Using Queued State Machines for Data Acquisition

Georgi Nikolov[1] and Boyanka Nikolova[2]

Abstract – **An approach for design development and implementation of data acquisitioning systems controlled by LabVIEW queued state machine architecture is suggested in this paper. The building blocks and functionality of such innovative type of state machines are described. A design steps based on unified modeling language are summarized, considered and introduced. In order to proof usability of suggested approach, in the end of the paper results achieved by developed temperature monitoring system are appended.**

Keywords – **Queued state machine, Virtual instrumentation, Paralleled DAQ processing, LabVIEW programming.**

I. Introduction

By definition a finite state machine is model of behavior composed of finite number of states, transitions between those states, and actions. The classical state machine is made up of entry, exit, input, and transition actions. This abstract machine defines a finite set of conditions of existence, a set of behaviors or actions performed in each of those states, and a set of events which cause changes in states according to a finite and well-defined rule set. State machines are the primary means within the Unified Modeling Language (UML) for capturing complex dynamic behavior. They are described by comprehensive set of notations named statecharts [2, 3].

In addition classical state machines are the most highly touted LabVIEW design patterns. There are many variations, most of which consist of a *Case structure* within a *While Loop*, with a *Shift register* or messaging construct wired to the case selector terminal. Each case of the *Case structure* contains a subdiagram corresponding to a state of the application. The case selector is an integer, string, or enumerated data type identifying the states. The *Shift register* or messaging construct passes the next state selection from a previous case to the selector terminal in the next loop iteration. In a typical application, the state selection is determined by an event on the user interface, by a step in a sequential test or measurement routine, or from the result of a previous state.

The Classic State Machine is appropriate for programs and routines of low to medium complexity but is not flexible enough for complex virtual instruments (VI), top-level programs, and graphical user interfaces. Alternative state machine implementations that utilize queues and *Event structures* are more functional and efficient for these applications.

Over the last three years, the queued state machine (QSM) has gained support and widespread use in large LabVIEW based applications in the developer community [1, 5]. QSM architecture, is one essential architecture that significantly facilitates programming advanced LabVIEW based projects. A common application for the QSM architecture is in programming applications that send commands for asynchronous processing in a parallel loop so that event cases can exit code execution quickly and avoid lockup. Another application is in multiple parallel virtual instruments programming such as in parallel data acquisition, alarm monitoring, and results analysis, where this method empowers any parallel application to send and receive commands and data across other parallel applications with no data loss.

The intermediate to advanced nature of the objects that make up the queued state machines architecture, taking full advantage of this template requires detailed knowledge of the its various characteristic design aspects. Especially attention must be kept when in project are involved data acquisition (DAQ) drivers. This paper suggest, illustrates, and describes an approach to use the various elements of the QSM architecture to design and build up parallel running data acquisition application in LabVIEW environment.

II. Queued State Machines

A. A High Level Layout of QSM Architecture

Generally, a queued state machine is a LabVIEW programming method that sends commands and other data from multiple source points, such as from user events and from one or more parallel processes, and gets these handled in one state machine process in the order in which they were added to the queue [1]. With such approach, a state can determine not only the next state to be performed, but a series of states that must be performed in order. The series of states that must be executed are placed in a queue. The states are removed from the queue one at a time and executed in the order they were inserted into the queue.

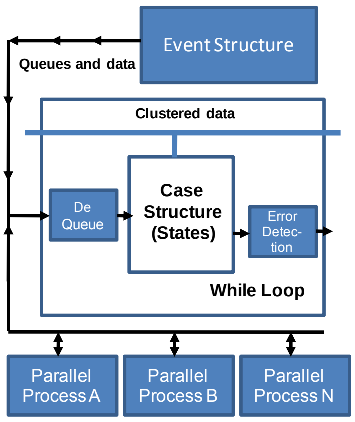

Fig. 1. High Level QSM Architecture

[1]Georgi Nikolov is with the Faculty of Electronics and Technologies at Technical University of Sofia, 8 Kl. Ohridski Blvd, Sofia 1000, Bulgaria, E-mail: gnikolov@tu-sofia.bg.

[2]Boyanka Nikolova is with the Faculty of Telecommunications at Technical University of Sofia, 8 Kl. Ohridski Blvd, Sofia 1000, Bulgaria. E-mail: bnikol@tu-sofia.bg.

In figure 1 is shown the simplest high-level illustration of the QSM design. The base building blocks of the architecture are *Event structure*, *Queues* and data, *While loop*, *Dequeuing* element, *Case structure*, *Error handling* element and one or more parallel processes objects. *Event structure* and parallel processes objects are the multiple producer processes responsible for sourcing commands and data and adding them to the queue. The *Dequeued* element removes commands and data from the queue and acts on these in *Case structure* in the order added to the queue. The new element in this architecture introduced in [1] is that it uses a queue element data type consisting of a cluster that contains the enumerated type definition bundled together with a *variant*. The enumeration contains the desired state for the case selector as normal. The *variant* is used to pass data from one state to another, using the queue functions instead of shift registers.

Another innovative feature suggested for the QSM architecture from reference [1] is the *shift registered* cluster data flow line that passes through all state machine cases. This flow line avails and allows update of parameters and variables, as needed, inside every state machine case. *Unbundled-by-name* utility is used to access parameters and variables and use the *bundle-by-name* utility to update the same.

Programs (or *SubVIs*) that run in parallel with the main consumer process can be data communications *VIs*, data acquisition *VIs*, results analysis *VIs*, and much more. These parallel *subVIs* primarily behave as producer processes and access the queue reference by name, that is shared with the consumer process. This method of access to the queue reference precludes the need to wire the queue reference to *SubVIs*, which creates transparent routes of communication and simplifies the block diagram

B. Benefits of QMS

There are many benefits of the presented queued state machine architecture according classical state machines. From data acquisition point of view the more important are:

- *Parallel Process Enabler.* This architecture establishes the use of queue references as data messaging pipelines that communicates information between parallel processes in timely manner. This type of communication solves one of the serious challenges in parallel process programming for data acquisition, alarm monitoring and results analysis.
- *Multiple Producer and Single Consumer Points.* In this type of state machine, queue data elements can be added from various points in the code known as *producer* points. However, queue elements are taken out of the queue from only one destination point, called the *consumer* point. This consumer point is considered to be the owner of the queue reference.
- *Global Access to the Queue via the Queue Name.* This means that the queue can be seen by other processes without the need of wiring the program components to a queue reference. Such approach creates transparent routes of communication and greatly simplifies the code.
- *Can be used Run Time Logic.* QSM programs can implement logic to change the latest command sequence by adding commands to the front of the queue or by emptying the queue to reset the program flow and add new commands thereafter.
- *Multi Consumer Queue References Creates a Network of Data Pathways.* Parallel process which themselves use the QSM architecture create a network of communication pathways with multiple producer and consumer points. This allows one parallel process to control multiple parallel processes.

III. QSM Base Design Steps

There are many references and manuals describing in details methods and steps to design classical state machines in unified manner [2, 3]. More of them are directed to textual object oriented programming languages. Concerning graphical languages can be mentioned LabVIEW Statechart Module [4]. With this module is possible to design LabVIEW applications with statechart diagrams, but it is relatively expensive and is not allowed for QSM. In this paper is introduced approach based on unified modeling language for design and develop data acquisition application based on QSM architecture. The presented approach consists from following steps.

A. Build Statechart

Statecharts are a methodology by which complex systems can be specified in an intuitive graphical manner. They enable complex relationships between concurrent states to be formed, through synchronization techniques and decomposition of states. This approach provides a high level of abstraction for designing applications using states, transitions, and events [3]. Statecharts are especially useful for designing a number of asynchronous parallel processes. As example in figure 2 is shown the statechart diagram based on UML notations. This diagram is created by authors in order to develop virtual system for temperature parameters investigation that is explained in next topics.

B. Define Transitions and States

After the statechart diagram is created the next step is to define various transitions, states and actions. The following special features must be considered:

Each transition contains three component – trigger, guard and action. Trigger is events that cause transition, guard is logic that can prevent a transition and action is what happens when transition is established.

Each state contains three types of action – entry, exit and static. Entry is what happens when data get in state, exit defines what happens when data leave, and static describe what happens while data are there.

When transition and states are defined statechart execution must be considered and verified. First is evaluated the trigger and guard logic for the transitions leaving the current state.

On first valid transition is executed the exit action for the current state, follow execution of the transition action, and finally is executed the entry action for all states being transitioned on. If no transitions are valid first is evaluated the trigger and guard logic for all static reactions configured for the current state. The last is executed the action code for all valid reaction.

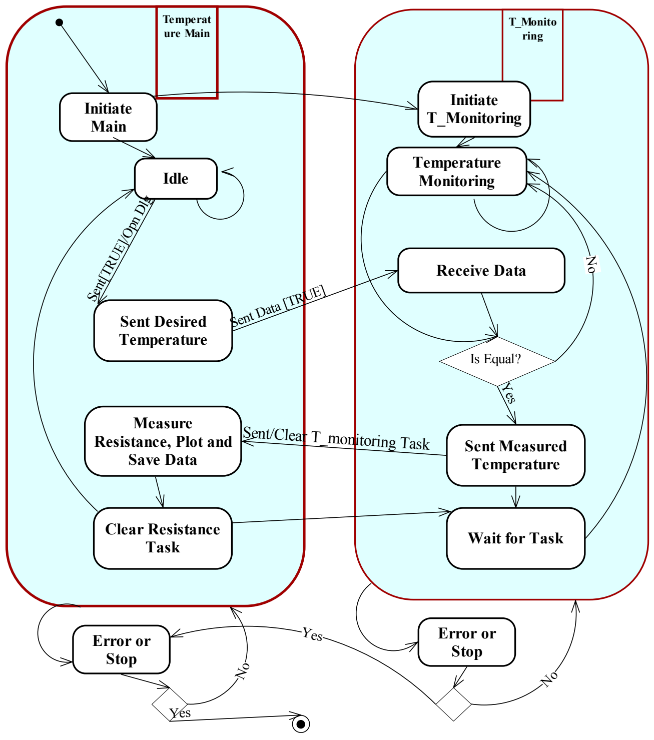

Fig. 2. UML statechart for virtual measurement system

C. Create typedef enumerated constant according defined states

After transition and states are defined the LabVIEW code can be developed. For beginning it is needed to create *typedef* enumerated constants that corresponded to each defined state. The *typedef* enumerated constant enlists chosen names of the state machine cases. Each time a command is added to the queue, the *enum* should be set to the machine's state name which will handle or process the command. It is needed to ensure that the *enum* constant is a copy of a *typedef*-based custom control so to give opportunity to add or remove command items from the *enum* and make changes to all instances of the *typedef* constant throughout all LabVIEW code.

D. Define and create Event structure and Queues sending commands and data

Event structure has one or more diagrams, or event cases, exactly one of which executes when the structure executes. The *Event structure* waits until an event happens on the front panel, then executes the appropriate case to handle that event. The structure is responsible for user defined events generated from user interface.

The LabVIEW queue implementation creates a queue reference of given name, using functions from LabVIEW's queue palette. Subsequent and repeated implementation of this same code will grab an existing queue reference of the specified name. This is typically done to give access of queue reference to *subVIs* which also avoids the need to wire a queue reference to the *subVI*.

E. Configure data acquisition task

A data acquisition task is a collection of one or more virtual channels with scaling, timing, triggering and other properties. Many of the built-in measurement and automation explorer (MAX) constructs are well suited for implementing many common data acquisition programming tasks, such as task creation, input or output operations and task destruction.

Special attention must be kept with DAQ tasks when an asynchronous parallel processes are developed. If the tasks is defined as continuous it is not possible to stop and clear it in order to give up DAQ resources for other process.

F. Develop supporting LabVIEW code

After the designed transitions and states are developed and verified, to complete the LabVIEW programming code or so called block diagram, an additional functions and blocks for data manipulation must be created. This process is strongly dependent upon concrete application and programmer's experience [5].

IV. Investigation of Temperature Parameters Using QSM

In order to verify suggested approach for data acquisitioning using QSM, a virtual system for investigation of resistance versus temperature dependency of various materials was developed. The hardware organization of the system is shown in fig. 3. The idea is when the temperature chamber is turned on DAQ board begin to monitor continuously the temperature in the chamber with analog input 0 (AI0). As temperature sensor the integrated one is used (AD22100). The user define desired temperature values for which the resistance of the device under test (DUT) must be measured. When the current temperature reach specified one the DAQ task is changed, channel 1 (AI1) measure resistance only once and give up resource to temperature monitoring again. With such a way the measured data is reduced only to this needed for investigation.

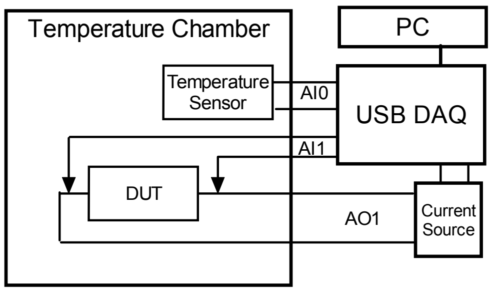

Fig. 3. Hardware organization of the developed system

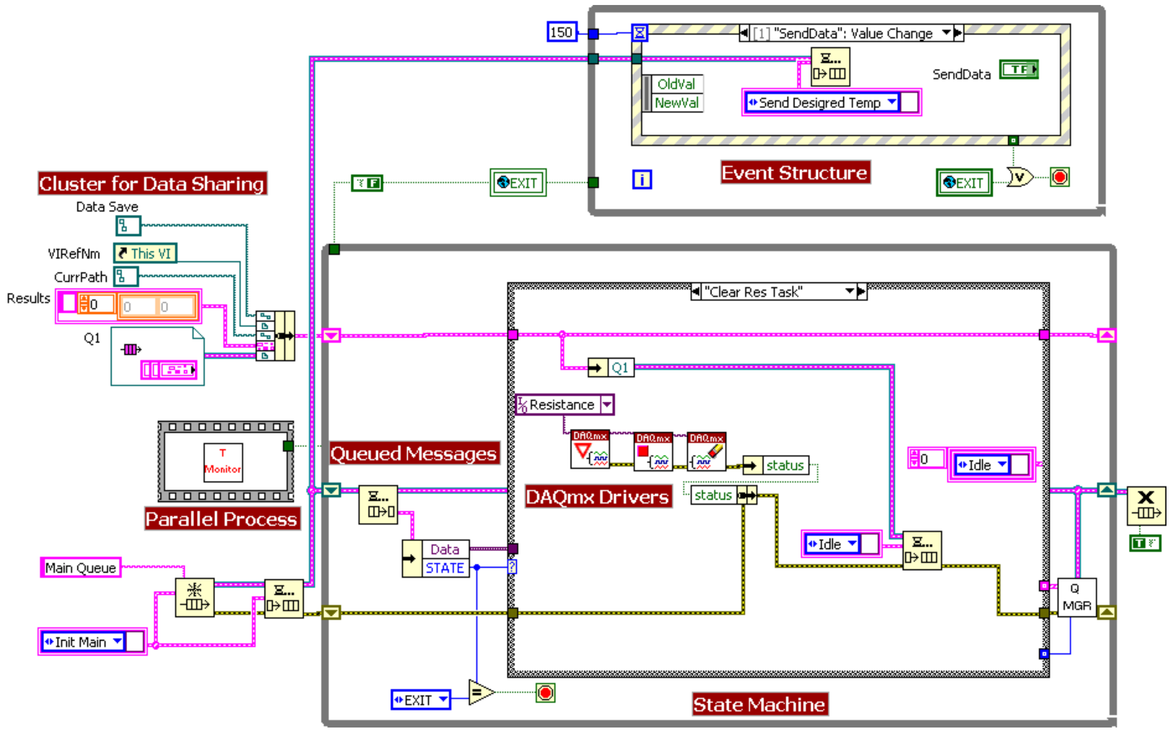

Fig. 4. The LabVIEW block diagram of the resistance measurement process

In addition when measurement data is acquired for given temperature this temperature is removed from the array that guarantee only one resistance measurement for each temperature point. The transitions and states of described process is illustrated in UML diagram from fig. 2. As can be seen there are two asynchronous data acquisition processes that exchange commands and data using queues. The LabVIEW block diagram of the resistance measurement process (Temperature main superstate in left on fig.2) is shown in fig. 4. As can be seen there are various elements from the QSM architecture. The block diagram is relatively simple, and can be easily used as a template for larger scale user interface design. Front panels of both processes are shown in fig 5. The results from investigation of 10 kΩ thermistor (NTC) is shown in order to illustrate usability of the created system.

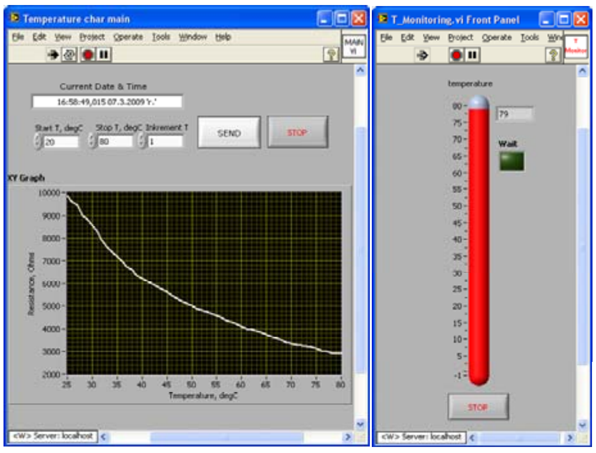

Fig. 5. Front panels of virtual system for investigation of resistance versus temperature dependency

V. Conclusion

Design, development and implementation of virtual measurement system based on queued state machine architecture is presented in this paper. The described approach can be used for various applications where paralleled asynchronous data acquisition processes are required.

Acknowledgement

This investigation has been carried out in the framework of the research projects № Д01-1251/18.12.07 and № 092 ни 044-03.

References

[1] A. Lukindo, “LabVIEW Queued State Machine Architecture”, Expression Flow, 2007, http://expressionflow.com/2007/10/01/.
[2] P. Stevens, Pooley, R., *Using UML. Software Engineering with Objects and Components*, 2nd edition, Addison-Wesley, ISBN-13: 978-32126-967-6, 2006.
[3] D. Drusinsky, *Modeling and Verification Using UML Statecharts,* Elsevier Inc., ISBN 0-7506-7949-2, 2006.
[4] National Instruments, “Build a Hybrid Control System with NI LabVIEW Statechart and Control Design and Simulation Tools”, 2006.
[5] P. A. Blume, *The LabVIEW Style Book*, Prentice Hall, ISBN 0-13-145835-3, 2007.