**Instrumentation Idea Rapid Prototyping Framework**

**Documentation**

**Author: Carmine Paolino Rev. A 2017/6/19**

**Index**

[Preface 1](#_Toc485644962)

[1 Introduction 1](#_Toc485644963)

[2 Continuous acquisition framework 2](#_Toc485644964)

[3 System level description 3](#_Toc485644965)

[3.1 Notifier data structure 4](#_Toc485644966)

[3.2 Visual indicators management 4](#_Toc485644967)

[4 Settings Dialogue 5](#_Toc485644968)

[4.1 Operations 6](#_Toc485644969)

[4.2 DAQ Type value change event 7](#_Toc485644970)

[4.2.1 DMM7510 case 7](#_Toc485644971)

[4.2.2 PicoScope 5000 series case 9](#_Toc485644972)

[4.3 Other device-dependent value change events 9](#_Toc485644973)

[4.4 OK button value change event 10](#_Toc485644974)

[4.4.1 Buffer file size 11](#_Toc485644975)

[4.4.2 Channels settings multiplexing 11](#_Toc485644976)

[5 Acquisition Message Loop 12](#_Toc485644977)

[5.1 Loop structure 12](#_Toc485644978)

[5.2 Interfacing the DMM7510 13](#_Toc485644979)

[5.2.1 Determining SCPI commands 13](#_Toc485644980)

[5.2.2 Building a test VI 14](#_Toc485644981)

[5.2.3 Configure Hardware subVI 16](#_Toc485644982)

[5.2.4 Acquire subVI 16](#_Toc485644983)

[5.3 Interfacing the PicoTech PicoScope 2000a Series 17](#_Toc485644984)

[5.3.1 Calling C functions 17](#_Toc485644985)

[5.3.2 Accessing the memory buffer 18](#_Toc485644986)

[5.3.3Calibration 20](#_Toc485644987)

[5.4 Interfacing other devices 21](#_Toc485644988)

[Bibliography 22](#_Toc485644989)

# Preface

The following is an excerpt of the graduation thesis the Author discussed at Tongji University at the end of his internship. It only contains the part relevant to the Continuous Acquisition Framework and can be used to understand some of the design choices.

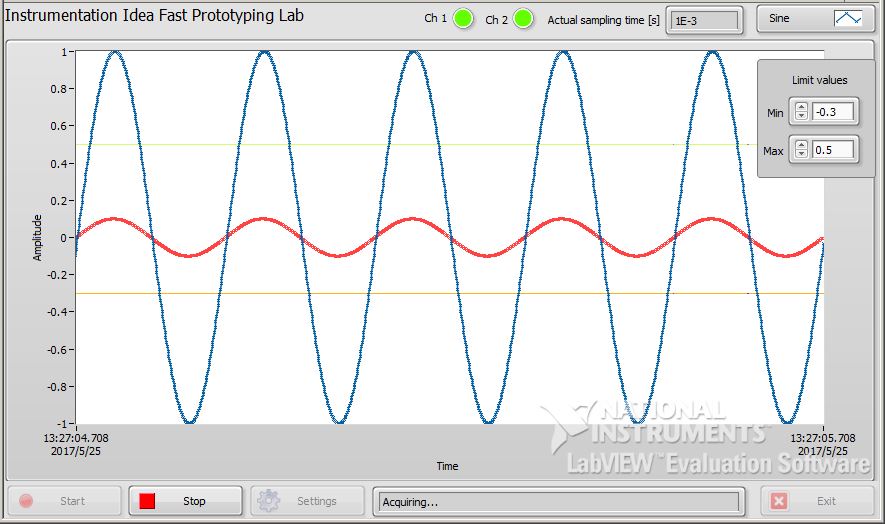
# 1 Introduction

The internship was conducted at China Innovation Center (CIC), a research, development and training center created in Shanghai by Thermo Fisher Scientific to support the high-growth Chinese markets. It focuses on biology, chemistry and engineering and plans to provide packaged solutions for more efficient product development.

Coherently with CIC goals, the main objective of the internship has been to develop a software framework capable of managing different acquisition devices, such as digital multimeters (DMM), oscilloscopes and digital acquisition boards (DAQ), while providing a common graphical interface to the user. The framework, described in LabVIEW, has to manage the most common acquisition parameters and provide the possibility to further elaborate samples by means of filtering, decimation, or even custom algorithms.

# 2 Continuous acquisition framework

The framework is forked from National Instruments’ Continuous Measurement and Logging project, therefore it retains its overall structure. The project was already developed with a focus on modularity, so its choice as an efficient platform to be customized for our needs was almost natural.

The focus of our work has been the management of the physical devices through proprietary drivers or standard interfaces (i.e. VISA). The result is shown below; it is the main window while capturing two sinusoidal signals from two simulated generators.

2.1 Main UI while acquiring two signals.

The two LEDs on top show both channels are active; the sampling time depends on the configuration of the generator and, in this case, is equal to 1 ms. Moreover, two horizontal lines allow for a quick visual check on whether the signal under acquisition is within a user-specified range. The message bar on the bottom tells the user the system is now acquiring samples.

In the following sections, we are going to discuss the implementation details and the challenges faced while adding functionalities to the framework.

Some familiarity with the LabVIEW environment is assumed, therefore no explanation will be provided on the most basic elements of LabVIEW programming.

# 3 System level description

At the highest level, five loops drive the acquisition:

* Event Handling: waits for changes on the main UI, like button presses or window closing; in response to these events, it enqueues a new command in the UI queue, so that action can be taken.
* UI Message: routes the actions requested by the previous loop to other specific blocks by operating on the Acquisition and Logging queues and prompts messages to the user in response to system events;
* Acquisition Message: initializes the communication to the device, configuring its parameters, fetching samples and releasing the resources in case of error or user intervention;
* Logging Message: stores, if required, the samples into a TDMS file for further elaboration, including metadata such as the name of the user, date, name given to the acquisition, as well as more specific parameters such as maximum and minimum values and total acquisition length in number of samples;
* Data Display: manages the graph in the main UI, as well as the other visual indicators and the threshold controls described in the previous section.

All these loops run in parallel, eventually waiting for a trigger before executing, like a new message in one queue or a system event. They in fact communicate through queues to pass commands or data around and through clusters containing configuration parameters.

Event Handling Loop

UI Message

Loop

Acquisition Message Loop

Logging Message Loop

Data Display Loop

Data queue

Data notifier

UI queue

Acquisition

and UI queue

Logging

and UI queue

3.1 Top level block diagram highlighting the communication channels.

## 3.1 Notifier data structure

It is interesting to look at the notifier data structure [1] used as the Data Display Loop input, comparing it to the queue that is used to log the same data. They are similar in the sense that both have operations that, before executing, wait until new data is available, but the notifier is a lossy mechanism, necessary to avoid UI delays affecting the rest of the system; when new information is produced by the Acquisition Message loop, any value currently on the channel is overwritten.

If a queue were to be used, the acquisition would have to wait until the Data Display loop had consumed the data. Since graphing the data involves many operating system procedures, avoiding possible stalls is extremely important.

Another feature of the notifier, not used here, is that it is by its nature a broadcast channel, meaning that any receiver listening would be able to see an incoming message; instead, in a queue, after the fastest block dequeues the element, no one else is able to see it, thus the channel is a unicast one.

## 3.2 Visual indicators management

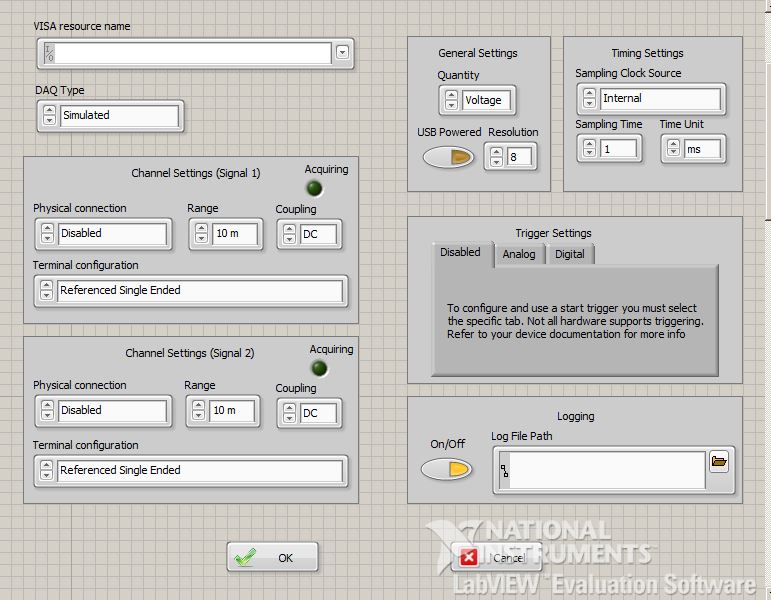
In the Data Display loop, three global variables manage the active channel LEDs and the sampling-time field; they are initialized at the beginning of the Acquisition Message loop and updated when the acquisition has started, meaning that they are turned on only after a successful initialization of the device.

In LabVIEW, global variables work as a subVI without any block diagram; their declaration is performed by inserting indicators with the proper name and type in the front panel of this VI. They are instantiated by dragging the subVI to a block diagram and selecting from a drop-down menu the specific control one wants to use.

The addition of the two thresholds uses an express VI provided by LabVIEW, which combines the original signal (or signals) with the new horizontal lines. It also allows to emphasize the samples outside the selected range by changing their colour, but in order to do this, it outputs a dynamic data type, which needs to be converted again to an array of waveforms in order to be displayed correctly; this conversion removes this highlighting functionality. Actually even a dynamic data type could be directly displayed, but since throughout the framework some of the chart properties are modified, changing its input type would require to update all these references to the new element. It could be done in a future release in order to take advantage of the visual aid provided by the highlighting functionality.

# 4 Settings Dialogue

When the Settings button gets pressed, the Event Handling loop enqueues the message “Launch settings dialog” into the UI queue. The UI Message loop then dequeues the message and opens the window up.

The resulting window is shown below; as previously said it allows the user to configure the most important acquisition parameters; not all the options a device may support are present, but still, this window offers a lot of flexibility.

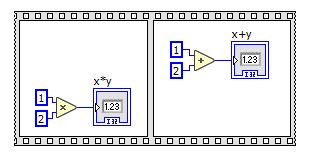
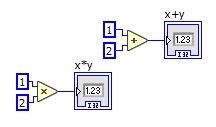
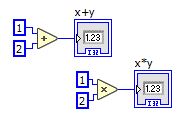
4.1 Settings UI

The settings are applied only if the OK button is pressed, in which case the “Broadcast Settings” message is enqueued in the UI queue and the associated action is to update both the Hardware and the Logging configurations.

The options provided are actually dynamic, meaning that changing one parameter modifies the choices available for the others. This requires more work on the programmer’s side to avoid the selection of unsupported device configurations; even if some of the devices may well be able to choose the most suitable parameters in case one option is not managed correctly, others can’t, therefore we decided to perform the same checks on every board for the sake of consistency.

The dynamic behavior is determined by a flat sequence structure, where the order of the operations is strictly enforced by the programmer. This structure is the only way to control the timing relationship between independent blocks, since their position on the block diagram doesn’t influence the data flow interpretation.

The following figure shows how a flat sequence works. The two boxes on the top row have the exact same behavior. The bottom structure forces the multiplication to be executed first. In this specific example there is no need to enforce any order, but, as we will see later, in some circumstances it is required

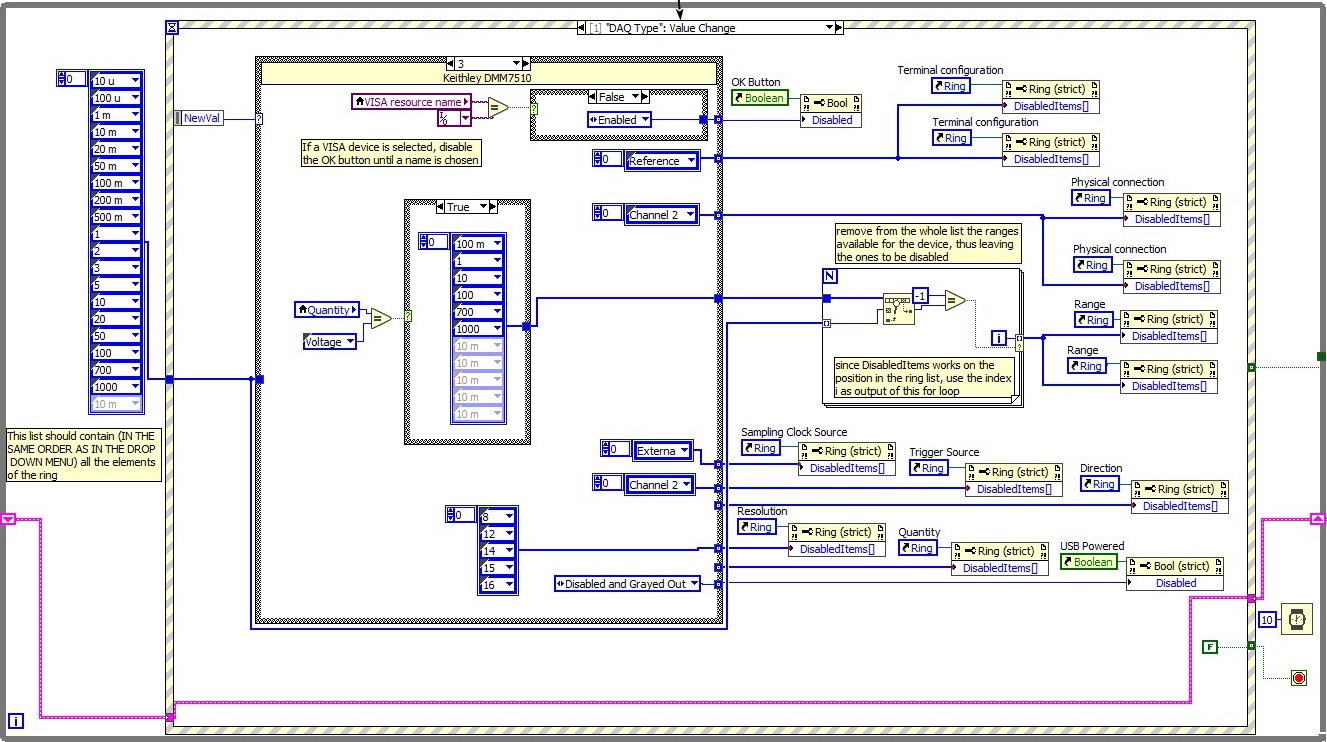


4.2 How to force relative execution time.

## 4.1 Operations

The first action (not shown in Figure 4.3), executed just after showing the window, is to update the value of the DAQ Type control to its previous one. The operation is required in order to trigger a Value Change event that will evaluate which options to activate at the very beginning, selecting only the ones actually supported by the device. The need for this comes from the fact that the last settings configuration is stored in a file so that it can be reused when restarting the framework; loading the DAQ Type does not trigger the dynamic selection of the possible options, thus leaving the possibility to ask the device for unsupported configurations.

The controls are then periodically monitored, every 10 ms (Figure 4.3). Changing one parameter on the Settings window may trigger some modifications in the other fields so that, for example, the available ranges selectable by the user are only the ones actually supported by the current device.

4.3 Settings flat sequence block diagram

Until the OK or Cancel buttons are pressed, the condition terminal on the bottom right of the picture is wired to a False constant, meaning that the loop will continue to iterate. Only when either of the buttons is pressed, a True value is passed on, so the window is closed and, eventually, the settings are propagated and stored on disk in order to get them back when the software is restarted.

Let’s now take an in-depth look at the DAQ Type Value Change event, since it’s the one with the largest number of consequences on the available options; later we will briefly describe the other events so that the entire behaviour of the VI will be unveiled.

## 4.2 DAQ Type value change event

The most recently selected DAQ Type value (NewVal in Figure 4.3) controls the case structure in the middle.

### 4.2.1 DMM7510 case

In the picture above we show the case (NewVal = 3) of Keithley’s DMM7510, a 7 1/2 digits multimeter capable of measuring different quantities on a single channel. Being a high resolution multimeter, its ADC is of the integrating type, therefore its performances are not directly expressed in bits, but depend on the aperture time of the converter; decreasing the acquisition speed, directly increases the resolution and noise rejection.

The way to configure the options available to be selected by the user, while disabling all the rest, is to create a reference to the control element used on the front panel and connect it to a property node. Many properties can be read or written at the same time by dragging the bottom horizontal line of the node symbol in order to make new options appear on the block diagram

Wiring a boolean to the Disabled property greys-out the entire control; the procedure is different if one wants to select which items in a ring control should be available. A ring control appears as a drop-down menu and is a numeric object that associates numeric values with strings; the user works on the strings, while internally it is the numeric value that matters, e.g. in comparisons or case selections. Disabling each single option can be performed by wiring to the DisabledItems[] property an array containing the values of the unwanted items. Before actually disabling them, the property enables every one so that only the last modification takes place.

Looking at the options the DMM7510 case structure has to configure, from the top of Figure 4.3 we see that, when the device is selected in the settings UI:

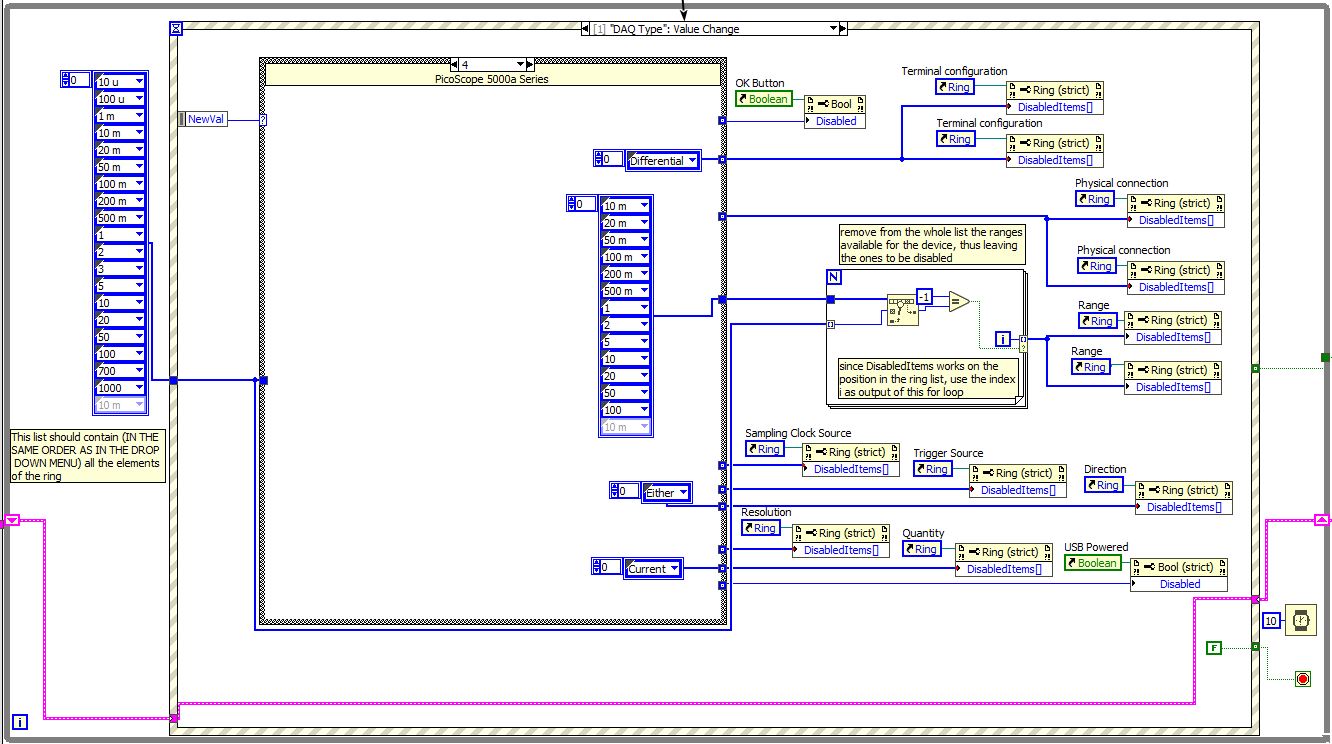
* the OK button, which closes the settings and applies the changes, is disabled until the user selects the VISA resource name, avoiding possible stalls in the following loops when trying to open an unspecified communication port; the comparison is done against a constant holding an empty name;
* the Referenced Single Ended option is disabled from the Terminal configuration; the only one remaining is the Differential one, which is what the device provides; even if this parameter is not used anywhere else and does not affect the behavior of other blocks, it has to be correctly set to be consistent with other devices that may have many available configurations;
* Channel 2 is disabled, since the DMM is a single channel device;
* Based on the physical quantity the user wants to measure (the DMM is the only supported device that can also measure a current), the available ranges are subtracted from the complete list on the far left of the picture in order to disable the remaining ones. A different approach would be to provide directly the items to be disabled, but this way, adding a new device, with its own supported ranges, all the other lists will have to be updated to remove the newly added ones. In our case, instead, adding a device only requires to select the supported ranges in its own case structure, making them effectively independent;
* The External Clock Source is disabled;
* Channel 2 is also removed as a possible trigger source;
* All the resolution options are disabled, since the DMM resolution is determined by the aperture time of the integrating ADC which, also depends on the selected sampling time;
* The USB Powered option is disabled.

In the previous picture, some of the Property Nodes seem to be duplicated. Looking back at the Settings UI of Figure 4.1, we see that for both Signal 1 and Signal 2 the options have the same name and that is why their block diagram symbols do, too. Double clicking on the block symbols highlights to which UI control they actually refer, so it is easy to check they are linked to different items.

Since disabling the right ranges is the trickiest operation in the block, we will further elaborate it.

A linear search is performed on the available ranges (which are device-dependent) for any single item in the complete ring (the long list on the left). The Search 1-D Array function returns the index of the element, if found, otherwise -1. In the latter case, a conditional, auto-indexed tunnel is enabled, so the iteration index is appended to a temporary array that, at the end, will contain the position of all the ranges the device does not support. These are then wired to the DisableItems[] property and determine the appearance of the control on the UI.

### 4.2.2 PicoScope 5000 series case

By comparing the DMM7510 case to the PicoScope 5000 series one (NewVal = 4), which is shown below, we notice that the selections are different, since this device supports different options. The disabled Terminal configuration is the Differential one, being the negative probe physically connected to GND; there are many more available ranges, the triggering options do not allow both Rising and Falling (Either) edges detection and there is no Current measurement capability.

4.4 DAQ Type value change event – DMM7510 case

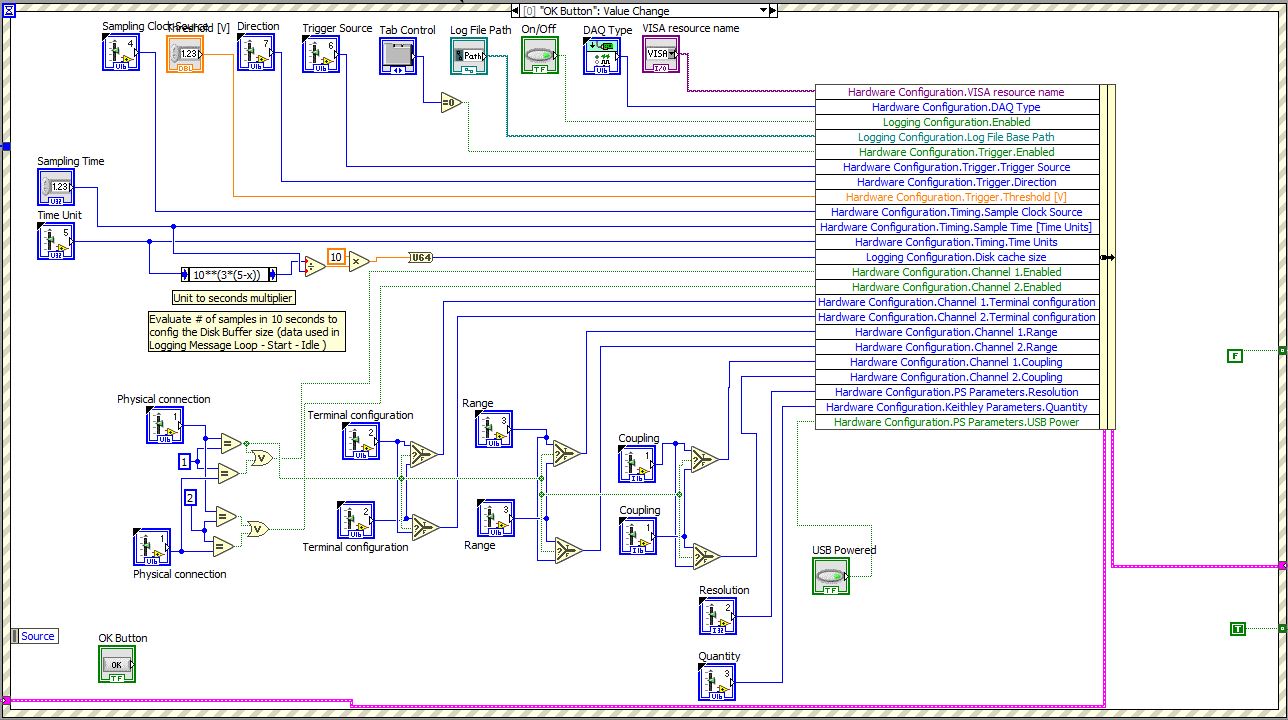
## 4.3 Other device-dependent value change events

All the previously described modifications take place when the device is changed, but some of them should be executed even if the device remains the same, for example, when the measured quantity is modified from voltage to current and different ranges should become available.

Therefore, other value change events are managed such as:

* Resolution: when the selected Resolution is equal to 16 bits (only the PicoScope 5000 series support this value) the available channels are reduced to 1.
* Physical Connection: in order to avoid the selection of the same channel for both signals, the Physical Connection control is monitored and, as an example, if Signal 1 gets associated to Channel 1 while Signal 2 was also linked to it, Signal 1 effectively is updated, but Signal 2 is switched to hold Signal 1 previous value, which may be either Disabled or Channel 2.
* Visa Resource Name: in the section above, we said we are disabling the OK button when there is no VISA port selected. This case, however, only manages the event that there is no selected name, and the DMM7510 device is chosen. To also address the case when the DMM is already chosen, and the VISA resource name is being deselected, its value is monitored and the OK button gets disabled.
* Quantity: as in the previous section, the case where the quantity is changed while the DMM is selected needs to be addressed, therefore the same block present in the DMM7510 case in the DAQ Type value change is inserted in its own event.

## 4.4 OK button value change event

Finally, the control event that copies all the new values onto the Hardware Resources Cluster is the OK Button Value Change event.

4.5 OK button value change event

Other than direct associations between UI control values and cluster items, some basic processing occurs here; specifically the size of the TDMS file buffer is evaluated and the Signal 1 and 2 parameters are eventually exchanged in order to be stored in terms of the selected channel.

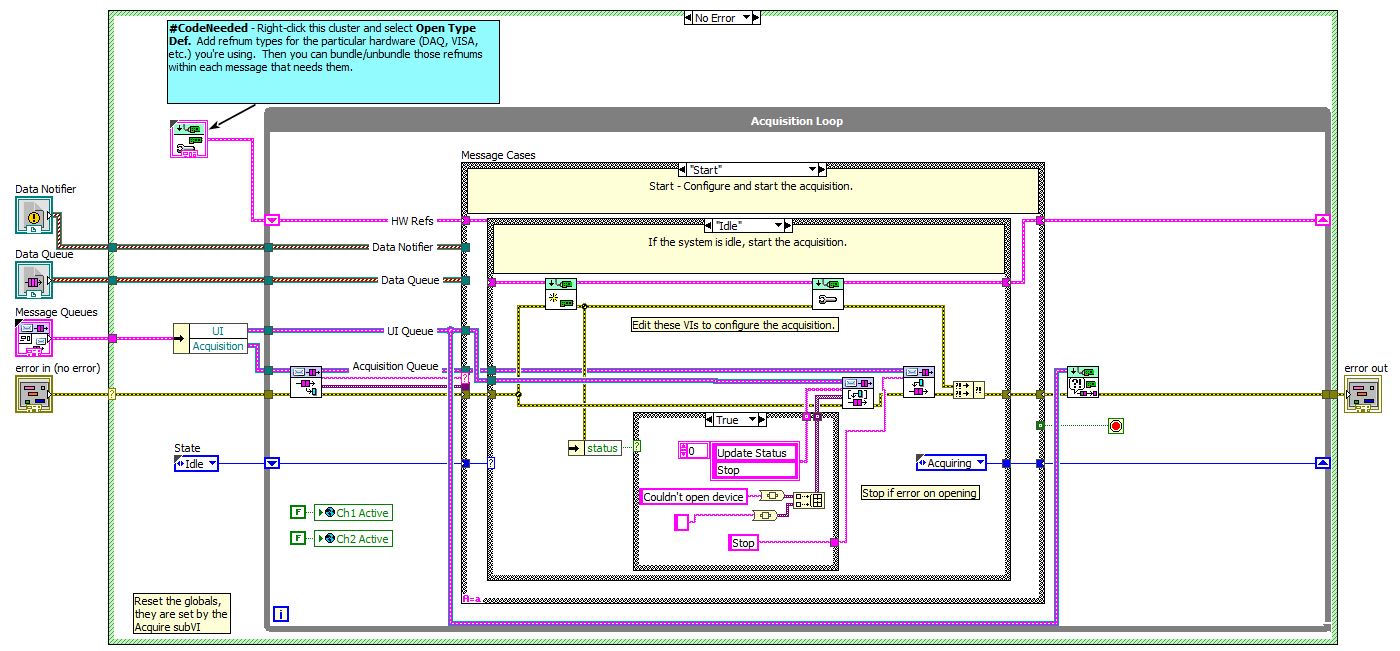
### 4.4.1 Buffer file size

Measurements are stored on disk in the TDMS file format, which was created by National Instruments specifically for acquisition applications. To avoid the overhead of writing to the hard drive too often, LabVIEW also manages a local buffer, stored in RAM, that is flushed to disk when full. However the default size is so big that hour-long acquisitions at a reasonable sampling rate may still not be able to fill it. In order to control how often data is flushed, the proper size is determined so that this happens every 10 seconds or so. This way the flushing doesn't cause too much overhead and still allows the user to work on the data while the acquisition is going on.

### 4.4.2 Channels settings multiplexing

Regarding the Physical connection control, since it allows to perform any mapping between Signals and Channels (see Figure 4.1), a cross-multiplexer structure is implemented, exchanging the values if needed. This way the cluster assignments are performed on the exact Channels and the rest of the software does not need to be aware of the mapping.

# 5 Acquisition Message Loop

It is the core of the framework, managing all the communications with the device. It is composed of a central case structure, driven by messages in the Acquisition queue and reports back to the main UI through the UI queue.

5.1 Acquisition Message loop structure

## 5.1 Loop structure

The main available states are:

* Start: opens the communication channel to the device, checking if the operation ended correctly and, in case, proceeding to configure the board with the settings chosen by the user; it is executed just once, right after hitting the Start button on the main UI.
* Acquire: continuously fetch data from the device memory, or trigger the device to write into a specified buffer in RAM.
* Stop: release the device driver.

Into each of these states, a nested case structure determines, based on the device selected by the user, which operations to execute. Therefore, the procedure to include a new device would be to define a new case in which device-specific actions take place, either by using its own driver functions, external VIs or standard LabVIEW functions.

In the following sections, we will describe what has been done in the case of the DMM7510 and the PicoScope 2000 Series, which exemplify two different approaches in how a device can be managed and also show some peculiarities of LabVIEW data-flow-based programming.

## 5.2 Interfacing the DMM7510

### 5.2.1 Determining SCPI commands

The device supports SCPI commands over the USB bus [2]. SCPI stands for Standards Commands for Programmable Instruments and defines a syntax and commands to be used with programmable test and measurement devices [3].

It was created on top of the IEEE 488.2 standard, which also defined a communication protocol to be used on the GPIB interface. The goal of SCPI was to provide a common way to express similar functionalities in different devices. However, there is no limit on the physical bus over which communication takes place, as long as the devices are capable of understanding the syntax.

The structure of the commands is tree-like, with a ":" separating submenu items, as shown below:

[SENSe[1]]

:DATA?

:FUNCtion "<name>"

:VOLTage

:DC

:RANGe

[:UPPer] <n>

:AUTO <Boolean> | ONCE

:REFerence <n>

:STATe <Boolean>

:ACQuire

To find out the correct sequence of commands configuring the DMM into a continuous acquisition state, after carefully reading the instruction manual, we used a Python module to rapidly test different options; it is called PyVISA and it provides its own shell interface, meaning that each command can be immediately executed and its behavior verified.

Below we show the listing of SCPI instructions that correctly configure the device, showing between <> user-definable parameters or variables.

TRAC:MAKE “RPFbuff” 10000000 // create user buffer

TRAC:FILL:MODE CONT “RPFbuff” // wrap around when full

TRIG:TIM1:COUN 0; // restart Timer1 indefinitely

TRIG:TIM1:DEL <samplingTime>; // define overflow period

TRIG:TIM1:STIM COMM // start on software trigger

TRIG:TIM1:STAT ON // enable timer trigger state

SENS1:VOLT:DC:RANG <manualRange>; // setup DC voltage range

SENS1:VOLT:DC:APER <max(0.24, samplingTime)>; // select resolution

TRIG:MEAS:STIM TIM1 // sample on Timer1 overflow

VOLT:DC:AZERo:STATe OFF // disable the autozero feature

\*TRG // send Software Trigger

The behavior of the device can be summarized as follows: after the \*TRG command, the timer will start, generating an overflow every <samplingTime> seconds; each overflow will trigger an acquisition, which will possibly last the entire samplingTime, to maximize the resolution and noise rejection performances; the sample will be placed in a buffer named “RPFbuff” and, upon reaching the last element of the array, it will start overwriting values from index 1 (wrapping).

The device only supports aperture times up to 0.24 seconds therefore, if the acquisition is too slow, the maximum value of 0.24 will be used.

At this point, the device starts acquiring and filling the buffer, so we need commands to find the index of the last sampled data and to retrieve the new values. Those commands should be executed periodically to avoid a buffer overrun.

TRAC:ACT:END? “RPFbuff” // get index of last element

TRACE:DATA? <previousIndex+1>, <currentIndex>, “RPFbuff”, REL

// fetch buffer slice

The result is the device returning a sequence of strings composed of the voltage value and its relative timestamp, as shown below.

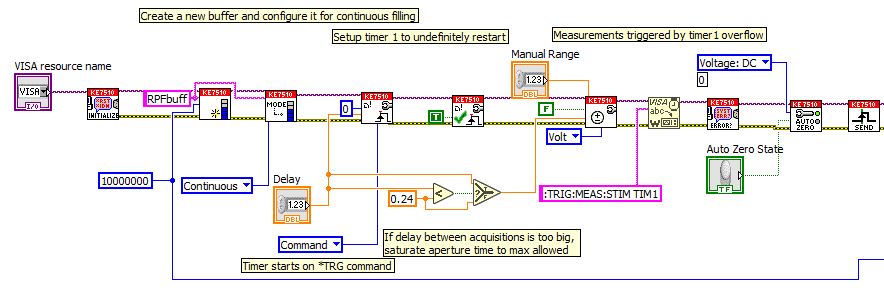
“<sample1\_str>,<time1\_str>,<sample2\_str>,<time2\_str>,…”

The TRAC:DATA? command requires the end index of the slice to be greater than the start. We have therefore to account for the case when the index wraps around the buffer, requiring two consecutive readings to be performed: one from the start of the slice up to the end of the buffer, the other from index 1 to the current index, at the end of the slice.

The REL parameter appends to each sample the relative time since the start of the acquisition in order to monitor the timing.

### 5.2.2 Building a test VI

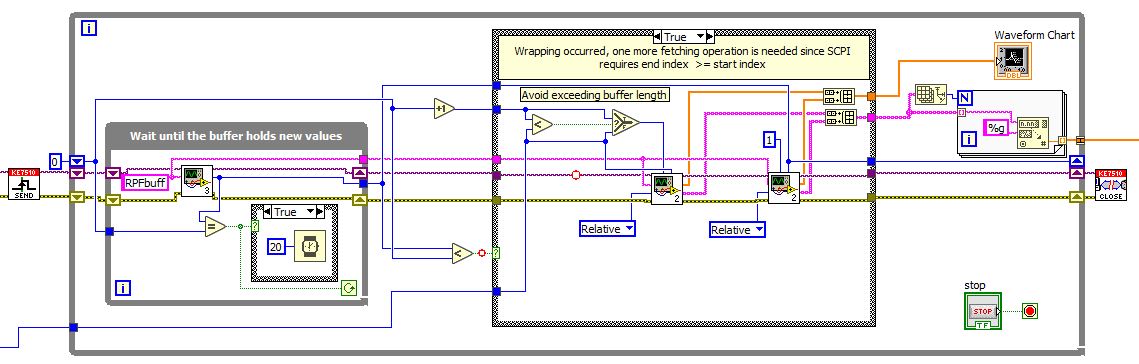
Having determined the actual instructions, a VI has been built, using the blocks already provided by the manufacturer. When necessary, they have been modified in order to correct some problems that were overlooked by the original creator.



5.2 DMM7510 test VI part 1- Configuration.

The internal implementation of the blocks in Figure 5.2 is basically a VISA Write function sending one of the commands previously shown, exactly as it is shown in the picture, fourth block from the right. Since there was no subVI that could configure the timer 1 as a trigger source, we had to manually send the right string.

Below is the fetching loop, continuously polling the current buffer index and, upon its change, retrieving the new samples.



5.3 DMM7510 test VI part 2 – Samples retrieval.

The instructions managing the index wrapping around the buffer, represented by the case structure in the middle of the picture, can be expressed by the following pseudocode statements.

if Current\_Index < Previous\_Index then

Fetch\_Buffer[Previous\_Index + 1:Current\_Index]

else

Fetch\_Buffer[Previous\_Index + 1:Buffer\_Size]

Fetch\_Buffer[1:Current\_Index]

end

where Fetch\_Buffer sends the

TRACE:DATA? <previousIndex+1>, <currentIndex>, “RPFbuff”, REL

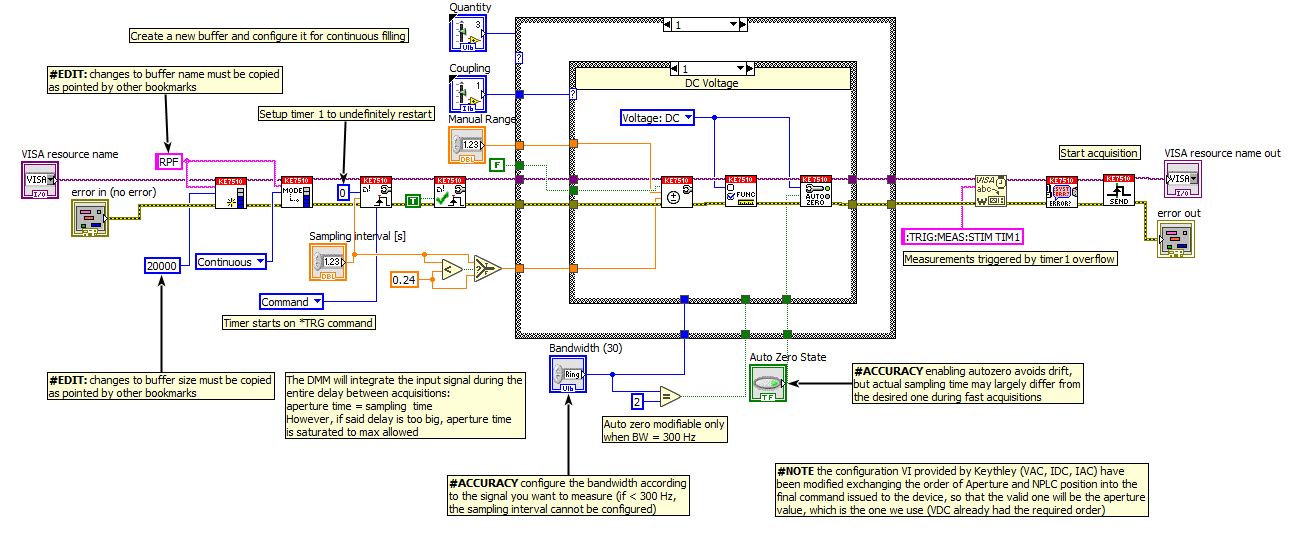
query to the device and reads the resulting string. The result is then parsed through the Spreadsheet String to Array function in order to get, from a string of the form

“<sample1\_str>,<time1\_str>,<sample2\_str>,<time2\_str>,…”

an interleaved array of samples and times, in double precision format. This array is further separated into two subarrays and each one of them is processed as required.

When the behaviour of the device was exactly as wanted, the VI elements have been separated and merged into the correct subVI of the framework.

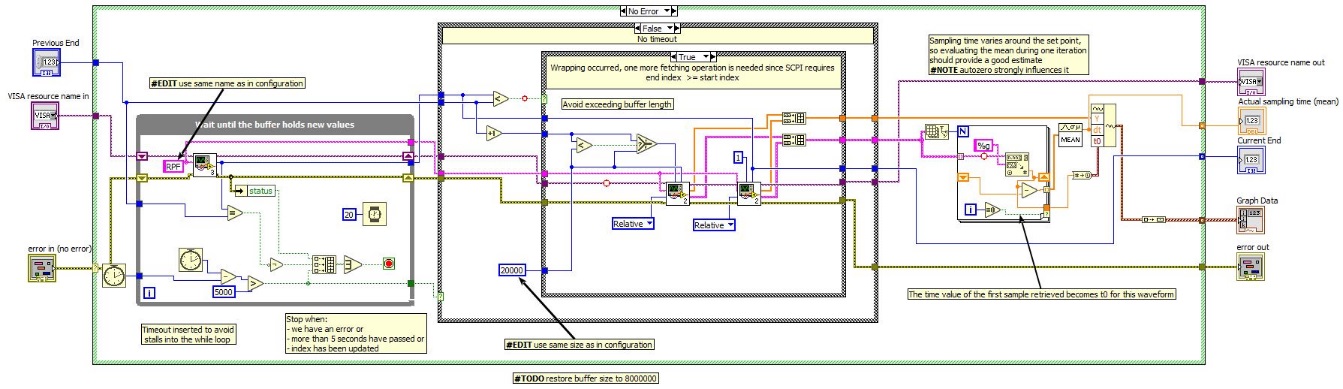
### 5.2.3 Configure Hardware subVI

With respect to the test VI previously discussed, the only difference here is the presence of two nested case structures managing the different quantity and coupling configurations; the former defines voltage or current measurements, while the latter determines AC or DC mode. In the picture we show the DC Voltage case.

5.4 DMM7510 framework configuration.

### 5.2.4 Acquire subVI

A better error management has been required in the actual acquisition loop as compared to the test VI. What has been done was to insert a timeout in the polling loop on the far left of the picture. It is executed until the device sends a new index and, at worst, it is terminated every 5 seconds so the software can proceed to the next iteration.

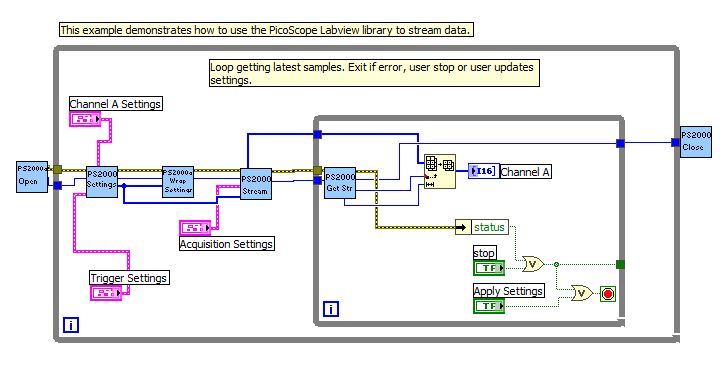
The termination condition of the polling loop now is the logical sum of an eventual error returned from the device, a change in the current buffer index from previous iteration or the timeout condition; those three signals are combined in an array in order to use the OR Array Elements function which, in a compact form, performs what we need.

5.5 DMM7510 framework samples retrieval.

Regarding the timing of the acquisition, we noticed some variability in the actual sampling intervals.

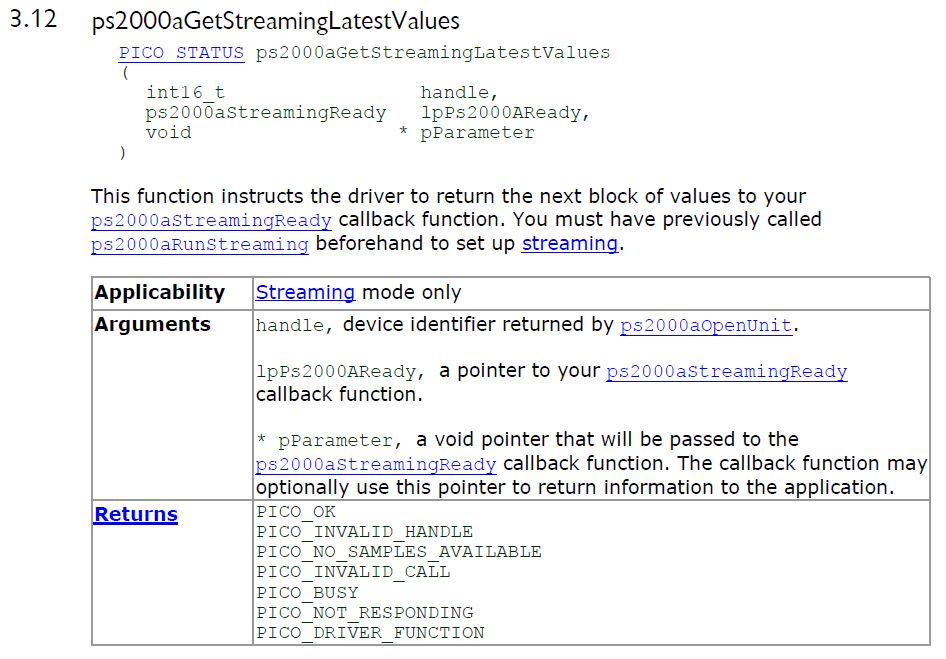
Unfortunately, the Waveform data type assumes a constant spacing between samples; therefore, from the array of relative time instants we are collecting, together with the samples, through the REL parameter, we evaluate the mean value for each batch of measurements. This cumulative quantity is then used as the dt parameter of the constructed waveform; t0 takes the value of the relative time of the first sample in the batch, thus having continuity between consecutive batches.

## 5.3 Interfacing the PicoTech PicoScope 2000a Series

This acquisition board comes with its own software, as well as ready-made LabVIEW examples, as shown below, that we exploited to speed up the integration process.

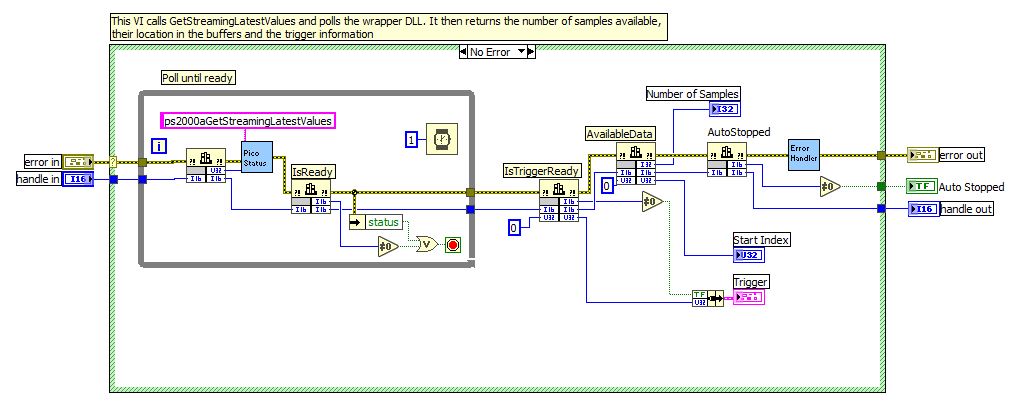
5.6 PicoScope continuous acquisition example VI.

### 5.3.1 Calling C functions

Differently from the case of the DMM, the VI uses C function calls to communicate to the device [4]. An excerpt from the API documentation describe how those functions are defined.

5.7 PicoScope API example

All of the functions require the device identifier (handle), which is an integer value, and return a status code, whose meaning is explained in a chapter of the API specification. Some input parameters may hold updated values upon return, so their pointer is passed in order to make the changes visible outside the function scope.

The way those functions can be used into a LabVIEW VI is through the so called Call Library Function Node, whose block symbol is a two-columns table (Figure 5.8 contains five of them). The left column is wired to input parameters, while the right one either to the return value (top cell) or to the parameters passed by referenced, which may be have been modified by the function.

5.8 Calling C funtions in a LabVIEW VI.

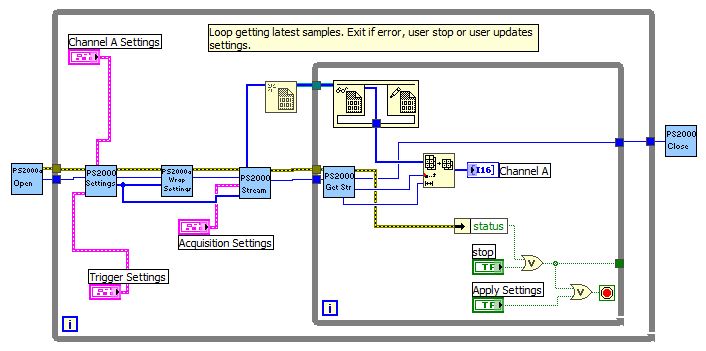
### 5.3.2 Accessing the memory buffer

Another significant difference with respect to the DMM7510 is that the board doesn't return the acquired samples directly to the software; instead the low-level driver triggers the device to write into a reserved region of RAM. The buffer is created in the StartStreaming subVI and is represented by a thick blue line leaving the block from the top and entering the inner loop (see Figure 5.6).

This works fine when the entire software fits into one VI, but in our case, having to separate the different blocks in order to include them into different parts of the framework, obtaining the correct behaviour was tricky.

The inner fetching loop will end-up in the Acquire subVI, while the configuration will be done by another block; the only way for them to exchange information is through the Hardware Resources Cluster. Therefore, the simplest method to make the memory buffer available to the rest of the software is to include the buffer as a cluster element and access it from there.

The reason why this solution does not work is that every assignment made to a cluster element creates a copy of the original values. As the buffer is created in memory, it is initialized to be filled with zeroes; when leaving the configuration subVI, we copy the buffer to the cluster element but what happens is that now we have a new array, in another part of the memory, which just holds zeroes. Triggering the board to store the samples in the original buffer works fine, since nothing has changed from the API perspective, but accessing the buffer from the cluster points us to the wrong section of memory. Therefore we will continue to read zeroes.

From a C developer's perspective, this is how C function parameters work, therefore the idea to tackle the problem is to find the LabVIEW equivalent of a “pass by reference”, where the information carried around is an identifier of the object (i.e. its address).

5.9 PicoScope continuous acquisition example VI modified using reference of the memory buffer.

In this new example, the data entering the loop is the reference to the buffer, graphically shown as an even thicker, blurred blue line, created by the New Data Value Reference function. An In Place Element is then used to get the array back in order to retrieve the samples from memory.

This structure is suitable to be used with the Hardware Resources Cluster, since a copy of the reference will still point to the right memory locations, but it introduces a new problem: any change to the input signal appears to the user after a certain delay.

An example can provide a clearer explanation of the phenomena. Using a signal generator, a sine wave was provided as input to the PicoScope board; the expected result would be to immediately see the waveform on the main UI, but what actually happened was that, for many seconds, the user would just get zeroes, then the waveform would appear. If the generator was turned off, the sine wave would persist on screen for the same amount of delay previously experienced.

Monitoring the actual buffer index returned by the getStreamingValues subVI, we have found that if one event happened at index x its result appeared on the main UI when the buffer index reached again position x, but after wrapping around the entire buffer length.

A workaround was to use a small buffer, so that the delay became negligible, but this limited the maximum acquisition speed to avoid a buffer overrun.

Looking at what happens in time, rather than in memory space, the definitive solution is easier to grasp. The two operations that have to happen to obtain the samples are:

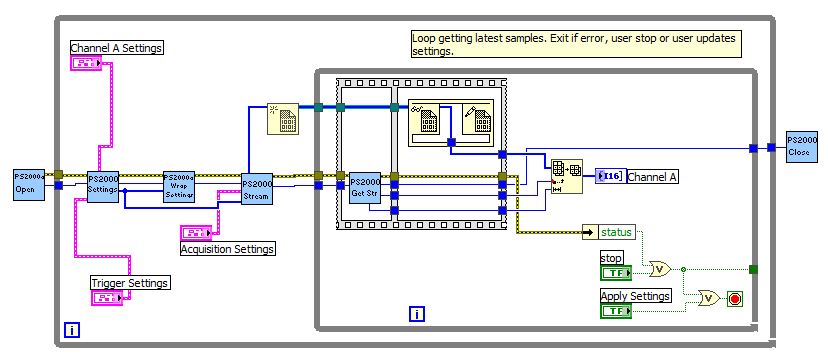
* trigger the board to copy the samples from its own buffer to the one in RAM memory and get the indexes of the first and last sample;
* retrieve the buffer slice from RAM.

The first operation is managed by the getStreamingLatestValues library function, while the last one is done by LabVIEW by dereferencing the buffer address from the Hardware Resources Cluster and extracting one slice from it.

What happens if the order of these operations is exchanged is that we firstly retrieve the buffer slice from the memory, getting the old data written there in a previous iteration and then trigger the board to update the buffer. We would then get the new values after an entire wrapping around the buffer, coming back to the same index.

This is exactly our situation, and the confirmation came by using the Highlight Execution option provided by LabVIEW when running a VI. This mode allows the developer to graphically see when data travels on a wire. Although the operations of the software are heavily affected by the activation of this mode, the relative execution in time of the blocks can easily be verified.

Since the execution time of independent blocks in a VI is not user-definable just by placing one more on the right than the other, as described in Chapter 4, the Flat Sequence structure is used to enforce a specific order.

The updated initial example therefore looks like this.

5.10 PicoScope continuous acquisition example VI modified using reference of the memory buffer and correct timing.

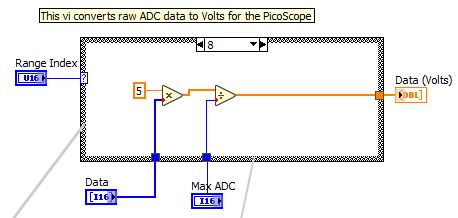
Now the various elements are able to communicate through a bus holding copies of the original values, as the Hardware Resources Cluster is, and the timing is correct, therefore the example can be split and included in the different subVIs of the framework.

As done for the DMM7510, new cases have been added, and no further modifications were required.

### 5.3.3Calibration

In order to get meaningful information out of a measurement, the most important prerequisite is to have calibrated devices. If the device is new, factory calibration may still be valid, unless extreme conditions have been encountered, such as mechanical, thermal or electrical stresses.

The PicoScope devices do not directly return the voltage values of the samples they collect, instead they provide the ADC codes. The conversion has to be done in a subsequent step and this gives access to the gain and offset constants. Therefore, by comparison to a calibrated device or by measuring some reference standard, one can verify, or eventually adjust, those constants.

In the framework, the ADCtoVolts subVI is basically a case structure, driven by the range index corresponding to the string selected in the settings, which performs the conversion.

5.11 PicoScope ADC to volts VI allows device calibration.

Once the actual gain and offset have been evaluated for the specific device, the calibration procedure can be completed by modifying the constants in the subVI, eventually adding the necessary compensation just after the division operation.

This procedure will not be available to the user, but anyone having the project sources or wanting to use the PicoScope SDK could be able to easily perform the required adjustments.

## 5.4 Interfacing other devices

Using the same approach, a couple more devices have been added, namely the PicoScope 5000 series and Arduino boards. The former requires the exact same procedure as its 2000 series little brother [5], while the latter uses an 8N1-mode serial interface, with an 8-bit payload, no parity and one stop bit.

# Bibliography

1. National Instruments. LabVIEW 2016 Help. http://zone.ni.com/reference/en-XX/help/371361N-01/
2. Keithley, DMM7510 Reference Manual. Rev. B – May 2015
3. Pieper J. M. SCPI and VISA a valuable combination. ACEA, The Netherlands.
4. Pico Technology Ltd. PicoScope 2000 Series (A API) Programmer’s Guide. Rev. 9 – 2016
5. Pico Technology Ltd. PicoScope 5000 Series (A API) Programmer’s Guide. Rev. 3 – 2016