20

Graphical Programming Techniques for Effective, Fast and Responsive Execution

Marko Jankovec
Faculty of Electrical Engineering,
University of Ljubljana,
Slovenija

1. Introduction

Graphical programming languages (G-language) in LabVIEW provide easy and intuitive coding, where even a beginner without any programming skills can build simple software and get results back quickly. The versatile collection of blocks providing most functions needed for a general user and simple graphical representation of dataflow that resembles flow charts are main advantages of G-language over text-based programming. However, when project grows and the complexity increases, the one dimensional programming space in text based languages starts to show advantages since it allows program flow in vertical direction only making it easier to follow and organize. The programming space in case of G-language is at least two dimensional with no data flow direction constraints. Furthermore, the use of structures for data flow manipulation (e.g. sequence, event structure etc.), at least one additional dimension is introduced.

Thus, a project that started as a simple and easy task that could be handled by a non-programmer quickly becomes vast and hard to understand. Usually the graphical code quickly grows over one computer screen making data flow very hard to follow and it becomes difficult to insert or change any functionality later on. Finally, it comes to a point, when one asks himself whether it would not be better to start all over again keeping in mind all the required software functionality. At that stage, the programmer already has much more clear ideas how his software should work and thus he can build the software from beginning in a better, much more organized way that it would be easy to modify and upgrade later on.

This happens quite often to beginners and for that reason it is good to know some common graphical programming rules. They are recommended to follow from the beginning of graphical code development in order to avoid such problems and save a lot of valuable time later on when the code grows.

When making software where very fast execution is required, (eq. handling fast data stream in real time) a detailed knowledge of memory and execution time requirements is essential. By knowing how LabVIEW handles data, one can build the program in a way that handles the data effectively without any unnecessary memory or execution overhead.

Software that interacts with the end user needs a kind of a user interface, which is one of the most important features of LabVIEW. The controls and indicators on front panel even substitute variables in text-based languages. User interface thus grows together with program code and cannot be apart from each other. The execution flow of a simple LabVIEW code is basically linear from data source over some processing nodes to the

indicators with regard to data flow. Advanced software usually has to provide more functionality by executing various pieces of code when pressing buttons or performing other actions on front panel. This is well handled in LabVIEW by Event structure, which is constantly gaining more and more functionality since version 7.1. Nevertheless, there are several issues still present with regard to the responsiveness of the user interface when time consuming operations are performed in the background or in the case of slow or even none response of hardware, such as external instrument.

2. Graphical programming techniques for better VI performance

An efficient LabVIEW application is designed without unnecessary operations, with minimal memory occupation including code, data, block diagram and front panel, GUI updates and data manipulations. The VI profiling tools in LabVIEW offer a detailed overview of execution time and memory occupation of the main VI and its SubVIs and should be used to find and correct all execution issues of designed application. Even though graphical programming looks trivial and bullet proof many rules are good to follow in order to achieve memory efficient and fast code.

2.1 Execution order

The execution order is solely defined by dataflow and not by position of the nodes in the diagram as one would intuitively presume. If no data dependency exists to determine the dataflow, a sequence structure can be implemented for that purpose, as shown in Fig. 1.

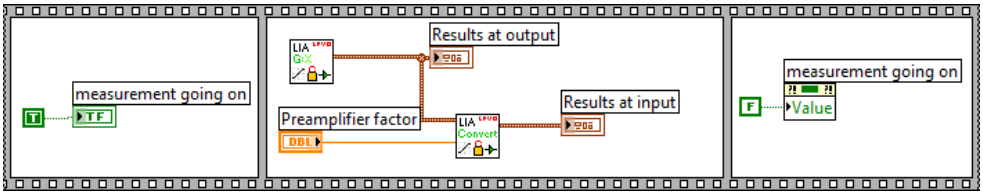


Fig. 1. Determining the execution order with flat sequence structure

In the presented case an indicator is lit before the measurement starts. The indicator goes off when data collection and calculation is finished. In such a case, the use of a sequence structure is appropriate.

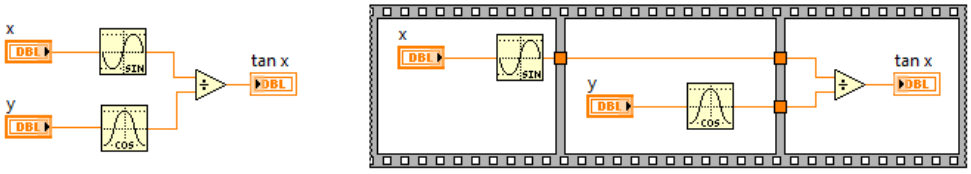


Fig. 2. By using sequence structure (right) the execution order is forced undermining possible data execution parallelism (left)

If the execution order is not important it is undesirable to use sequence structures just to save space in the diagram (e.g. stacked sequence - in Fig. 2 flat sequence is shown for better readability), since they undermine data flow principals and reduce the efficiency of code optimisation in the scope of LabVIEW's compiler. LabVIEW can distribute execution of parallel data paths applying them to more processor threads making code execution much

faster. Therefore it is advisory to use native dataflow principle to determine the execution order and leave parallel data paths if possible. Parallel data paths are desirable to use but one has to be careful if specific execution order is required.
Another chance of messing with dataflow is using local and global variables or property nodes. They can be used to write and read data of front panel controls and indicators at any desirable place and time in the block diagram. They make the code readability much harder and even cause some undesired dataflow issues leading to race conditions and software misbehaviour (Fig. 3).

Fig. 3. A simple example of race condition using local variables that are changed in two independently running loops

If no particular data is passed to the nodes to be executed next one can use an error wire instead of sequence structure to define dataflow order. For that all used SubVIs should have error inputs and outputs at least. Besides defining dataflow direction, a consistent usage of error handling greatly improves reliability and responsiveness of software, especially when software execution depends on external hardware.

2.2 Execution speed

When working with LabVIEW it is quite easy to forget how data is handled and how they are stored in memory. This is often one of the reasons for slow execution speed that can be dependant of many factors such as time elapsed since last program launch, amount of data already processed and similar.
In order to explore the effectiveness of using different variables in LabVIEW a test was performed, where the simple algorithm was repeatedly executed in a loop and the total loop execution was measured using "Tick count" functions just before and after the loop by using sequence structure, as shown in Fig. 4. The number of loop iterations in our case was set to one million times so that the time difference in ms scale could be measured accurately enough and that execution time of "Tick count" function is negligible with respect to the total loop execution time.

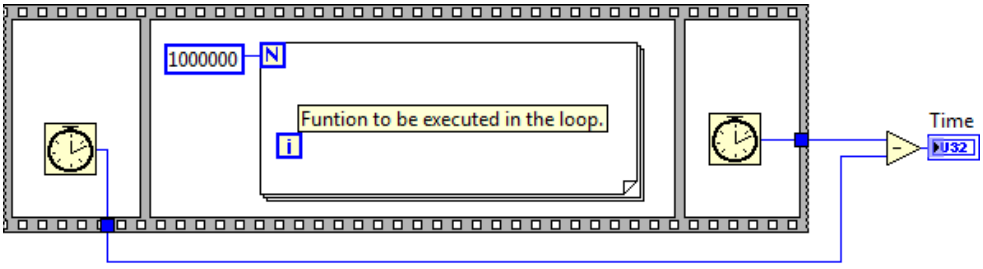


Fig. 4. Experiment of measuring execution time of one loop iteration

The execution priority of the VI was set to Highest (time critical priority) in order to eliminate the influence of operating system as much as possible. During the execution speed test no other programs were running and no disk or mouse activities were performed. The computer used for test has Intel Core i540 2.53 GHz processor, 4 GB RAM and Nvidia NVC 5100M graphic adaptor. The test was performed in 64-bit LabVIEW 2010 running on 64-bit Windows 7 Enterprise operating system. A test algorithm was the following: A random number between 0 and 1 is generated inside the loop and compared to a pre-set test value. In each loop iteration occurrences of test result are counted according to the outcome by two separate variables.
Beginners frequently use local and global variables in LabVIEW as they are used to text based programming languages. Besides already mentioned data flow undermining they are also slower to access and they increase memory occupation. Each local or global variable makes a copy of data of original control or indicator in memory, which becomes important when the data occupies large memory blocks.
Initially, front panel indicators named "less" and "more or equal" were used as counters (Fig. 5.). Local variables are used to read data from indicators. The VI was executed 100 times to get a good approximation of the average execution time. In this case the average time for 1 loop iteration was 680 ns.

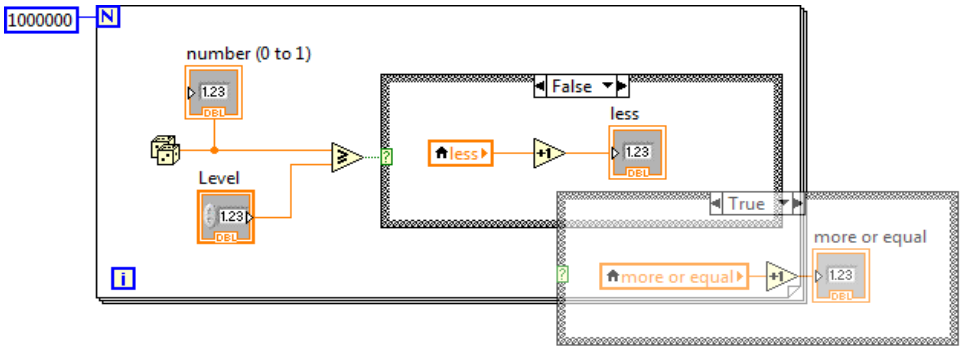


Fig. 5. An example of using local variables to read values of front panel indicators

Another possibility to read or write values of front panel controls and indicators is to use their "Value" property nodes. By that no additional memory allocation is needed to store variable data, since you write or read directly to variables' memory block. The downside of using property nodes is that they turn out to be extremely slow. By replacing local variable with "Value" property nodes in the presented case slowed down the execution to enormous 300 µs, which is much slower than using local variables. For that reason it is strongly advised not to use property nodes in places where fast execution is required.
Nevertheless, in many cases it is possible to avoid using local variables as well. A better solution of this simple problem is to use shift registers to store data instead, as shown in Fig. 6. They are inherent loop structures and thus perform much faster. Additionally, no front panel activities are necessary to update the counters' values. Measured time needed for one loop iteration was 83 ns on average. Further optimisation is possible by removing the indicator "number 0 to 1" out of the loop. Displaying this number inside the loop for each iteration is meaningless since no one can perceive updates at such rate. Execution time of one loop iteration thus drops significantly to only 41 ns. On the contrary, if we replace it with a more complex indicator e.g. Waveform Chart, the execution time would increase to 133 ns.

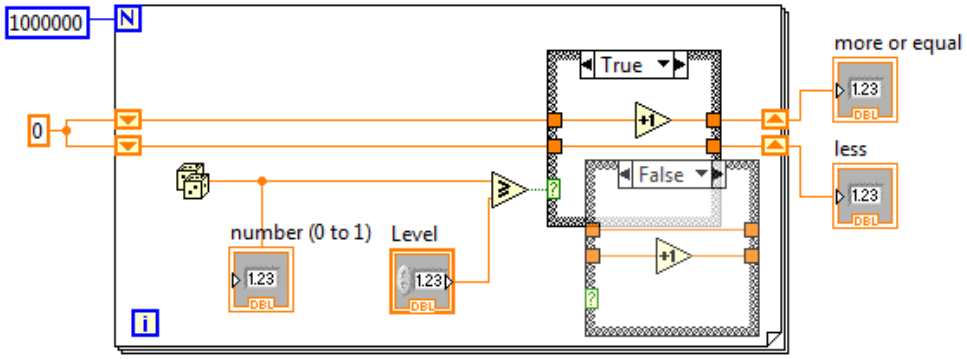


Fig. 6. An example of using shift registers to store values from previous loop iteration

When choosing data type the least time and memory consuming variable type that still complies with the algorithm should be used. If we replace the type of both counters from Double to 32-bit Integer, the execution time decreases from 41 ns to 35 ns. Finally, if "Level" control is not required to change during the loop iteration it is better to put it outside the loop so the control is read only once and after that its value is read from the input tunnel of the loop in each iteration. This further decreases the execution time to 33 ns.
It is important to avoid all unnecessary operations that are performed repeatedly within loops. All constants and calculations that are placed in a loop and result in constant values should be put outside. If possible also avoid data type coercions (marked by small red dots) since they insert additional operation and extra memory buffer that stores the new variables' data.

2.3 Memory management

Memory and disk operations are the main source of latencies in computes. If memory requirements are known in advance, memory can be allocated statically. When you launch the application, memory is allocated according to known global memory requirements of the application and it remains allocated while the application runs. Static memory allocation is simple to work with because the compiler handles all the details.
However, in most application it is impossible to determine what would be the memory requirement in advance. With dynamic memory allocation, memory is reserved when needed and freed afterwards when not used anymore. Dynamic allocation though requires more processor time than static, especially if memory consumption grows during the execution of an application, which results in slower execution speed and fragmented memory. LabVIEW utilises special Memory management module, which is a set special routines for allocating and deallocating memory. It can statically or dynamically allocate, manipulate and deallocate memory as needed and the developer doesn't have to concern about that as would be the case of using conventional C or C++ programming. However, although LabVIEW takes care of all memory management details, a serious developer should be aware of when the memory allocation is performed and how to tailor the code in the way that the particular data memory is reused instead of copied to a new allocated memory block.
Large amount of data is usually stored in a form of arrays or strings, where string is physically stored as an array of unsigned 8-bit integer. Both data types are stored in

continuous memory blocks. Conversion of string to unsigned 8-bit array or back is performed by function "String to Byte Array" or "Byte Array to String" which does not actually change the data stored in memory but only changes its representation. This operation is similar to "Type Cast" function which changes the memory interpretation to or from any kind of variable type. In general type casting is very useful to interpret the data in a binary format that are usually read from external devices, e.g. instruments. Binary data transfer is often used since it greatly improves the utilisation of the communication bandwidth of the bus, such as RS-232 or GPIB. However, typecasting does the same operation as flatten to string and unflatten from string and the new temporary buffer is allocated if data have to be reinterpreted and this cannot be avoided even by "In Place Element" structure that is intended to prevent new buffer allocations in the data flow.

When manipulating with arrays LabVIEW can utilise the same memory block of the source array to store the result, if possible. This is very important if the size of array is large saving a lot of time and memory needed for memory allocation and data copying. However, this is not always possible and depends also on the developer's approach of problem solving.

LabVIEW 2010 offers a tool called Show Buffer Allocations, found under Tools->Profile menu. By this tool you can actually see at which points in the diagram new memory buffers are allocated and let the developer to tailor his software in a way that allocated memory is reused more efficiently.

LabVIEW inherently tries to minimize new buffer allocations by analysing the data flow. An example in Fig. 7 shows two almost identical cases. An 8-bit integer array is initialised and after that each element is multiplied with a "Numeric" control value. A case in the right hand side uses 32-bit integer type of "Numeric" control to multiply the array elements. This requires array type conversion to 32-bit integer data type followed by a new memory allocation to store it. If the left diagram requires 1 MB to process the array data and 1 MB to display it (2 MB all together), the right one requires 1 MB for initialised array, 4 times that much to convert it to 32-bit integer and the same amount to display it (9 MB all together).

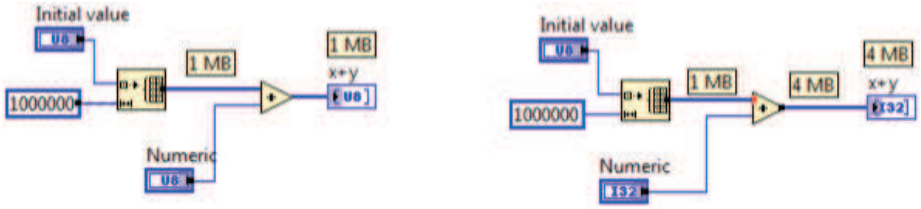


Fig. 7. An example of data type conversion (red dot at the upper input of addition function) requiring new buffer allocation (black square at the output of addition function)

It is clearly seen from results in Fig. 7 that one should avoid type conversion if large amount of data is processed. Developer should use consistent data types for arrays and watch for coercion dots when passing data to functions. Furthermore, displaying large arrays and strings on front panel also increases memory consumption since the array indicator uses its own memory buffer for data storage. In order to decrease memory consumption one should avoid using front panel indicators to store large amount of data. If the VI is used as a SubVI front panel should stay closed if not used thus the required memory block for front panel objects would not be allocated. However, if some property nodes connected to any front panel object are used in the diagram, the front panel memory allocates completely. Since

LabVIEW also uses front panel information to determine possible optimizations, wiring front panel controls holding large amount of data to the connector pane can reduce the number of buffer allocations.
LabVIEW tries to minimise the number of buffers according to the dataflow and functions used to process the data. If dataflow fans out to several branches the new buffer allocations are made according to the node functions that these branches are connected to. An example shown in Fig. 8 manipulates the same array data by functions in parallel.

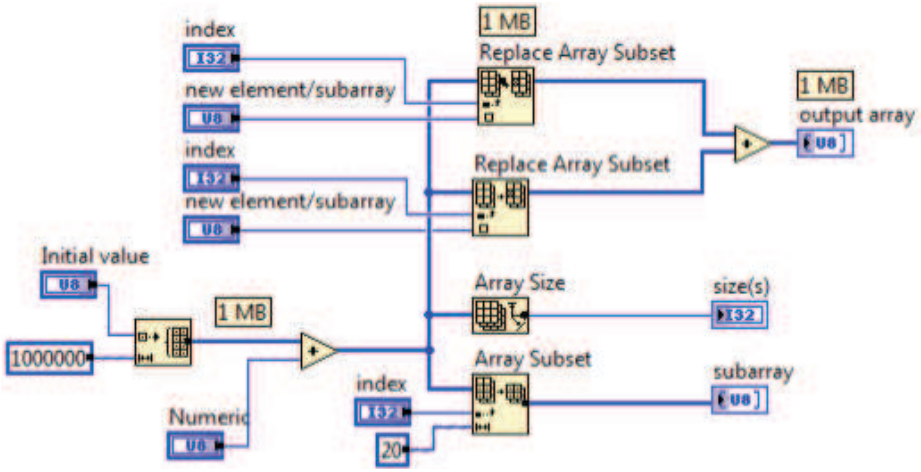


Fig. 8. An example of data path fan out and possible buffer allocations (denoted as small black squares in the block diagram)

Functions that do not alter content of the array can operate over the same buffer (Array Size, Array Subset, …) and allocate their own buffers to store their output data, which are usually smaller. The functions that alter the content of the array can also operate on the same buffer, if possible. Since array data at fan out flows in parallel, no execution order is forced in any way and LabVIEW can decide which function to execute first. In order to retain the required functionality without new buffer allocations LabVIEW chooses first to execute functions that don't alter the array data and afterwards all others. However, if two functions (e.g. Replace Array Subset) that alter the data are used in parallel an additional buffer copy has to be allocated since outputs of both functions must be consistent with the input data. Thus the example shown in Fig. 9 uses 3MB of memory to store the data. If only one "Replace array subset" was used for the same purpose, the additional 1MB buffer would not be allocated.
When designing the block diagram, a designer has to be aware of any areas where the size of data changes frequently in a flow, such us increasing of array or string size. Since strings and arrays are stored in a continuous memory blocks, at each node that increase the data size a new memory allocation occurs leading to fragmented memory which gives a lot of work to LabVIEW Memory Manager.
For array initialisation it is most convenient to use "Initialise Array" function if elements of constant value are permitted. If array values have to be dynamically generated the most efficient way is to use loops with output tunnels with enabled indexing. If loop count is known in advance the memory block of the requested size is allocated only once just before

entering the loop. If the array size is not known in advance (e.g. wiring the control to the count terminal), the memory is allocated dynamically during the loop iterations. However LabVIEW is smart enough not to allocate one array element at each loop iteration but rather allocates larger blocks of memory at each *n*-th run of the loop reducing the time overhead and memory fragmentation to minimum.

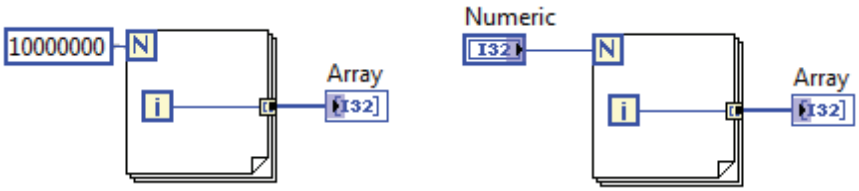


Fig. 9. Examples of the most efficient ways of creating arrays.

One of the problems that often occur is reading some data from external instrument in a continuous manner and storing it to an array. In such cases usually the amount of data is not known in advance and thus memory buffer cannot be preallocated. Fig. 10 shows a simple solution that is not recommended if large amount of data is expected to be read. The "Build array" function continuously increases the array size by allocating new and new chunks of memory as data are passed from the "485 read" SubVI. After a while the array that is passed by shift registers through loop iterations would be stored in a fragmented memory that would require more and more time for LabVIEW Memory Manager to allocate. This would finally start to slow down loop execution resulting in buffer overflows and possible data loss.

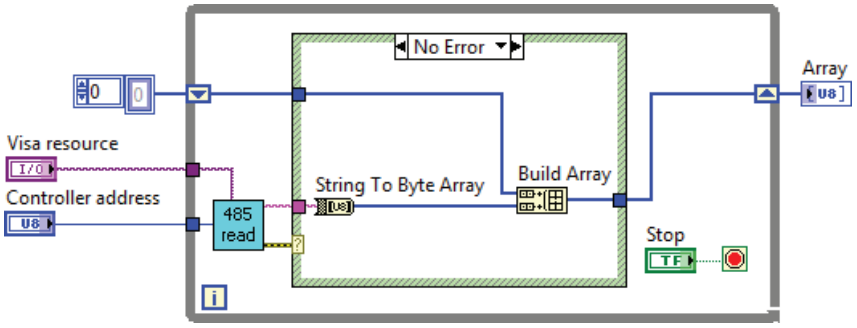


Fig. 10. The easiest but the least efficient solution to continuous data reading and storage in a dynamically allocated array.

In such cases it is advisory to avoid applying concatenate string and build array functions to large strings or arrays since every execution changes the size of data and requires new buffer allocation. A developer should use "replace string/array subset" functions instead and initialise data at the beginning if size is known in advance. If not, then increase the size of the array in large increments when necessary as the loop runs and use "replace array subset" otherwise. An example of improved algorithm for continuous data storage is shown in Fig. 11.

In this solution the current array write index has to be stored separately for "Replace Array Subset" function. Initially an empty array of certain size is allocated and passed to shift

register. After each execution of "485 read" SubVI the size of the stored data in the array together with the size of the returned data is calculated and compared to the total array size. If the returned data do not fit in the array the "Build array" function is called to increase the array. Initial array size should be chosen according to the expected data size, returned by "485 read" function and total amount of data. Smaller value means more often buffer allocations while larger value can lead to unnecessarily oversized buffer for array data storage.

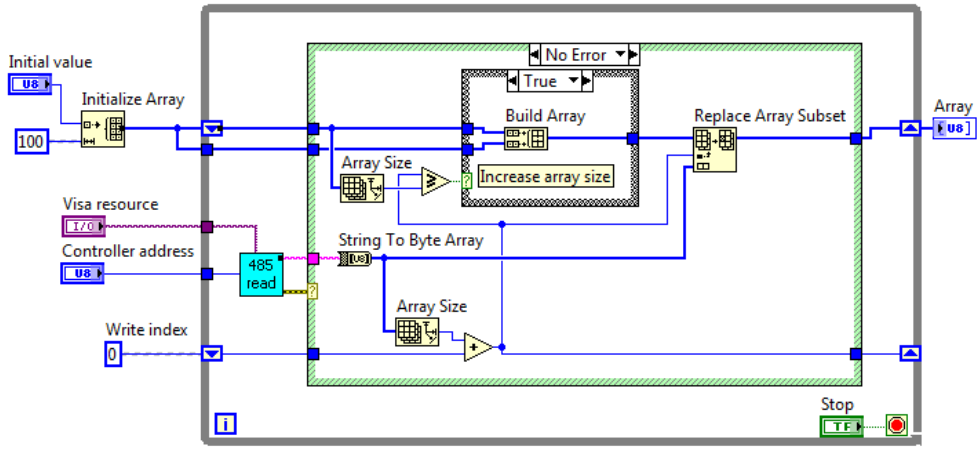


Fig. 11. An efficient way of dynamic memory allocation for data that are continuously read.

3. Organising the LabVIEW code

Dataflow concept of LabVIEW's code execution is very close to the engineering way of thinking and solving problems. Nevertheless, no one can expect a successful result without basic problem solving skills, regardless of how easy and user friendly the software environment is. LabVIEW offers many ways to solve the problem from beginning to an end and it is a developer's choice which way to follow. This is the first trap in which one can be caught. It is smart to ask yourself a question: Which way to take? And the answer would be: "It depends of the problem."

Let's presume that a problem is well defined. By knowing the problem one should have a clear idea of what are the required functionalities of the software. The requirements are usually formed together with an end user at the beginning but they can also evolve during the software development. A very common problem that people have is that they start to develop software by not knowing exactly what the requirements are. And the result is often a wasted effort when the software has to be partly or even completely rewritten when some new features need to be added that are slightly out of the chosen concept. Therefore, the first step in LabVIEW application development should be a clear definition of all requirements that are completely understood, agreed, approved by all partners and documented.

Starting from that point a developer can identify the inputs, outputs and finally design the algorithm. When the algorithm is well defined it is the right time to run the LabVIEW and start to translate the algorithm into code. The developer has to choose the most efficient way of applying it in to a graphical code, maximising readability and performance of the code.

LabVIEW provides several options to choose from; each optimal for a specific target application.

3.1 Linear data flow

If the problem is trivial and the software is intended for one's own use, linear data flow programming style can be used. This suits for cases, where you have some input data that have to be processed and results written out. The algorithm can have some loops or other structures implemented, but the solely task of the software is, that it executes once and then terminates, passing results to some outputs. From definition of dataflow execution concept it is obvious that every diagram node executes when all of its inputs are present and after the execution it passes data forward to other nodes. The dataflow is linear and defined by wire connections. In Fig. 12 an example of a linear dataflow code is shown that reads one single measurement from data acquisition (DAQ) interface.

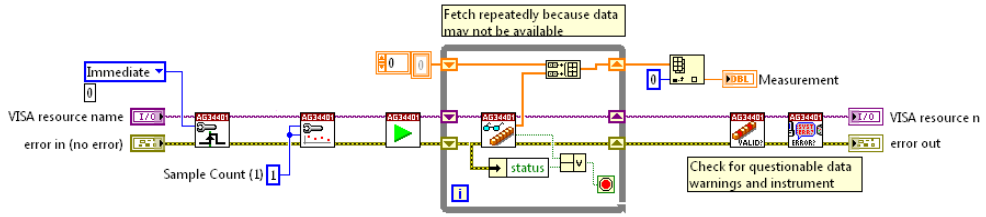


Fig. 12. Example of a linear dataflow programming style (taken from NI DAQ example)

Linear data flow style is usually the first approach that beginners start with regardless of the nature of the problem to be solved. In most cases this leads to several execution, memory and readability problems when the code evolves. In Fig. 13 an example is presented, showing such software, made in the linear data flow programming style. It evolved from simple task as shown in Fig. 12 and then expanded adding various features later on.

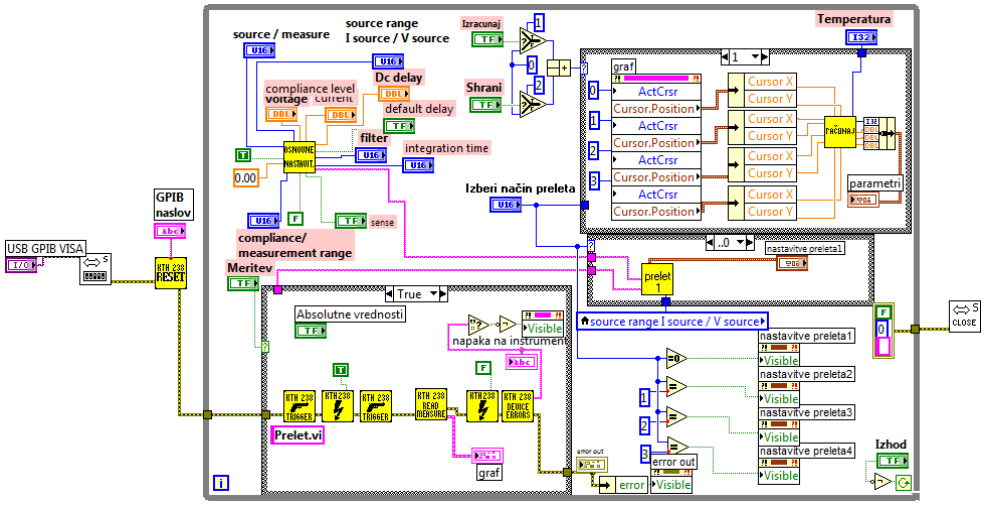


Fig. 13. External instrument driver via GPIB by using linear data flow programming style

This block diagram is well organised, all building blocks are aligned, dataflow direction goes from left to right as we are used to and errors are properly handled, but it is still pretty clumsy in the sense of its expandability. The software development started as a simple task to control an external instrument via GPIB and return measurement results. Although not intended at beginning it finally resulted in a top level application that interacts with end user via controls and indicators on front panel. Unsuitable design used at the very beginning resulted in a bad user experience:

- Initialisation of the GPIB interface is outside the main loop. The software has to be restarted in the case of GPIB connection error or just to update GPIB device changes.
- All user interfacing is done via buttons on front panel which is not common in modern computer application (e.g. File saving).
- The main loop is polling the values of front panel controls as fast as it can, thus taking much of the processor time that could be available to other applications. The same fact states for updating front panel indicators.

For that reason this software was completely overwritten making majority of the invested effort fruitless.

3.2 Handling events

Events represent the core functionality of any contemporary operating system. Introducing event handling in LabVIEW is almost inevitable when the target software has emphasis on user interface. Such an example is shown in Fig. 14. The event structure, which handles events that are generated from user activities on front panel, is placed in the main program loop. The software controls full colour LED reflector by sending appropriate string commands via USB interface.

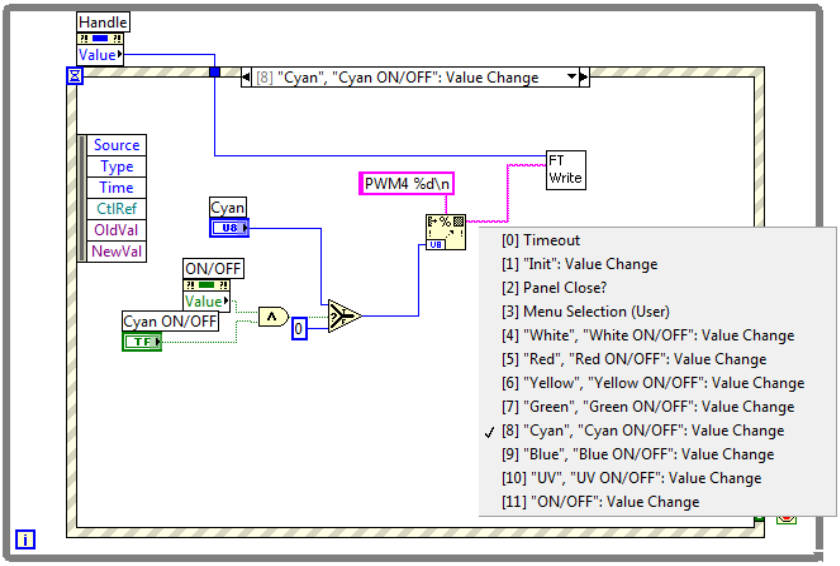


Fig. 14. Example of handling the user interface activities by event structure.

Each event case is assigned to one or more events that can be triggered by front panel object by user or code itself. In this particular case seven sliders control seven LEDs of different

colours together with ON/OFF knobs for each colour. USB write function (FT write) ends immediately if the write operation is not successful making this software responsive even if USB device is malfunctioning. USB interface is initialised in the "init:Value Change" case, which is assigned to a hidden "init" button. The dataflow is made in such a way that the "init:Value Change" event is triggered automatically when the software starts by "Value signaling" property node.
If the USB interface initialisation fails, it runs the same event again until initialisation completes or user closes the software as shown in Fig. 15. Making user defined events in such a way has lot of possibilities. Besides (hidden) front panel objects that can be used solely for software user event triggering LabVIEW offers special user event VIs that can generate software events. This makes event handling very versatile. The problem however occurs if a piece of code inside some event case gets stuck. This often happens trying to read data from external hardware that does not respond. The whole event structure waits till the individual event case completes and it cannot be interrupted with other events pending. Certainly, it is wise to implement a timeout in the functions that read data from external instruments. However, this only partly solves the problem since timeouts for some instruments are required to be in the range of 10 seconds or more.

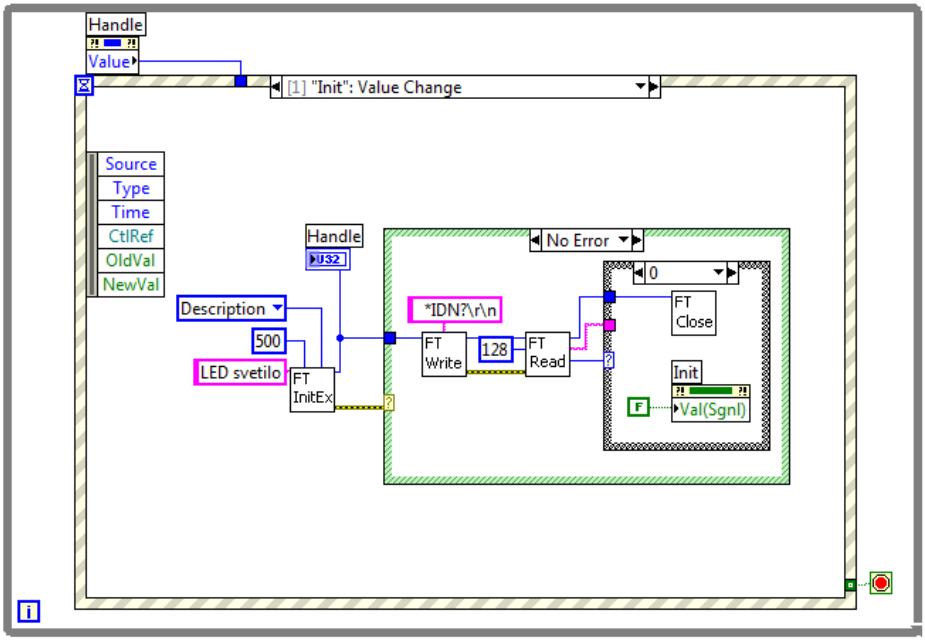


Fig. 15. "Init" case, where the USB interface is initialised and if it fails, the same event is called again by "Value signaling" property node of "Init" button

3.3 State machine

State machines in LabVIEW basically do not differ from other implementations. They represent the solution to a problem in a form of a state diagram, providing most clear and robust solution. In contrast to event handling, state machines are meant for fast execution

and data content handling in cases, where user's interaction is small or even not present. The implementation of a state machine is shown in Fig 16. It consist of a while loop, where for each state a separate case is defined. The state is stored in a shift register, which can hold any type of data in general. Nevertheless, the most appropriate data type would be (enumerated) integer allowing fastest loop execution and lowest memory occupation. The inner while loop passes the state value from previous to next iteration together with other relevant data via shift registers. In each case a code is executed according to the state specifications and the next state number is chosen based on the results.

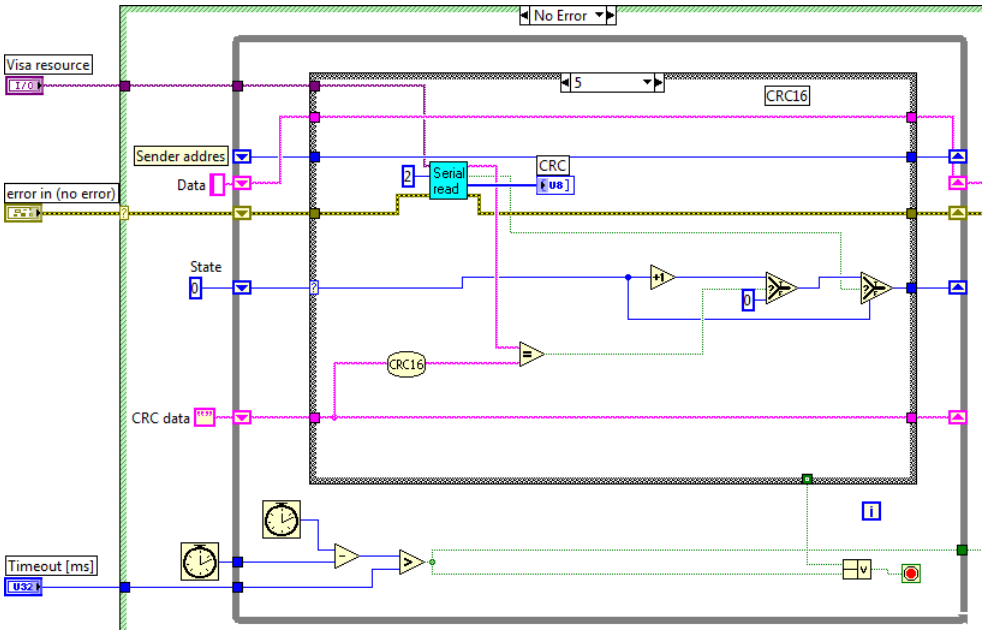


Fig. 16. Example of a state machine for handling RS-485 serial communication protocol

This particular state diagram handles data that are read from RS-485 port according to communication protocol requirements. The presented state reads 16-bit CRC value from input data stream and compares it to the calculated value. If the results match the next state is chosen, otherwise the state returns to 0, at which it waits for the first byte of the communication frame. If the "Serial read" function returns no data i.e. the data have not arrived yet, the same state repeats. Timeout is implemented by reading time in ms before entering the loop for a reference time and comparing it to a time inside loop iterations. Since the timer function returns ms timer value in 32-bit unsigned integer format it can overflow. For that a subtraction is performed of both timer values prior comparison thus eliminating the overflow problems.

By using state machine programming style it is very easy to change or add some new features, but it is not convenient for handling user interface actions. The main reason is that in every state all relevant buttons have to be handled resulting in unnecessary processor load. Furthermore, if several states are present it is a lot of work to add additional user interface features. For user interface event handling a separate default state can be defined

in which all buttons and other front panel object states can be polled. In such a way the user interface actions are polled only when no other operation is going on. Such an approach is easier to implement but prone to become irresponsive if state machine doesn't enter the default state frequently enough or stays in some other state for a longer time.

3.4 Master-slave design style

Most of the main applications require user interface handling with fast response and also fast code execution in the background. For that at least two parallel dataflows are required. They are arranged in two main loops, where the first (master loop) handles front panel activities usually by event structure and second (slave loop) executes all other software activities. An example is shown in Fig. 17. To assure parallel executions of both loops they shouldn't be connected by wires in the way that dataflow would force sequential execution of loops. In order to pass data between both loops the queue functions are most appropriate to use. The only connection between both loops is the obtained queue reference wire, which is the input data of both queues and thus parallel execution is allowed. Although the queue data can be of any type, the easiest way would be a string type in which any text message can be passed or an enumerated integer.

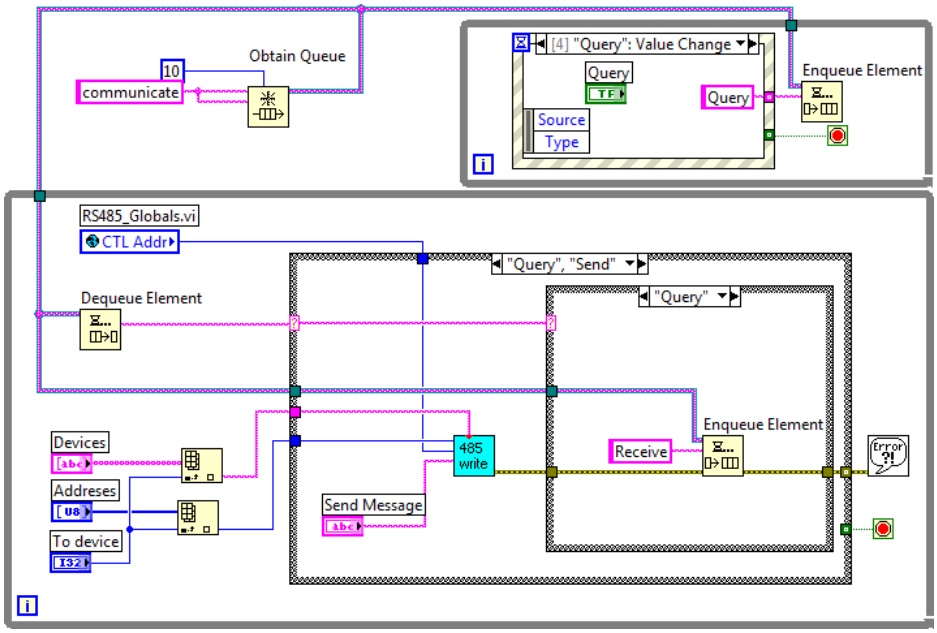


Fig. 17. Master-Slave design with two loops that run in parallel

In Fig. 17 a top level application is presented that acts as a communication terminal over RS-485 bus. Case structure in the slave loop executes different cases according to the de-queued string. Strings that are used as commands can be queued anywhere. In this particular case a value change event of query button (Button pressed) sends "Query" command to slave loop, where a "Query" case is executed. The same case executes also when "Send" command arrives, since both are required first to send message over RS-485 bus. But additionally in

the case of "Query", the "Receive" command is additionally queued if no error occurs. In the next run of slave loop the "Receive" string is dequeued and a corresponding case is executed. The advantage of this approach is that the front panel event handling is disconnected from code execution, allowing that both run independently. The problem can be divided into simple actions that can be easily implemented in the slave loop and these actions are then executed according to the front panel events and results of previous executions (e.g. error outputs or other). This makes such an approach much more versatile since many combinations of different cases and events are possible. Furthermore, if execution of a certain case freezes for longer time (waiting for receive communication timeout), the master loop is still running and captures front panel events. The problem of software responsiveness however is only partially solved since the slave loop still doesn't respond until the function in the failed case is timed out.
Thus we need to implement a way of breaking the execution of a SubVI, which waits for timeout. One way to do that is to read the queue size, i.e. number of elements pending for de-queuing. If the queue size is a larger than zero it means that a front panel event is pending and the current execution in the slave loop should break and continue to service another string in a queue. But this approach allows only one queued message at a time which can be quite a limitation in some cases. A special break message can also be reserved for slave loop breaking at which any slave function would break the execution. It can be queued from master loop by a special cancel key or event on exiting the software or similar.

4. Conclusion

Programming in LabVIEW looks easy, since the dataflow, functionality of execution nodes and data types of wire connections are visually presented. Peoples' graphical perception is much more efficient and faster way of acquiring information comparing it to textual, which makes graphical programming so attractive. However, an experienced LabVIEW developer should be aware of all possible pitfalls that are hidden behind the neat wires and functions. This chapter provides beginners and advanced developers with some general rules illustrated by examples taken out from real life applications. A developer should check his software with LabVIEW Profile tools on regular basis, isolate and correct the issues. It is not good to waste time striving to perfect every detail but rather isolate main problems that slow down the software execution and increases memory usage. These problems lie in complex or large memory data structures that are improperly handled and in frequently executed parts of the software that perform unnecessary calculations.

5. References

Bishop, R. H. (2004). *Learning with Labview 7 Express*, Pearson Prentice-Hall Int., ISBN 0-13-117605-6

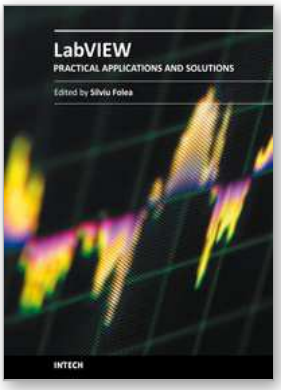
Gorup, Ž. (2006).*Uvod v Labview*, Fakultetazaelektrotehniko, ISBN 961-243-036-5, Ljubljana, Slovenija

Hogg, S. (November 2010). What is LabVIEW? In: NI developers zone, Available from http://zone.ni.com/devzone/cda/pub/p/id/1141

Kerrry, E. (April 2011) Memory management in LabVIEW 8.5, In: NI developers zone, Available from http://zone.ni.com/devzone/cda/pub/p/id/241

NI Tutorial. (October 2006). LabVIEW and Hyperthreading, In: NI developers zone, Available from http://zone.ni.com/devzone/cda/tut/p/id/3558

Travis, J.&Kring, J. (2006).*LabVIEW for Everyone: Graphical Programming Made Easy and Fun (3rd Edition)*, Prentice hall, ISBN 0-13-185672-3



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.

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