that infromation or move to other file or other computers.*

LabVIEW User Manual

A VI contains the following three components:

- **Front panel** – Serves as the user interface
- **Block diagram** – Contains the graphical source code (G language) that defines the functionality of the VI
- **Icon and connector pane** – Identifies the VI so that you can use the I in another VI. A VI within another VI is called subVI. A subVI corresponds to a subroutine in text-based programming languages.



After you build the front panel, you add code using graphical representations of functions to control the front panel objects. The block diagram contains this **graphical source code**

Introduction to VI

- Introduction to VI
- LabVIEW data types
- Block diagram nodes
- Built-in Functions
- Array & Clusters
- Waveforms
- Express VI & Dynamic Data
- Loops & Structures
- Tunnels & Indexing
- Shift Register & Feedback N.

Ing. Salvatore La Malfa, Sistemi Integrati di Misura, 3.4.2009

*LabVIEW programs are called virtual instruments (VI) because **their appearance and operation imitate physical instruments**, such as oscilloscopes and multimeters. Every VI uses functions that manipulate input from the user interface or other sources and display that infromation or move to other file or other computers.*

LabVIEW User Manual

A VI contains the following three components:

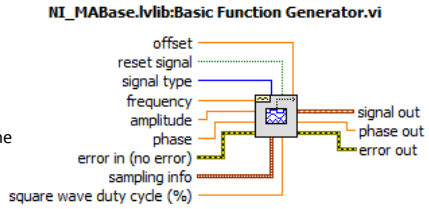
- **Front panel** – Serves as the user interface
- **Block diagram** – Contains the graphical source code (G language) that defines the functionality of the VI
- **Icon and connector pane** – Identifies the VI so that you can use the It in another VI. A VI within another VI is called subVI. A subVI corresponds to a subroutine in text-based programming languages.

Icon



Connectio pane





Creates an output waveform based on **signal type**.

The Icon identifies the VI so you can use it in another VI

The connector pane is a set of terminals that corresponds to the controls and indicators of that VI, *similar to the parameter list of a function call in text-based programming languages*. The connector pane defines the inputs and outputs you can wire to the VI

LabVIEW data types

In LabVIEW there are 32 different data types. The color and symbol of each terminal indicate the data type of the corresponding control or indicator. In the following table the 9 most common are reported.

Control	Indicator	Description
		**Float.** Double-precision, **floating-point numeric**
		**Integer.** 32-bit signed integer numeric
		**Boolean**. Stores Boolean (TRUE/FALSE) values.
		**String**. Provides a platform-independent format for information and data, which you can use to create simple text messages, pass and store numeric data, and so on.
		**Array**. Encloses the data type of its elements in square brackets and takes the color of that data type. As you add dimensions to the array, the brackets become thicker.
		**Cluster.** Encloses several data types. Cluster data types appear brown if all elements in the cluster are numeric or pink if all elements of the cluster are of different types. Error code clusters appear dark yellow, while LabVIEW class clusters are crimson by default or teal green for Report Generation VIs.
		**Dynamic data.** (Express VIs) Includes data associated with a signal and the attributes that provide information about the signal, such as the name of the signal or the date and time the data was acquired.
		**Waveform.** Carries the data, start time, and *t* of a waveform.
		**I/O.** Passes resources you configure to I/O VIs to communicate with an instrument or a measurement device.

Block Diagram Nodes

Nodes are objects on the block diagram that have inputs and/or outputs and perform operations when a VI runs.

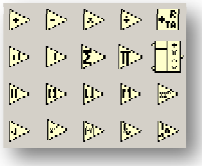
They are analogous to statements, operators, functions, and subroutines in text-based programming languages. LabVIEW includes the following types of nodes:

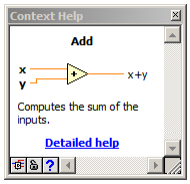
- **Functions**—Built-in execution elements, comparable to an operator, function, or statement.
- **SubVIs**—VIs used on the block diagram of another VI, comparable to subroutines.
- **Express VIs**—SubVIs designed to aid in common measurement tasks. You configure an Express VI using a configuration dialog box.
- **Structures**—Execution control elements, such as For Loops, While Loops, Case structures, Flat and Stacked Sequence structures, Timed structures, and Event structures.
- **Formula and Expression Nodes**—Formula Nodes are resizable structures for entering equations directly into a block diagram. Expression Nodes are structures for calculating expressions that contain a single variable.
- **Property and Invoke Node**—Property Nodes are structures for setting or finding properties of a class. Invoke Nodes are structures for executing methods of a class.
- **Call by Reference Nodes**—Structures for calling a dynamically loaded VI.
- **Call Library Function Nodes**—Structures for calling most standard shared libraries or DLLs.
- **Code Interface Nodes (CINs)**—Structures for calling code written in text-based programming languages.

Functions

Nodes are objects on the block diagram that have inputs and/or outputs and perform operations when a VI runs.

- **Functions are the essential operating elements of LabVIEW**. Function icons on the **Functions** palette have pale yellow backgrounds and black foregrounds.
- Functions do not have front panels or block diagrams but do have connector panes.
- **You cannot open or edit a function**.





You can change the number of terminals for some functions. You can add terminals to functions by using the Positioning tool to **drag the top or bottom borders** of the function up or down, respectively



You also can add or remove terminals by right-clicking one of the terminals of the function and selecting **Add Input**, **Add Output**, **Remove Input**, or **Remove Output** from the shortcut menu. Depending on the function, you can add terminals for inputs, outputs, or refnum controls. The **Add Input** and **Add Output** shortcut menu items add a terminal immediately after the terminal you right-clicked. The **Remove Input** and **Remove Output** shortcut menu items remove the terminal you right-clicked. If you use the shortcut menu items to remove a wired terminal, LabVIEW removes the terminal and disconnects the wire.

Array & Clusters

Use the **array** and **cluster** controls and functions **to group data**.

Arrays group data elements of the same type. Clusters group data elements of mixed types.

Arrays

An array consists of elements and dimensions. Elements are the data that make up the array. A dimension is the length, height, or depth of an array.

You can build arrays of numeric, Boolean, path, string, waveform, and cluster data types. Consider using arrays when you work with a collection of similar data and when you perform repetitive computations. *Arrays are ideal for storing data you collect from waveforms or data generated in loops, where each iteration of a loop produces one element of the array.*

Array indexes in LabVIEW are zero-based.

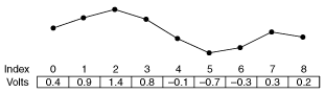
Array elements are ordered. An array uses an index so you can readily access any particular element. The index is zero-based, which means it is in the range 0 to *n* – 1, where *n* is the number of elements in the array.

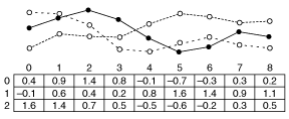
An example of an array is a waveform represented as a numeric array in which each successive element is the voltage value at successive time intervals.

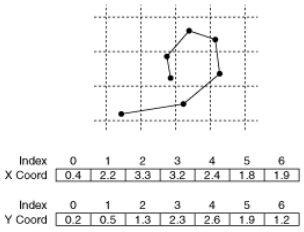
A more complex example of an array is a graph represented as an array of points where each point is a cluster containing a pair of numeric values that represent the X and Y coordinates.

A 2D array stores elements in a grid. It requires a column index and a row index to locate an element, both of which are zero-based.

The row index selects the waveform, and the column index selects the point on the waveform.







Array & Clusters

Use the **array** and **cluster** controls and functions **to group data**.

Arrays group data elements of the same type. Clusters group data elements of mixed types.

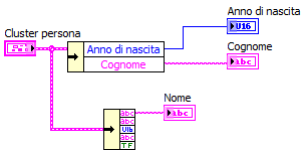
Cluster

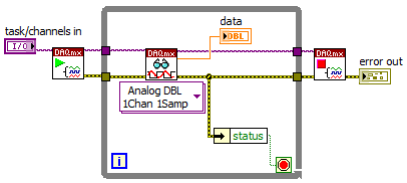
Clusters group data elements of mixed types. *A cluster is similar to a record or a struct in text-based programming languages.*
Bundling several data elements into clusters **eliminates wire clutter on the block diagram** and **reduces the number of connector pane** terminals that subVIs need. The connector pane has, at most, 28 terminals. If your front panel contains more than 28 controls and indicators that you want to pass to another VI, group some of them into a cluster and assign the cluster to a terminal on the connector pane.
Most clusters on the block diagram have a pink wire pattern and data type terminal. Error clusters have a dark yellow wire pattern and data type terminal. Clusters of numeric values, sometimes referred to as points, have a brown wire pattern and data type terminal. You can wire brown numeric clusters to Numeric functions, such as Add or Square Root, to perform the same operation simultaneously on all elements of the cluster.

The **error in** and **error out** clusters include the following components of information:

status is a Boolean value that reports TRUE if an error occurred.
code is a 32-bit signed integer that identifies the error numerically. A nonzero error code coupled with a **status** of FALSE signals a warning rather than a error.
source is a string that identifies where the error occurred.
Some VIs, functions, and structures that accept Boolean data also recognize an error cluster. For example, you can wire an error cluster to the Boolean inputs of the Select, Quit LabVIEW, or Stop functions. If an error occurs, the error cluster passes a TRUE value to the function.



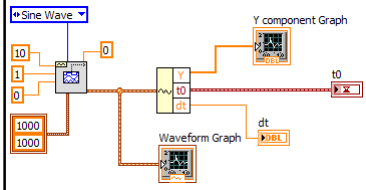


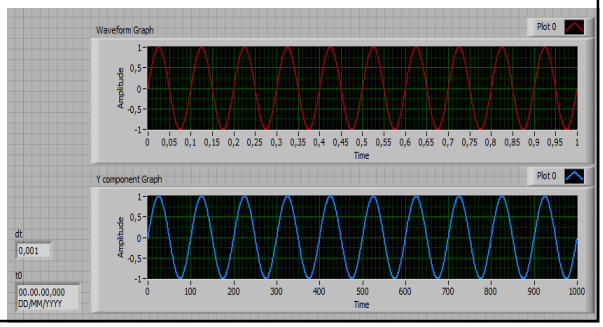


Waveforms

The waveform data type carries the **data**, **start time**, and **delta *t*** of a waveform.

- You can create a waveform using the **Build Waveform** function.
- Many of the VIs and functions you use to acquire or analyze waveforms accept and return waveform data by default.
- When you wire waveform data to a waveform graph or chart, the graph or chart **automatically** plots a waveform based on the data, start time, and delta *x* of the waveform.
- When you wire an array of waveform data to a waveform graph or chart, the graph or chart automatically plots all waveforms.

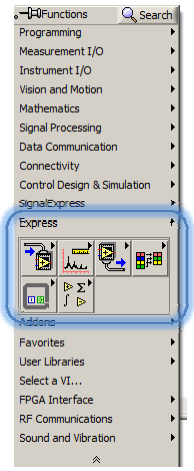




Express VI

Ing. Salvatore La Malfa, Sistemi Integrati di Misura, 3.4.2009

An **Express VI** is a VI whose settings you can configure interactively through a dialog box. Express VIs appear on the block diagram as expandable nodes with icons surrounded by a blue field.

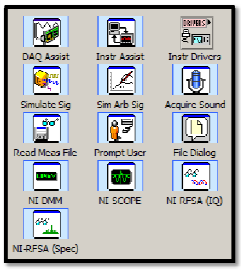


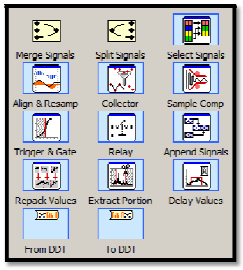
You can configure an Express VI by setting options in the **configuration dialog box** that appears when you place the Express VI on the block diagram.

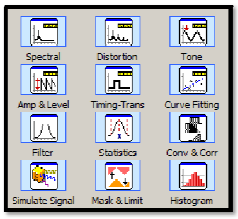
The primary benefit of Express VIs is their **interactive configurability**. Express VIs are useful when you want to give users a VI or library of VIs for building their own applications easily *with minimal programming expertise.*

Express VIs **do not provide run-time interactive configuration for VIs**. If you need run-time reconfiguration, build an application with a user interface that contains features similar to a configuration dialog box. Express VIs are designed for ease of use.

If you need an application to run with strict memory restrictions or high execution speeds, use standard VIs.







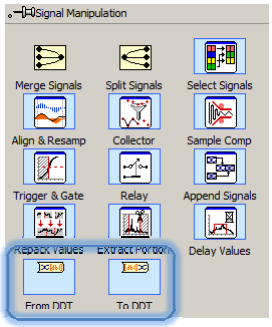
Dynamic Data

Ing. Salvatore La Malfa, Sistemi Integrati di Misura, 3.4.2009

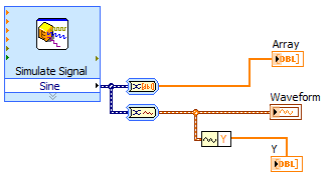
Most Express VI accept and/or return the dynamic data type.

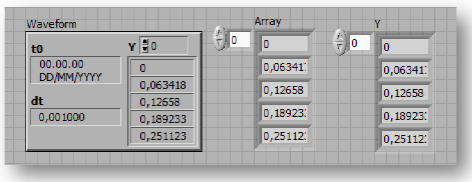
The dynamic data type accepts data from and sends data to the following data types, where the scalar data type is a floating-point number or a Boolean value:

- 1D array of waveforms
- 1D array of scalars
- 1D array of scalars—most recent value
- 1D array of scalars—single channel
- 2D array of scalars—columns are channels
- 2D array of scalars—rows are channels
- Single scalar
- Single waveform



The dynamic data type is for use with Express VIs. **Most other VIs and functions that ship with LabVIEW do not accept this data type**. To use a built-in VI or function to analyze or process the data the dynamic data type includes, you must convert the dynamic data type.



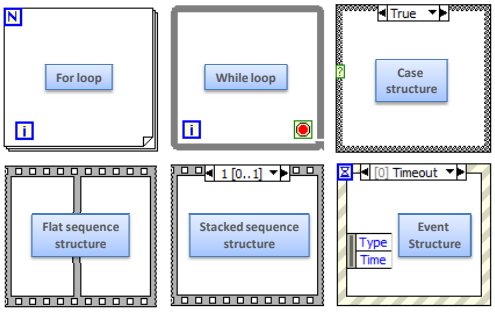


Loops and Structures

Structures are graphical representations of the loops and case statements of text-based programming languages. Use structures on the block diagram to repeat blocks of code and **to execute code conditionally or in a specific order**. Like other nodes, structures have terminals that connect them to other block diagram nodes, execute automatically when input data is available, **and supply data to output wires when execution completes**.

Each structure has a distinctive, resizable **border** to enclose the section of the block diagram that executes according to the rules of the structure. The section of the block diagram inside the structure border is called a subdiagram. The terminals that feed data into and out of structures are called **tunnels**. A tunnel is a connection point on a structure border.

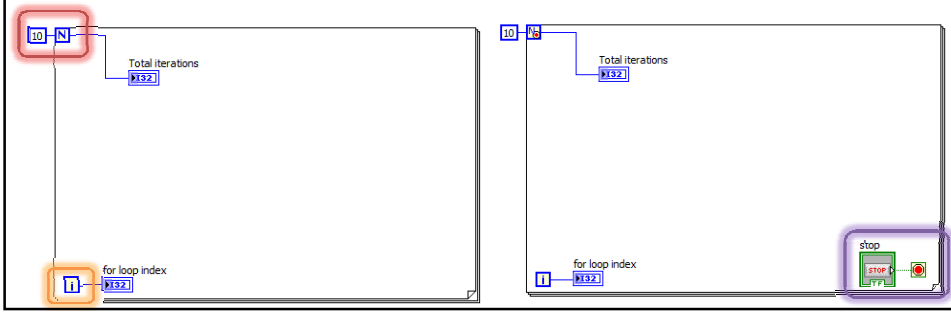
For Loop—Executes a subdiagram a set number of times or, if you add a conditional terminal, until a Boolean condition or an error occurs.
While Loop—Executes a subdiagram until a Boolean condition or an error occurs.
Case structure—Contains multiple subdiagrams, only one of which executes depending on the input value passed to the structure.
Sequence structure—Contains one or more subdiagrams that execute in sequential order.
Event structure—Contains one or more subdiagrams that execute when events are generated by user interaction.
Timed structures—Execute one or more subdiagrams with time bounds and delays.
Conditional Disable structure—Contains one or more subdiagrams, exactly one of which compiles and executes at run-time.
Diagram Disable structure—Contains one or more subdiagrams, exactly one of which compiles and executes at run-time.



For Loop

A For Loop executes a subdiagram a set number of times.
•**The count terminal** is an input terminal whose value indicates how many times to repeat the subdiagram
Set the count explicitly by wiring a value from outside the loop to the left or top side of the count terminal, *or set the count implicitly with auto-indexing.*
•**The iteration terminal** is an output terminal that contains the number of completed iterations. The iteration count always starts at zero.
Both the count and iteration terminals are *32-bit signed integers*. If you wire a floating-point or fixed-point number to the count terminal, LabVIEW rounds it and coerces it to within range.

You can add a conditional terminal to configure a For Loop to stop when a Boolean condition or an error occurs. A For Loop with a conditional terminal executes until the condition occurs or until all iterations complete, whichever happens first. For example, if you want a loop to execute a set number of times unless an error occurs, you can use a For Loop with a conditional terminal and wire an error cluster to the conditional terminal, shown as follows.



While Loop

Similar to a Do Loop or a Repeat-Until Loop in text-based programming languages, a **While Loop** executes a subdiagram until a condition occurs. The While Loop executes the subdiagram until the conditional terminal, an input terminal, receives a specific Boolean value.

The While Loop does not include a set iteration count and **runs infinitely if the condition never occurs.**

If a conditional terminal is **Stop if True**, you place the terminal of a Boolean control outside a While Loop, and the control is FALSE when the loop starts, you cause an infinite loop, as shown in the following example. You also cause an infinite loop if the conditional terminal is **Continue if True** and the control outside the loop is set to TRUE.
The iteration terminal, shown as follows, is an output terminal that contains the number of completed iterations.
The iteration count always starts at zero. During the first iteration, the iteration terminal returns 0.



Auto Indexing Loops

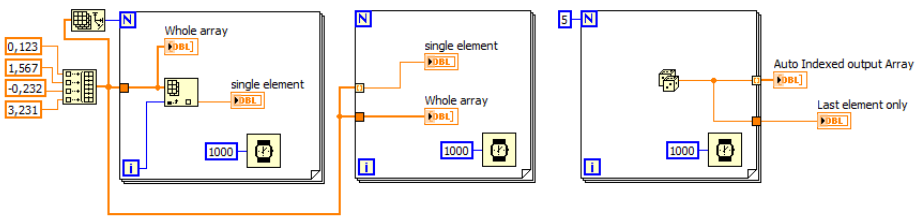
If you wire an array to the input tunnel of a For Loop or While Loop, you can read and process every element in that array by enabling auto-indexing.

When you wire an array from an external node to an input tunnel on the loop border and enable auto-indexing on the input tunnel, elements of that array enter the loop one at a time, starting with the first element.
When auto-indexing is disabled, the entire array is passed into the loop.

Right-click the tunnel at the loop border and select **Enable Indexing** or **Disable Indexing** from the shortcut menu to enable or disable auto-indexing. Auto-indexing for While Loops is disabled by default.

The loop indexes scalar elements from 1D arrays, 1D arrays from 2D arrays, and so on. The opposite occurs at output tunnels. Scalar elements accumulate sequentially into 1D arrays, 1D arrays accumulate into 2D arrays, and so on.

If you enable auto-indexing on an array wired to a For Loop input terminal, LabVIEW sets the count terminal to the array size so you do not need to wire the count terminal.

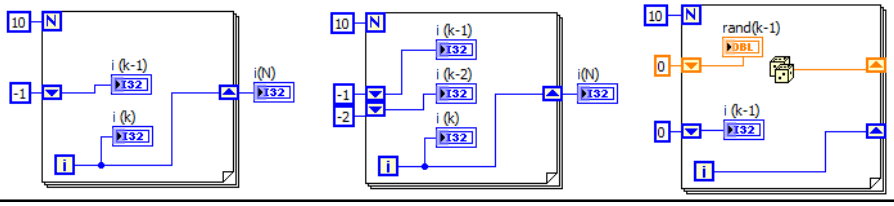


Shift Registers

Ing. Salvatore La Malfa, Sistemi Integrati di Misura, 3.4.2009

Use shift registers with For Loops or While Loops to transfer values from one loop iteration to the next.

A shift register appears as a **pair of terminals** directly opposite each other on the vertical sides of the loop border.
The terminal on the right side of the loop contains an up arrow and stores data on the completion of an iteration.
LabVIEW transfers the data stored in the right terminal of the shift register to the left terminal. The loop then uses the data from the left terminal as the initial values for the next iteration. This process continues until all iterations of the loop execute. After the loop executes, the terminal on the right side of the loop returns the last value stored in the shift register. *Create a shift register by right-clicking the left or right border of a loop and selecting **Add Shift Register** from the shortcut menu.*
You can add more than one shift register to a loop. If you have multiple operations that use previous iteration values within a loop, use multiple shift registers to store the data values from those different processes in the structure, as shown in the following block diagram.
Initializing a shift register resets the value the shift register passes to the first iteration of the loop when the VI runs. If you do not initialize the shift register, the loop uses the value written to the shift register when the loop last executed or, if the loop has never executed, the default value for the data type.
Use an uninitialized shift register to preserve state information between subsequent executions of a VI.
Stacked shift registers let you access data from previous loop iterations. Stacked shift registers remember values from multiple previous iterations and carry those values to the next iterations. *To create a stacked shift register, right-click the left terminal and select **Add Element** from the shortcut menu.*



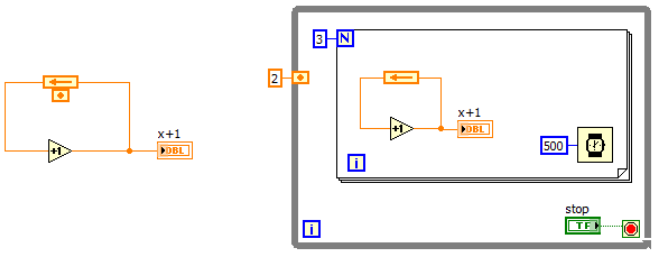
Feedback Node

Ing. Salvatore La Malfa, Sistemi Integrati di Misura, 3.4.2009

Use Feedback Node with For Loops or While Loops to transfer values from one loop iteration to the next.

The feedback Node operates similarly to shift registers in loops

The Feedback Node stores data from only the previous iteration.
To initialize a Feedback Node when a loop executes, right-click the initializer terminal and select **Move Initializer One Loop Out** or **Move Initializer One Loop In** from the shortcut menu to move the initializer terminal. The Feedback Node then initializes before the first iteration of each execution of the loop. You must wire an initial value to the initializer terminal if you move it to the left edge of a loop.
If the initializer terminal is on the edge of the outermost or innermost loop LabVIEW dims the corresponding option to **Move Initializer One Loop Out** or **Move Initializer One Loop In** in the shortcut menu.
If you do not move the initializer terminal to the left edge of a loop, the Feedback Node **globally initializes**. You can use a Feedback Node that globally initializes anywhere in a block diagram. If a Feedback Node globally initializes and you set an initial value, the Feedback Node initializes to that value on the first call of the VI in an execution.
If you do not wire an input value to the initializer terminal, the initial input of the Feedback Node for the first execution is the default value for each data type**. Each time the VI runs after the first execution, the initial value is the last value from the previous execution.**

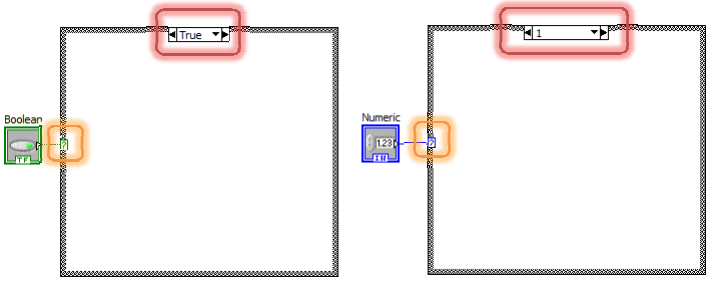


Case Structure

A **Case structure** has two or more subdiagrams, or cases. Only one subdiagram is visible at a time, and **the structure executes only one case at a time**. An input value determines which subdiagram executes.
The Case structure is similar to switch statements or if...then...else statements in text-based programming languages.

The **case selector** label at the top of the Case structure contains the name of the selector value that corresponds to the case in the center and decrement and increment arrows on each side.
Wire an input value, or selector, to the **selector terminal**, to determine which case executes.
You must wire an integer, Boolean value, string, or enumerated type value to the selector terminal.
If you do not specify a **default** for the Case structure to handle out-of-range values, you must explicitly list every possible input value. For example, if the selector is an integer and you specify cases for 1, 2, and 3, you must specify a default case to execute if the input value is 4 or any other unspecified integer value.

You can create multiple input and output tunnels for a Case structure. Inputs are available to all cases, but cases do not have to use each input. However, **you must define each output tunnel for each case**.



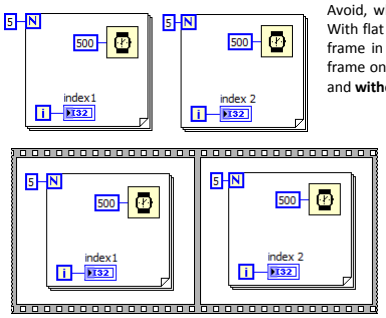
Sequence Structure

A **sequence structure** contains one or more subdiagrams, or frames, that execute in **sequential order**.
Within each frame of a sequence structure, as in the rest of the block diagram, data dependency determines the execution order of nodes.
There are two types of sequence structures:

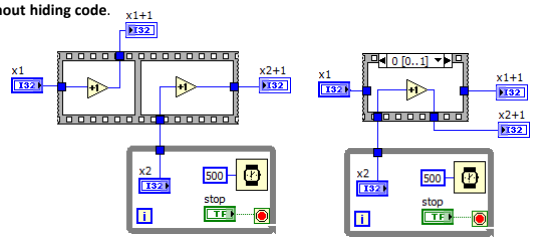
- **Flat Sequence structure** executes frames from left to right and when all data values wired to a frame are available. The data leaves each frame as the frame finishes executing. This means the input of one frame can depend on the output of another frame.
- **Stacked Sequence structure** The Stacked Sequence structure, shown as follows, stacks each frame so you see only one frame at a time and executes frame 0, then frame 1, and so on until the last frame executes. The Stacked Sequence structure returns data only after the last frame executes. Use the Stacked Sequence structure if you want to conserve space on the block diagram.

Rely on data flow rather than sequence structures to control the order of execution. (for example by the use error clusters)

To take **advantage of the inherent parallelism in LabVIEW**, avoid overusing sequence structures. Sequence structures guarantee the order of execution and prohibit parallel operations. For example, asynchronous tasks that use I/O devices, such as PXI, GPIB, serial ports, and DAQ devices, can run concurrently with other operations if sequence structures do not prevent them from doing so.



Avoid, when possible, using stacked sequence structures. Use flat sequences instead. With flat sequences you do not need to use **sequence locals** to pass data from frame to frame in the Flat Sequence structure. Since the Flat Sequence structure displays each frame on the block diagram, you can wire from frame to frame without sequence locals and **without hiding code**.

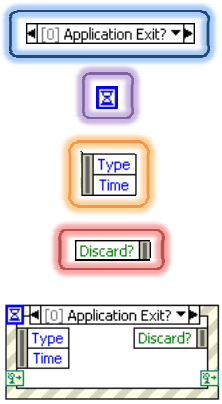


Event Structure

Use the Event structure to handle events in a VI.
The Event structure works like a *Case structure* with a built-in Wait on notification function.
The Event structure can have multiple cases, each of which is a separate event-handling routine.
You can configure each case to handle one or more events, but only one of these events can occur at a time.

When the Event structure executes, it waits until one of the configured events occur, then executes the corresponding case for that event. The Event structure completes execution after handling exactly one event. It does not implicitly loop to handle multiple events. Like a Wait on Notification function, the Event structure can time out while waiting for notification of an event. When this occurs, a specific Timeout case executes.

- **The Event Selector Label** at the top of the Event structure indicates which events cause the currently displayed case to execute. View other event cases by clicking the down arrow next to the case name and selecting another case from the shortcut menu.
- **The Timeout Terminal** at the top left corner of the Event structure specifies the number of milliseconds to wait for an event before timing out. The default is –1, which specifies to wait indefinitely for an event to occur. If you wire a value to the Timeout terminal, you must provide a *Timeout* case.
- **The Event Data Node** behaves similarly to the Unbundle by name function. This node is attached to the inside left border of each event case. The node identifies the data LabVIEW provides when an event occurs. You can resize this node vertically to add more data items, and you can set each data item in the node to access any event data element. The node provides different data elements in each case of the Event structure depending on which event(s) you configure that case to handle. If you configure a single case to handle multiple events, the Event Data Node provides only the event data elements that are common to all the events configured for that case.
- **The Event Filter Node**, is similar to the Event Data Node. This node is attached to the inside right border of filter event cases. The node identifies the subset of data available in the Event Data Node that the event case can modify. The node displays different data depending on which event(s) you configure that case to handle. By default, these items are inplace to the corresponding data items in the Event Data Node. If you do not wire a value to a data item of an Event Filter Node, that data item remains unchanged.



LabVIEW™

LABVIEW BASICS II

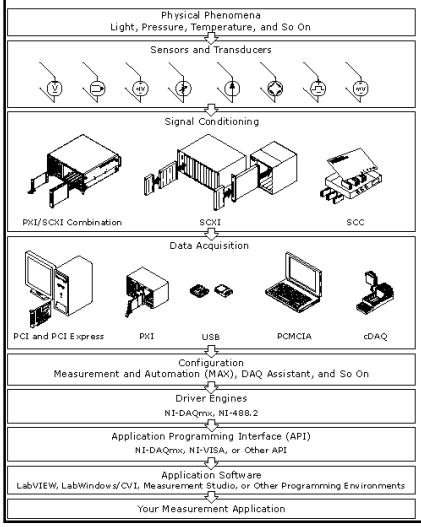
Eng. Salvatore La Malfa

Eng. Salvatore La Malfa
PhD Student
D.I.E.E.S. - Department of Electrical, Electronic and Systems Engineering
University of Catania - Faculty of Engineering
V.le A. Doria, 6 - 95125 Catania, Italy
Tel: +39-095-7382341
Fax: +39-095-330793
e-mail: salvatore.lamalfa@diees.unict.it

NATIONAL INSTRUMENTS

Measurements Fundamentals

NI measurement devices and application software are packaged with NI-DAQ driver software to program all the features of your NI measurement device such as **configuring**, **acquiring**, and **generating data from and sending data to NI measurement devices**. **Using NI-DAQ saves you from having to write these programs yourself.** Application software, such as LabVIEW, LabWindows™/CVI™, and Measurement Studio, sends the commands to the driver, such as acquire and return a thermocouple reading, and then displays and analyzes the data acquired.

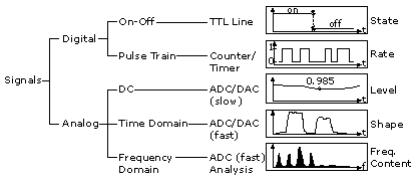


- **Sensors and transducers** detect physical phenomena.
- **Signal conditioning** components condition physical phenomena so that the measurement device can receive the data.
- The computer receives the data through the **measurement device**.
- **Software** controls the measurement system, telling the measurement device when and from which channels to acquire or generate data.
- Software also takes the raw data, analyzes it, and **presents it in a form you can understand**, such as a graph, chart, or file for a report.

A signal is classified as **analog** or **digital** by the way it conveys information.

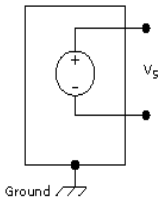
- A digital (or binary) signal has only two possible discrete levels—high level (on) or low level (off).
- An analog signal, on the other hand, contains information in the continuous variation of the signal with respect to time.

A breakdown of the main signal types is shown in the following figure.

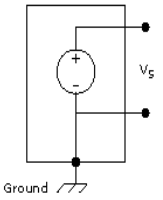


Analog Connection considerations

To measure analog signals, you need to know the signal source
grounded or floating.
You also must consider the measurement system
differential , referenced single-ended, or nonreferenced single-ended.

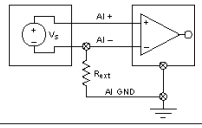
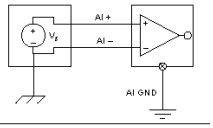
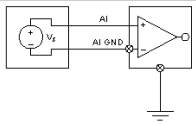
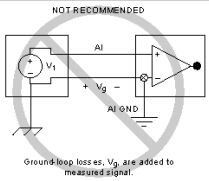
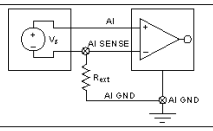
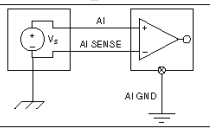
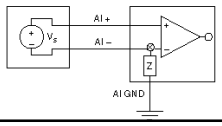
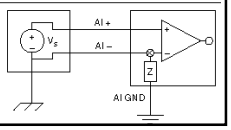


You can measure **floating signal sources** with both **differential** and **single-ended** measurement systems. In the case of the differential measurement system, however, make sure the common-mode voltage level of the signal with respect to the measurement system ground remains in the common-mode input range of the measurement device. A variety of phenomena—for example, the instrumentation amplifier input bias currents—can move the voltage level of the floating source out of the valid range of the input stage of a DAQ device.



A **grounded signal source** is best measured with a **differential** or an **nonreferenced single-ended** measurement system. *If you use an RSE measurement system with a grounded source, the result is typically a noisy measurement system* often showing power-line frequency (60 Hz) components in the readings. Ground-loop introduced noise can have both AC and DC components, introducing offset errors and noise in the measurements. The potential difference between the two grounds causes a current to flow in the interconnection. This current is called ground-loop current.
However, you can still use an RSE measurement system if the signal voltage levels are high and the interconnection wiring between the source and the measurement device has a low impedance. In this case, the signal voltage measurement is degraded by ground loop, but the degradation may be tolerable. You must **observe the polarity** of a grounded signal source before connecting the signal to a ground-referenced measurement system because the signal source can be short-circuited to ground, which can damage the signal source.

Analog Connection considerations

Input	Signal Source Type	
	Floating Signal Source (Not Connected to Building Ground) Examples: Ungrounded thermocouples, signal conditioning with isolated outputs, battery devices	Grounded Signal Source Example: Instruments with nonisolated outputs
Differential (DIFF) A differential measurement system has neither of its inputs tied to a fixed reference, such as earth or building ground. A differential measurement system is similar to a floating signal source in that the measurement is made with respect to a floating ground that is different from the measurement system ground		
Ground Referenced Single-Ended (RSE) Note: AI GND is shared as a reference for all RSE channels Referenced and nonreferenced single-ended measurement systems are similar to grounded sources in that the measurement is made with respect to ground. A referenced single-ended (RSE) measurement system measures voltage with respect to the ground, AIGND in the figure, which is directly connected to the measurement system ground.		
Nonreferenced Single-Ended (NRSE) Note: AI SENSE is shared as a reference for all NRSE channels In an NRSE measurement system, all measurements are still made with respect to a single-node analog input sense (AISENSE), but the potential at this node can vary with respect to the measurement system ground.		
Pseudodifferential Pseudodifferential input configurations are common in simultaneous sampling and dynamic signal acquisition (DSA) devices that do not employ a multiplexed signal architecture. A pseudodifferential system is well-suited for measuring the output of floating or isolated devices under test such as battery-powered instruments or most accelerometers.		

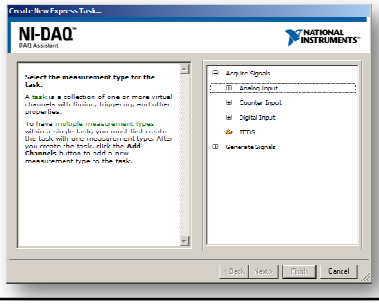
NI DAQmx key concepts

Virtual channels and tasks are fundamental components of NI-DAQmx.
• Virtual channels or sometimes referred to generically as channels, are software entities that encapsulate the physical channel along with other channel specific information—range, terminal configuration, and custom scaling—that formats the data
• Tasks are collections of one or more virtual channels with timing, triggering, and other properties.

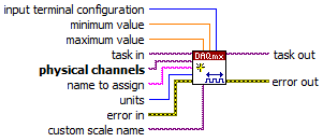
A *physical channel* is a terminal or pin at which you can measure or generate an analog or digital signal. A single physical channel can include more than one terminal, as in the case of a differential analog input channel or a digital port of eight lines. Every physical channel on a device has a unique name (for instance, SC1Mod4/ai0, Dev2/ao5, and Dev6/ctr3).

To create virtual channels, use the DAQmx Create Virtual Channel function/VI or the DAQ Assistant.

Virtual channels created with the DAQmx Create Virtual Channel function/VI are called *local virtual channels* and can only be used within the task.If you create virtual channels with the DAQ Assistant, you can use them in other tasks and reference them outside the context of a task. Because these channels can apply to multiple tasks, they are called *global virtual channels*. You can select global virtual channels with the NI-DAQmx API or DAQ Assistant and add them to a task.



DAQmx Create Channel (AI-Voltage-Basic).vi



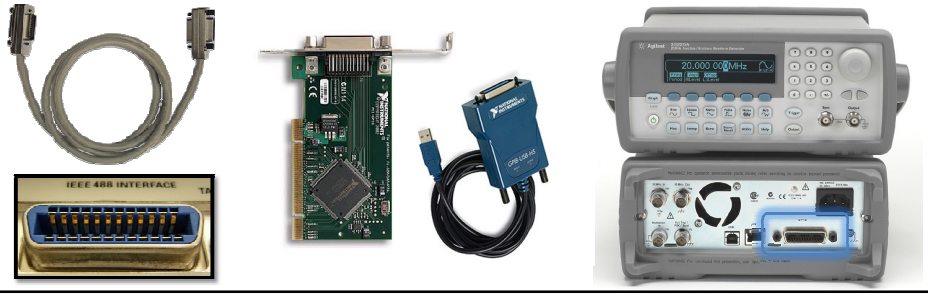
Creates channel(s) to measure voltage. If the measurement requires the use of internal excitation or you need excitation to scale the voltage, use the AI Custom Voltage with Excitation instance of this VI.

IEEE 488.2 (GPIB)

GPIB, or *General Purpose Interface Bus*, instruments offer test and manufacturing engineers the widest selection of vendors and instruments for general-purpose to specialized vertical market test applications. GPIB instruments are often used as stand-alone benchtop instruments where measurements are taken by hand.

You can automate these measurements by using a PC to control the GPIB instruments.

The GPIB protocol categorizes devices as *controllers*, *talkers*, or *listeners* to determine which device has active control of the bus. Each device has a unique GPIB primary address between 0 and 30. The Controller defines the communication links, responds to devices that request service, sends GPIB commands, and passes/receives control of the bus. Controllers instruct Talkers to talk and to place data on the GPIB. You can address only one device at a time to talk. The Controller addresses the Listener to listen and to read data from the GPIB. You can address several devices to listen.
The GPIB is a digital, 24-conductor parallel bus. The GPIB uses an 8-bit parallel, byte-serial, asynchronous data transfer scheme, which means that whole bytes are sequentially handshaked across the bus at a speed that the slowest participant in the transfer determines. Because the unit of data on the GPIB is a byte, the messages transferred are frequently encoded as ASCII character strings. Refer to the GPIB documentation for more information about the hardware specifications of GPIB devices.
You can obtain faster data rates with HS488 devices and controllers. HS488 is an extension to GPIB that most NI controllers support.



NI DAQ



Input analogico		
Numero di canali	8 SE/4DI	16 SE/8 DI
Frequenza di campionamento	48 kS/s	1.25 MS/s
Risoluzione	14 bits	16 bits
Campionamento simultaneo	No	No
Intervallo massimo di tensione	-10..10 V	-10..10 V
Intervallo di accuratezza	138 mV	1920 µV
Intervallo minimo di tensione	-1..1 V	-100..100 mV
Intervallo di accuratezza	37.5 mV	52 µV
Output analogico		
Numero di canali	2	2
Velocità di aggiornamento	150 S/s	2.86 MS/s
Risoluzione	12 bits	16 bits
Intervallo massimo di tensione	0..5 V	-10..10 V
Intervallo di accuratezza	7 mV	2080 µV
Intervallo minimo di tensione	0..5 V	-5..5 V
Intervallo di accuratezza	7 mV	1045 µV
I/O digitale		
Numero di canali	12 DIO	24 DIO
Temporizzazione	Software	Hardware, Software
Livelli di logica	TTL	TTL
Intervallo input massimo	0..5 V	0..5 V
Intervallo output massimo	0..5 V	0..5 V
Contatori/Timer		
Numero di Contatori/Timer	1	2
Risoluzione	32 bits	32 bits
Frequenza di origine massima	5 MHz	80 MHz
Minima ampiezza di impulsi input	100 ns	12.5 ns
Livelli di logica	TTL	TTL
Intervallo massimo	0..5 V	0..5 V

National Instruments offers several hardware platforms for data acquisition. ***The most readily available platform is the desktop computer****. National Instruments offers PCI DAQ boards that plug into any desktop computer. In addition, NI makes DAQ modules for PXI/CompactPCI, a more rugged modular computer platform specifically for measurement and automation applications. For distributed measurements, the National Instruments Compact FieldPoint platform delivers modular I/O, embedded operation, and Ethernet communication. For portable or handheld measurements, National Instruments DAQ devices for USB and PCMCIA work with laptops or Windows Mobile PDAs. In addition, National Instruments has launched DAQ devices for PCI Express, the next generation PC I/O bus and for PXI Express, the high performance PXI bus.*

