We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900
Open access books available

185,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



21

The Importance of a Deep Knowledge of LabVIEW Environment and Techniques in Order to Develop Effective Applications

Riccardo de Asmundis
Istituto Nazionale di fisica nucleare, Section of Napoli,
Physics Departement University "Federico II",
Italy

1. Introduction

LabVIEW development system is spreading out in many applicative fields: industry, research fields, controls and monitoring designs; automatic measurement systems as well as machine controls or robotics; FPGA design associated with real time applications, artificial vision and enhanced controls in industrial processes. These circumstances create the need to prepare many people in the correct approach in the design, manipulation and management of both simple, complex and extensive LabVIEW Applications and Projects.
All these conditions imply the necessity of having good programmers involved in such a job: very often this statement is not at all reflected by the real working conditions with evident critical consequences on the resulting style and quality of the Applications delivered. You must be aware from the easy paradigm of the type "I understood, it's easy to do", without investing time and human resources to correctly study the development system before making Applications of a high level (Bishop 2009). Programming in LabVIEW must be considered as a serious job exactly like programming in any other language; expertise and professionalism play an essential role in the correct design in order to accomplish the final result of having a solution which is effective, robust, scalable in the future, reliable and pleasant to use or manage. If these conditions are unsatisfactory I would suggest either avoid programming in LabVIEW or delegating someone else with the correct expertise in making programming on it, or spend some time on your own in learning and becoming proficient in this knowledge process in order to obtain the best results for your Applications.

1.1 What the bad programmer loses?

A bad programmer is usually not conscious of being a bad programmer. Just like the mistakes made while learning a new human language, a bad programmer isn't aware of the mistakes he is making, either since he does not know about different solutions or he is not conscious of using a bad one, or he underestimates the consequences of an erroneous or poor design; these consequences are frequently "masked" by the exceptional quality of the G-Compiler, Memory Manager or other elements contained in the Run Time Engine, or the

machine performance and whatever is done behind the scenes in helping arrive at a final successful execution and good apparent behaviour: in other words, LabVIEW is strongly "robust against stupidity" and the programmer often doesn't see it. Please, try to avoid the principle of "whatever works is ok for me".

The most evident damages of bad programming can be visible in both the short and long terms: in the short term the programmer can undergo a possible personal stress in manipulating the generally extended front panels, with poorly identifiable objects onto it; complex block diagrams are usually related to them and have typical drawbacks: for instance they do not fit into the PC window of the screen and oblige the programmer to move the window up-down and left-right continuously, creating incredible difficulties in following the wire routing. Under these conditions it becomes hard to include new parts of code to extend functionalities, for both loss in space into the diagram and loss in technicalities which help in this process. As a result the solution is unsatisfactory and the programmer will dislike the development system and probably the language itself, or, worse, he will continue to design Applications in this way without taking advantage of the advanced tools and clean designing which are all possible in LabVIEW (Essic 2008) (Johnson G. W. 1994).

On the other hand, the heritage of such a badly written code by a different programmer is usually a shocking event whose solution passes several times through a complete redesigning of the Project. In the long term, developers like these do not encourage the use of the system at all: they tend not to choose LabVIEW because they finally don't know how the system really works and how to use it in terms of programming. They think it is not sufficiently clear, that the desired results cannot be easily reached, and sometimes that "LabVIEW is slow". All these bad opinions come uniquely from their own responsibility: the one of not knowing the system correctly and, sometimes, of simply having to accept the fact that they will have to spend sufficient time in learning all its features.

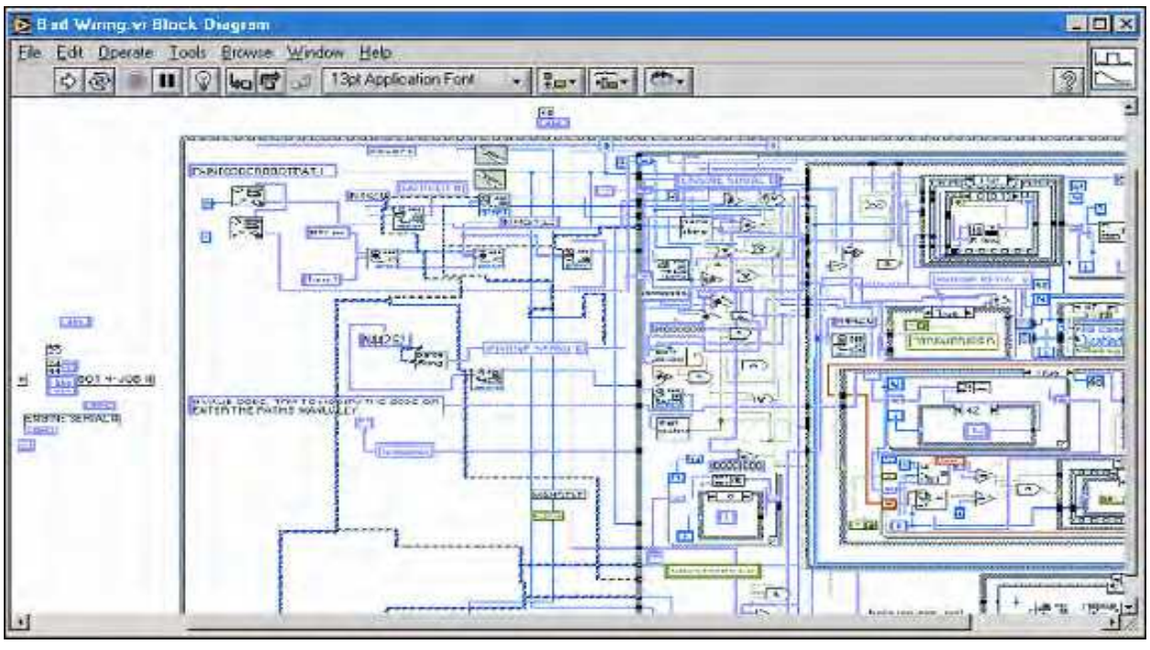


Fig. 1. Example of a typical result of a "badly designed" block diagram.

Lot of recommendations come even from official LabVIEW courses: if you look at the Figure 1 the example reported comes from one of the official LabVIEW Intermediate courses just as an example of what *should never happen* when programming in LabVIEW (National Instruments 2010).

From here I wish to start with my series of recommendations, which I will try to indicate in such a way in order to allow the reader to keep track of them and use them in the future as a reference.

> **Recommendation 1) Don't allow diagrams to go out of the development window sizes; if this cannot be avoided for grouping reasons ([1]) do it in a single direction, the best would be in the horizontal one.**
>
> Avoid diagrams whose development goes out of the screen. If your diagram is starting to have an aspect similar to the ones in Figure 1 (out of the screen, too many wires, too many superposition, confusion,…), stop your development and think about the right solutions: adopt the modularity model typical of LabVIEW by using SubVIs. Open a new VI and start to write your code into it. Then you will define the final character and finality of the SubVI you are developing in order to include it into the whole project (main VI) as a subpart. In other words *think modularly!*

1.2 The data dependency model

LabVIEW uses a "Data Dependency Model". This concept is strongly stressed into all main and basic courses like the LabVIEW Core 1 course. Self-teaching developers tends to underestimate this concept, even if they have seen it in the Core 1 self-course. If the developer does not adequately consider the data dependency he will experience several problems. One is to

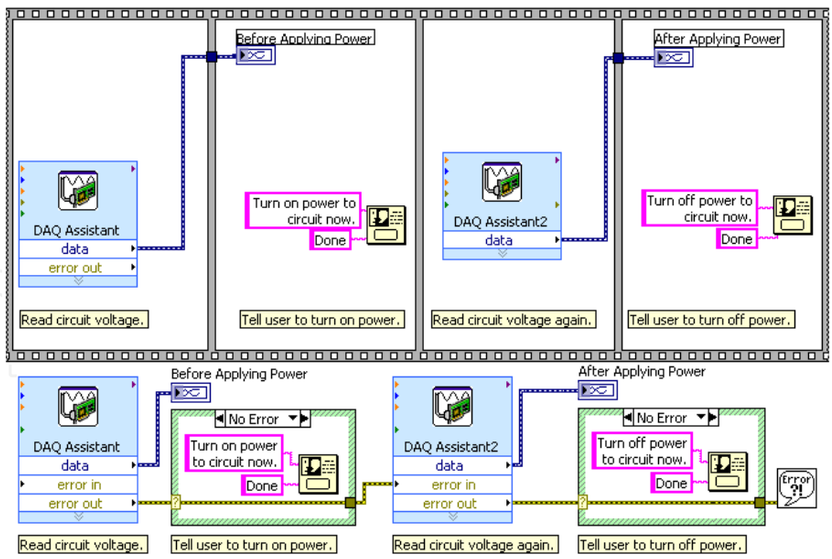


Fig. 2. Two different ways of designing sequential operations. (From LabVIEW 2009-Core 1 course manual, Lesson 8 "Using Sequential Programming").

[1] Like a SubVI which has to complete some operation in a whole (i.e. a complete communication with an instrument or a complete access to a file,…)

arrange the parts of his VI badly, which have to be executed in sequence: hyper-usage of Sequences Structures generally indicates a non-understanding of the data dependency issues, poor management of the available structures, a strong mental conditioning coming from classical text-based programming languages probably utilized in the past. And here we can introduce the next recommendation:

> **Recommendation 2) Avoid using the Sequence Structures.** Sequences are not considered in the official courses unless for marginal purposes: they overcome the data dependency paradigm which is the main point of force in LabVIEW. Make the effort in using data dependency instead. Mainly **use the error cluster** to create data dependency which forces the sequence of execution and, at the same time, takes care of produced errors.

Error cluster is a main topic in the LabVIEW environment. Their use is not only recommended but is vital for good programming structures and final reliability of the application. Lots of aspects have to be analysed concerning the Error treatments and a specific paragraph will be dedicated to it. In the Figure 2, the contrast between the two indicated solutions to create a sequence is shown. The first is "forced" by the flat sequence structure; the second is naturally invited to be sequential thanks to the data dependency: the error cluster makes the operations naturally in sequence and its usage allows it to act with the right messages to the user in case of error/no-error. Try to comprehend this example to amplify the concept of data dependency model: *LabVIEW executes nodes as data are available at its entrance.*

Finally the Sequence Structures have another drawback: several developers tend to affectionate to them avoiding using of most of the existing alternatives: State Machines, for examples, can be used in the place of many Sequences Structures, with the big advantage of letting the program flow to "choice" the step to be performed, which is, on the contrary, fixed for the Sequences.

1.3 The development environment: project explorer and the organization of the proper job

In my daily experience I see many programmers who don't have good organization of their work in terms of the storing resources in the PC: precise rules should be used for the locations in which the VIs related to a project or VIs coming from third parts (like Instruments Drivers) must be stored (Sumathi, Surekha 2007). Very frequently people can be confused regarding this issue, and this causes problems: the most frequent problem is losing control of what version of certain SubVI the developer is using: this aspect can lead to very serious problems, like the non-functioning of a VI (even the "main one") that worked the day before! Such a situation can happen if a version conflict is engaged due to different and old versions of VIs on the machine.

Here are some suggestions to consider for good organization for a correct outcome:

> Rule 1) Decide an "official" area on the machine in which a**ll LabVIEW Projects or Applications must be stored** and use that area at all times.
>
> a. **Do not use the Windows "Desktop"** to store work, or folders located on the desktop.
> b. Chose a **subfolder of the main "Document"** folder relative to the developer account, which must have Administrator's privileges.
> c. As an alternative to point b., use the **"LabVIEW Data" folder,** automatically created by the LabVIEW installation.

Rule 2) **Create a subfolder for every project** you develop: all material developed uniquely for that project must be stored in that folder.
Rule 3) **Subdivide the folder of the project into several subfolder**, each of them containing VIs, Controls, Menu, etc. for specific tasks: see figure 3 for an example.
Rule 4) **Use the project explorer:** the project explorer must reproduce the organization you are keeping in the file system. See figure 4 for the aspect of the project explorer related to the Project.

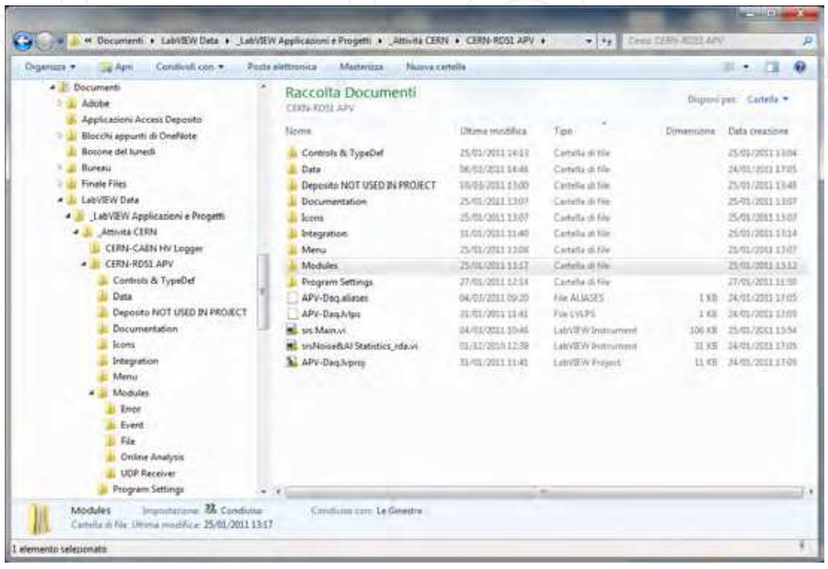


Fig. 3. An example of a folder containing a project. Remark the readability in such an organization created thanks to a subdivision in lower subfolder.

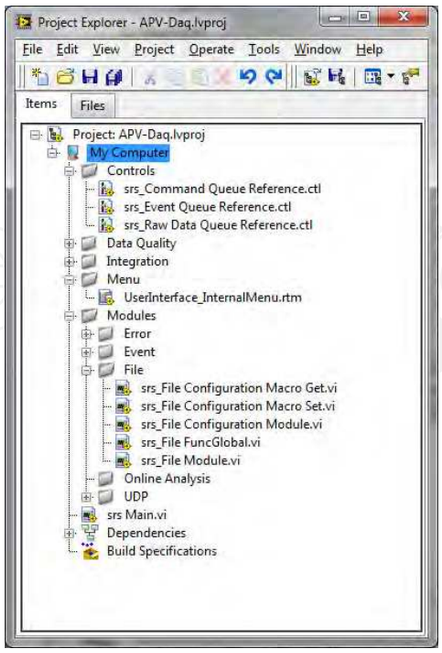


Fig. 4. Aspect of the Project Explorer window relative to the example above.

Project Explorer not only allows you to keep track of the organization of all files (VIs, data files, LabVIEW menus, other accompanying files for documentation too) but also gives several instruments not available in other ways: building a stand-alone Application is an example, as well as creating and managing Shared Variables.
Remember that you have at least two ways to create subfolders in the Project Explorer: the "Virtual Folder" and the "Auto populating Folder". The latter replicates what you do in the actual Windows directories, while with the former you need to explicitly manipulate them in the Project Explorer in order to organize the materials. The former is not always preferable: in my example I used Virtual Folders (i.e. "manual" organization) only.

Rule 5) Be careful if you save new versions of SubVIs with the same name but under different location. It's possible that, at a new reload of the Project, the system links an older version of these SubVI's, causing incomprehensive malfunctioning. If you need to change the location only of a sub-VI, **pay attention in removing the old version by deleting it** or, if useful for future references, move them to a storage area called "deposit" or similar, with a different name (i.e. "mySubVI_**OLD**.vi").

Rule 6) Last: if you need to use a third part software like Instruments Drivers, take care that the **whole driver folder is located in the official Instruments directory of LabVIEW,** like C:\Program Files\National Instruments\LabVIEW 2009\instr.lib, depending on the LabVIEW version you have. Official subdirectory must be "**instr**" folder. This allows LabVIEW to look at the existing drivers and to create the appropriate buttons in the Functions Palette window. Drivers stored there, are not erased when you uninstall the LabVIEW version in view of installing a new one: you just need to copy the whole driver folder to the new "instr" subfolder of the new installation (National Instruments 2010).

2. The correct design of front panels

Front Panels represent a critical job that is very often underestimated. It is true that, in case of a VI used as a quick, short and "fast" check, it is not worth spending a long time on the front panel: the problem is that from those poorly designed front panel, used in temporary VIs, we frequently evolve towards a large public utilization of the VI. The VI becomes from a short test, a "final user VI" and for this reason it must be correctly designed.
Poorly designed front panels usually suffer from the following drawbacks:

1. They are **larger than the screen size** and need to move window positioning bars, both horizontal and vertical.
2. They use lot of **"single" controls or indicators**, not separated among them well, creating confusion in the input and output objects.
3. They use **insupportable colours** on the objects or parts of the objects.
4. They use **inconsistent character sizes** and/or styles.
5. Controls and indicators **are labeled by their default names** (numeric 1, array 2,…).
6. They are not **user-friendly**.

In the Figure 5 you can see an example of a badly-designed front panel. Here a certain confusion exists in the destination of controls and indicators which are not organized well or logically and several character styles are used together (National Instruments 2010).

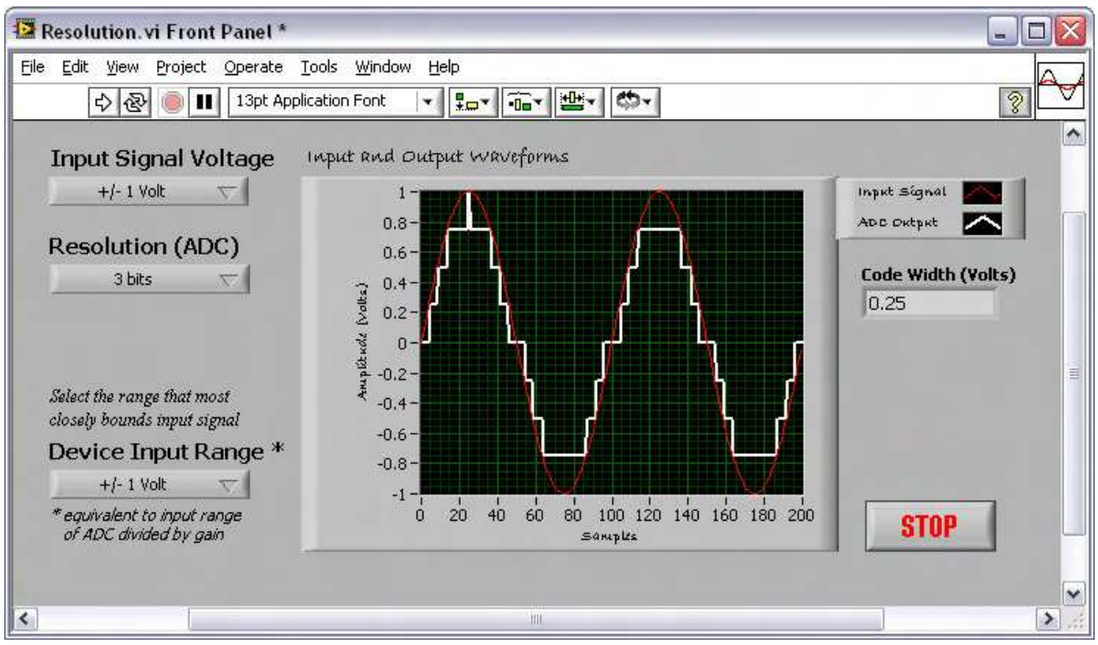


Fig. 5. A badly designed Front Panel. The use of different character styles, the non-clear assignment of controls and indicators cause confusion in the final user. (From National Instruments 2009-Core 1 Course presentation slides, Lesson 4 "implementing a VI").

Try to adopt certain rules in designing front panels. We can distinguish two categories of rules: the aesthetical ones and the functional ones. The problem is the two categories interfere each other. We try to resume in the following:

Aesthetical rules:

Rule 1) Avoid front panels that are bigger than the screen size. If more elements should be included use:

a. Tab Control to separate the elements in different logical context (i.e. all input/instrument settings, run control, output, error or alarm list, etc.).
b. Different windows as SubVIs which open for the running period only (i.e. small windows for setting, controlling etc.).

Rule 2) **Clearly organise controls and indicators for their correct purpose**: do not mix them in short space, but explicitly separate them for their functions. Regroup controls and indicators that are related to the same element (i.e. all controls of an instrument should be collected together in an unique portion of the screen or a specific tab of a Tab Control.

Rule 3) **Do not play with colours**: use default colours when possible. If an emphasis must be done use the light colours only and be uniform for entire parts (i.e. all controls of an instruments=light-blue).

Rule 4) Try to put yourself under the **final user point of view**; try to check the panel by thinking that the user has never seen it before. If necessary ask a colleague to check if the panel is easily understandable.

Functional rules:

Rule 5) Use specific labels for controls and indicators: **do not use default names** (numeric 1, 2, array 1, 2,…). Always give a representative name to the objects.

Rule 6) If labels became long, use a short one (mnemonic) and use **Caption instead on the front panel objects**. Establish your own rules for the name assignment for labels and/or captions and respect it. Example: Label CSM_BField; caption "Common Switch Magnet, Magnetic Field (Tesla)".

Rule 7) **Panels must respond quickly to any stimulus from the user** (coming from the mouse, TAB key,…) and must appear as animated (ex. a clock indicating day/time running).

The last point of the above list is strictly related to a correct design of the block diagram, but is very crucial for a good implementation. We will return on it when we discuss diagram issues. In the Figure 6 you can see an example of a well-designed Front Panel: note the absence not only of the displacement bars but also of all extra elements in the LabVIEW Menu on top of the window; such a solution contributes to the clearness of the VI and in avoiding unexpected behaviour at run time due to an improper use of the commands.

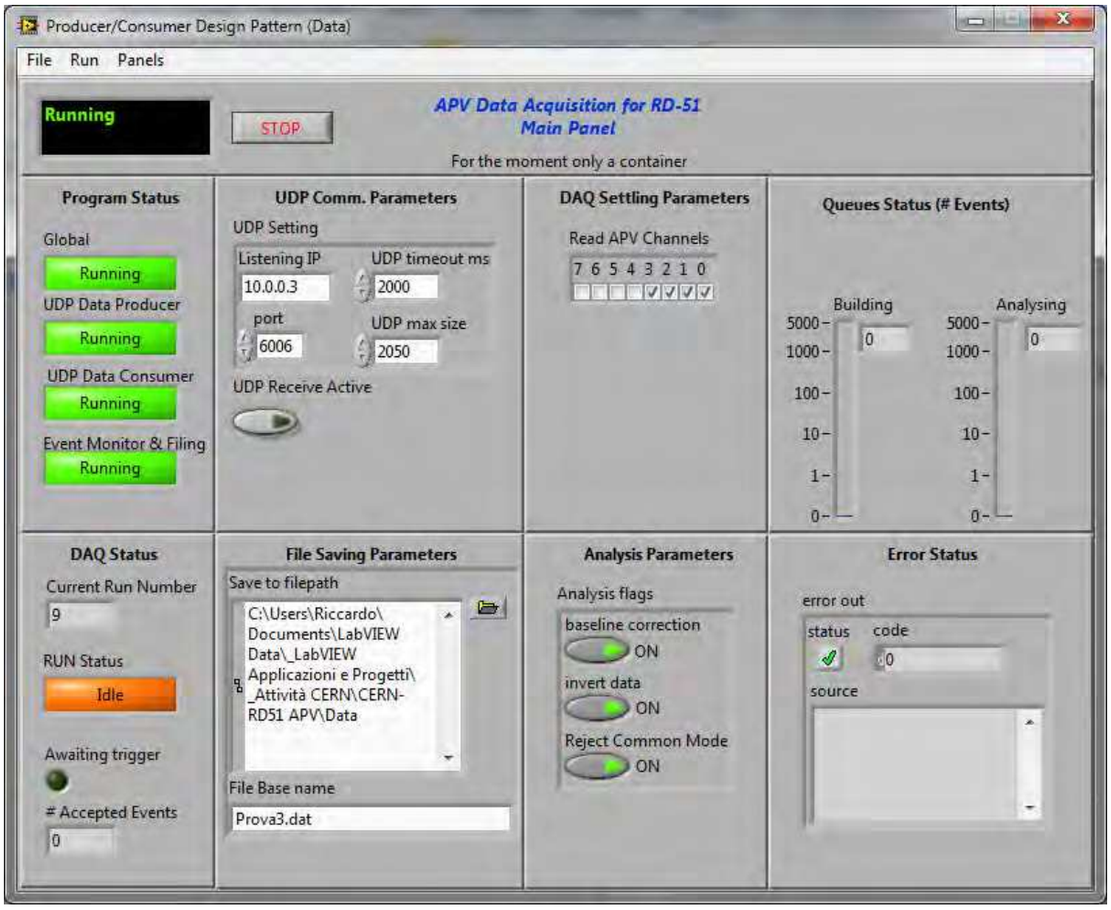


Fig. 6. An example of quite a good design for a Front Panel. Please note the subdivision of the panel into boxes clearly indicating the category or specificity of the elements included (from APV Data Acquisition System for CERN-Geneve, by R. de Asmundis).

In this case the VI presents only an overview of the execution status, indicating all the relevant elements involved: data are presented in separate panels. This solution has been adopted to further separate the logical aspect of the final Application.

In most cases, when a front panel must be revisited, **you may need a stop** to your development process, taking your time to decide how to reorganize it. Pressing in a fast design is the most dangerous aspect that can cause a bad design.

2.1 Custom menus

Usually all commands to be sent to a VI are implemented as graphical controls: buttons as Booleans, simple numeric controls or slides, dials etc. represent a natural "way of thinking" in LabVIEW. Nevertheless, if the number of implemented commands starts to be too high, basically on the main VI, different solutions are suggested. In this context **Custom Menus becomes a very useful alternative**. Custom Menus allow you to define your own menu items and organize them as you desire. Your custom menu is shown at run time on the VI window allowing the user to select from the menu voice the required function: the VI diagram has to handle the menu item voice by voice, by executing the related command, just as you would do by using Boolean buttons. Use event structure to handle, in an unique event case, all menu items. There are several useful examples concerning the menus which can be used as a starting point: this issue is quite a rich one and must be carefully studied. In the Figure 6 again, a custom menu is visible on top of the window, while in the figure 7, a possible programming solution for menu choice handling is shown.

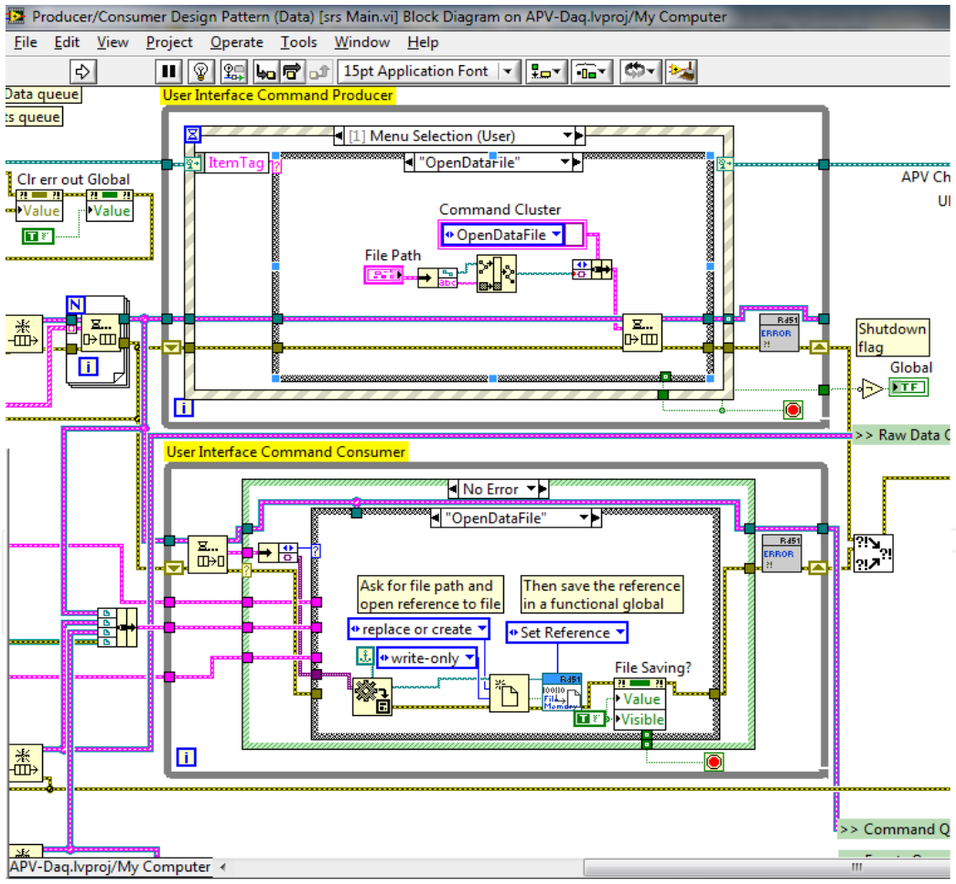


Fig. 7. Portion of the diagram which handles the Menu Item selection and acts in consequence. The bottom most loop handles the request coming from the top one (from APV Data Acquisition System for CERN-Geneve, by R. de Asmundis).

This handling is performed via a queue command exchange between the top and the bottom loop in Figure 7.

3. Elements of language: why to ignore them?

Several aspects related to the basic knowledge of LabVIEW language are, for some mysterious reason, often ignored by beginners. This is really a pity and demonstrates the weakness in which several people approach the language: they tend to underestimate the possibility offered and to overestimate their capability in developing with the few elements of language learned. We can individuate the following objects which are often misused, not well understood or completely ignored:

- **Relating data: Arrays and Clusters**. Correct use of them in several situations.
- **Shift register, a precious tool:** their "active" utilization instead of under staying to the needs of them.
- **Auto indexing of arrays into For** and (marginally) While Loops.
- Advanced data manipulation: Variant, bit manipulation, cast type, etc..
- The **References to objects** and their use in the Property Nodes setting actions.

A deep discussion on all of these topics would make this Chapter too long, so, for some of them, I can only make a citation.

3.1 Arrays

These elements are often misused. Cluster not used at all. Let me try to give some suggestions.

Arrays and Clusters are "musts" in LabVIEW environment: Arrays constitute a mandatory data structure for several situations, like data acquisition coming from samplings (at slow or high rates), oscilloscopes and data sampling equivalent acquisition cards, blocks of data to be processed and so on. Even weak programmers undergo the experience of using arrays because of their "natural" introduction by lots of drivers-VIs, subVIs or LabVIEW functions that only treat arrays. Nevertheless very often arrays are badly used since most of their mechanisms are not well understood. The most powerful of these mechanisms is the **auto indexing principle**: Arrays are, by default, **auto indexed when entering or leaving any** *For Loop* **structure,** while they are optionally auto indexed in *While Loops*. When using arrays, programmers must think from the "array point of view" and always considering auto indexing not only as an useful instrument, but also as the best way with which we treat arrays. Arrays tend naturally to be treated by auto indexing mechanisms (Travis 2004).

As an example consider the following problem: we want to generate a 10 by 10-elements matrix of random numbers where each row contains a random in the range $0\text{-}10^n$, where "n" is the row index starting from 0. Then we want to extract the main diagonal of this matrix into a second output array. In other words, each row of the 2D-matrix, will contain random numbers in the range 0-1, 0-10, 0-100,…$0\text{-}10^9$. In the following figure the two different solutions are reported:

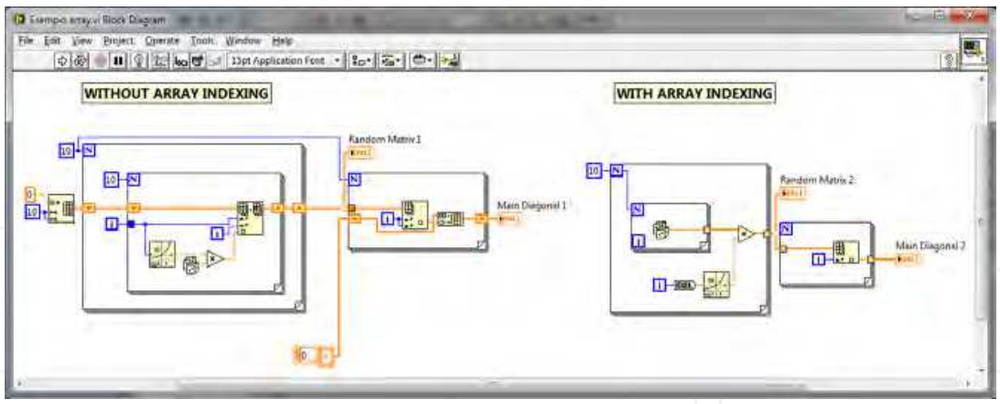


Fig. 8. Contrast between two different solutions in the use of indexing of arrays. Remark the elegance of the second (the right) solution.

In the leftmost solution auto-indexing of array is not used and more functions are involved, the structure is more complicated and the readability of the diagram lower. In particular, the second loop generates an array starting from an empty one by using a build array function in concatenate mode: this solution, although quite practical, is memory and time consuming for the machine and should be adopted only if strictly necessary.
The solution on the right uses auto-indexing **which is implicit in the For Loop structures**: the compactness and the elegance of the solution, in particular in the extraction of the main diagonal, is evident.
So, definitively:

- **Auto-indexing** should be used as frequently as possible when manipulating Arrays.
- Sometimes auto indexing **can be useful with While Loops** too (example to record the history of some parameters once the loop has terminated): be careful in this case, because While Loops can run for extremely extended time and accumulation of array on the border is greatly memory consuming in those conditions. For this reason auto indexing in While Loops is disabled by default.

We cannot spend very long on arrays since there are a lot of other issues to be considered in the Chapter. Anyway, make the effort to think and plan better and better when working with the arrays, even by investing some time in understanding related functionalities.

3.2 Clusters

Clusters are basically ignored by most beginning programmers. This is a paradox since the cluster is the **equivalent in LabVIEW to the "Structures" in C or C++ languages, or Records in Pascal**. Roughly speaking, ignoring clusters is exactly like ignoring one of the most important data structure of programming languages: it is like completely cutting out an entire vital existence of LabVIEW. People who do so often make incorrect sentences like "data structures in LabVIEW are complicated and too many wires and connections are necessary to move data". Reasons for this lack of knowledge always comes from the approximated way of learning the language, without spending any time in the pure study of it, but going directly to "try to develop" something: it becomes natural that in such cases, people use what they have learned until that time and tend not to go on in their acquisition of knowledge.
So, stop for a while and **study Clusters. Then use them as frequently as possible** to represent data which is **logically related one another**. Clusters are extremely useful, for

instance, for grouping setting parameters for an instrument, variables coming from the same source that must be displayed together, entire groups of data to be presented like atmospheric parameters in an environmental control, and so on. Moreover many internal structures in LabVIEW are clusters, like data handled by X-Y graphs, so you must know them. Clusters allow:

1. The **collection of several variables** (In or Out) in an unique container; by choosing to "clusterize" variables which are logically coherent, you have a natural way of logic separation of data into different categories.
2. **Wiring becomes simpler**: very complex data can be exchanged using a single wire.
3. Clusters can be **assembled or disassembled by name**: this introduces an intrinsic clearness into the diagram, by citing the name of the variable transported.
4. Uniform clusters (i.e. all numbers, all characters) can be **sent directly to mathematical (or character) operations** thanks to the polymorphic nature of the functions, so it is not necessary to unbundle and re-bundle them.
5. Clusters allow **improvement in clearness of front panels**. A Front Panel organized in clusters is clear, compact, logically subdivided in a natural way and simpler to maintain.

Consider, as an example, the front panel showed in Figure 9 it is quite well designed, but **it does not use Clusters**. Besides the hard work to obtain it (all boxes around Controls are put by hand), the real problem is into the diagram (Figure 10) where values coming from different controls are picked up in an apparently spread out way. Clusters would help a lot in grouping together the data and make the diagram more friendly, as you can see in the bottom part of Figure 10, where two Clusters have only been introduced. Notice the *Unbundle by Name* function which greatly increases visibility of the diagram. You can imagine the advantage in using Clusters for **all Controls and Indicators** in this diagram.

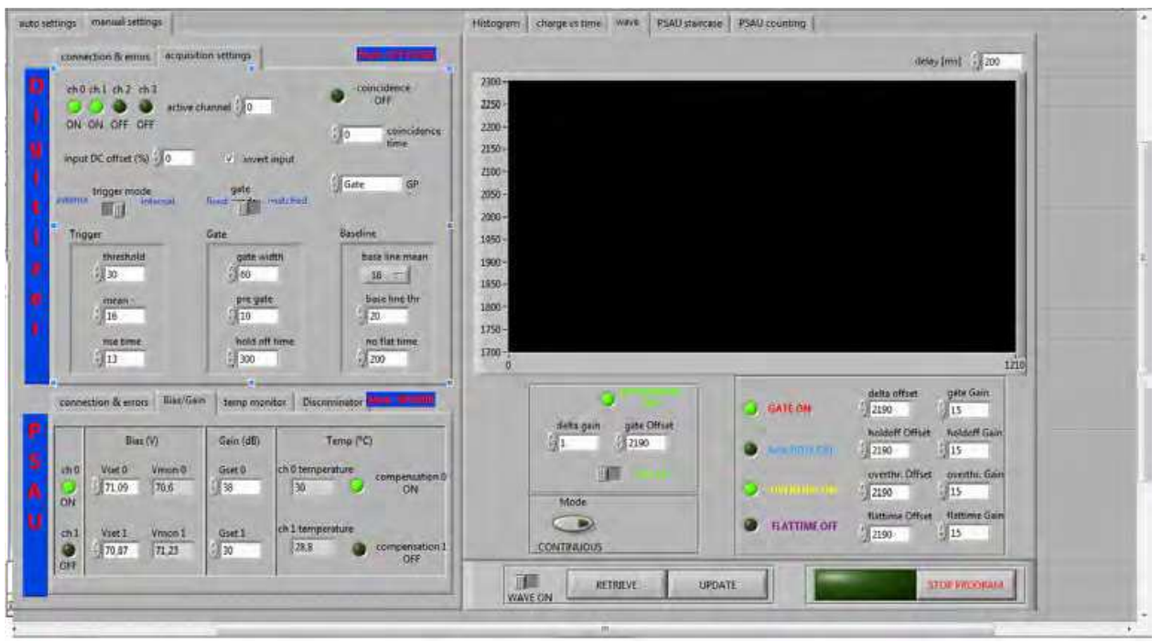


Fig. 9. Example of a front panel which do not use Clusters: even if it appears quite good, a lot of time has been spent to obtain this result. Boxes around groups of Controls or Indicators have been put in by hand: Clusters would provide a natural way of doing this.

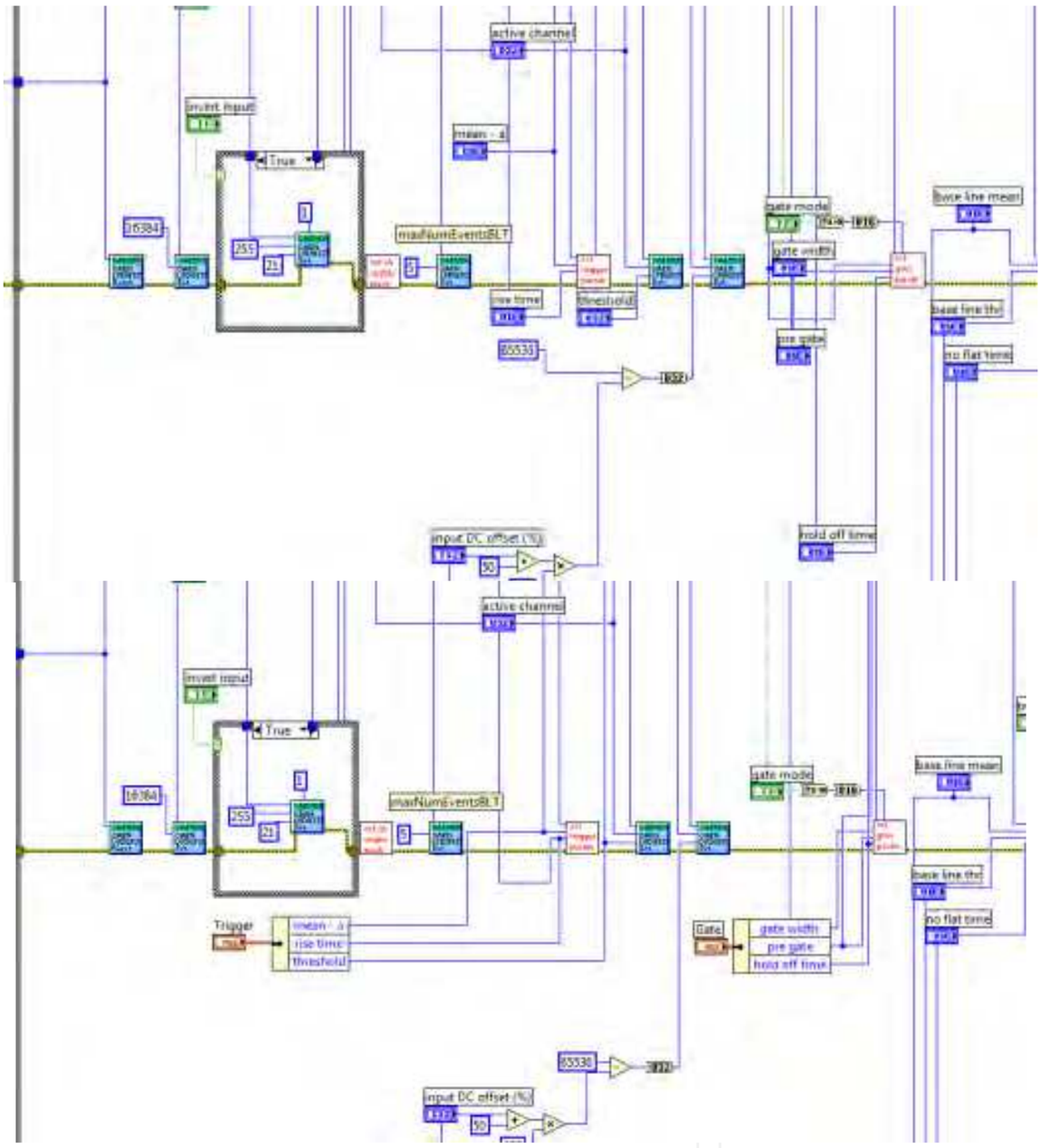


Fig. 10. On top a portion of the diagram relative to the Figure 9. In the bottom the same portion in which only two clusters, each containing three elements, have been introduced. You have to imagine the final result when the clusterization of data is completed.

3.3 Type definitions

Clusters, but not only them, are often involved in **Type Definitions**. If Clusters are "seen but ignored", type definition are often **not known at all**. Again this is a paradox, since they are the equivalent, in C/C++, of "Typedef" statement. In particular C-statement "typedef struct" **is translated in LabVIEW as** "Type definition made using Clusters". For this reasons, Clusters and Type Definitions are often used together.

Use type definition to **create your own data type for your Applications.** Include type definitions into your project using a specific folder. Once you create a Type Definition, you can use **instances** of it wherever you want in all VIs of your Project. Instances are **bound** to the original Type Definition, so that, if you modify it, all instances are automatically updated.
Type definitions are made **by using the Control Editor** and this may be the reason for which they are so ignored: editing of Controls are, in fact, considered as a "far" item, not useful in the daily practice and to be used rarely. Although it is true that we do not modify controls to create our own controls every day, Type Definitions can be manipulated in this environment only.
It can be convenient to try with the following suggestions:
Suggestion 1) If you think a control/indicator, even a simple one like a small cluster expressly prepared, is subject to be used several time, then **define it as a Type Definition.**
Suggestion 2) To do so, simply select your control, go to "Edit → Customize Control" menu item. The Control Editor window will open. Select "Type Definition" or "Strictly typed Type Definition" as the type of control you are editing. Customize it if necessary and save it as a new, separate type. When you close the Editing window, answer YES to the question asking if old control must be bound with Type Definition.

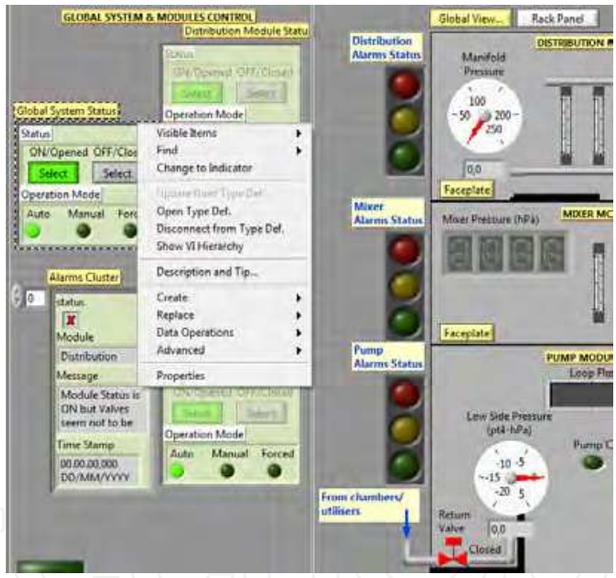


Fig. 11. Example of a Control Panel which uses several Type Definition. Remark the contextual menu with relative voices like "Open Type Def" or "Disconnect from Type Def". If the pointed Cluster is modified in the Control Editor as Type Def, all instances are automatically updated (from a Real Time Gas Control System implemented in the INFN Naples, R. de Asmundis)

In the example of Figure 11 several Type Definitions are present. On one of them the contextual menu is opened. All Type Definitions are Clusters, and this is not by chance: if a cluster must be repeated in several points of the VI because it has several utilization, it is convenient to define it as Type Definition.

- Remember that if you need to change anything in your Type Definition, **just open it and make your modifications**: all instances will be automatically updated.

One of the most useful aspects of adopting Type Definition is in the case of advanced Design Patterns into diagrams. We will return to this in the relative paragraph.
Remember: it is not convenient to ignore Type Definitions.

3.4 Local and global variables. Using shift registers as Global Variables

Shift Registers represent a very powerful tool generally used to report, in repetitive structures (For or While Loops) data coming from previous cycles. We do not enter in the details because shift registers are so useful that they are surely known even by beginners. Almost every repetitive structure uses shift registers because of the needs of the process itself.
Don't forget that Shift Register **are actually a memory storage** for any type of data. Consequently we can use Shift Register as a memory storage to pass data among VIs or part of a VI. This solution, strongly recommended by National Instruments knowledge base, is called "**Functional Global Variable**".
The reader surely knows Local and Global Variables: they are an useful way of accessing to Controls and Indicators in different parts of a diagram or among SubVIs. Excessive use of Variables is, indeed, to be strongly avoided. I have seen several beginners using local variables in the place of drawing wires to transmit data within the diagram. Well, this is **a very incorrect way of programming** for several reasons:

1. You lose the main principle of LabVIEW language, again the **Data Dependency** paradigm. LabVIEW performs operation once data is available at the input of functions. Variables alter this paradigm, by inducing an immediate **availability of data, but with no guaranteed it will be up to date.** This leads to 2.
2. **Race conditions** can be introduced: a function (far) uses data (picked up from a Variable) that has not been updated yet, because, in another point of the diagram, the Variable is still waiting to be updated., In this way we *planned* to send correct data to the target VIs or portion of diagram, but this is not the case.
3. Race condition **must be avoided** since they represent big "bugs" in your program. Diagrams can run correctly even thousands of times and fail at a thousand and one!

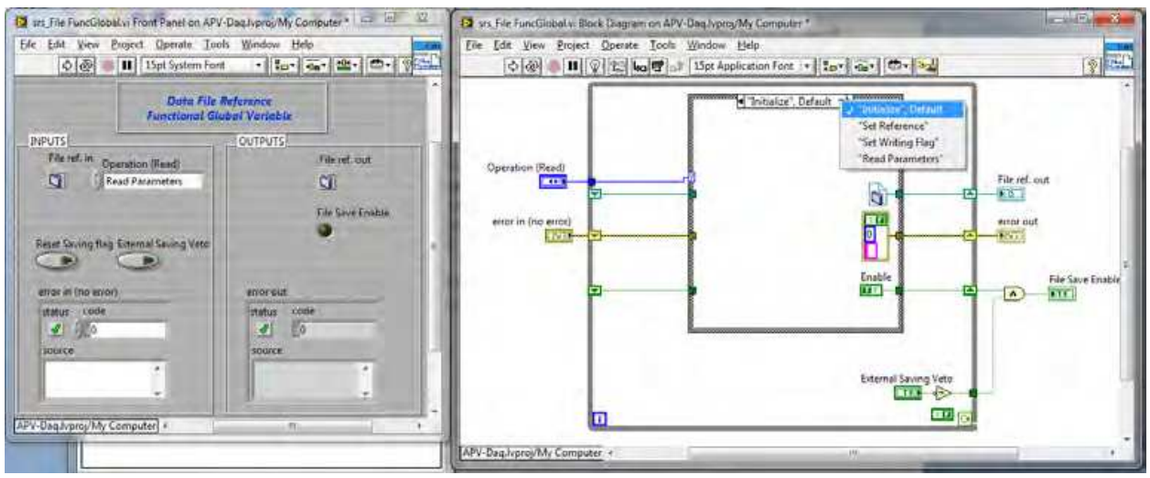


Fig. 12. Example of a Functional Global Variable (FGV). To be noted, the different choices with which the FGV can be called: "initialize" to clear the output, "Set Reference" to save values to be successively used, then "Read Parameters" to use them.

Please consider Figure 12. the front panel and diagram of a Functional Global variable which uses shift registers to store data is represented. The problem solved here is the following: in a professional application, the User can decide when saving *streams* of data onto a file and when not to. Where to open and handle the file in such a process? Usually data streaming consists in opening the file before a loop, writing continuously within the loop and closing it after (see paragraph on the "files" in this Chapter). If the diagram is quite complex and this sequence for file handling is not convenient to be designed in the standard way, the *opening and closing* of the file only can be done in the command section of the diagram: there a **Functional Global Variable is used** to store the **file reference** and, if needed, other information too. Then, in the part of the diagram which writes the file, the **same functional global** is used to read the file reference number and whatever is needed. You can, in this way, store and retrieve information at different locations in a diagram without using a variable. Since Functional Globals are SubVI the data dependency paradigm tend to be correctly applied, automatically avoiding race conditions.

4. References and variants

References and Variants represent two other data type which is often ignored in LabVIEW. We encountered **References** in the above paragraph, talking about files. References do not apply to files only: basically each object can be referenced, and actions can be taken on the objects by using references. References are *numbers* which we do not decide, create or define: just **use.** References are created by right click on terminals in the diagram and from there on, this number can be used for several purposes:



Fig. 13. The "Reference" palette menu.

Reference to a file, as above, to write, read and do any other special operation on file; reference to an object on the front panel can be used to access it and programmatically modify **any attribute it has**. It is difficult here to collect all situations where References can be used or are useful. Just as an example, consider using the **property nodes** to change property programmatically (i.e. changing the colour of an indicator depending on data, make a control active or inactive greyed-out and so on). Property Nodes need a considerable space in the diagram and it can be useful to move them in a SubVI. In that case **References** to the modifying objects must be sent to the SubVI: it will be received in the SubVI as a "Control Reference". Figure 14 shows a cluster of reference created to send compactly references to SubVIs.

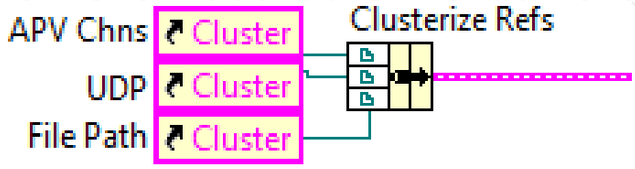


Fig. 14. A Cluster of Reference is created in order to transmit compactly References to specific SubVI.

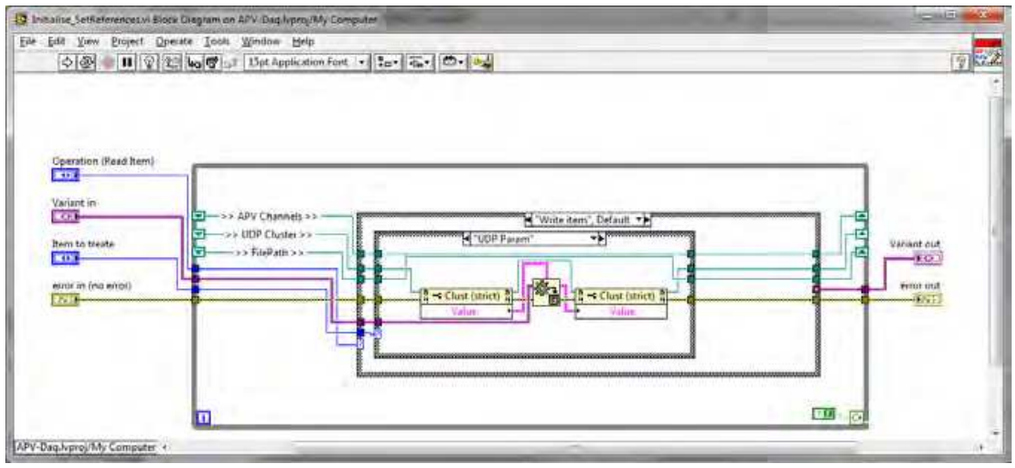


Fig. 15. The subVI in which these References are used.

Figure 15 shows the diagram of a SubVI which uses these References: note the Property Node which accesses to the object and read/write data on it, remotely (i.e. in a SubVI) and again without using Global Variables. Property nodes act in a better way on objects with respect to Variables, and allow Error treatment.
In the same figure also a **Variant** is used. Variant can be very practical, and it is another item defined in most programming languages. It simply contains "any" kind of data. You can use Variants when you need to send various-shaped data using a single "channel". An example is the handling of a user menu choice: depending on the selected menu item, you need to send different data types to a consumer into the diagram. You simply send a Variant at **every menu item,** so the structure to send data is always the same: the receiver must interpret the Variant by knowing what data type is associated with it in order to reconvert. Variants are a very flexible way of data exchange for several situations. It is useful to spend some time in testing and learning them in view of an advantageous utilization starting from available built-in examples on Variants.

5. The hierarchy

LabVIEW is conceived as a **hierarchical environment**: different tasks or jobs can be performed in different subprograms, namely different SubVIs. It often happens that programs are written without using SubVI, writing everything in an unique diagram, namely the main VI. This is not a good choice. In this way diagrams tend to quickly become too big, going over the screen size and incomprehensive; it becomes difficult to manage, to be updated and is not as scalable as it should be. In my opinion there are two main reasons for this tendency:

1. There is insufficient familiarity in creating SubVIs at the beginning, so their usage is not taken into account.
2. Developers do not "think hierarchically", meaning they have not figured it out, before writing the code, the best way of organizing the diagram and basically on how to **subdivide tasks.**

Regarding the first point, it is sufficient a simple tour in creating some SubVIs, be careful where to store them or giving the correct names to SubVIs, correctly choose the data in/out and the terminal connection and so on.

As for the second point, don't forget that SubVIs are the equivalent of Functions and Subroutines in text-based programming languages (C, C++, Pascal,…). When learning C, for instance, one of the first points stressed is the use of Functions: C and C++ that are **made by Functions**. Pascal pushed this concept to the extreme: Functions and Subroutines had to be declared in the main program. All these aspects force the developer to "think hierarchically": when a C developer has a specific task that the main program has to do, a specialized one or a task which has to be repeated several times, he/she immediately starts to define a new **C Function.** It must be the same in LabVIEW: **opening a new VI** and making it a SubVI should be a natural process.

In the Figure 16 we can have an idea of what a disastrous diagram can come out when not constructed with the principle of a good planned hierarchy.

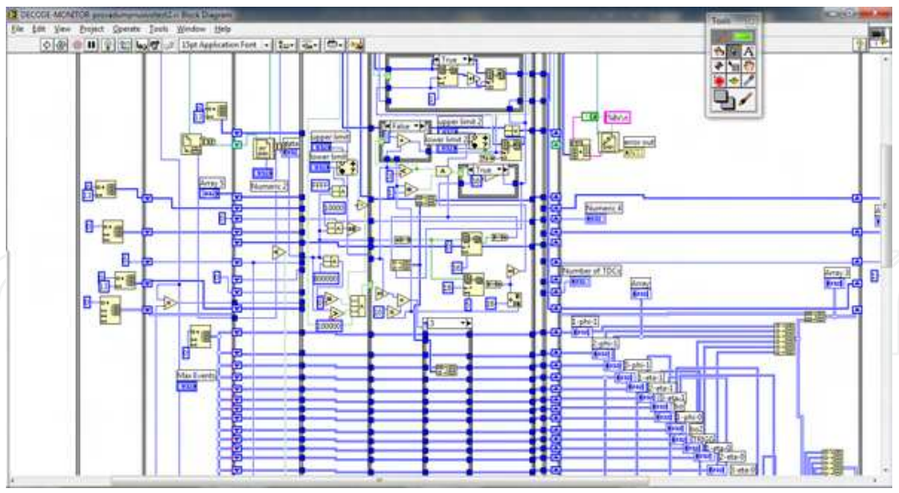


Fig. 16. An example of a very badly designed diagram in which no use of SubVIs has been adopted.

Situations like the diagram of Figure 16 lead to serious difficulties in understanding and possibly rescaling the program (i.e. the diagram) for future developments. It can be impossible to understand it even after reopening just two months later: the confusion can be total and the robustness is strongly compromised.

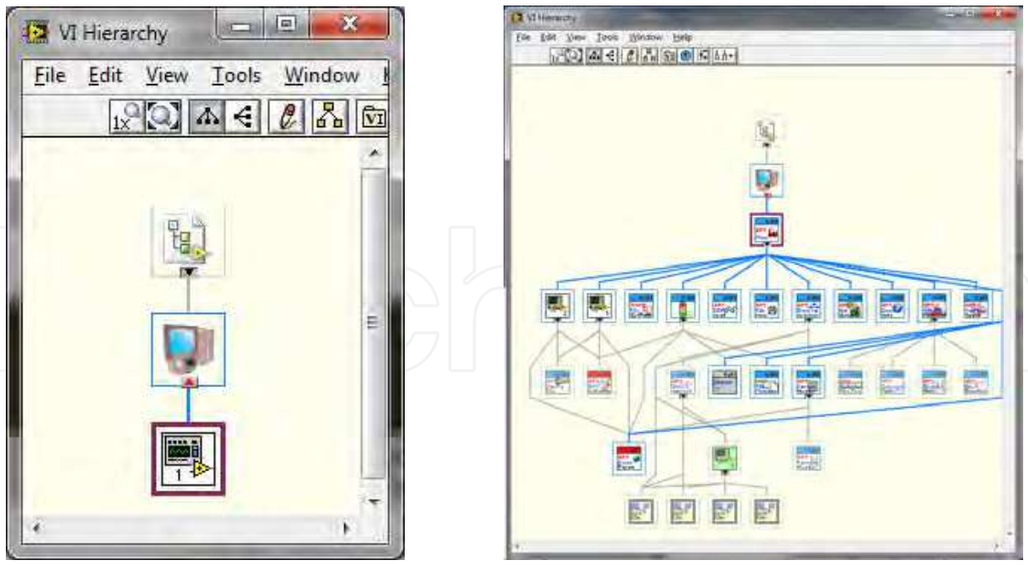


Fig. 17. Left: the "Hierarchy Window" of the above diagram has no meaning due to the poor hierarchical design. On the right a Hierarchy Window for a correct designed Application which adequately uses SubVIs.

The **Hierarchy Window** helps a lot in keeping the control of the development process. It would be useful to refer to it during hierarchy definition. Starting from the **first SubVI** you introduce into your application, open the Hierarchy Window to have a look at the modularity you are adopting in creating your Application. SubVIs can be easily accessed to open their front panel, to set VI properties, to edit their icons.
Here are some possible hints about Hierarchical developing and SubVIs in general:
Suggestion 1) Whenever you have a "sub-problem" in the sense that a certain task can be seen as independent, **open a new VI and use it as a SubVI.** You can develop robust, clear, independent VI that can be used as SubVI in several points of your Application.
Suggestion 2) SubVIs allow you also to create parts of your program that can be tested and debugged independently from the whole Application: once your SubVIs work fine you are saved from one of the most frequent source of bugs in your Application.
Suggestion 3) When your main diagram starts getting too big, individuate a part that can be performed by SubVs: then select that part and make **"edit→ create SubVI" command from menu.** Then re-organize your SubVI (terminals, Icon, etc.).
Suggestion 4) When you open a new VI to be used as a SubVI, **immediately drag the icon terminal** pane in the main VI: this immediately imposes the desired hierarchy, and the SubVI is visible in the Hierarchy Window. You organize your hierarchy even before finish developing of your SubVIs.
Suggestion 5) The last hint is somewhat more complicated: think **what you don't want to see** into the diagram of the Main VI of your Application and what you do see. For example **decide what not to have in the main VI** as, for instance, low level functions: file open, write/read, close; access to instruments drivers; TCP/UDP access; etc. Decide that **all low level functions** of any kind **must be written in SubVI.** This solution allows you to create a **"Layers" Hierarchy** in which you can have different visibility of your work: it is a great help for big Projects and Applications.
Obviously most of these hints must be checked and a lot of experience must be done to absorb them. But please, start to adopt these strategies immediately.

5.1 Documenting VIs

VI documentation is another missing point: basically all developers, also experienced ones, tend to skip this step completely. Diagrams are poor in explanations, notes or internal documentation. This is another mistake. VIs without any documentation are hard to understand and if you mix this with the general weakness in the diagrams clarity the result is hard to understand.

LabVIEW is a self-documenting environment and you do not need a lot of work to create a good documentation. It is sufficient to keep in mind some standard steps to adopt during the development processes to get an automatically documented code. Documentation issues include:

- A **correct name** given to the VIs.
- A **correct allocation** of VIs in your file system (as already described).
- A correct **naming of Controls and Indicators** on the Front Panels.
- Designing of **a good Front Panel**, in order to self-guide the user in understanding it. This should be a good practice for SubVIs too, even **if they are not normally open or visible to the final user**.
- Good aspect of **VIs Diagrams**.
- An adequate choice and design of the **VIs Icons**.
- A **correct positioning of input and output** terminals on the SubVIs Connection Panes.
- Some words in the "documentation" tab of the **VI Properties page for all VIs involved**.

Most of the above issues are clear to understand. For others, further clarification may be useful. As you can see, basically, all these rules do not imply extra-work to your job. They must become habitual. When you open a new VI, for instance, the first thing to be done **is saving it in the right folder and giving it a good name** which is indicative of its attitudes.

The same thing when you introduce any Control or Indicator: the name label is automatically selected to allow you to input the specific name: please, do it. Do not leave "numeric 1, numeric2, array1"… names.

For VI icons you can have suggestions by the same Figure 17 right side, in which two icons only must still be designed. There is a logic in the choice of the icons:

1. **An icon Template** has been adopted: it is a simple framework design (two rectangles in light blue in Figure 17) with a short explanation as a short name of the Project or the activity for which the Project has been made. Create it and save it as an Icon Template (from version 2009 on).
2. One specific **Template should be used for each Project**, giving a signature to the project.
3. The background colour is modified into **two-three options** (Red or Pink) to locate different purposes of SubVIs: normal VIs are in the original colour (blue); Functional Global variable are in Red and Type Definitions are in Pink.
4. At **every icon of the SubVI in the Application** three lines of text, one glyph and the cited framework identify the purpose of the SubVI itself.
5. Icons that are different in styles **come from the LabVIEW system** (as internal functions made by SubVIs) or from my personal "User Library": obviously they keep their old icon design.

In the two figures 18 and 19 you can see the front panel and diagram of a SubVI **that is normally closed**, but optionally opened and shown to the User.

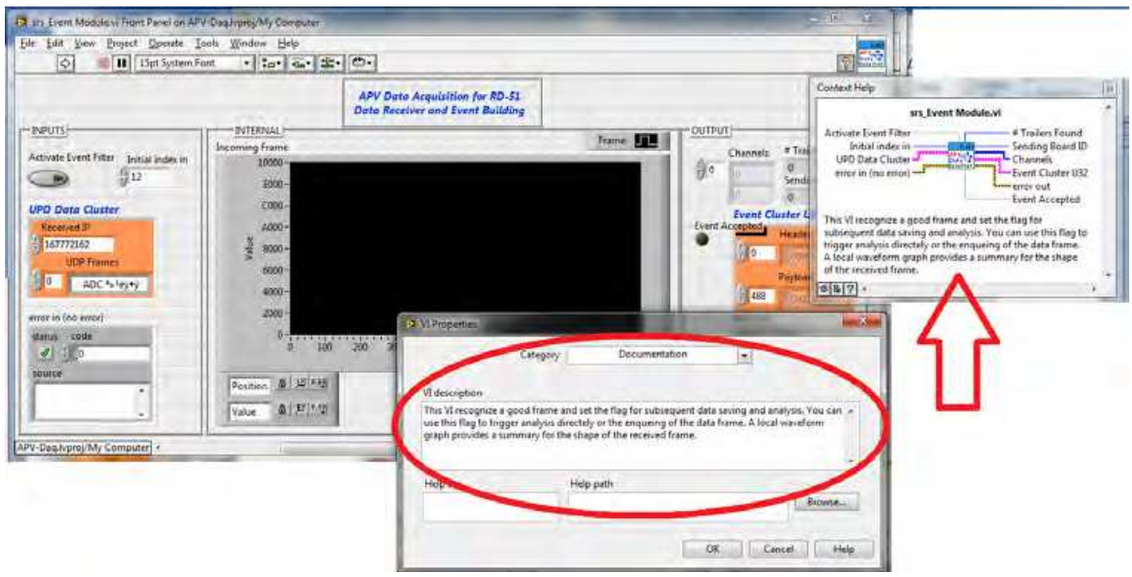


Fig. 18. Example of a documentation introduced in a SubVI.

Note the care taken in the arrangement of the objects on the front panel: Input, Internal and Output frames create a visual impact which immediately guides the user; the coloured object are Type Definitions and this strategy is useful to locate them in panels; the message written in the "Description" page (indicated by the oval (in figure 18)) of the VI-Property window is automatically shown in the LabVIEW Context Help.

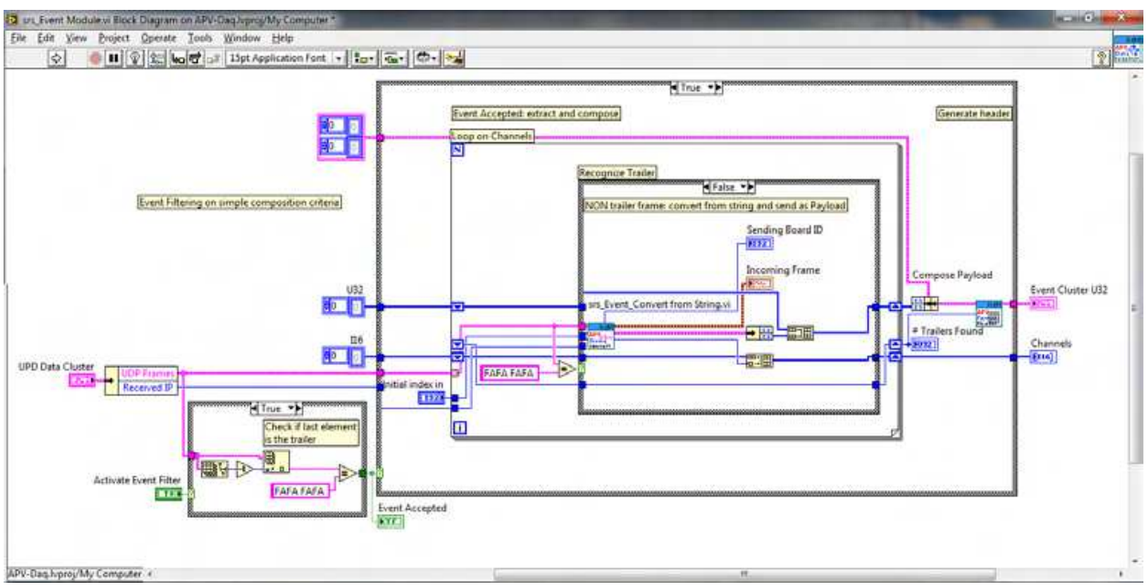


Fig. 19. The diagram of the SubVI of figure 18.

In the second figure the corresponding diagram is shown. Note the presence of several free labels used as internal documentation of the diagram itself. They help a lot in reminding the purpose of objects used, of SubVIs etc. Here are some hints concerning the diagram:
Suggestion 1) It is useful to document specific operations by choosing to **show the label on structures** (For and While Loops, Events structures), giving them a clear name which

indicates the function. You can do the same on some function (it is done on a Bundle Function in the Figure 19).
Suggestion 2) Using **free labels** to document the diagram is highly suggested.
Suggestion 3) From version 2010 on **use the "wire label tool"** to give a label to single wires, indicating the data transported.
With a well-organized and documented set of SubVIs you do not risk that even after two months, when you reopen your project, you will not remember what you did.

6. Data filing in LabVIEW

Data filing is not a trivial job in any programming language: LabVIEW makes no exception in this regard. There is, in fact, a very wide range of choices concerning how you can save your data on files, and the decision of using one format instead of another must be steered by the following two points:

- **What kind of data**, in a "logic sense" you are saving.
- What is the **future utilization** of this data?

These two points are not clear at all to the novel developer, and sometimes even to an experienced one.
Typically, once learned how to save data onto spread-sheet files, the tendency is to continue using spread-sheets even in cases that would require a more convenient solution: the Spread-sheet format is supported by the fact that it is Excel compatible, in the sense that files can be directly read by Excel and eventually transformed into Excel format. But we mustn't forget that there exist a lot of other possibilities whose utilizations can be strongly recommended.
First of all, let's try to give some explanations of the statements above.

- The "logic sense" of data to be saved means a sort of category to which data needed or produced by an Application belong. Here is a possible list reported as a series of examples:
  a. [configuration data] The physical channels to which the program must acquire data from the field via ADCs or other related hardware.
  b. [configuration data] On-Off status which decide if some Data Acquisition Channels must be taken or not (to temporary inhibit DAQ from certain channels).
  c. [Internal data] Information related to the user (ID, account, a possible password).
  d. [internal logging] Log information concerning the correct functioning of an Application for debugging purpose.
  e. [internal logging] Journal file of actions taken by the user / Journal file of actions taken by the Application for getting a strong redundancy on delicate processes.
  f. [log raw data] Real (physical) data acquired that must be saved on disc for future analysis.
  g. [log analysed data] Online analysis is performed and we want to save basics results of analysed data.

The list is far from being complete. Pay attention to the fact that we are not speaking about **the format**. The format, in fact, **is an independent choice:** all the logical categories of data indicated in the above list are subject to be saved into different formats, but there are no fixed rules for this. One must clarify, **before deciding the format**, what is the logical aspect of data to be saved. For each of them a different and more indicated solution is suggested.

- The second point is in some way related to the first and, from a certain point of view, the choice must be done by considering both aspects. Saved file can be:

a. Successively used as source of data **to be analysed**.
b. **Retrieved intact**, just as they have been written.
c. Subject to be **archived** in a long term historical archive and sometimes retrieved for future reference.
d. Hidden because it is used **by the program itself** for its internal setups (typical case of Configuration Files).
e. Subject to be used on the **same machine** which took it or moved to be opened/used on a **different machine**.
f. Their destination is on the same **Operating System** or on a different one.

And this list could also continue.
Another aspect is the format: **text file or binary file**? What exactly is the difference? What is more convenient and for what reasons? I found that sometimes even experienced people make some trivial mistakes by choosing a wrong solution on this point.

6.1 An overall analysis of the file functions

The File Functions are very well divided and organised in LabVIEW. After an overall analysis of them we will switch to a series of useful hints for deciding file saving in LabVIEW. The Figure 20 represents the main File Function palette at the centre with four sub-palettes opened. The first row of the main palette contains the "basic" File Functions, the ones you use when you don't want to worry about *how,* technically speaking, *you are saving* your files.

6.1.1 Basic file functions

This first line provides two different and pre-ordered ways of file saving: spread-sheet file (Write and Read respectively) and express VIs which use LabVIEW Measurement file format.

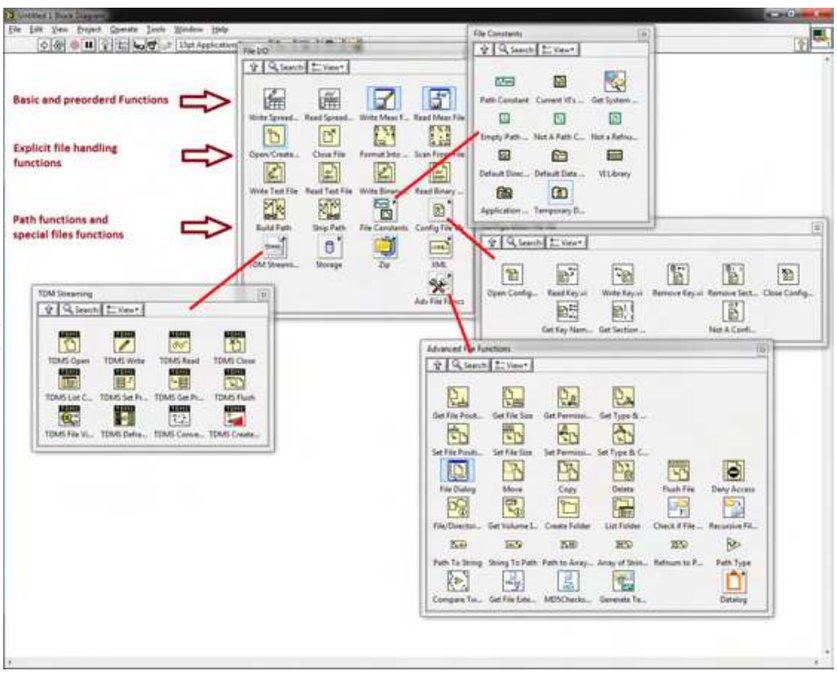


Fig. 20. The very extended and rich File Functions palette of LabVIEW.

Even if these two methods are useful in several situations, they suffer from some drawbacks:

- Spread-sheet file

a. Spread-sheet files **can only save numeric values** in double precision representation. Moreover some old LabVIEW versions saved these data as single precision representation. If your floating data are usually treated in double precision**, a loss of representation or loss of data** can be introduced if you saved them with the Write to Spread-sheet File Function.
b. You **cannot write on your file any other auxiliary information** like strings, Booleans, annotations etc.: you **cannot save date and time string** information; date and time can only be saved as floating point, i.e. using the absolute timestamp of LabVIEW, but that is to be decoded by another program and is not clearly understandable by human operators if written as a double number and not interpreted.
c. The machine format is a text file, whose fields are separated by tab character. This means that huge files can be created even for not such a large amount of data to be stored.
d. When you record spread-sheet files you must be careful in developing the file **towards bottom** direction instead of to the right one: in other words, spread-sheets must grow toward the bottom by appending **lines**, avoiding too many columns on the right. In this case, spread-sheet programs can fail in opening these files.
e. The numeric precision can be lost because **you chose the number of significant digits** at the moment of the file save, just as in a print-out sheet.

The last point is particularly significant when critical numerical data must be saved. It is an important issue concerning text file which will be considered later.

- LabVIEW Measurement files. They can be very useful, but I suggest to adopt this solution for a relatively small amount of data. They are driven by an Express VI, so a specific setup panel is opened when you introduce these functions into your diagram. The main advantage of using LabVIEW Measurement Files is the rapidity in defining the file and obtaining it. The read-back of the file is also easy.
  Here are some drawbacks:
  a. They are usually text file unless you select "binary (TDMS)" into the setup page, so they reflect the usual problems of text files.
  b. Their utilization is basically foreseen as a LabVIEW tool, being incompatible with any other analysis software or platform.
  c. Files can grow a lot if you try to save uncontrolled data banks, like big multi-dimensional arrays and so on.

The last point can be a very critical issue. Often the data size treated by an Application is hidden by the apparent ease in which the diagram handles it: a simple array, for instance, **can grow to hundreds of Megabytes** without any particular indication or problem. This aspect is so transparent that risks can be under evaluated by the developer: in the case of saving such an amount of data, very huge files can be abruptly created, causing slowness in the machine response.

In conclusion use the **first line of File Function** if you have little data, in small or moderate Applications, or to do initial tests of data saving of a new Application. Then consider using a **more sophisticated method** for **file saving** even for moderate Applications and whenever more control and care must be taken in saving data.

6.1.2 Owner's file functions

The second and third lines (on Figure 20) provide a complete control of file writing and reading which should be well understood. These functions are able to manipulate files as

you want: you can open, write/read data repeatedly and finally close. You can use these function for every situations of data I/O to disc. You can decide:

- If the file is a text or a binary file;
- The data representation in the case of binary files;
- The number of characters/data elements to be read;
- The adoption of **data streaming** technique.

6.1.3 File Constants Palette

The upper right palette of figure 20 is the **File Constant Palette.** It is an important series of tools useful for path definition and handling. Consider using them to create the correct path to access the disc area in which files are stored (the whole path to the actual folder). In particular the "Current VIs Path" function is extremely useful for getting the current path in which the VI is acting: wherever you move the VI, on different machines also, the function shows you the current and complete path to locate the VI. From there, with the **strip and building path functions**, you can define your own path.

6.1.4 Advanced file functions

This is a very plenty toolkit with all sorts of functions: you can move within an (opened) file, by positioning the "pick up head" wherever you choose, calculate file sizes, list directories, create or destroy folders or files, and so on. These functions are not frequently used, but when needed they are really useful. A typical use is the positioning within a file with Get/Set File Position functions, which allows you to move in binary files like on a tape recorder. **Some experience is needed to use these functions; always start with little tests**: create new, small VIs to do the test and study the effects and the results. Then integrate the test VIs as SubVIs into your final application. This process is the only way to learn and clarify the utility of these functions kit. Since the functions are **low level Functions**, you can almost do anything on your File System, even delete important System Files! They must be used carefully.

6.1.5 Text files or binary files?

At this point in our discussion, some time must been spent on the format. Text files are usually preferred to binary ones, at least because when we open them with a text editor we can *see the contents*. The same thing does not happen for binary files which are somehow unreadable. A binary file can only be read by using the same procedure as when it was written: the same data type must be declared when we read and the number of bytes of *elements* must be input to the read function.

But what is the real difference between the two types ?

Consider that a file (binary or text) is composed of a sequence of bytes, groups of 8 bits on the disk (independently from how the File System of the machine has fragmented them on the disk). A single byte can have a value that, if translated into integer, goes from 0 to 255.

Well, a file which can potentially contain all combinations of the 8 bits within the bytes, i.e. bytes which contain all values from 0 to 255, is called a **binary file.** On the contrary, if the values are limited to a certain subset of the range 0..255, they are called **text files**. The subset is a precise one: "allowed codes" cover all ASCII printable characters (form 'space' to 'z', numerically 32..127 code range) and some other "control character" which indicates line or page feed, carriage return etc.. Under certain points of view there is not a big difference between the two categories, but practically the difference is enormous.

The following table reports some hints concerning the possible choices for file saving:

Table of suggestions for choosing format of files	
Use Text files when	**Use Binary files when**
You do not have big amount of data to be saved.	The amount of data starts to be significant.
You want to have the "feel" on the data just saved by watching them.	Your utilization of data in the future is for analysis or archiving purposes.
You are not concerned about space on the disc.	You are concerned about saving space on the disc: Binary files are extremely more compact than Text ones.
You don't have to worry about speed in data writing or reading, since text file operations are slow.	You are concerned about speed: if you need to get data quickly and save on disc "into the loop" Binary files is the only solution, often accompanied with the "data streaming technique" (see later).
You don't have to worry about machine overload: text files are somehow a heavy process for it.	You are concerned about machine overload: writing in binary is a light process and no data conversion is needed.
You do not care about precision for saving numerical data: suppose you have a 15 significant digits number to be saved. When saving it in a text file you de facto print it on the disc by using a character representation. If you, by mistake, ask for 5 significant digits to be saved, the remaining 10 digits are definitively lost.	You are interested in definitively storing data with the original precision or the original conditions (i.e. all data must be saved "as they are").
Your data must be read from other programs, different machines or different Operating Systems.	Your data must be read in the same environment (LabVIEW); some extra work is needed as a specifically-written program into a different Language (C, C++, …), even on a different Operating System, in order to read binary file on other platforms.

Table 1. A compendium of indications for steering the choice of a correct file format.

Binary files seem to be more complicated or difficult to manipulate. Well, it is not so: they are quite simple and easy to understand. Consider the following example. We have the following eight numbers (as signed integers) to be saved:
667283 -134452 235567 7894451 -5569852 7789663 -3363331 -445874
Each of them, to be stored in memory, needs a 32 bits Integer (I32) representation and takes 4 bytes in the computer memory.

- If we save them into a Text file, we need approximately 67 bytes: each digit is, in fact, transformed in an ASCII character (1 byte) and **we do not save numbers** but their representation in a character writing. It is like **printing a sheet of paper and saving it on the disc**. The text files need some control characters like 'tab' to move from one number to the next, 'EndOfLine' to end the rows and so on.
- If we save them into a Binary file, the effective file size containing data is 32 bytes: 8 numbers times 4 bytes each, simply the dump of the memory onto the disc. Negative numbers are already taken into account thanks to the two's complement representation.

Try to figure out how they work: take some time before deciding and do not make rapid decisions which usually end up in text files.

6.1.6 TDMS Files

In recent years, starting from version 7, National Instruments introduced the LabVIEW Measurement Files format (LVM files). We already cited this format while talking about the Basic Files Functions: it is a text file well organized to record on disc "sessions" of data taking under the form

[Date-Time] | [X] | Channel 1 | [Channel 2] | …

Where '[]' means optional fields. The format and storing parameters (name, paths, automatic naming,…) can be set up by Express VIs. A prefix header can be optionally recorded to store further descriptions.

A few years later the TDMS format was introduced, together with the TDM Streaming palette (the bottom-left palette in figure 20). TDMS File format is an evolution of LVM, recorded in **binary mode** and optimized for search and access thanks to a specific indexing technique. The details of how the files are stored or indexed are transparent to the developer, who can take advantage from the intrinsic structures and handling related to this format. TDMS is a LabVIEW "internal" format, in the sense that it is conceived as an efficient method for **data storage and retrieving in LabVIEW environment**: you can save large amounts of data, perform data streaming and classify your data into different channels and categories. This choice can be particularly useful in several situations in which you have a consistent amount of data and/or numerous sources of it (i.e. several channels which produce data).

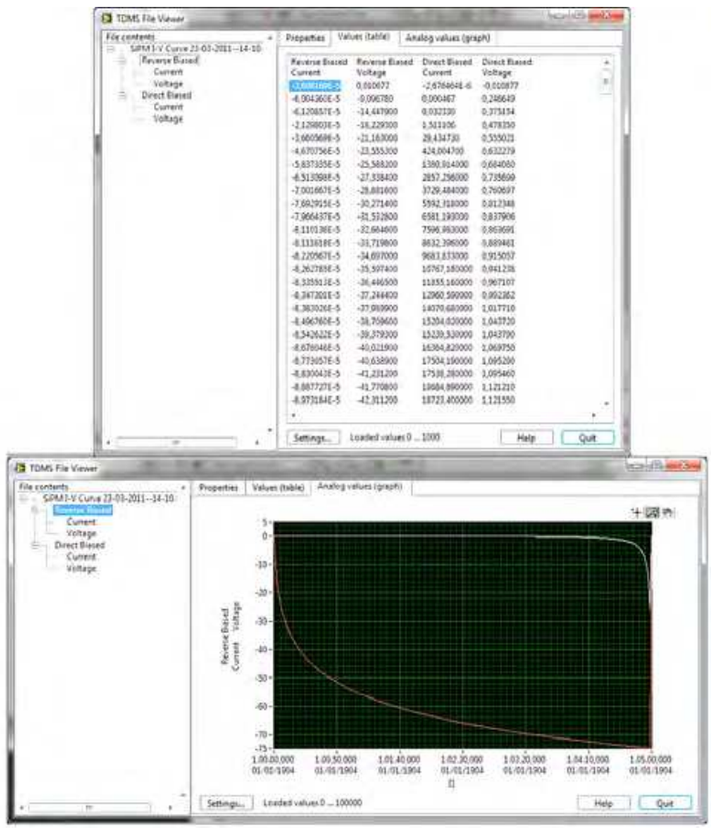


Fig. 21. An example of the TDMS file Viewer in which some data is shown.

In the Figure 21, two screenshots of the "TDMS File Viewer" (the bottom-left VI in the TDM palette of figure 20) are shown. I'm indicating here two possible presentations of data: table form and graph form. You can imagine that the number of channels visible as in the top table can increase, or the number of curves on the same graph could increase as well. It depends on how you organise the data saving operation in your writing VI and on how you access to their view by using the TDMS File viewer: this function works as an interactive browser that you can expressly open or programmatically open from a VIs.

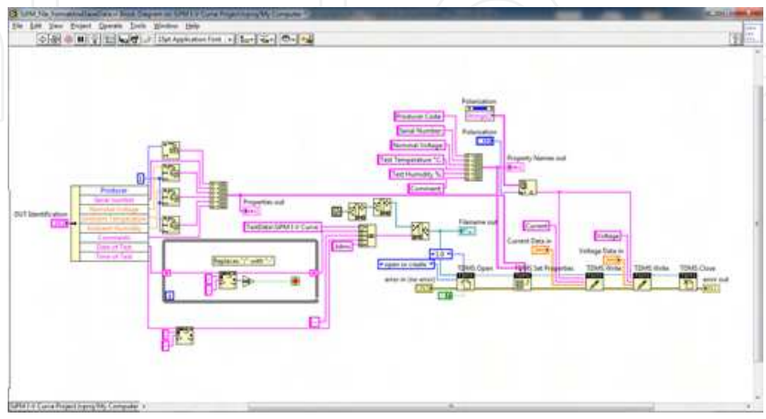


Fig. 22. Portion of file writing VI which saved data of figure 21.

In the Figure 22 you can see how the data are saved into a TDMS file: two writing operations are used ("TDMS Write" to create the two Channels "Voltage" and "Current"), while the same Group Name is used (coming from "Polarization" string set). Note that before writing the file, a series of auxiliary data is stored by using "TDMS Set Properties" function: it is a very useful feature that allows you to store a header on the file whose format is completely custom. The parameters, conditions and serial number of the device under test are written in the header in the test bench Application chosen for this example.
In conclusion: if you have a consistent amount of data that must be visible internally in LabVIEW, **without any doubt use TDMS Files**. In the case you need to extract data and use it in a different environment (i.e. convert to ordinary binary or to text file) it is easy to write a conversion program.

6.1.7 Configuration and Datalog Files

We have yet to speak about two special categories of files: Configuration files and Datalog Files.
Use **Configuration Files** if you need to store **internal characteristics or parameters of your VIs or Application**: examples are initial setup of a panel or of a bench instrument, different configurations for instruments or for any connected hardware; choices of taking or not data from certain channels of your field and so on. Configuration files resemble the old ".ini" files in Windows: they are text files in which a series of "Sections" are listed and, within any "Section", information is identified using "Keys". One advantage is that they can be opened in a text editor and, if needed, edited by hand (even if this is not a suggested action) and one can always keep track and "understand" what is written. One disadvantage is that, as the number of Keys and Sections grows, writing and reading the file can start to be tedious because you need to expressly call the "Read Key", "Write Key" or similar VIs for every Section and Key you have. It is convenient to organize the Configuration File reading and writing process well:

1. Perform Read and Write on Configuration files in **specifically written SubVIs**; it is not convenient to implement this aspect into the Main VI.
2. Carefully chose the **Sections** (how many and their names) to be used and **Keys** within sections.
3. If possible **use automatic naming in For Loops** for assigning names to Keys or Sections. Automatic naming can be formed by a base (i.e. "Voltage_") followed by the index of the loop iteration

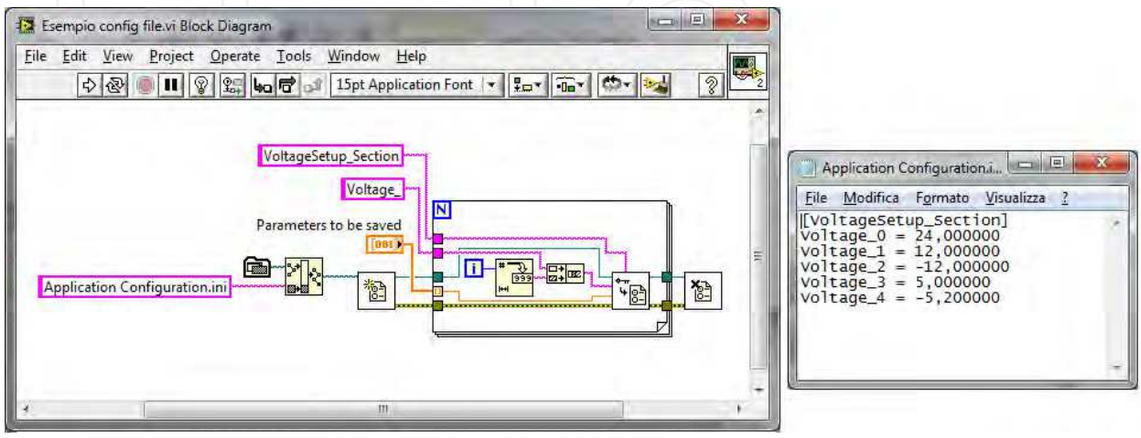


Fig. 23. Example of Configuration File write using an iterative technique and its result.

To have an idea, the Figure 23 reports the code to write a DBL floating point array to a Configuration File in iterative mode, together with the resulting file. All types of information needed by your application can be collected in configuration files: integers, doubles, Booleans, etc., even Variants data type. One should decide to use **Variants** when the amount of data to be stored and retrieved is quite consistent. Variants are stored as a series of codes and strings, where the former classify the type of data contained in the latter. Information can be reconstructed using the Variant as data type to be read from the file and finally by reconverting Variants to the original type using the "Variant to Data" function. Using Variants allow you to quickly write and read on configuration files, but the result is somewhat incomprehensible at a first glance.

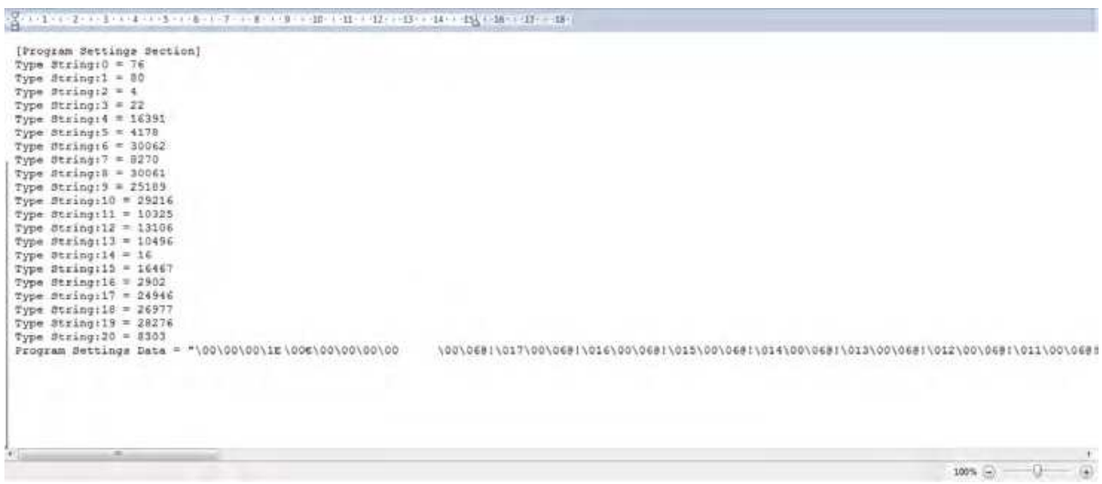


Fig. 24. Example of the aspect of a Configuration File using a Variant writing.

Use **Datalog Files** when a series of data organized as "Records" must be saved. Datalog files are a special kind of binary files basically conceived for storing Clusters contents: suppose you want to save a group of inhomogeneous data, like the content of a cluster composed by a timestamp, some Boolean, integers, strings, and so on, and several times during your execution. Datalog file makes a single Record at every saving action and stores it in this way. Single records can be accessed when reading back the Datalog file, by specifying its index: the first, second record etc.. Moreover the record position can be specified before reading, avoiding a sequential access which is usually slow.

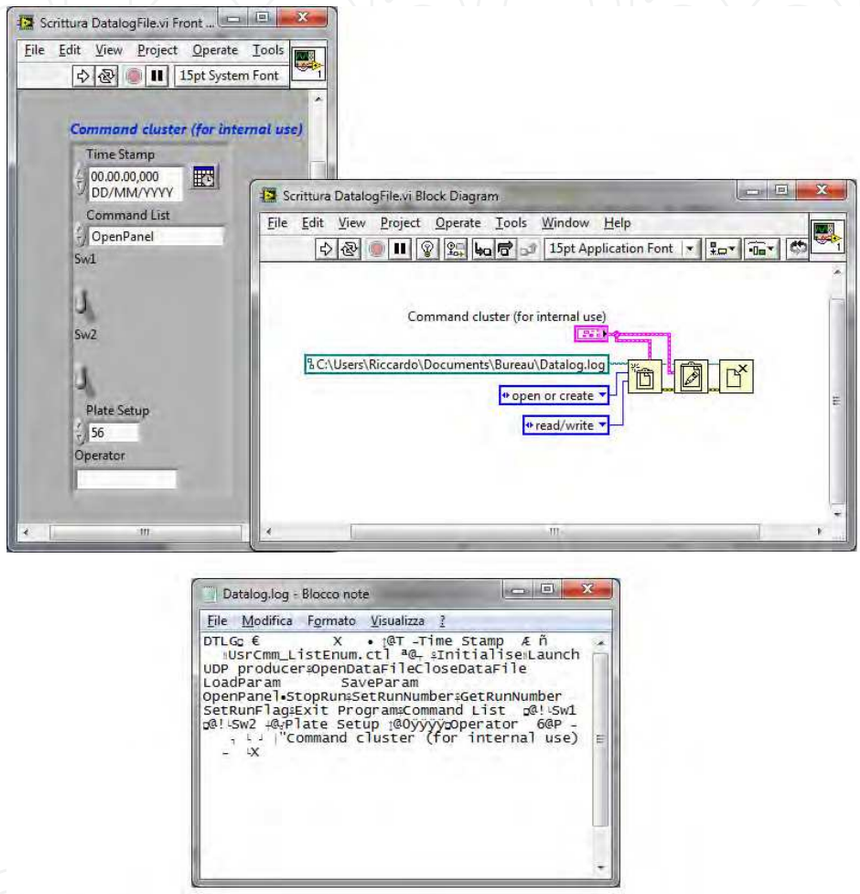


Fig. 25. Example of use of a Datalog File. The cluster is transcript as records in the file. The resulting file contents (which is binary) is also represented.

6.1.8 The concept of the "Data Streaming"

A very important concept which is typically ignored by new or inexperienced LabVIEW developers is the **Data Streaming.** If an Application produces a flux of data whose duration in time or size is long, the Data Streaming technique is almost a "must", to save data. New developers tend to accumulate data internally in the VIs, under the form of arrays of different sizes: then, at a certain moment in the program, they "save" these arrays into some files. This is not a good solution for several reasons:

- First, accumulation of large amounts of data in some structures of the VI is not convenient for the machine overload it can create, in particular regarding the LabVIEW Memory Manager.

- Second, if the program halts or is subject to an internal error which induces the VI to abort, all accumulated data can be corrupted or is lost.
- Third, since accumulation memory cannot be infinite, you need to stop some processes to dedicate machine time to write the data in a shot: the writing time can have, unfortunately, a duration of several seconds in the case of a large amounts of data to be saved. This situation can be attenuated with the correct handling of **parallel processes**, which constitute another important aspect of good programming in LabVIEW.

For all these reasons, in similar cases, use **Data Streaming**. The concept is simple and is based on the following processes:

1. open your data file
2. enter a loop which takes data and write immediately into the file, usually in binary format
3. stop the loop when finished and close the data file
4. check for errors.

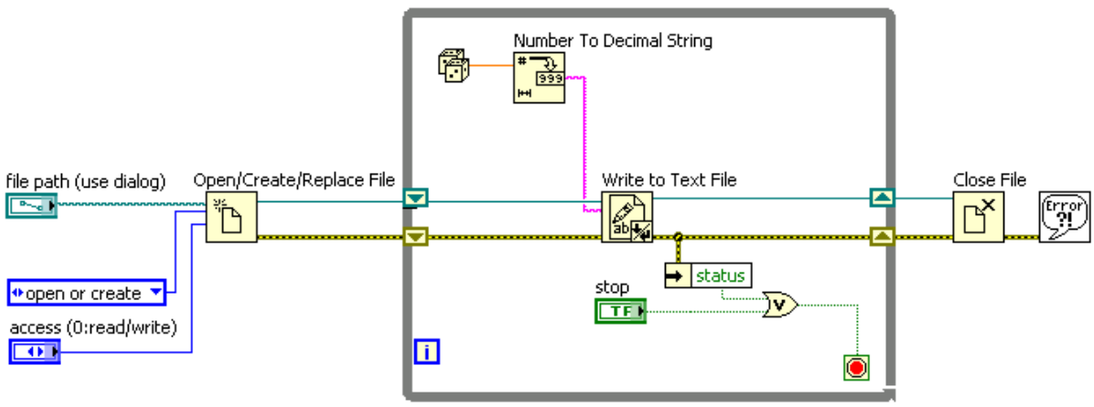


Fig. 26. Example of a simple Data Streaming implemented with a text file. (From National Instruments 2009-Core 1 Course presentation slides, Lesson 6 "Managing Resources").

In the Figure 26 the simplest data streaming technique is reported, in this case made by writing on a text file. Such a structure must be implemented as an independent one with respect to the rest of a VI, like parts which operate on the User Interface, for instance. The next paragraph will present some complex structures and Design Patterns which help for such a kind of design.

6.2 A conclusion concerning the files

Filing in LabVIEW (and in all programming languages) is a very extensive topic which has lots of implications. A mistaken choice in file format or recording technique can cause serious problems in using the data stored in your DAQ systems. It often happens that a series of conversion programs (under the form of several new VIs in the case of LabVIEW) must be written just to convert data files in order to make them compatible with some analysis system. A list of suggestions follows:

Suggestion 1) **Plan carefully** regarding your file format before starting to write code abruptly.

Suggestion 2) Decide, on the basis of the amount and type of data to be stored, **how many files you need to write** "in parallel": LVM files, Configuration, TDMS, Datalog, actual row Data Files, analysed results, etc..

Suggestion 3) **Choose carefully the files locations** (folders) related to your application, and use an automatic naming technique for data files which is based on a correct archiving logic:

for example a basic file name followed by date and time of your DAQ and a standard file extension.
Suggestion 4) **Decide about the format** for your data file**:** text file or binary ones. Both of them have advantages and disadvantages. A binary file is compact, rapid to write and read, and reports the actual values taken during DAQ; if correctly formatted, it can be read under different platforms too, even if this is an advanced task. Text files are nice because they are comprehensible, but take more time to be written and read, they need more space on the disc and limit the numerical precision because you cast your native precision by an "internal printing" operation.
Suggestion 5) **Save files in streaming mode** using an adequate structure if the DAQ or related processes are long term processes. Avoid accumulating large amounts of data and save them at the end of your process or from time to time by interrupting your DAQ.

7. Structures and parallel programming in LabVIEW

Structures in LabVIEW include For and While (Timed or not) loops, Case and Event structures and Sequences (flat or staked). Most beginners use structures in an incorrect way: they superimpose one structure over another by adding pieces which try to satisfy the increasing demands in the features of the application under development, again without any planning.
First of all **try to avoid using the Sequences Structures.** All LabVIEW programs can be done without them, and data dependency can fully substitute the sequencing of operations in most cases. Sequence Structures must be used in very restricted cases, when, for example, stringent timing issues are present during, suppose, a dialogue with an instrument. Sequences Structure is in contrast with the data dependency paradigm typical of LabVIEW, and must be used expressly to circumvent it **when necessary.** They tend to hide code in different frames, generating poor diagrams not clearly understandable.
Bad usage of structures implies several problems and under-utilization of LabVIEW system features. LabVIEW is an implicit **parallel environment** in the sense that it is designed to generate parallel and optimised code without particular intervention of the developer: but the latter **must know how to arrange structures** in order to induce LabVIEW to work parallel.
A last point is that Structures are related to the so-called **Design Patterns** techniques: this topic is a very important point which is expressly studied in several official LabVIEW courses (from Core 1 on).

7.1 Parallel processing issues

Putting more than one loop structure on the same diagram, automatically generates parallel code in LabVIEW, **if no data dependency exists among the loops.** Loops (While or For, but the usefulness is basically on While ones) cannot be interconnected to one another with wires, because this would create a data dependency, and the second loop should wait for the end of the previous to start its execution. Even transmitting a "STOP" flag needs a specific technique. LabVIEW compiler optimizes code in order to be executed in separate threads of the Operating System or, if it is the case, in separate sub-processor of a multi-core system.
Similar precautions must be taken to transmit **data** among parallel loops, and for this aspect several synchronization VIs are available (see the "Synchronization Function" palette). We cannot do a large *excursus* here, but I want to show you some general principles in order to encourage you to continue in carefully study and learn about these techniques.

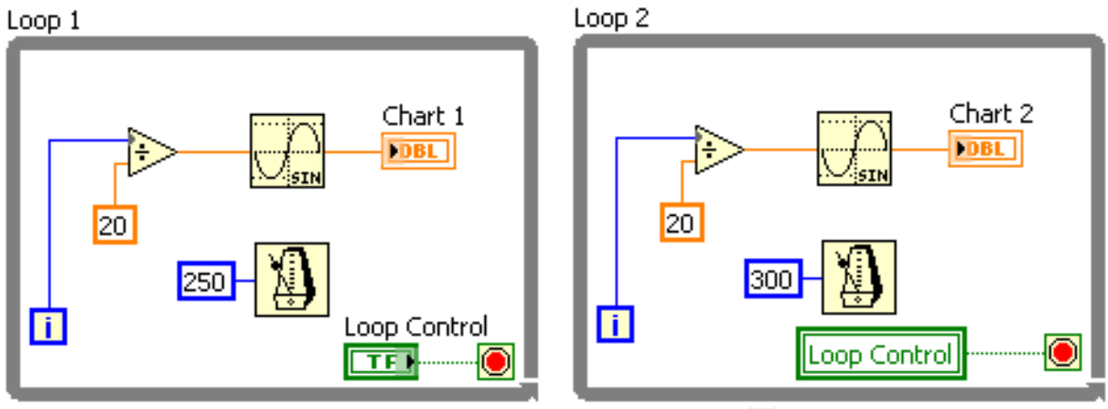


Fig. 27. Example of a very simple parallel structure. The Stop signaling must be passed via Local Variable (as in the case of figure), Global or Shared Variable or Propery node (on Value item). (From LabVIEW 2009-Core 1 course presentation slides, Lesson 9 "Using Variables").

Consider the example in Figure 27 the two loops are timed using the metronome function and the stop signal is sent via local variable. This is the simplest way for implementing parallel processing.
If you need to send data from one loop to another, which is the case for some particular **design patterns**, you must use Synchronization Functions.

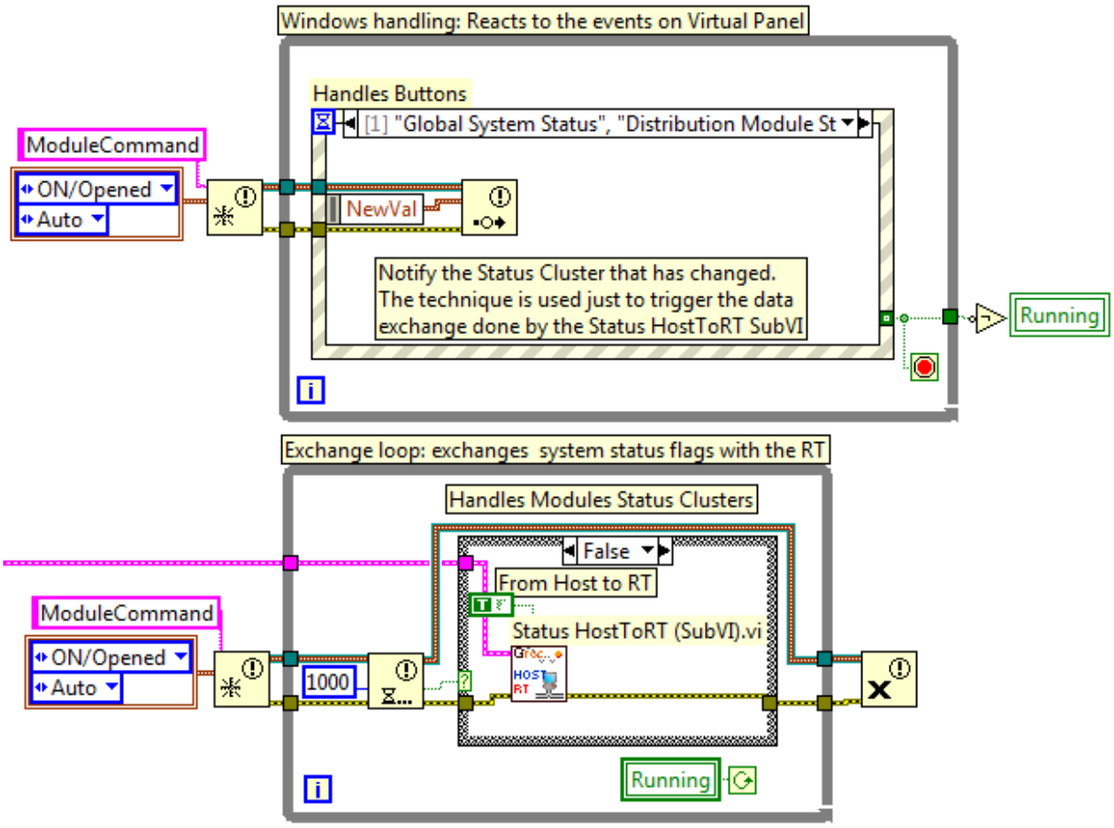


Fig. 28. Two parallel processes which exchange data via Notification Functions and control the end of process using the "Running" flag.

By looking at the Figure 28, several techniques for parallel processing and data exchange are presented:

1. The loops exchange information using **Notification VIs Functions.**
2. Notifiers allow you to pass a **single data** to the receiver.
3. The data can be of any type: here a cluster containing two enumerated types is passed.
4. You can ask to create "the same notifier" (whose name here is "ModuleCommand"): it will be created only once by the first Create Notification Function which is executed. Using multiple initializations with the same notifier name adds clarity to the diagram.
5. The upper loop signals the action of the user that has been taken into account by the Event Structure, by writing the "new value" of the notified variable (the cluster) into the notifier reference.
6. The lower loop receives the data and processes it. You have the possibility of tracing, in the lower loop, if any data has arrived, thanks to the "timeout" technique: the Boolean coming out from the notification receiver (lower loop) is the timeout flag.
7. Both loops need no timing: the upper one waits for an event from the users that is processed by the Events Structure (typical way of functioning for this structure); the lower loop uses the timeout of the notification function as internal timing, and performs different operations in the case the notifier has sent a data or not (timed out or not).

This technique is an application of the so called "**Master-Slave Design Pattern**", in which a short information, like a sort of flag, is sent from a "Producer" loop to alert a "Consumer" loop to do something.

An extension of this technique is the "real" **Producer-Consumer Design Pattern** in which **Queues Functions** are used instead of Notification Functions. Use Producer-Consumer design patterns, when more data must be sent from an acquisition loop to a data treatment loop, for example. Using notifier can cause data to be lost since it can treat one item at a time: queue functions, on the contrary, allow you to send streams of data from a producer loop to a consumer loop without losing any.

In the figure 29 **Data Consumer** is the bottommost loop. It is a state machine which "knows", from outside, if a Data Acquisition Process is in action or not.

- In the first case it provides to extract data packets from the queue, analyse and write them into an archiving data file **using data streaming on a binary file**; it finally sends formatted events (I mean *physical events*) data to a second queue for event monitoring purpose: a third loop (not represented in the figure) provides for this extraction and online presentation.
- In the second case it just does a monitoring of the incoming events.

This example is extracted from an actual Data Acquisition System for the "RD51" research activity at CERN (Geneva, Switzerland), on a Data Acquisition System for MicroMGas particle Detectors. This example carries several technicalities into it: two queues are used, one for UDP (raw) data and the second for pre-analysed data (built physical events). A Producer-Consumer design pattern is used where the consumer features a state machine which handles Run status on the experiment: state machine stays "idle" if no Run is active, and makes a series of operations (like initialization, file opening, flags settings,…) when a Run is started. Then it takes data, analyse it creating actual physical events and save it into formatted, binary streaming file; and eventually sends events to the online monitor via the second queue.

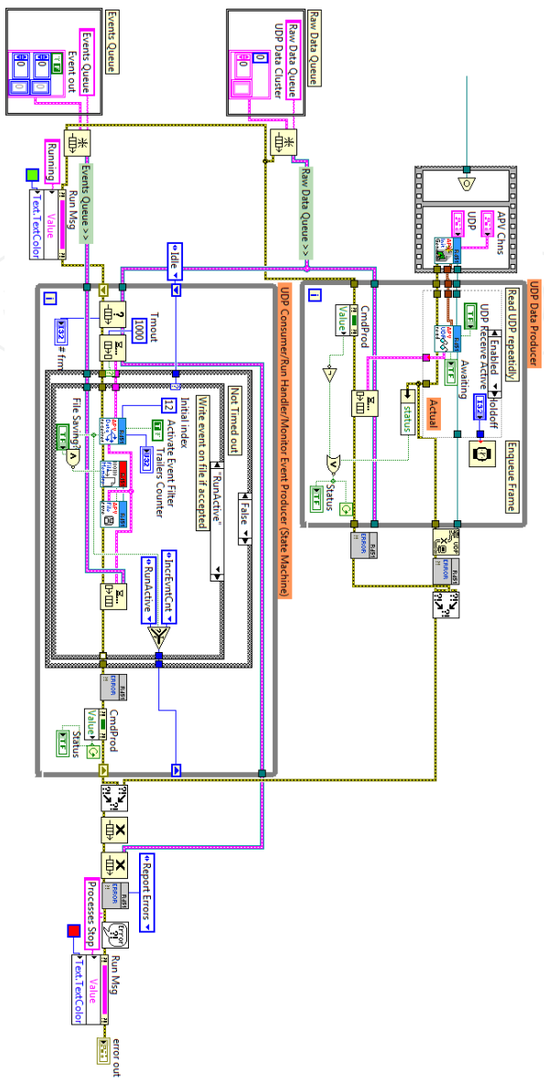


Fig. 29. A more complex example of parallel loops in a "Producer-Consumer" Design Pattern.

Using design pattern is extremely advantageous for a lot of reasons:

1. It is clear at the beginning of the development operation, how to implement things in the VIs.
2. Design patterns feature the logic and the "place" for implementing all that you need in an Application: from User Interface which gets commands from the User and formats information toward the "Command Consumer" which actually executes the command, to data getting and distributing onto secondary processes.
3. The design becomes clear and standard: a known framework helps to insert code. You already know where and how to process an User command, for instance, where to get data or write a file.

Definitively the **work can be well organised in a rational, standard way**. Future extensions of code can be added easily, but also reusing of code in new applications, since the framework is basically the same.

8. Conclusions

This Chapter does not want to substitute official LabVIEW courses, but, on the contrary, pushes in the direction of encouraging following some course, at least Core 1 and Core 2 if you are a beginning programmer. It is only a collection of considerations and suggestions in the direction of improving the knowledge in what a developer should know as a minimum to develop rational, well organized and effective Applications. Extending Applications in the future, which is a common job for programmers, can be done without altering the existant structure with the risk of obtaining a non effective solution. Remember that it is not convenient that diagrams go over the screen size, for clarity and readability: if that is not important make sure that the Diagram goes out of the screen in one direction only; horizontal direction is preferred. If you are starting to go over the screen, stop for a while and reorganize by creating more SubVIs, compacting structures and wiring and so on. Avoid excessive usage of local and global variables and sequence structures, since these features overcome the data dependency paradigm.

Many other suggestions can be made. A lot of other topics exist in LabVIEW programming, from Active-X usage to priority setting of SubVIs, from extensive use of Variants to complex data structures, from interaction of machine with the external field (via specific hardware) to Instrument Driver designing techniques. This list can be basically infinite.

I hope this work can stimulate new or inexperienced developers in adopting a well-planned strategy for development their job, in the direction of creating effective Applications which are finally understandable, reliable and scalable. This strategy starts from a good knowledge of the language features that can only be reached by a careful study before and during LabVIEW development process. In this way a complete experience can be accumulated which make, at the end, a very capable LabVIEW programmer.

In the end I would like to conclude with a final remark. The Figure 16 is an indication in understanding to which level of difficulty a diagram can arrive, when the developer *thinks to "know everything" concerning* LabVIEW. Several people, in particular if coming from the scientific environments, are convinced that just by giving few spots on the LabVIEW development system they are capable of doing everything: this is a big mistake, and anyone who wants to develop effective Applications in LabVIEW should have the ability to stop at a certain point and take the time to *study* LabVIEW.

9. References

Bishop R.H. (2009) *LabVIEW 2009 Student Edition*, National Instruments

Essic J. (2008) *Hands-On Introduction to LabVIEW for Scientists and Engineers*, Oxford Press

Johnson G. W. (1994) *LabVIEW® Graphical Programming*, Mc Graw Hill

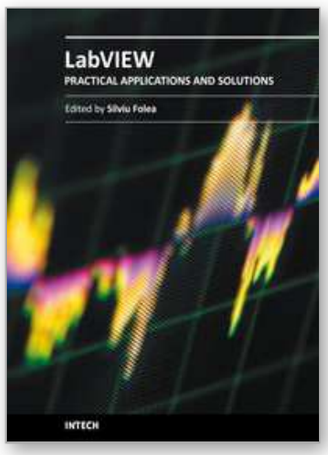
National Instruments (2010) *LabVIEW Core 1 Course Manual and presentation slides*, ©National Instruments

National Instruments (2010) *LabVIEW Core 2 Course Manual and presentation slides*, ©National Instruments

National Instruments (2010) *LabVIEW Core 3 Course Manual and presentation slides*, ©National Instruments

Sumathi S., Surekha P. (2007) *LabVIEW based Advanced Instrumentation Systems*, Springer-Verlag

Travis J., Kring J. (2004) *LabVIEW for everyone*, Prentice Hall



Practical Applications and Solutions Using LabVIEW™ Software

Edited by Dr. Silviu Folea

ISBN 978-953-307-650-8
Hard cover, 472 pages
Publisher InTech
Published online 01, August, 2011
Published in print edition August, 2011

The book consists of 21 chapters which present interesting applications implemented using the LabVIEW environment, belonging to several distinct fields such as engineering, fault diagnosis, medicine, remote access laboratory, internet communications, chemistry, physics, etc. The virtual instruments designed and implemented in LabVIEW provide the advantages of being more intuitive, of reducing the implementation time and of being portable. The audience for this book includes PhD students, researchers, engineers and professionals who are interested in finding out new tools developed using LabVIEW. Some chapters present interesting ideas and very detailed solutions which offer the immediate possibility of making fast innovations and of generating better products for the market. The effort made by all the scientists who contributed to editing this book was significant and as a result new and viable applications were presented.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Riccardo de Asmundis (2011). The Importance of a Deep Knowledge of LabVIEW Environment and Techniques in Order to Develop Effective Applications, Practical Applications and Solutions Using LabVIEW™ Software, Dr. Silviu Folea (Ed.), ISBN: 978-953-307-650-8, InTech, Available from: http://www.intechopen.com/books/practical-applications-and-solutions-using-labview-software/the-importance-of-a-deep-knowledge-of-labview-environment-and-techniques-in-order-to-develop-effecti

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2011 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License, which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.

IntechOpen

IntechOpen