Using LabVIEW to Create Multithreaded DAQ Applications

Updated Feb 4, 2020

Overview

This document describes how to use LabVIEW to isolate different tasks of an application. It shows several different approaches to writing a data acquisition (DAQ) application and describes the advantages and disadvantages of each approach. This document assumes a basic knowledge of DAQ, multitasking, threads, and LabVIEW programming. Refer to the Further Topics section for more information about the technologies mentioned in this document.

Contents

Introduction

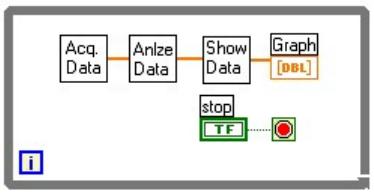
Consider an application where you have the following three tasks:

1. Acquire Data
2. Analyze Data
3. Present Data

You can approach this example in the following three ways.

Approach 1: Ordered Example

You can approach each task sequentially, as is shown in the following example.



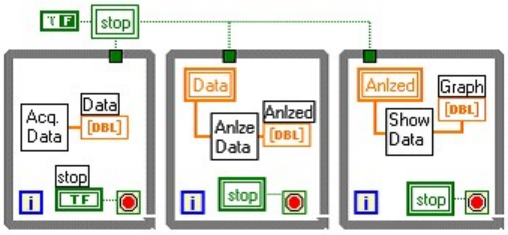
Note the following issues in this approach:

- The data is acquired, analyzed, and displayed in that order.
- All data that is acquired also is analyzed and displayed.
- While the data is being acquired, it cannot be analyzed or displayed.
- The application is simple.
- The three tasks run in the same thread because they are connected sequentially.

This is the most common approach, and for many applications it is the best approach.

Approach 2: Multiple Tasks Example (Simple)

In the following example, each task runs independently and at different rates from the other tasks.



Data passes between tasks using local variables.

Note: This examples uses local variables for presentation purposes only. Using local variables might result in lost or duplicated data.

Note the following issues in this approach:

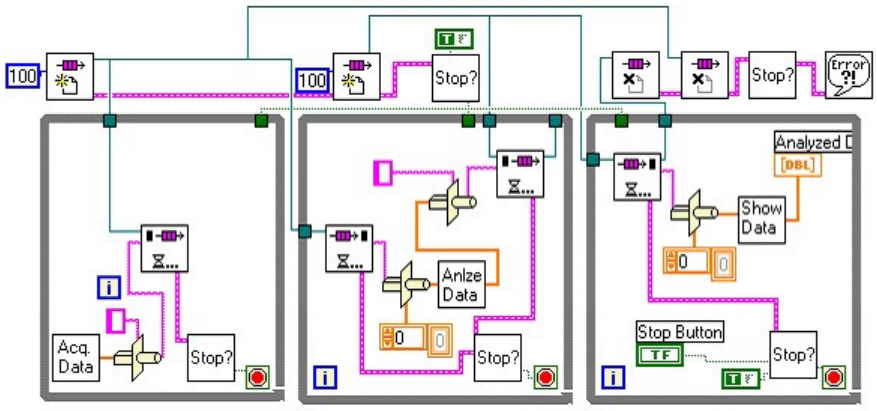
- The tasks no longer run in a particular order.
- Depending on the rate the Acquire Data task runs compared to the Analyze Data task, some data might not be analyzed and some data might be analyzed twice.
- You can write the tasks in such a way as to allow the Acquire Data task to run quickly and always acquire all data, while the Analyze Data and Show Data tasks only occur frequently enough to show the user what is happening.
- Each task can run in its own thread.

This first two issues seem like disadvantages, but you can take advantage of the issues. For example, suppose that the Acquire Data task also saves the acquired data to disk. This saved copy of the acquired data has all the data if you need a complete log of the data later. The other tasks can then run slower, skipping some data, and display only the latest data while analyzing the data.

Another advantage of this approach is that it allows prioritization. With some additional work, you can configure the three tasks so the Acquire Data task gets priority of execution over analysis and presentation. That way, slower computers can acquire and save data without getting errors.

Approach 3: Multiple Tasks Example (Advanced)

Finally, consider the following more advanced approach.



Note the following issues in this approach:

- There are still 3 separate tasks, which means you can use different threads.
- This approach does not use local variables, so you can place the three loops in separate VIs.
- This approach uses the Queue VIs, preventing any data from being lost or duplicated.
- The Queue VIs provide synchronization between tasks.

Note: Because of the nature of queues, if the Analyze Data task does not keep up with the Acquire Data task, the queue fills up with data to be analyzed and the Acquire Data task must wait for the Analyze Data task to remove data before it can place additional data in the queue.

Data Acquisition Applications

Data acquisition makes a particularly good example of this approach because buffered acquisition reliably stores data into a buffer and it is up to the program to remove data from the buffer before it is overwritten. If the data is overwritten, an error is returned and the acquisition stops. While this makes acquisition reliable because you never lose any data without an error, it also means that you must make sure acquisition tasks execute fast enough to avoid errors.

If you do not read data from the buffer fast enough to avoid getting an error and still perform analysis on the data, you have the following options:

- Save the acquired data to disk and perform analysis on it at a later time. Refer to the High Speed Data Logging example linked below for more information.
- Programmatically ignore data if you do not have time to analyze it. This data will be lost, but your data acquisition loop will continue running. This is the approach taken in the Multiple Tasks Example (Simple) above.
- Use more computing power so the analysis occurs faster. Options include buying a faster computer, dividing the analysis so that it occurs on multiple computers, or using a computer with multiple processors. This is the approach taken in the Multiple Tasks Example (Advanced) above. Notice that a program must be multithreaded in order to take advantage of a multiple processor computer. For this reason, running the Ordered Example on a Multiple Processor computer does not have any advantages.

Conclusion

If you use the multiple task approach in your application, a number of advanced topics can be powerful tools with that approach:

- Use DataSocket instead of local variables to allow tasks to run on multiple computers or multiple VIs.
- Make maximum use of multitasking or multi-processor computers.
- Use synchronization, such as queues, notifiers, rendezvous, and semaphores, to time different tasks and ensure that no data is lost.
- Refer to *LabVIEW Help: Suggestions for Using Execution Systems and Priorities* (linked below) for information about options you can select to ensure maximum benefits from multithreading technology.

Refer to Running Multiple Tasks document (linked below) for VIs that show the three approaches described in this document.

See Also:
Optimizing Test with Multiprocessor Machines
Running Multiple Tasks
LabVIEW Help: Suggestions for Using Execution Systems and Priorities