**LabVIEW Queued State Machine Consumer Producer Architecture**

By: Anthony Lukindo, alukindo@mezintel.com, Mezintel Incorporated, Calgary, Canada

Sept, 2007

---

**Summary**

The Queued State Machine -Producer Consumer Architecture; abbreviated in this article as QSM-PC is one essential architecture that significantly facilitates the programming of mid-sized to advanced LabVIEW-based projects that constitute 100 or more VIs. In light of the intermediate to advanced nature of the objects that make up the QSM-PC architecture, taking full advantage of this design template requires detailed knowledge of the ‘why’ and ‘how’ of the various design aspects that characterizes this template. This article defines, illustrates, and describes at length the various elements of the QSM-PC architecture.

Note that you can access a producer consumer design template that ships with LabVIEW from a VI, or a project menu, by selecting: [File] >> [New...] and then from the ensuing dialog, and in the [Create New] treeview, choose [VI] >> [From Template] >> [Design Patterns] >> [Producer Consumer Design Pattern –Data] option. This article adds useful features in the template that ships with LabVIEW and uses a QSM-PC template that was arrived-at after the study and use of a number of other similar templates. The template proposed here is considered by the author as one of best of breed templates for QSM-PC architecture.

**Background**

Historically, the Queued State Machine programming method evolved from the LabVIEW developer community through articles in the now discontinued publication: “LabVIEW Technical Resource Guide”. The subject was introduced by Jeff Parker (Winter, 1995), and elaborated and modified by Greg Fowler (Spring, 1996) and Lance Butler (Winter, 1996). The National Instruments website hosts various knowledgebase articles and tutorials matching the key word: “Queued State Machine”. National Instruments’ Kevin Hogan, produced a useful presentation and web cast on this same subject matter that continues to be available as of this writing.

**Queued State Machine –Consumer Producer Design Objects**

Generally, a queued state machine is a LabVIEW programming method that sends commands and other data from multiple source points (i.e. producer points), such as from user events and from one or more parallel processes, and gets these handled in one state machine process (i.e. destination consumer point) in the order in which they were added to the queue. Figure 1 is the simplest high-level illustration of the QSM-PC design. Figure 2 adds detail to this illustration, and Figure 3 is the equivalent LabVIEW implementation. These illustrations will be used throughout this article to clarify the features and functions of the QSM-PC architecture.
Figure 1. Shows four objects of high-level QSM –PC architecture. 1 = Queue reference; 2 = User events object (producer of commands and data); 3 = Commands processor (consumer process that handles commands and data); 4 = Parallel SubVI processes (producer of commands and data)
Figure 2. QSM –PC architecture showing data flow and objects inside the consumer process
Figure 3. LabVIEW implementation of the producer consumer architecture. Labeled objects 1 through 4 match those of the high level illustrations in Figure 1 and Figure 2.
A High Level Layout of QSM-PC Architecture

A typical general layout of QSM-PC architecture comprises of the four objects annotated by the encircled numerals 1 to 4 shown in Figure 1. These objects are namely: (1) Queue reference object; (2) User events object (optional for head-less embedded applications that require no user interaction); (3) Main state machine object; (4) One or more parallel processes subVI objects. Note that label 1.1 in Figure 1 is a broken-line representation of the queue reference object. The broken-line is meant to show that the subVI processes are able to access and interact with the queue from within the SubVI by referencing the queue by name without having to wire the queue reference to the SubVIs. See the LabVIEW implementation of this illustration in Figure 3. Queue reference access by name also allows a queue reference to be use by dynamically launched VIs by VI Server and is discussed further under the section: Item (1) Queue Reference Object.

Items (2) and (4) are the multiple producer processes responsible for sourcing commands and data and adding them to the queue. Item (3) is the single consumer process which removes commands and data from the queue and acts on these in the order added to the queue. One important rule is that, for any one queue reference, you should only have one consumer process while there can be one or more producer processes. This rule ensures that the consumer process is the central command processing center where each and every command added to the queue reference gets handled. In the LabVIEW QSM-PC layout of Figure 3 the consumer process is part of the open top level block diagram, rather than a subVI. This is to facilitate manipulation of controls, indicators and properties of front panel objects. The discussion that follows elaborates further on items 1 through 4 and shows their LabVIEW implementation examples.

Item (1) The Queue Reference

The queue reference in the QSM-PC architecture emulates a command and data messaging pipe-line where the queue accepts data packets added by one or more producer objects and releases these data packets to one consumer object. Figure 2 elaborates on the queue reference message communication function by showing data packet migration through the queue originating from producer objects and ending at the main consumer object. The consumer object loop in item 3, Figure 2, includes queue management VIs. These VIs are from LabVIEW’s queue pallete VI shown in Figure 4 to accomplish various functions such as creating the queue, adding data to the queue and releasing data from queue. These functions are covered in this section.

Figure 4. LabVIEW’s queue palette VIs.
Creating The Queue Reference or Grabing Existing Queue Reference:
The LabVIEW queue implementation shown in Figure 3 creates a queue reference of the name: ‘Main Queue’ using the Obtain Queue VI from LabVIEW’s queue palette; see example LabVIEW code in Figure 5. 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.

![Figure 5. LabVIEW code for creating a queue reference](image)

Two inputs are essential for creating the queue: (a) The Queue Name: In this example the queue name is ‘Queue Main’; and (b) Data Packet Definition:

Shown annotated in Figure 6 is a sample data packet recommended for the queued state machine discussed in this article. This data packet constitutes a cluster containing two pieces of information: (i) A type def enumerated constant also known as a Function ‘STATE’ constant; and (ii) a variant ‘DATA’ item. This data packet definition means that the resulting queue reference can only carry data elements that conform to this data packet.

![Figure 6. Recommended data packet for QSM architecture](image)

The typedef enumerated constant enlists your chosen names of the state machine cases in the consumer process. 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. Therefore, the type-def STATE items are like command names that designate which state name contains the relevant code for processing the command. Typical function enum state names: ‘INITIALIZE’, ‘IDLE’, ‘ERROR’, and ‘EXIT’ should be included.

You must make the enum constant a typedef-based custom control so that you can add or remove command items from the enum and have your changes propagated to all instances of the typedef constant throughout your LabVIEW code. See LabVIEW literature on how to create typedef enum constants. Note that alternative queued state machine templates can also use string data elements for commands in place of the typedef enum. However, string-based state machine cases require exact syntax spelling of state names and are also case sensitive. Typed def enums help reduce programming errors because, once defined, you can only choose an item from the available list of valid items.
The second item in the cluster is the variant data element which can carry any data. This data element allows you to package data along with the command for use in processing. Sending data along with the command using the variant data element is optional. Note that alternative queued state machine templates do not include the variant data element which means that only commands can be sent via the queue. You would need to use functional globals or other means to send data to the processing loop. The variant data element recommended in this QSM-PC architecture gives you a convenient and useful option to package your commands with any data.

Adding a Command and Data to the Queue Reference
You use the ‘Enqueue Element’ VI from LabVIEW queue palette to add data elements to the queue. The example in Figure 7(a) is an instance where a producer process adds a data element to the queue with the enum message set as STATE 1 but with an empty variant data. Figure 7(b) is where a producer process adds a queue data element with enum set as STATE 4 and packaged with a data cluster bundled as variant data. The bundled data is intended for processing by the consumer object under STATE 4. Any data type can be bundled as a variant data making this a flexible way of sending any data in the queue pipeline. The state machine that will handle the data will have to decode the variant data inside the case named STATE 4. This is elaborated under the section: Item(2) Consumer Object Stats Machine of this article.

Figure 7 (a): Single data packet added to enqueue VI. STATE set as STATE 1 and empty variant data

Figure 7(b): Single data packet added to VI. The STATE set as STATE 4 and variant data packages a data cluster using the bundle by name utility.

Figure 7(c): Multiple data packets added by building them into an array and using a ‘For Loop’ to add the data to the enqueue VI.

Note that you should remember to use the en-queue element at opposite end of queue VI, (available from LabVIEW’s queue palette) to add a data element to the queue which needs priority handling. For example
the exit and error state cases are typically added to the front of the queue so that they can be handled immediately by the consumer object process.

**Removing Data Element from Queue for Handling by Consumer Object**

You use the ‘De-queue Element’ VI from LabVIEW’s queue palette to remove data elements from the queue for processing by the consumer object. The de-queue process is annotated as item 1.2 in Figure 2. After de-queuing, the data packet is unbundled to reveal the STATE name which is wired to the selector terminal of the State Machine. A case in the state machine which matches the packet’s STATE Name will handle the data packet that was input to selector terminal. The example in Figure 8(a) is a section of the consumer dequeue process taken form Figure 2. A commented section of equivalent LabVIEW implementation code taken from Figure 3 is given in Figure 8(a).

After a dequeued data packet is unbundled to reveal the STATE name, the STATE name is compared with the STATE name ‘EXIT’. Should the comparison be TRUE, the EXIT State will run and thereafter the consumer process loop will terminate executing. This assumes that the STATE name: EXIT, is included in the enum list, and that this STATE name is intended to quit the LabVIEW program.
De-Queue VI Unwired Time-Out Input Terminal Defaults to (-1)
Notice that the de-queue VI’s ‘Time Out’ input terminal is unwired and will therefore default to -1. This means that the consumer dequeue process will wait indefinitely or until an element is ready to be dequeued. A way to make the consumer process loop continuously is described under this article’s section: ‘Consumer Object State Machine’.

The Queue Manager VI in the Consumer Loop
A Queue Manager VI is located right after execution of each individual case in the consumer’s state machine. See item 1.3 in Figure 2 and item 1.3 in Figure 3. Note that this VI is strategically placed so that errors and commands can be capture upon execution of the individual state machine cases. The Queue Manager VI has three main purposes.

![Diagram of Queue Manager VI](image)

**Figure 9.** Queue manager VI executes after a state finishes running in the consumer process. This VI handles errors and allows run-time logic implementation, and monitors the need to exit.

1. **Handle Errors From State Cases**
   Note that the queue VIs are considered ‘mission critical’ VIs and are therefore not wired to the error line. This allows these VIs to continue to work even when errors are captured by the queue manager VI. If such an error is captured, the ERROR state name is added to the front of the queue, see Figure 9. This forces priority processing of the shift-registered error before other states can run. You must clear the error in the error handler case.

2. **Implement Case-by-Case Run-time Logic**
   The Queue manager VI can accept an array of STATE commands from any case in the state machine. The array-type input allows a state case to input no state (empty array), just one state (single element array), or multiple states. This avails the capability for consumer state machine case to implement ‘as-needed’ run-
time logic. Run-time logic is where a consumer state case behaves as a producer by adding commands, as needed, to the queue for processing by the consumer loop. For example, a consumer state case after processing data from the queue can determine that an alarm condition needs to be raised. The state can then add the ALARM state name in the front of the queue or it can pre-empt all elements in the queue, and in their place, just add the ALARM state name. The default is to add data packets to the back of the queue.

(3) Monitor Global Exit
Monitoring the global EXIT condition only works if the VI is in constant polling mode. This can be achieved by forcing the consumer VI to visit the idle case at specified wait intervals. By doing this the queue manager VI can poll for the boolean EXIT global to determine if the consumer loop should terminate. If polling is not done, then the EXIT enum state must be input to the queue to instruct the consumer loop to exit. Note that you could also make the de-queue VI come out of the wait state by closing the queue reference from another process in the code see Figure 10. All VIs waiting on the queue will come-out of the wait state and produce error number 1122. You can check for this error and terminate the consumer loop accordingly. However, this method of exiting from queue VIs’ wait state is not used in the QSM implementation of Figure 3.

Figure 10. You can STOP the consumer loop by destroying the queue reference which causes VIs waiting on the queue to come-out of the wait state and generate a error #1122. An error state of TRUE from wait VI is used as a condition for exit.

Item (2) Consumer Object State Machine

Note that the consumer process is considered to be the owner of the queue named: ‘Main Queue’ because it is from only this consumer process where queue elements are dequeued from the ‘Main Queue’ reference. Producer processes access or borrow the queue reference from multiple points only to add elements to the queue.

Once a data packet is removed from the queue, the consumer process will execute the state case of the unbundled state name from the data packet. The code inside this designated case will then run to process the command and accompanying data, if any. The consumer object state machine is labeled as item 3 in the Figures 1, Figure 2, and Figure 3.
Other than the case names: INITIALIZE, IDLE, ERROR, and EXIT you can add and name other state cases that will perform useful functions based on your application’s needs. For example you can designate four cases: (a) READ; (b) READ_STEP_1; (c) READ_STEP_2; and (d) READ_STEP_3 and spread-out your READ code to run under these four sequential case steps. A producer will only need to add the enum Case READ for the code to run all 4 read steps. This is especially useful if you want more space to write your code rather than packaging all your code in one subVI. Furthermore, each step can execute run-time logic to exit the consumer loop should this be required. **Figure 11** shows the READ case with the other three cases added for immediate sequential execution in the front of the queue.

![Figure 11](image.png)

*Figure 11. Consumer process case executing the READ command in three additional READ steps that will execute immediately thereafter by the addition of sequential READ steps to the front of the queue.*

**Consumer Process IDLE State Machine Case**

Be sure to include the IDLE case in the consumer’s state machine if you want your consumer process to continue running while in standby mode without necessarily visiting other state machine cases. You can loop through the idle case to poll for time-elapsed and to adjust for wait times using the wait until next multiple timer VI. To accomplish continuous looping include the IDLE enum case state when initializing the consumer loop and keep adding the IDLE enum case state while inside the IDLE case. See **Figure 12** and **Figure 13**.

![Figure 12](image.png)

*Figure 12. Example of how to perform continuous loops via the IDLE case state*
Figure 13. Consumer process example showing how to code the IDLE case for continuous and constant loop cycle time. Note special use of the ‘Wait till next ms multiple VI’ inside a sequence frame.

Decoding Data Packets Originating From Producer Processes
Producer processes send commands via the enum selector. Data of any type can be sent along with the commands by converting the data to a variant data type and then bundling variant data along with the enum command. However, for the consumer process to use this data, the data packet must be appropriately decoded from the variant data type. Variant data decoding must be done in the designated enum consumer case. Figure 12 shows variant data decoding in state machine case: STATE 4, of a data packet input to the queue as was shown in Figure 6(b) and Figure 6(c).

Figure 14. Decoding variant data inside STATE 4 for data that was enqueued from the example in Figure 6(a) and Figure 6(b).

More on Run-Time Logic Implementation in Consumer Case
Each time a consumer state machine case finishes executing, there is opportunity for intervention of program flow by the addition of enum state commands at the back or front of queue or even to replace the queue elements with new enum command instructions. This feature, known as run-time logic programming is an important advantage of the QSM architecture. Figure 15 shows use of a SubVI to execute STATE 5. If the SubVI in STATE 5 can instruct detects emergency alarm conditions. The program will immediately
pre-empt (replace) all enum commands in the queue and execute the emergency shut down enum case instead.

**Figure 15.** Run time logic programming with option to pre-empt program flow.

**Avail and Update Parameters and Variables in Any State Machine Case**

Another feature worthy of note in the QSM architecture is the shift registered cluster data flow line that passes through all state machine cases see **Figure 16** (not shown in **Figure 3**). This flow line avails and allows update of parameters and variables as needed inside every state machine case. You use LabVIEW’s unbundled-by-name utility to access parameters and variables and use the bundle-by-name utility to update the same. **Figure 16** shows these operations within enum case: STATE 6 of the consumer loop.

**Figure 16.** Shift registered cluster of parameters and variables that can be accessed and updated within every case of the state machine in the consumer process
**Item (4) Parallel Process SubVIs**

SubVIs that run in parallel with the main consumer process complement and add useful functions to the QSM-PC architecture; see item 4 in Figure 1, Figure 2, and Figure 3. SubVIs that can be run in this manner are data communications, data acquisition (DAQ), results analysis, and much more. These SubVIs primarily behave as producer processes and access the queue reference by name, using the method described in Figure 5. This is the same queue reference 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 as seen in the LabVIEW implementation of Figure 3 which creates transparent routes of communication. The block diagram looks tidy when SubVIs access queues in this manner.

**Building Both Multi-Consumer Functionality in LabVIEW Code**

By creating multiple queue references of unique names you can have multiple consumer points in your LabVIEW code. Each consumer point (i.e. at the dequeue VI) will own the respective queue reference of a unique name and will behave as the only destination point for all packets sent via that queue reference. In this way you can implement a network of transparent data migration paths from multiple producer points to predefined destination points in your code. The one important rule here is that to send data to a given consumer point you must use the queue reference owned by the respective consumer process.

The example in Figure 17 shows a block diagram for SubVI 1 which behaves as a consumer process for the queue name: ‘Q1’ but which also grabs queue references: ‘Main Queue’; Q2; and Q3. In this case, SubVI 1 can use queue reference Q1 to accept commands and data from other producer points while and at the same time SubVI 1 can send commands and data to consumer processes that own the other queue references.

![Figure 17. Block diagram for a subVI that behaves as the consumer process for Q1 but is also the producer process for queue references: (1) Main Queue; (2) Q2; and (3) Q3](image-url)
Summary

The QSM –PC architecture design attributes, object features, and function are covered in the forgoing discussion. The goal is to help master the fundamentals in the applied use of this method to develop parallel process LabVIEW programs, and specifically the strategic use of queue references as a network of communication pathways for commands and data transfer between multiple parallel processes. The fundamentals discussed here are applicable to many other variations of the QSM -PC architecture. This article will conclude with pertinent highlights covered in the discussion.

1. **QSM-PC is a Parallel Process Enabler**
   The QSM –PC architecture establishes the use queue references as data messaging pipelines that communicates information between parallel processes in an as-needed and timely manner. This type of communication solves one of the serious challenges in parallel process programming.

2. **Multiple Producer and Single Consumer Points**
   In the QSM –PC architecture, 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.

3. **Global Access to the Queue via the Queue Name**
   Global access to a queue reference means that the queue can bee seen by SubVI processes without the need of wiring the SubVIs to a queue reference. This creates transparent routes of communication and greatly simplifies the code.

4. **Empowers Programs to Build Run Time Logic.**
   QSM –PC 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.

5. **Multi Consumer Queue References Creates a Network of Data Pathways**
   Parallel process SubVIs which themselves use the QSM-PC architecture create a network of communication pathways with multiple producer and consumer points. This allows one parallel process to control the flow multiple parallel process subVIs.

**References**


**Example included with This Article**

A LabVIEW program is included with this article on QSM PC architecture implementation. The file QSM –PC EXAMPLE.LLB contains a main program and two subVIs that run in parallel. This example implement most of the QSM –PC command and data communication features discussed in this article.