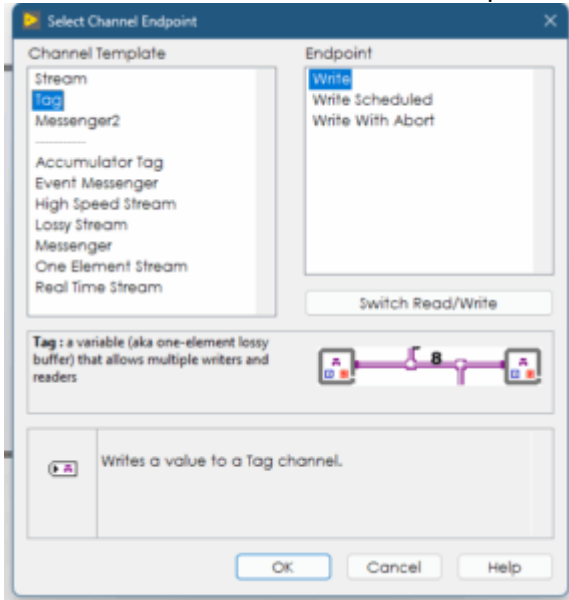


LabVIEW Channel Wires vs Queue and Notifiers

Category: Blog, LabVIEW Talk

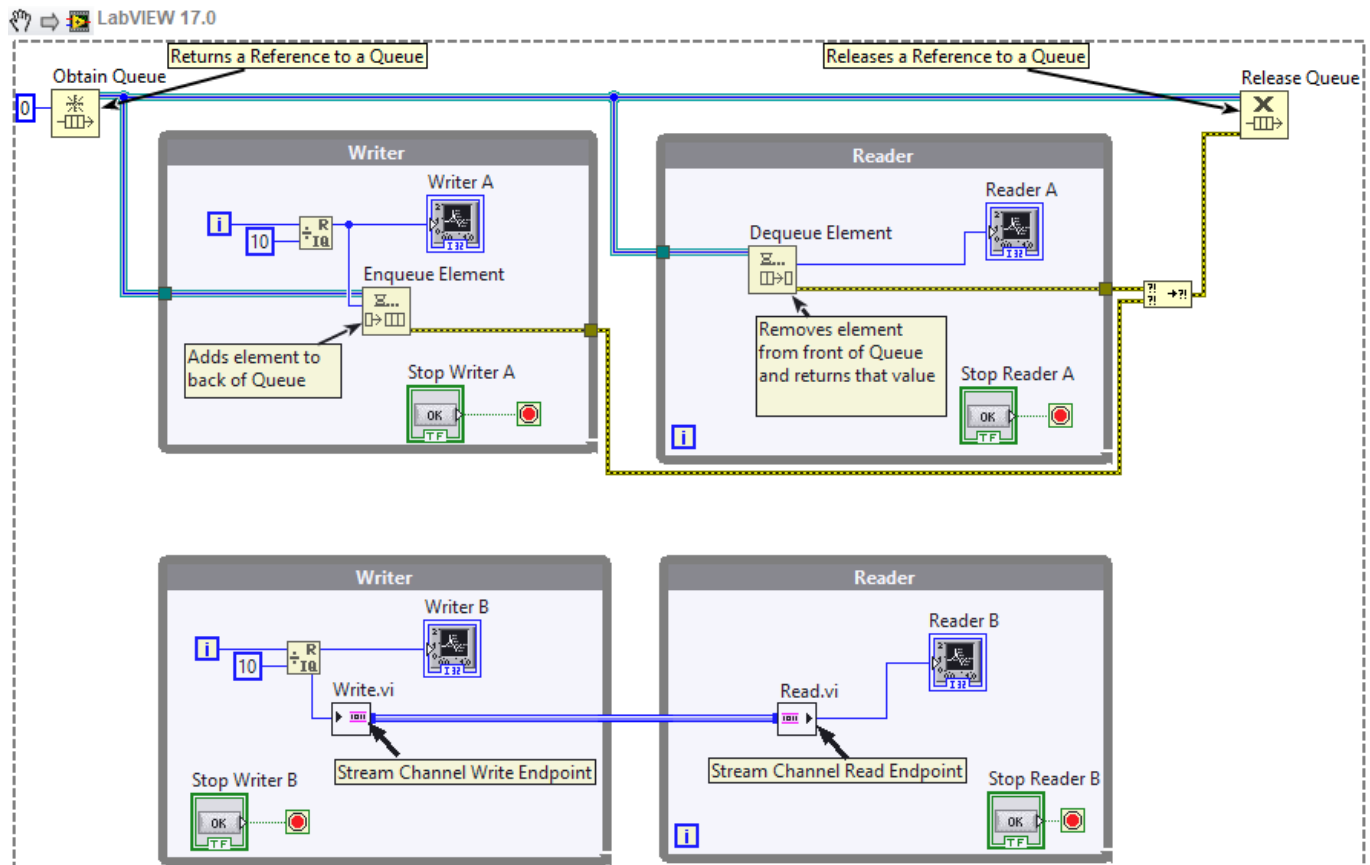
scritto da Nicola Bavarone | Ottobre 5, 2024



When developing complex applications in **LabVIEW**, managing the flow of data between different sections of code is essential to ensure synchronized and coherent behavior.

LabVIEW provides various tools for handling this, including [channel wires](#) and **reference-based** solutions like **queues** and **notifiers**.

In this article, we will explore the main characteristics of these tools, compare them, and help identify which approach might be better suited depending on the architecture's requirements.



Channel Wires: What They Are and How to Implement Them

Channel wires are one of the newer features introduced in LabVIEW, designed to simplify communication between different **parallel loops** or blocks of code.

Unlike queues and notifiers, which use references to manage communication, channel wires offer a more graphical and intuitive approach.

A **channel wire** is a visual wire that connects two sections of code and allows asynchronous data transfer, respecting the **dataflow** pattern.

Advantages of Channel Wires

1. **Intuitiveness:** Due to their graphical nature, channel wires are easier to understand for those with limited experience in concurrency management.
2. **Integration with dataflow:** They maintain a structure visually similar to traditional LabVIEW wires, making the code easier to read.
3. **Communication between parallel loops:** Ideal for simplifying communication between loops that need to operate concurrently without direct interference.

Implementing Channel Wires

To use channel wires, you select the type of channel you need, such as a **Tag Channel** (for single data items) or a **Stream Channel** (for a continuous data stream).

Once selected, the wire is created like a standard wire and connects the code sections, enabling data transfer between various nodes.

Queues and Notifiers: A Reference-Based Solution

Queues and **notifiers** represent a more traditional and well-established method for managing communication between parallel loops in LabVIEW.

They use a reference-based system, meaning data is stored in memory and accessed through explicit calls to **enqueue** and **dequeue** functions (for queues) or through notifications (for notifiers).

Advantages of Queues and Notifiers

1. **Explicit control:** With queues and notifiers, control over data flow is explicit and managed through references, allowing more detailed synchronization management.
2. **Advanced queue management:** Queues allow the creation of true data buffers, which can be helpful for managing variable data flows without losing information.
3. **Flexibility:** The handling of queues and notifiers is highly configurable, allowing them to model various types of concurrent architectures.
4. **Persistent references:** Since they are reference-based, these tools are ideal for more complex architectures that require persistent data across multiple components of an application.

Implementing Queues and Notifiers

In LabVIEW, **queues** and **notifiers** are powerful tools for managing communication between parallel loops, but they differ significantly in how they handle data and the communication models they implement.

Queues and the Stream and Message Communication Model

Queues are used to implement a communication model based on **FIFO** (First In, First Out) or **LIFO** (Last In, First Out) memory structures, meaning that data is stored in a sequential manner and handled in the order it was inserted (FIFO) or in reverse order (LIFO). This system allows for data transfer between loops running at different speeds, ensuring that **no data is lost** in the process (no memory loss).

To implement queues, you create a queue using the **Obtain Queue** function and manage data transfer between loops using **Enqueue Element** (to add data to the

queue) and **Dequeue Element** (to extract data). Thanks to this storage mechanism, queues are ideal for building architectures such as:

- **Message Handlers**, where messages sent from different parts of the system can be processed in sequence, ensuring that no messages are lost.
- **Producer-Consumer**, a widely used architecture where one loop produces data and another loop consumes it, even if they operate at different speeds.
- **Pipelines**, where tasks are divided across multiple loops or tasks to improve overall efficiency, managing a continuous data flow between different sections of code.

In general, the advantage of **queues** is their ability to retain all data, making them an essential tool when data loss is unacceptable and when there's a need to synchronize loops or tasks running at different speeds.

Notifiers and the Tag Communication Model

On the other hand, **notifiers** work on a **single-memory** principle, where only the **last value** is shared. Notifiers are implemented by creating a **notifier** with the **Obtain Notifier** function, and using the **Send Notification** function to send a value, and **Wait on Notification** to receive it. However, the **single-memory** behavior means that every new value overwrites the previous one, leading to **data loss** for unread values.

This mechanism makes notifiers ideal for implementing a **Tag** communication model, which is suited for scenarios where there's no need to maintain a full history of values, but it's important to communicate only the **latest state change**. **Tag Channels** are used, for example, to synchronize events or signal a state change in a system, such as:

- **Event synchronization**, where it is sufficient to notify when an event occurs without needing to handle a sequence of notifications.
- **State updates**, to share the latest version of a parameter or variable with other parts of the code.

In summary, while **queues** implement a continuous or discrete data flow, allowing for robust architectures with no data loss, **notifiers** are limited to transmitting only the most recent available value, losing the previous data. This difference makes **queues** ideal for handling data and message flows, whereas **notifiers** are better suited for notifying single events or state changes.

Channel Wires vs Queues and Notifiers: A Comparison

Channel wires provide a visually simpler solution, allowing loops to connect in an intuitive manner. However, this simplicity can be a drawback in more

complex applications, where the explicit control and flexibility of **queues** and **notifiers** become more advantageous.

While **queues** and **notifiers** require more attention to manage references, they offer:

- Greater flexibility in buffering data.
- Precise control over asynchronous behavior.
- Adaptability in architectures with high concurrency.

Channel wires may be useful in simpler applications where communication between loops is minimal, and maintaining visual simplicity is a priority.

However, when complex data buffering or precise synchronization of multiple threads is required, reference-based solutions offer greater robustness.

Our Preference at Bytelabs

At Bytelabs, we prefer using **Queues** and **Notifiers** for several reasons:

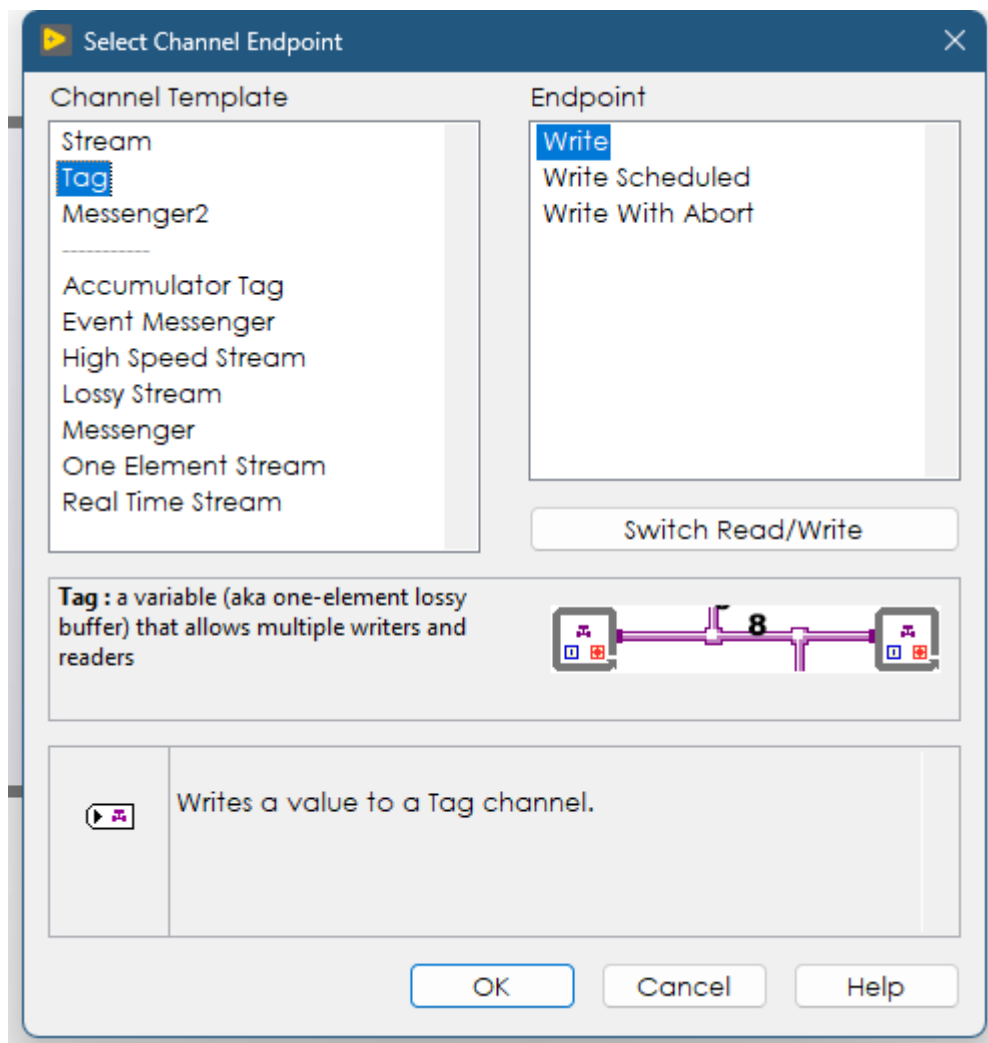
1. **Code Portability and Familiarity:** Our experience with using queues and notifiers makes them a natural choice, as they align well with the habits and practices we've developed over time.
2. **Code Readability and Dataflow Principles:** One key aspect of LabVIEW is the way data flows through the code, especially through **tunnels** in structures. We are accustomed to the clear dataflow behavior in iterative structures, where data passes in or out **once** per iteration. The **channel wire's tunnel** behaves differently, introducing what is essentially a reference-based approach but without the explicit reference notation. We prefer seeing upfront whether a wire carries data or a reference for better code readability and clarity.
3. **API Familiarity:** The API for **queues** and **notifiers** reflects a natural programming approach, using library functions that most developers are already familiar with. This makes it easier to develop and maintain complex systems without needing to adapt to a specialized library like **channel wires**.
4. **Maintaining Readability in Complex Architectures:** As code becomes more complex, maintaining readability becomes a significant challenge. **Channel wires** can obscure the flow of execution, whereas using **references** allows for more flexible patterns like **Functional Global Variables (FGV)** or **shift registers**, which provide a clear way to track state and manage data across loops.

As **Certified LabVIEW Developers (CPI)**, we need to teach these concepts in the **Core 2** curriculum. We agree with the broader CPI community that explaining **channel wires** as a mechanism for parallel loop communication can be more intuitive for beginners. However, as developers, we also focus on long-term

solutions, thinking ahead to when the code grows in complexity. As developers, not just instructors, it's important to guide others in understanding that while simple problems can be solved with simple tools, professional and moderately complex architectures require a more conscious choice of tools, prioritizing **robustness** and **maintainability**.

We also see the value in comparing LabVIEW's abstractions with those of other programming languages and tools, such as flowcharts, UML, and **C-like Languages**. This comparison is important because, as we move toward more sophisticated topics like **Core 3** where the much-loved **Queued Message Handler (QMH)** transitions to a **Channel-based** architecture, understanding these layers of abstraction becomes essential.

Our philosophy at Bytelabs is to approach programming with a forward-looking mindset, ensuring we use the right tools with agility and awareness of future complexities.



Conclusion

In conclusion, we recognize the intuitive appeal of **channel wires**, especially for those just beginning with parallel loop communication.

However, as systems grow in complexity, we firmly believe that **queues** and **notifiers** provide a more robust and maintainable solution.

The explicit management of references allows for greater flexibility, and the ability to implement patterns like **FGVs** and **shift registers** becomes invaluable when scaling up architectures. For us, adhering to dataflow principles and ensuring **code readability** over time is paramount, particularly as we develop professional-grade systems.

This approach aligns with our philosophy that choosing the right tool goes beyond ease of use; it's about anticipating future challenges and maintaining **control** over execution. In this sense, we value the clear and structured approach of **queues** and **notifiers**, which allow us to design architectures with precision, robustness, and adaptability.

Citing the **Zen of Python**, we love the phrase:

"There should be one— and preferably only one—obvious way to do it."

While LabVIEW offers multiple ways to handle parallel communication, we believe that **queues** and **notifiers** provide that one obvious, reliable way to achieve long-term success in complex applications.

References

[Introduction to Channel Wires \(NI.com\)](#)

[NIWeek 2019/Using and Abusing Channel Wires: An Exercise in Flexibility](#)

[Channel wires vs Queues \(Forum NI\)](#)