

Synchronization

- causing task occur at the same time or in unions by using clock triggers or events.
- level of synchronization depends on the application.
- synchronization not required in some slower control applications.
- synchronization required to acquire and transfer lossless data at known rate.
- The data communication>>synchronization palette contains the building blocks for complex, parallel, application logic, where dataflow becomes a little more fuzzy.

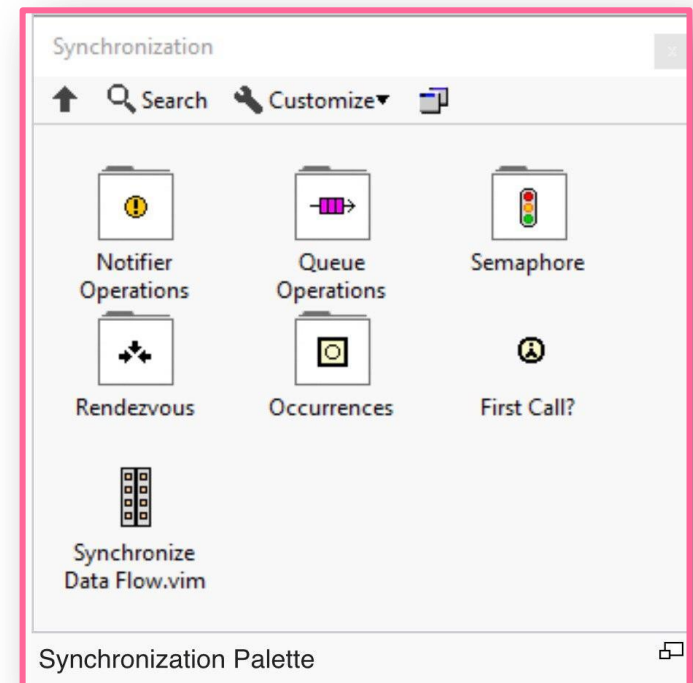
LabVIEW offers several types of synchronization tools.

They can be placed into two groups:

1. Semaphore and Rendezvous.
2. Occurrence, Notifier, and Queue.

the first group pauses execution of a piece of code until data is available or a condition is met.

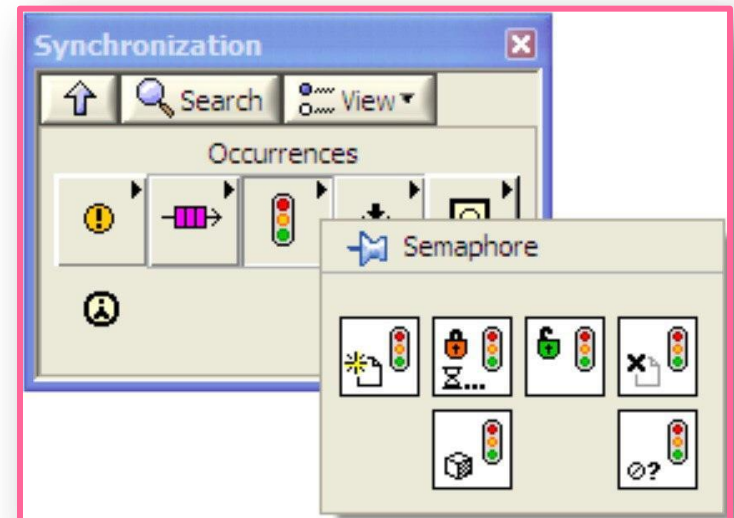
The second group manages access to a piece of code based on the actions of multiple data sources.



Semaphore

The semaphore is a piece of code that gates access to a resource. When writing your code, you create the semaphore and pass its reference to each loop that may need to access the resource in question. When access is desired, you use the semaphore reference to request access, or “acquire” the semaphore.

- The Semaphore tools are found in the sync palette at data communication>>synchronization >Semaphore palette, such in figure below.
- The Semaphore has the following basic operations: Create, Acquire, Release, and Destroy.
- A Semaphore acts like a 4-way stop sign: multiple sections of code want to go, but the stop signs only allow one car at a time.



Hints for using semaphore

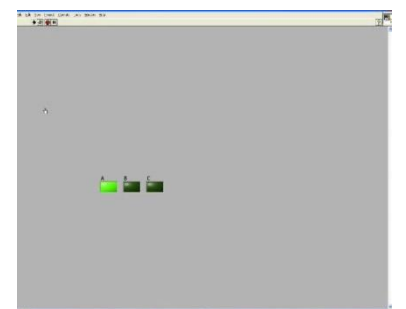
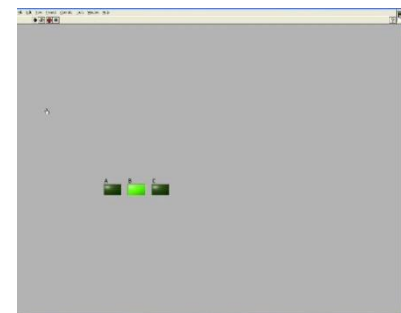
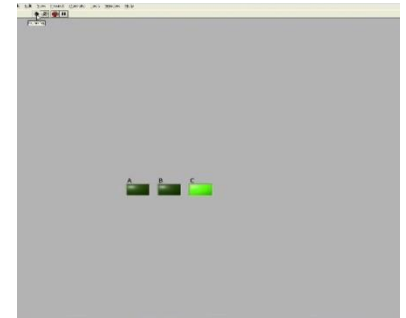
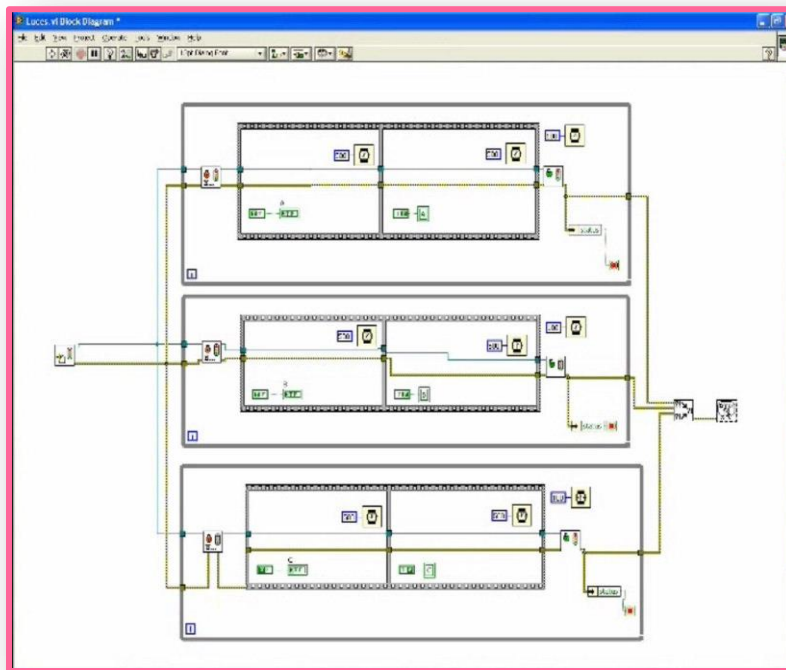
- Use a semaphore to lock and unlock a shared resource that can only be accessed by one location at a time (or by a fixed number of locations at a time).
- You cannot use the Not a Number/Path/Refnum function with a semaphore refnum. You must use the Not a Semaphore function for this purpose.
- You can obtain a reference to a named semaphore in multiple locations in your application. Just wire the semaphore name into the name input of the Create Semaphore function.
- Make sure to call Destroy Semaphore only once on a semaphore, and only when you are ready to destroy it.

Advantages of Semaphore

- A semaphore is a way to limit the number of tasks that can simultaneously operate on a shared, protected resource. A protected resource or critical section of code might include writing to global variables or communicating with external instruments. You can use semaphores to make your code thread-safe, and to prevent race conditions. A semaphore is similar to a mutex (Mutual exclusion object) in other programming languages.
- It is useful for protecting two or more critical sections of code that should not be called concurrently. Before entering a critical section, the thread must acquire a semaphore. If no thread is already in the critical section, the thread proceeds to enter that part of the diagram immediately. This thread must release the semaphore once the critical section is complete. Other threads that want to enter the critical section must wait until the first thread releases the semaphore.

- obtain a semaphore reference and then place the Acquire Semaphore and Release Semaphore VIs at the beginning and end, respectively, of each critical section. Then each critical section can acquire and release the semaphore one at a time to assure proper data control. At the end of the code use the Release Semaphore Reference VI.

Simple example about semaphore with implementation



Types of Semaphores

1. Binary Semaphore: Binary semaphore is used when there is only one shared resource.

Binary semaphore exists in two states ie.Acquired(Take), Released(Give). Binary semaphores have no ownership and can be released by any task or ISR regardless of who performed the last take operation. Because of this binary semaphores are often used to synchronize tasks with external events implemented as ISRs, for example waiting for a packet from a network or waiting for a button is pressed.

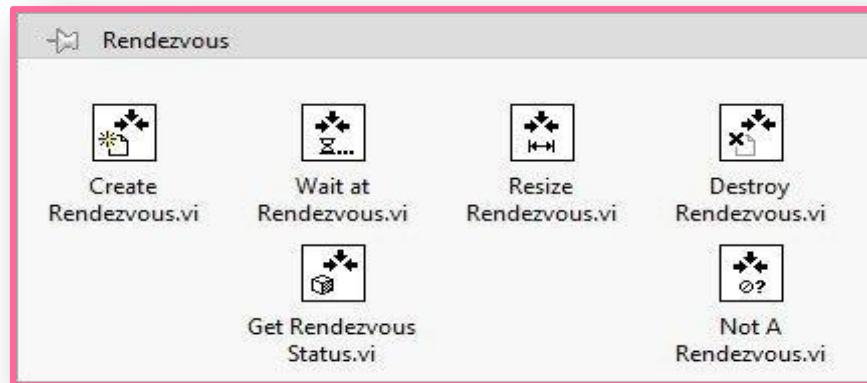
Because there is no ownership concept a binary semaphore object can be created to be either in the “taken” or “not taken” state initially.

2. Counting Semaphore: To handle more than one shared resource of the same type, counting semaphore is used. Counting semaphore will be initialized with the count(N) and it will allocate the resource as long as count becomes zero after which the requesting task will enter blocked state.

3. Mutex Semaphore: Mutex is very much similar to binary semaphore and takes care of priority inversion, ownership, and recursion.

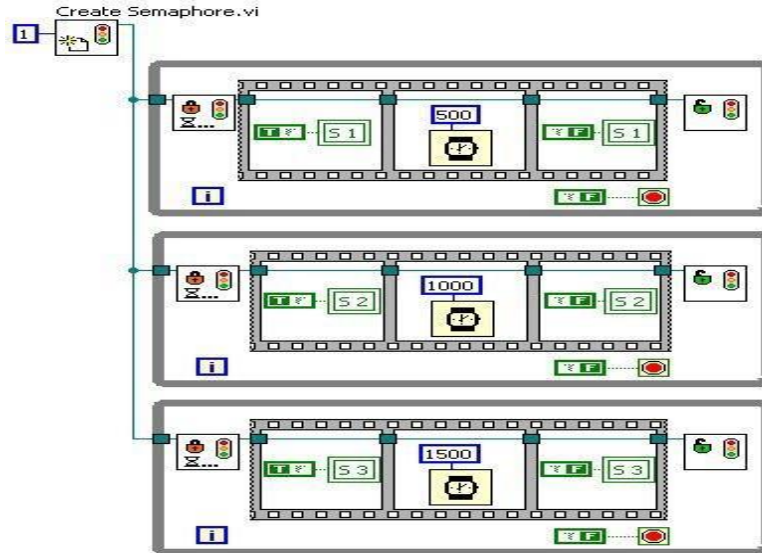
Rendezvous

A rendezvous ensures that several threads or VIs wait at a certain execution point before proceeding. Each task that reaches the rendezvous waits until the specified number of tasks are waiting, at that point all tasks proceed with execution. A rendezvous is useful for synchronization of two or more different sections of code. The effect is similar to that of creating one occurrence and using it to trigger several Wait on Occurrence nodes elsewhere in the same or other VIs. The Rendezvous Palette is found in the Functions Palette under Programming»Synchronization»Rendezvous as figure below.

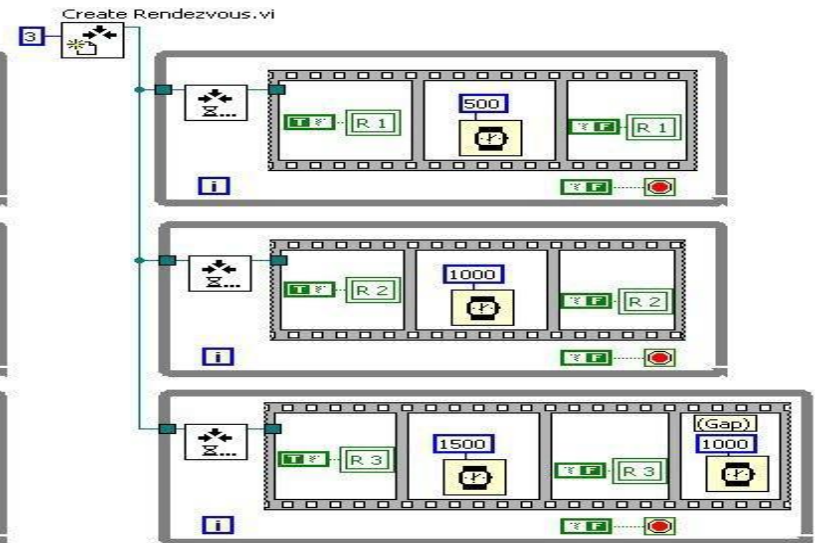


The advantage of the rendezvous is that the user can define and programmatically change the number of code sections that are required to meet at the rendezvous before all the involved Wait at Rendezvous nodes return.

Semaphore vs. Rendezvous



- Multiple sections of code want access to a single device (e.g. a DAQ board).
- Only allow one section of code access to a Notifier at a time.
- A read/write data manager is not read until a write is completed (or vice versa).

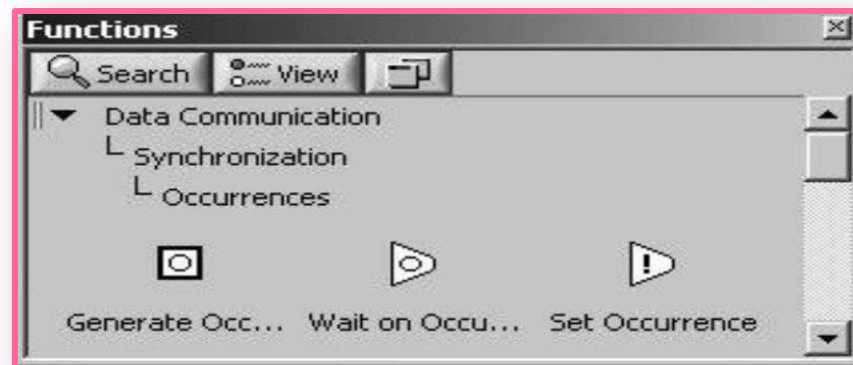


- Perform calculations only after all input channels have been acquired.
- Move to the next position after all the motion controller axes are stopped moving.
- Open all tank valves only after all tanks are filled as indicated by level sensors.

Occurrence

allow you to control separate, synchronous activities, in particular, when you want one VI or part of a block diagram to wait until another VI or part of a block diagram finishes a task without forcing LabVIEW to poll. Unlike the other synchronization VIs, there is no destroy function for occurrence search instance of the Generate Occurrence function acts similarly to a constant, outputting the same value each time it is called. Additionally, occurrences have no data they are used purely for synchronization where you want to wait until an event occurs in some other location.

GenerateOccurrence(DataCommunication>>Synchronization>>Occurrences palette).



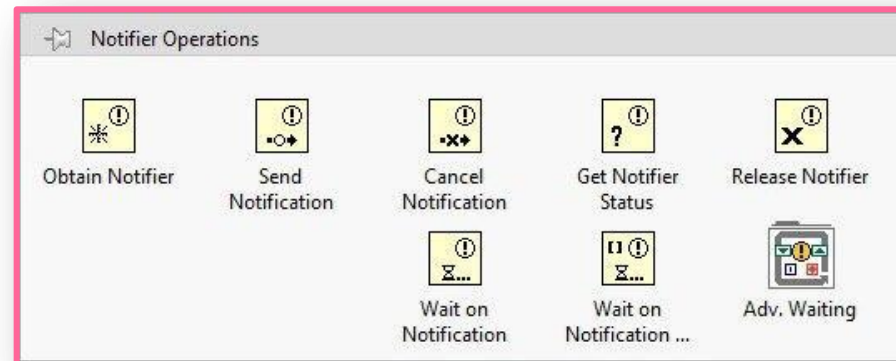
Generally, you should use occurrences only for memory and performance optimization. Try using notifies instead of occurrences.

Occurrences don't have a destroy function (like other synchronization functions). Each Generate Occurrence function acts as a constant, outputting the same occurrence reference each time it is called.

Notifiers

Notifiers are a tool for communicating between two independent parts of a block diagram or between two or more VIs running on the same machine. Notifiers cannot communicate across networks or the VI Server. This communication is generally for the purpose of synchronizing two independent processes.

Unlike Queue Operations functions, Notifier Operations functions do not buffer sent messages. If no nodes are waiting on a message when it is sent, the data is lost if another message is sent. Notifiers behave like single-element, bounded, loss-y queues. `GenerateNotifiers(Programming » Synchronization » Notifier Operations)`.



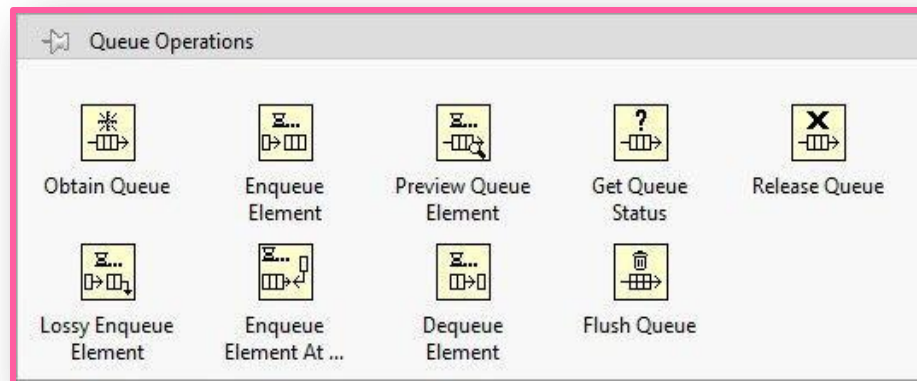
One benefit of notifiers is that a process that receives information completely stops execution while waiting, and starts only when new data becomes available. This reduces excess processing power being wasted on unneeded polling.


Queue

A queue maintains a first in/first out (FIFO) order of data items. For instance, the customers waiting in line at a fast food restaurant are in a queue. The first customer to arrive is going to be the first one served.

A queue is useful in producer/consumer situations, where one portion of code is creating data to be used by another portion. These can e.g. be two loops running in parallel. The advantage of using a queue is that the producer and consumer rates do not have to be identical. If consumption is slower than production, the queue will eventually become full and the producer code will be forced to wait until the consumer has dequeued an element before a new element can be queued up.

Generate queue (Programming » Synchronization » Queue Operations).





Unlike an array, it is not possible to randomly access elements in a queue. It is strictly a buffer that provides you the ability to enqueue (add/insert) and dequeue (subtract/remove) elements. The only way to view all the elements in a queue is to dequeue them one by one. You cannot perform data manipulation to all the elements in a queue either.

Unlike Queue Operations functions, Notifier Operations functions do not buffer sent messages. If no nodes are waiting on a message when it is sent, the data is lost if another message is sent. Notifiers behave like single-element, bounded, loss-y queues.

References

<https://www.viewpointusa.com/>

<https://knowledge.ni.com/>

<https://lavag.org/>

<https://en.m.wikipedia.org/>