Everything You Ever Wanted To Know About Functional Global Variables

(Use DVRs Instead)

Nancy Hollenback, CLA

NI Field Architect – Americas Central West

LabVIEWjournal.com

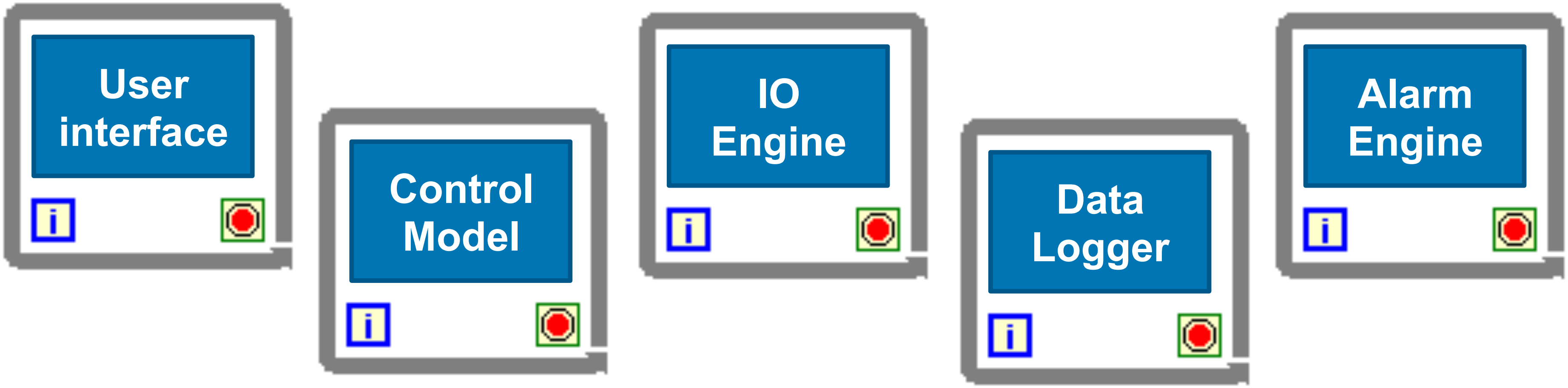
Nancy.Hollenback@ni.com

Agenda

- What is a functional global variable (FGV)?
- Does the FGV prevent race conditions?
- Is the FGV better than the global variable?
- Which use cases are a good fit for FGVs
- Is there a better way? (DVRs)
- Cool stuff with DVRs and Classes

Why Do We Need Functional Global Variables?

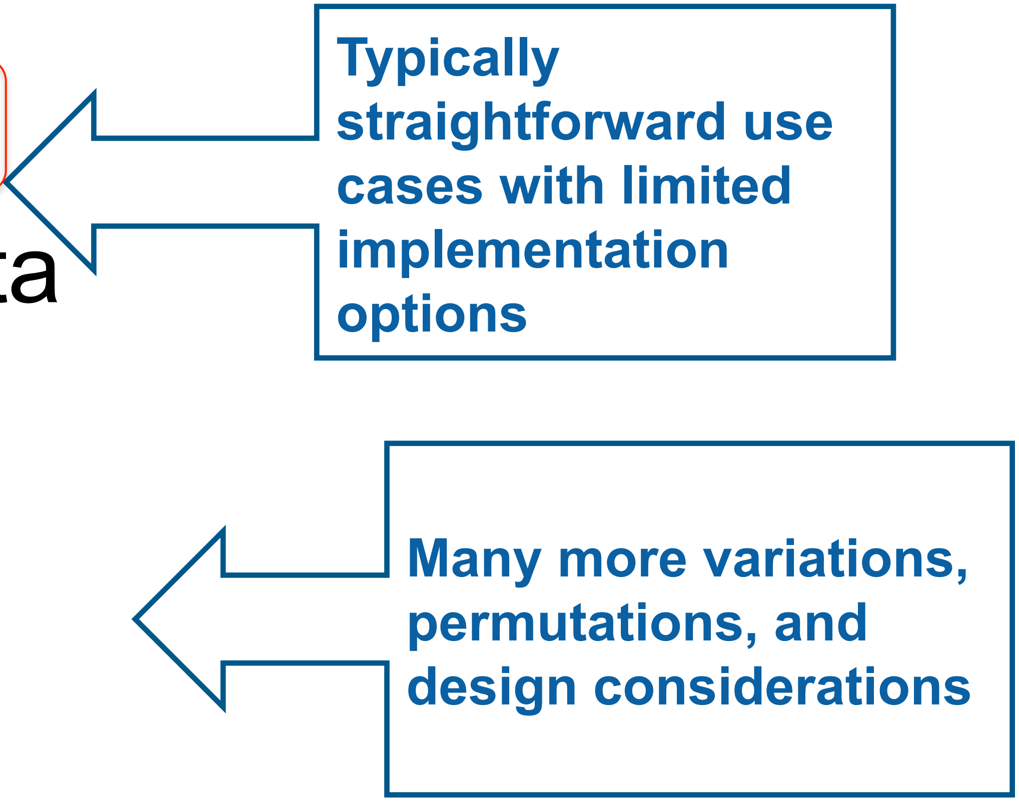
- A large application usually has many processes executing concurrently
- Processes need to share data or send and receive messages.



Inter Process Communication

- Store Data
- Stream Data

- Send Message

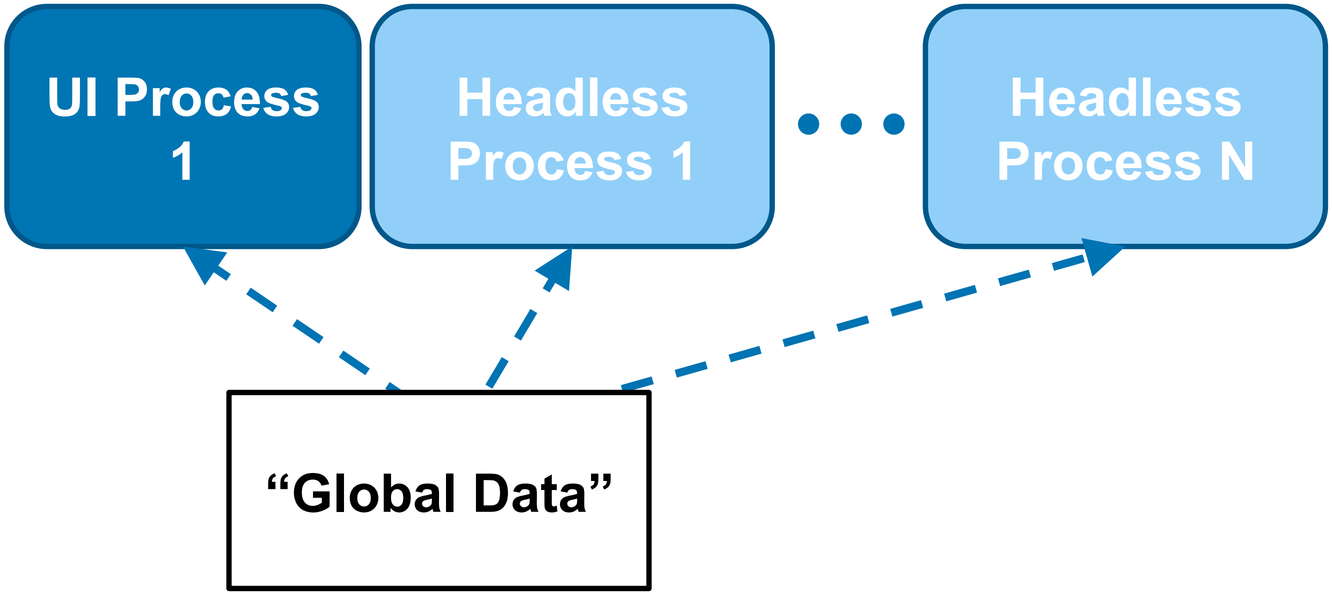


Typically straightforward use cases with limited implementation options

Many more variations, permutations, and design considerations

Store Data

- Data is stored and made “globally” accessible
- Storage mechanism holds only the current value
- Other code modules access the data as needed
- The potential for race conditions must be considered



Use Cases

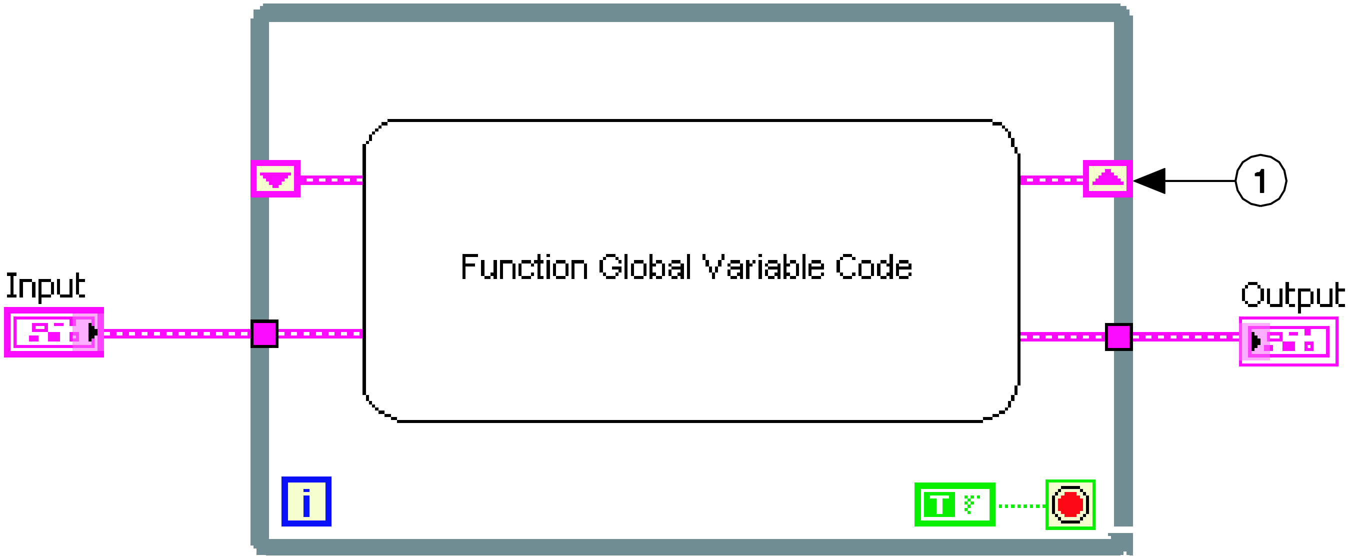
Configuration data
Slowly changing data
Non-critical messages

Functional Global Variables – Benefits

- Provide *global access to data* while also providing a framework to avoid potential race conditions.
- *Encapsulate data* so that debugging and maintenance is easier
- Facilitate the creation of *reusable modules* which simplifies writing and maintenance of code
- Program becomes more *readable*.

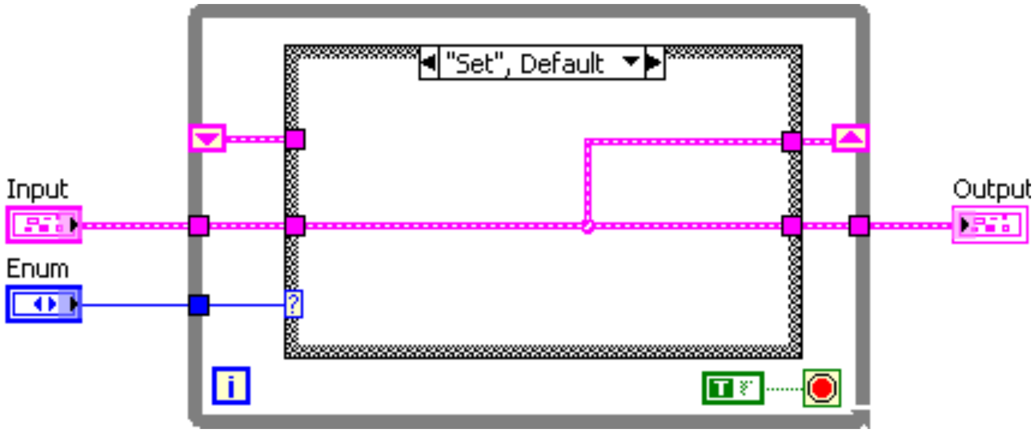
Functional Global Variable - Review

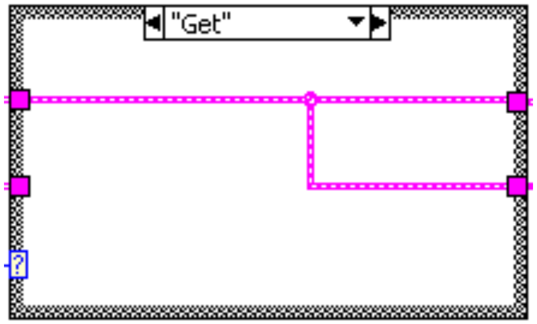
- The general form of a functional global variable includes an uninitialized shift register (1) with a single iteration For or While Loop



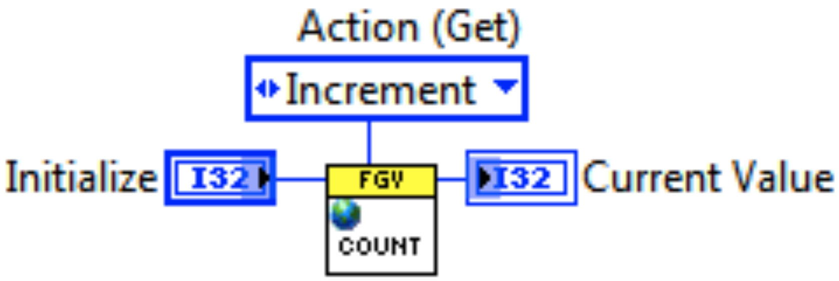
Functional Global Variables

- A functional global variable usually has an **action** input parameter that specifies which task the VI performs
- The VI uses an uninitialized shift register in a While Loop to hold the result of the operation





Best Practices for Documentation



- The action/method control should be a type defined enum.
- Make “get” the default action/method.
- Consider making the action/method required.
- Include this in the label.
- Wire to the top connector

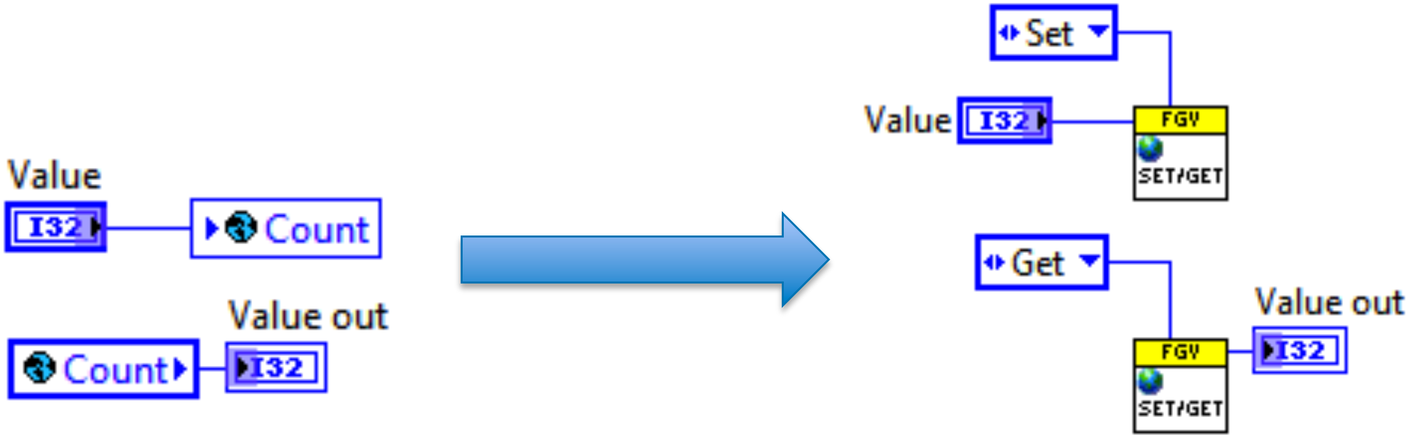
Functional Global Variables – History

- (LV2 Style Global, Action Engine, VIGlobals, USRs, Components)
  - Global data storage mechanism prior to the introduction of the global variable in LabVIEW 3
  - Foundational programming technique that has been in extensive use in the LabVIEW community

Note: The behavior of an uninitialized shift register was not defined in LabVIEW 1

Replacing Global Variables with FGVs

- This is a common initial use case.



Main – Using a Global

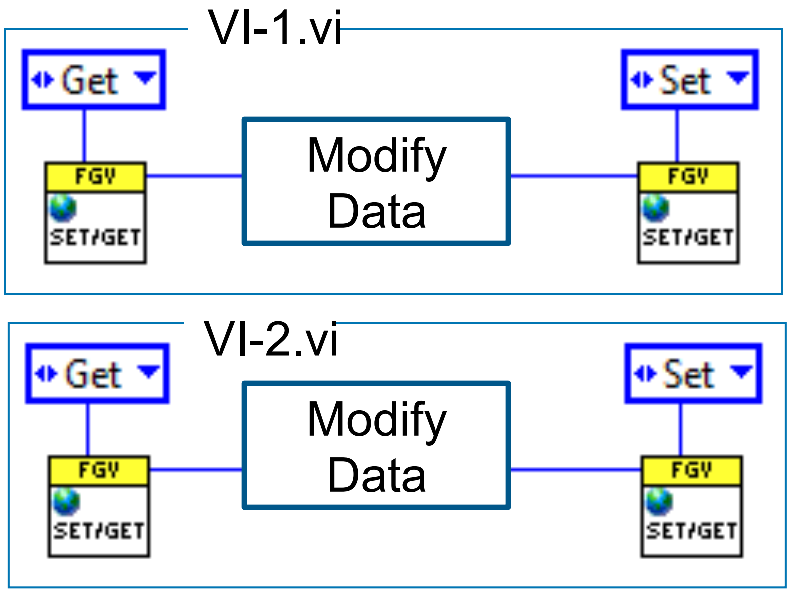
DEMO

Main – Using a Simple Set-Get FGV

DEMO

Do FGVs Eliminate Race Conditions?

- What if the FGV includes only set and get methods?



What happens when 2 VIs call the get and both modify the data before either has called the set?

Race Condition with a Set-Get Functional Global Variable

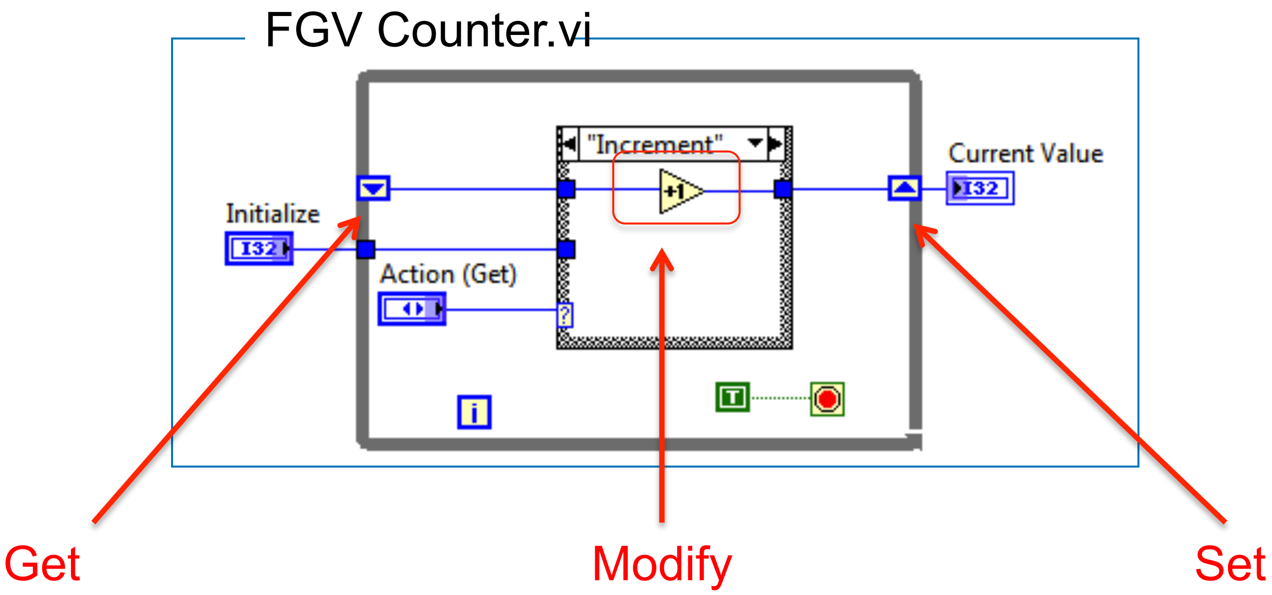
DEMO

Use FGVs to Protect Critical Sections of Code

- Identify a critical section of code, such as the modification of a counter value or a timer value.
- Identify the actions that modify the data (increment, decrement)
- Encapsulate the entire get/modify/set steps in the FGV

This is commonly called an Action Engine.
It is a special type of FGV.

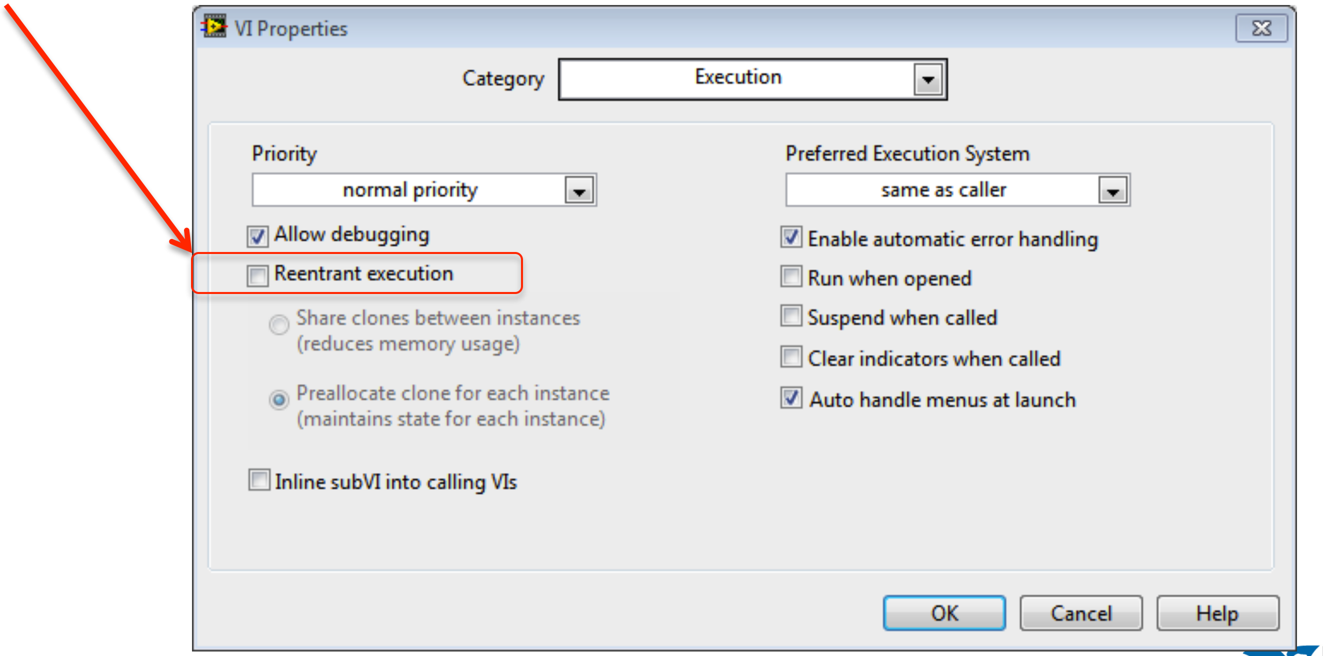
FGV – Action Engine Protects Critical Sections of Code



- This action engine wraps the “get/modify/set” around the critical section of code.

Sidebar: Execution Properties – Non Reentrant Execution

- VIs are non reentrant by default
- The LabVIEW execution system will not run multiple calls to the same SubVI simultaneously



VI Properties

Category: Execution

Priority: normal priority

☑ Allow debugging

□ Reentrant execution

○ Share clones between instances (reduces memory usage)

● Preallocate clone for each instance (maintains state for each instance)

□ Inline subVI into calling VIs

Preferred Execution System: same as caller

☑ Enable automatic error handling

□ Run when opened

□ Suspend when called

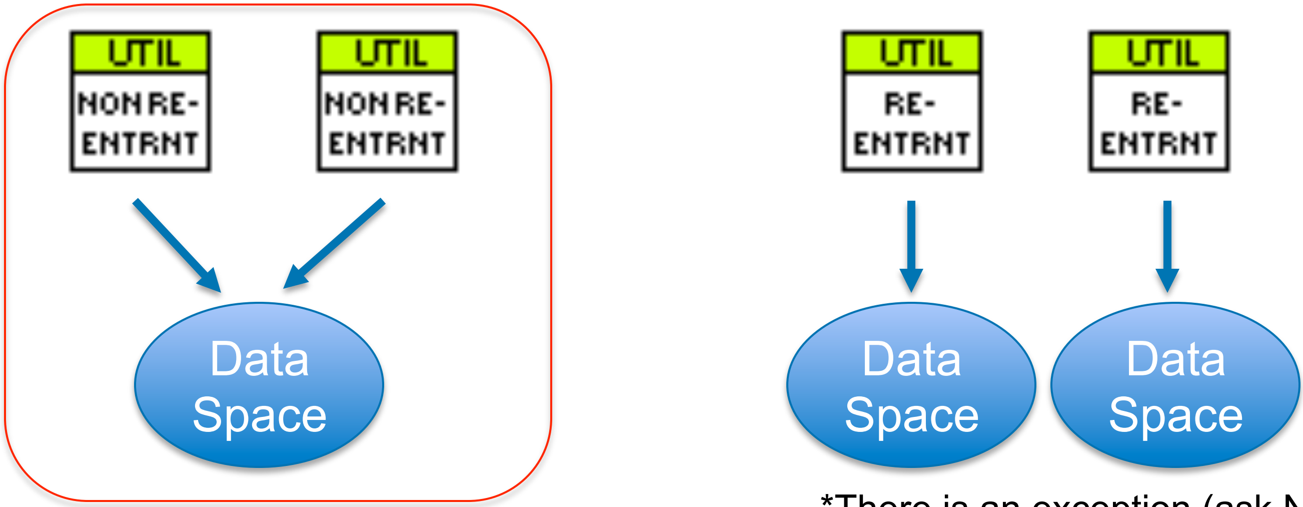
□ Clear indicators when called

☑ Auto handle menus at launch

Sidebar: Reentrant vs. Non-Reentrant

- Non reentrancy is required for FGVs*
- Reentrancy allows one subVI to be called simultaneously from different places.
  - To allow a subVI to be called in parallel
  - To allow a subVI instance to maintain its own state

State (or the data that resides in the uninitialized shift register) is maintained between all instances of the FGV



*There is an exception (ask Nate)

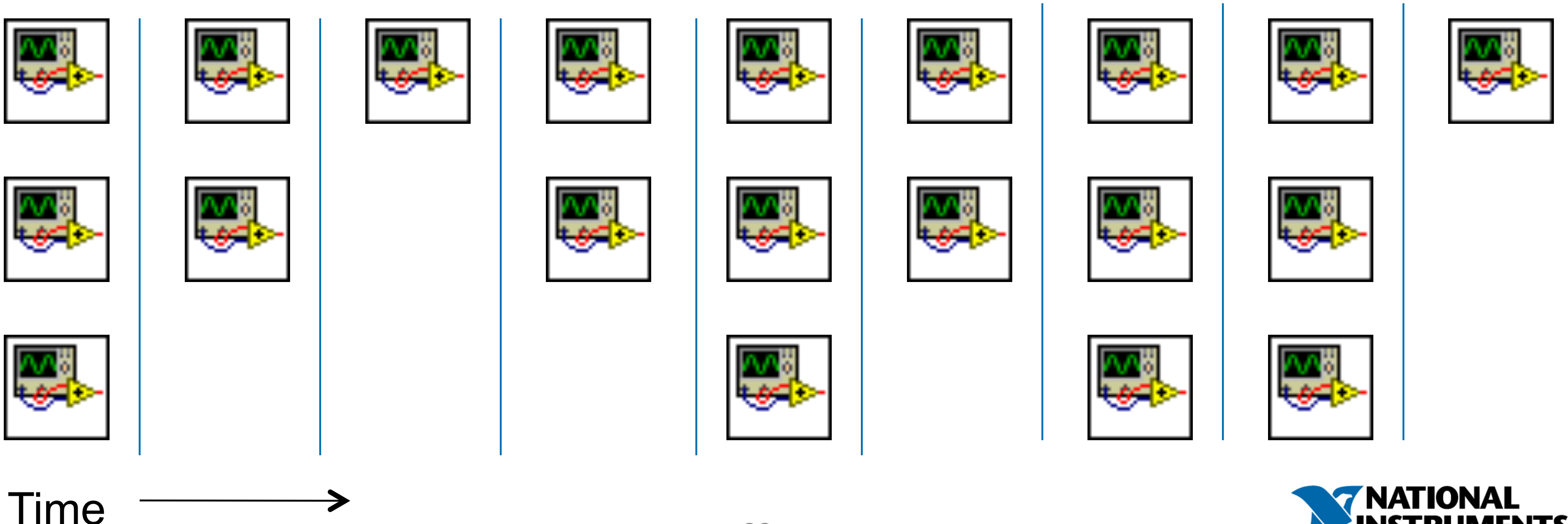
Non Reentrant VIs Block Other Calls

Increment
FGV
COUNT
Increment
FGV
COUNT

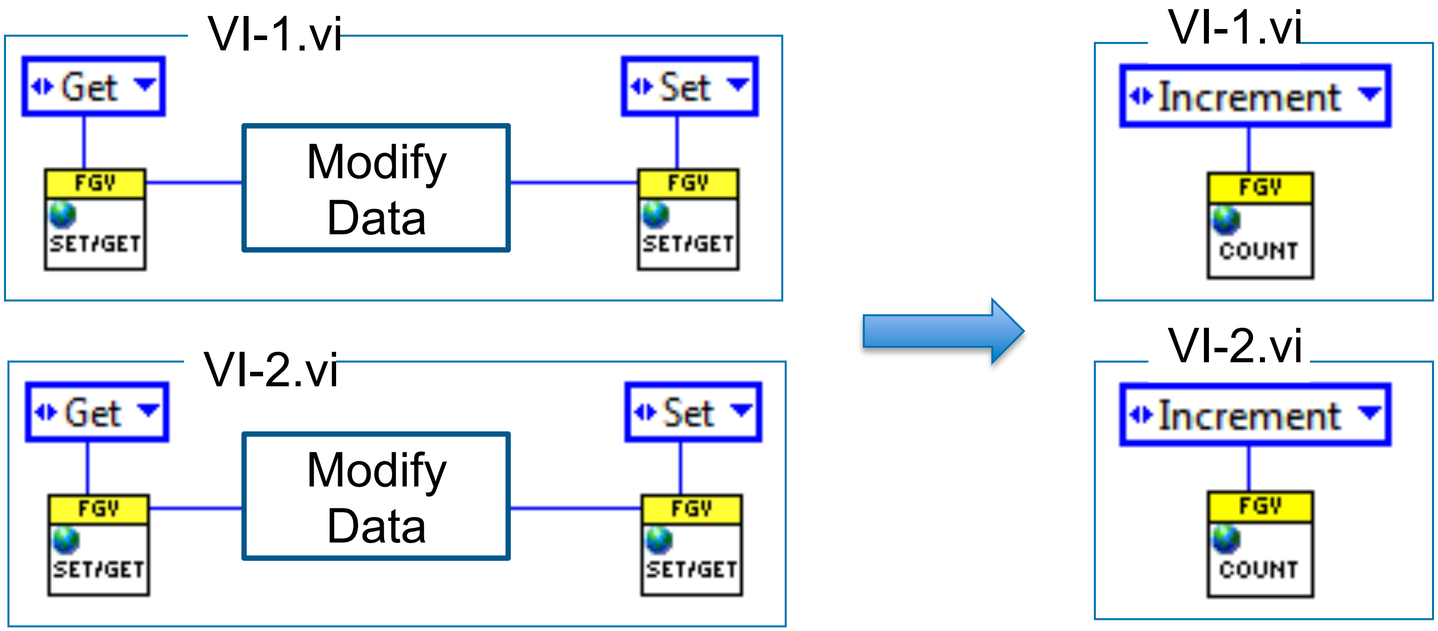
- These two VIs are non reentrant by default
- They cannot run simultaneously
- One will run until completion and block the other from running until completed.

Shared Clones vs. Preallocate clones

- 20 unique Instances of a reentrant VI
- During execution, max of 3 instances called simultaneously
- Preallocate – 20 Clones
- Shared Clones – 3 Clones



Action Engines Protect Critical Sections!



The FGV will block other instance from running until it has completed execution. Therefore, encapsulating the entire action prevents the potential race condition.

Avoid Race Conditions!!! Fully encapsulate the get/modify/set.

Action Engine FGV

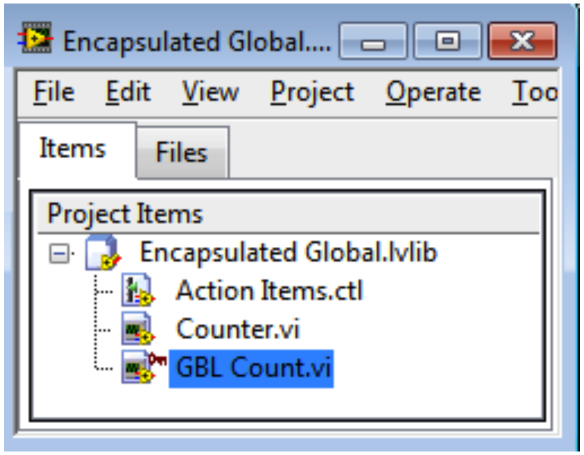
DEMO

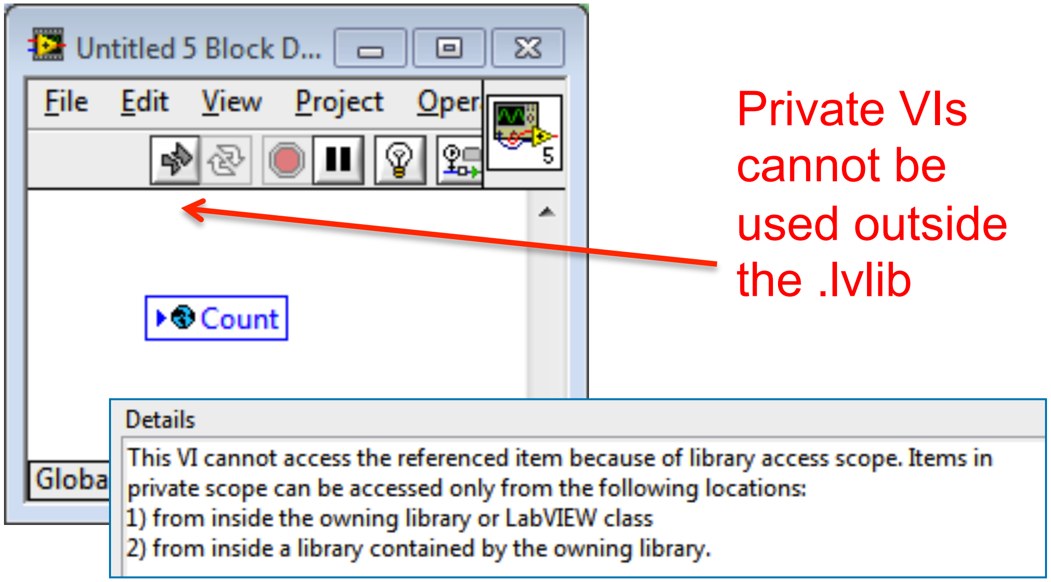
Globals vs FGVs

- Globals are significantly faster.
- FGVs allow for extra code to check for valid data.
- What if we used a project library to encapsulate a global?
  - Make the global private
  - Write VIs to access the data

Encapsulated Global

- Create a global variable
- Add it to a project library and set access scope to private





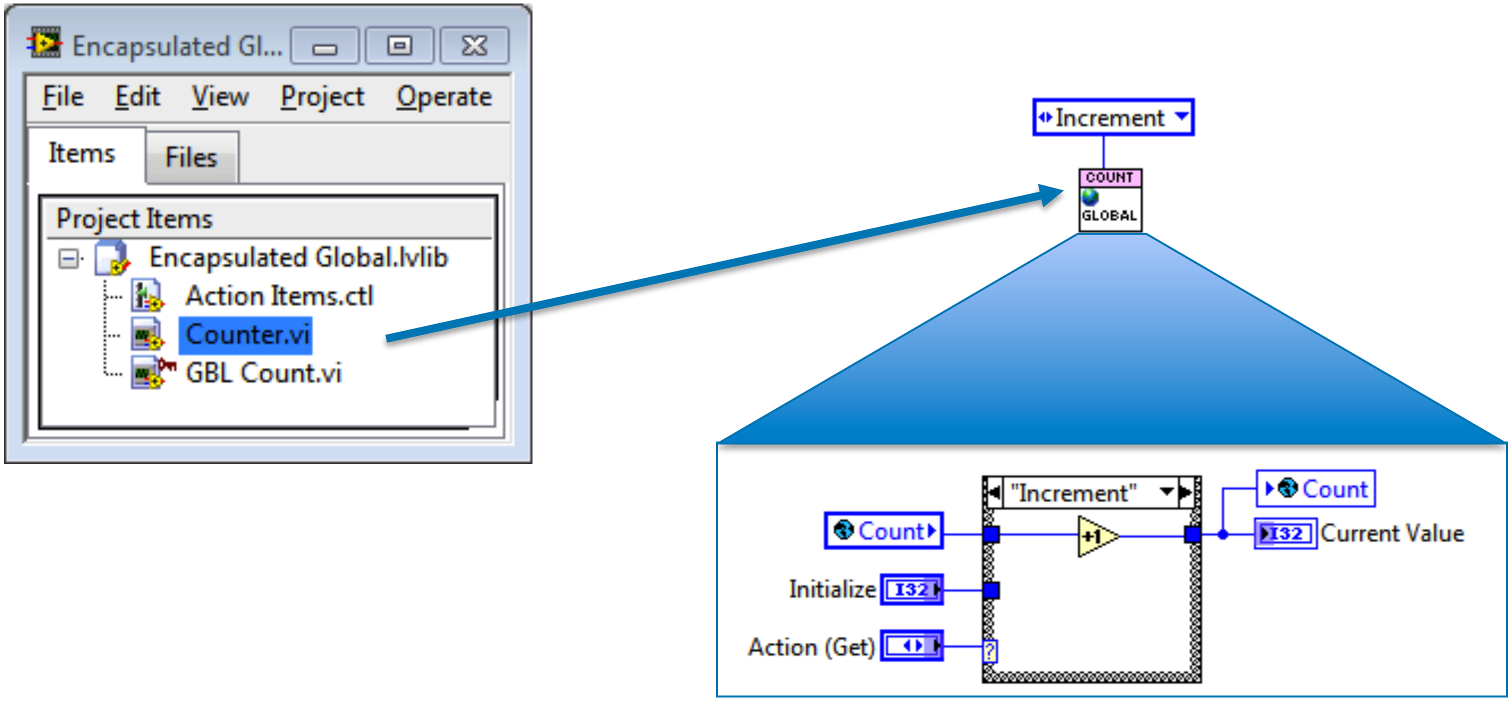
Details

This VI cannot access the referenced item because of library access scope. Items in private scope can be accessed only from the following locations:
1) from inside the owning library or LabVIEW class
2) from inside a library contained by the owning library.

Private VIs cannot be used outside the .lvlib

Encapsulated Global

- Create the VI in the lvlib, that will act on the privately scoped global variable.



Consider locking and password protecting the .lvlib

Encapsulated Global

DEMO

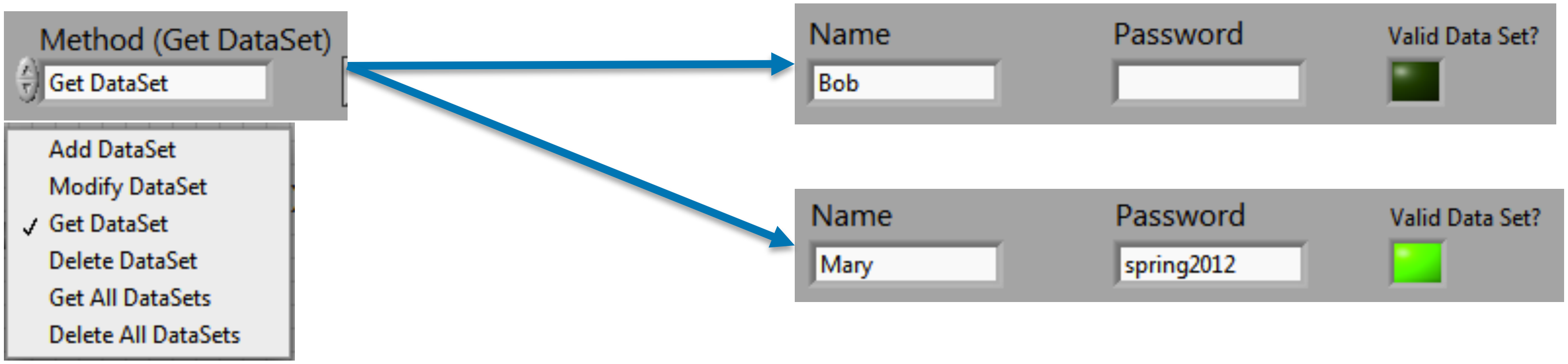
Reusable components with FGVs

- Recall that FGVs encapsulate the data and functionality and as such are a good design pattern for building reusable components
- Consider using a FGV as a look-up table.

Name	Password
John	66ford90
Mary	spring2012

Array of names has corresponding array of values or datasets

Name Value Look Up Table



Method (Get DataSet): Get DataSet

Name: Bob
Password:
Valid Data Set? ●

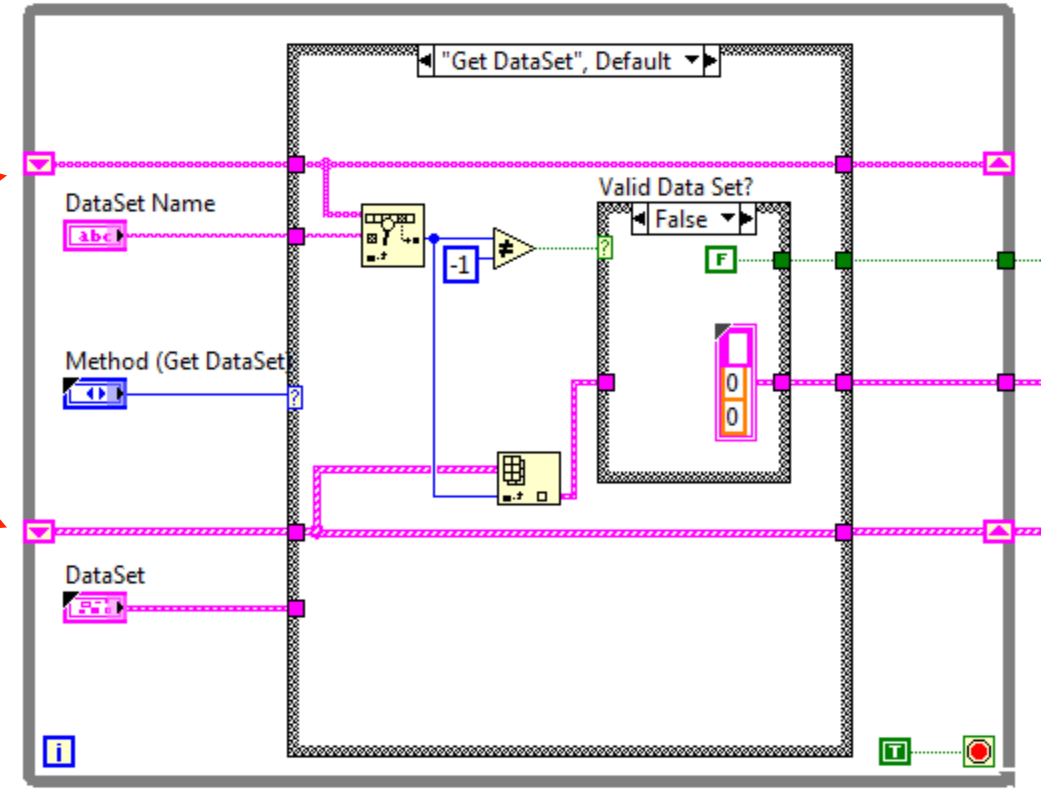
Name: Mary
Password: spring2012
Valid Data Set? 🟩

- Define the data type of the value that is associated with the name.
- Modify the method to include all actions to perform related to adding, getting, and deleting items from the list.
- Add code to ensure whether data is valid

FGV – Resource Storage

Design pattern for a key-value look up table.

- Array of names has a one-to-one correspondence to the array of data sets
- Does not protect against race conditions
- Allows for the qualification of valid data



FGV Password Storage

DEMO

Resource Storage FGVs

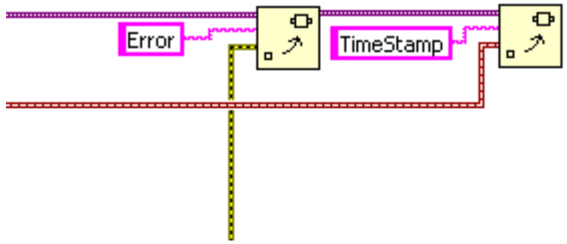
- Build drop-in reusable components.
- Provide protection and validation of data.
- Susceptible to race conditions.
- Can be used to store:
  - References (User Events, DVRs, etc)
  - Information about devices
  - Paths for data storage
  - Operator information
  - Anything that requires a name-value lookup

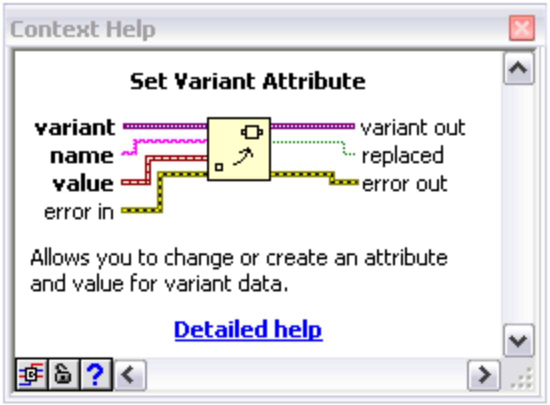
Creating Your Own Resource FGV

DEMO

Variant Attributes

- Very flexible mechanism for storing data
- Hash table in which the value can be any data type.





Set Variant Attribute

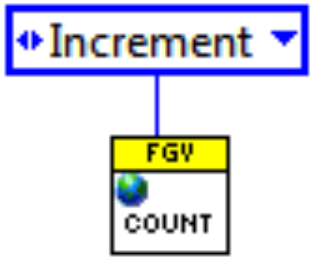
Allows you to change or create an attribute and value for variant data.

FGV with Variant Attribute

DEMO

What if You Need Multiple Counters…

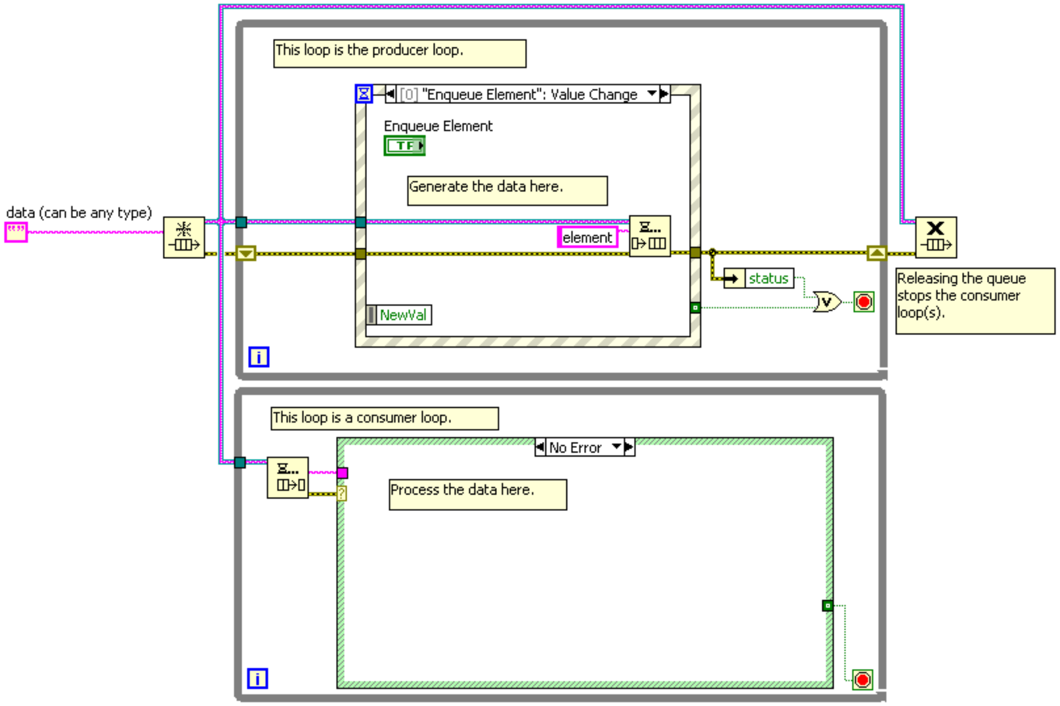
- Reentrant functional global?
- Array manipulation of the functional global data?
- Perhaps there is a better way…



?

Review of Queues and References

- Reference is a pointer to the data
- The wire contains the reference, not the data.
- Forking the wire creates a copy of the reference, not a copy of the data
- Access data through methods (VIs)
- Developer controls the creation and destruction of the data



This loop is the producer loop.

Generate the data here.

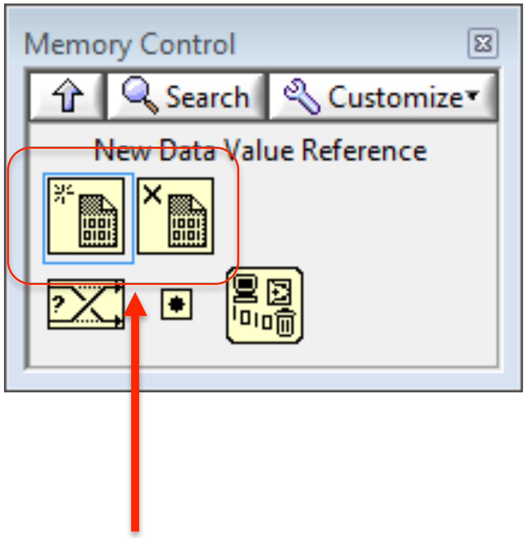
Releasing the queue stops the consumer loop(s).

This loop is a consumer loop.

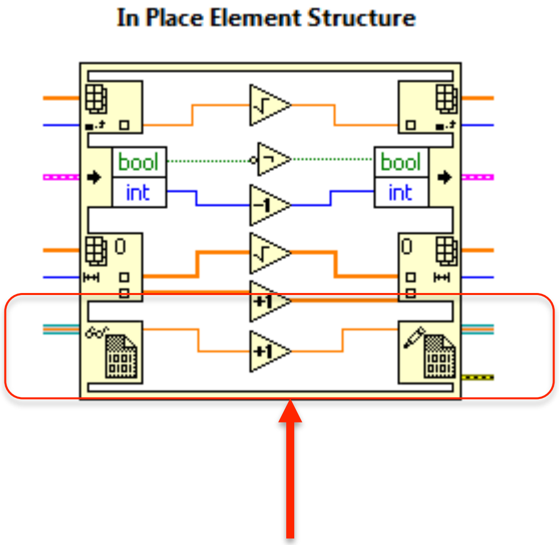
Process the data here.

What is the Data Value Reference (DVR)?

- This is a simple way to wrap a reference around any type of data.

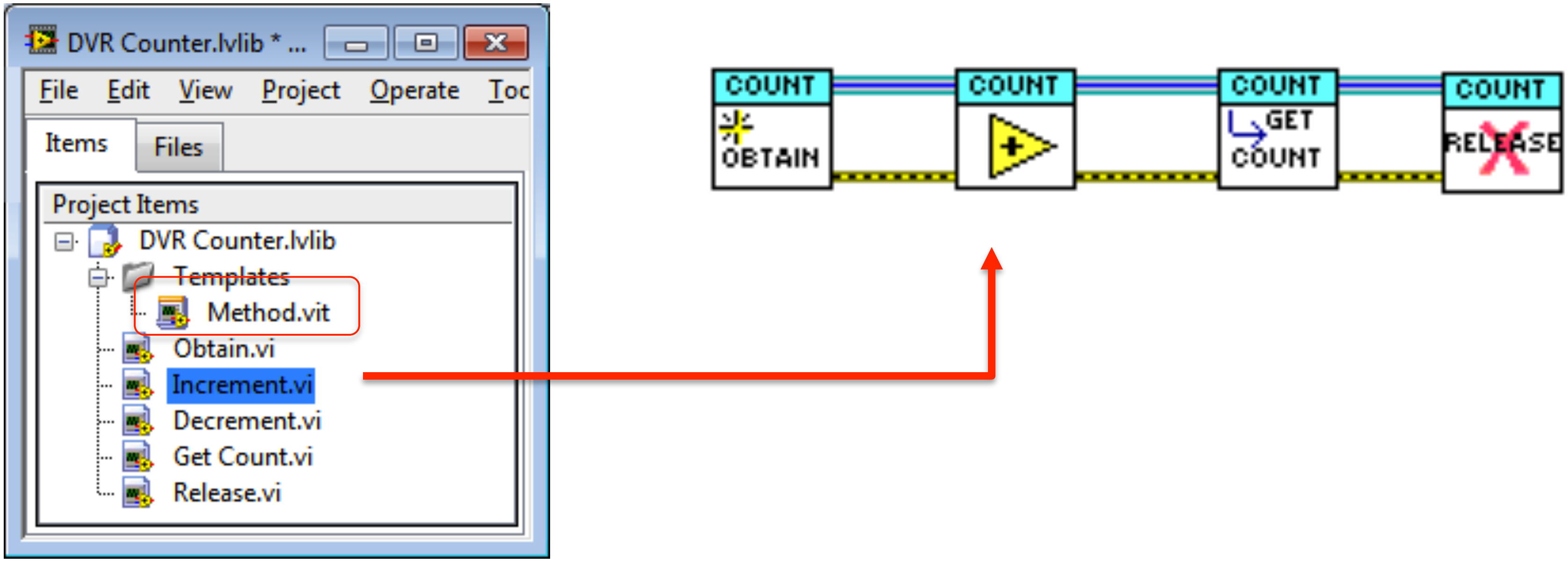


Create & Destroy



Modify

Data Value Reference (DVR) Library


DVR Counter.lvlib * ...

File Edit View Project Operate Toc

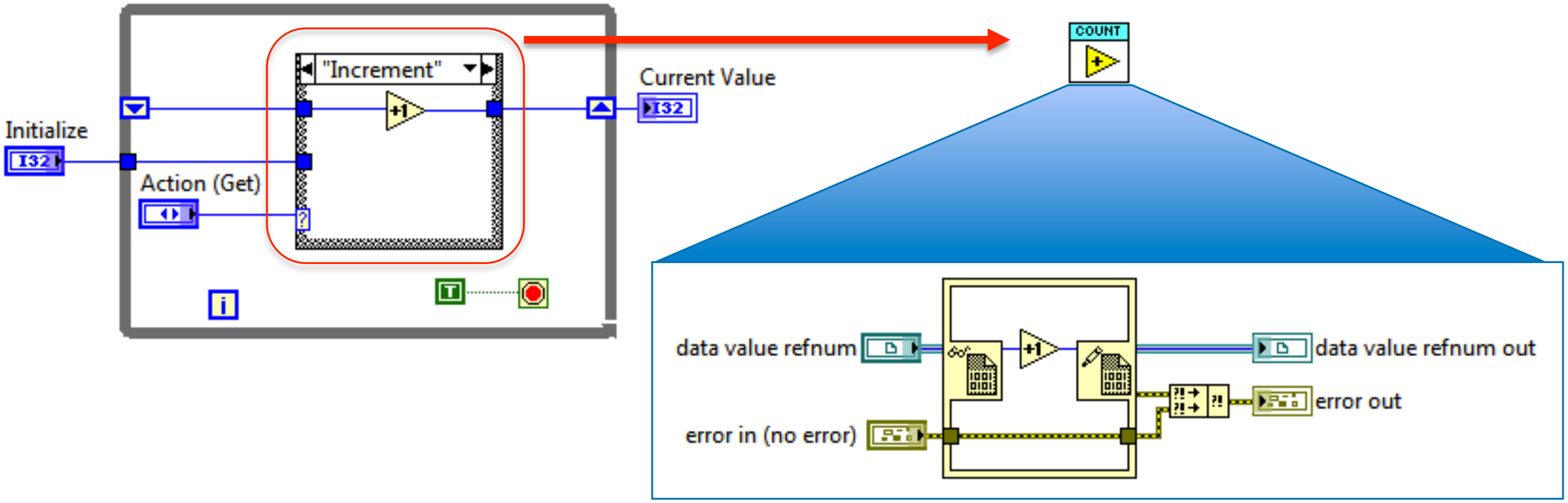
Items Files

- Project Items
  - DVR Counter.lvlib
    - Templates
      - Method.vit
    - Obtain.vi
    - Increment.vi
    - Decrement.vi
    - Get Count.vi
    - Release.vi

- Create a constructor and destructor.
- Create a template for the methods.
- Create a method for each case that will modify the data.

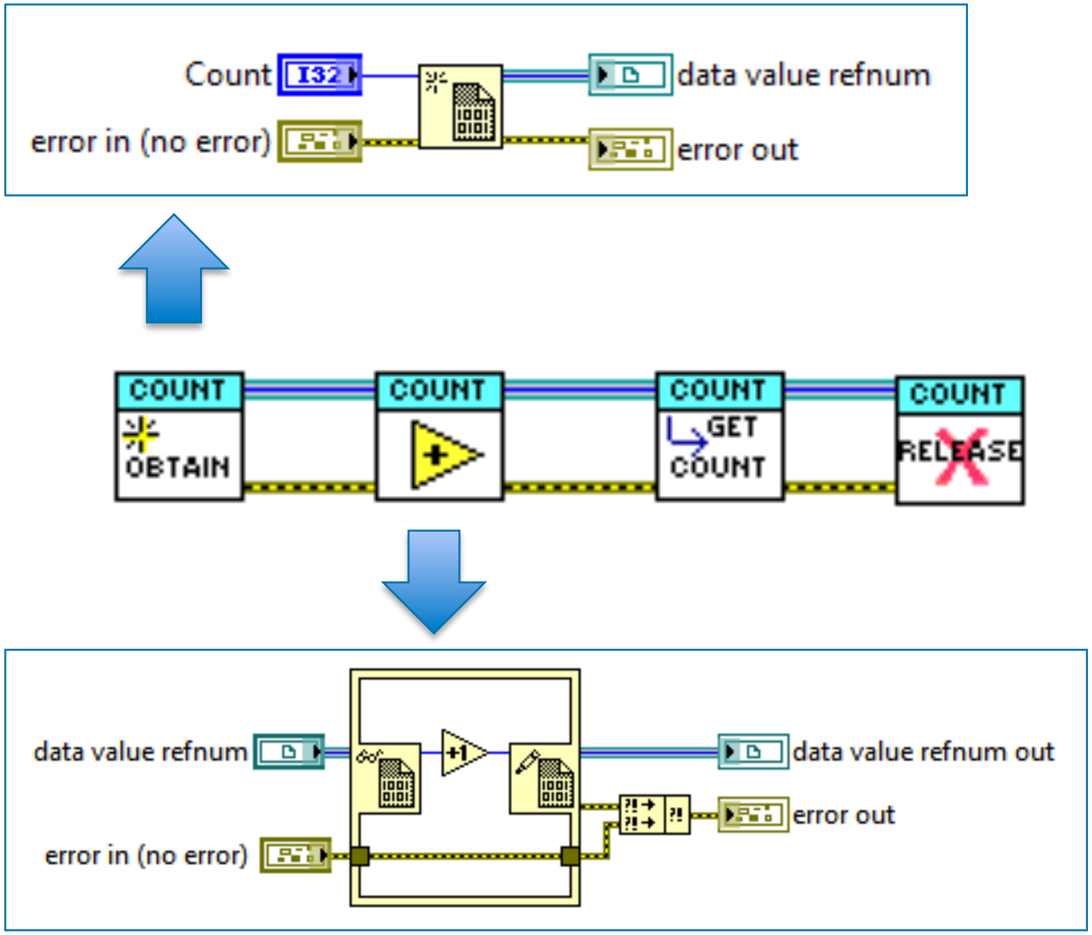
Creating a DVR from an FGV

- If you already have an FGV, you can easily transform it into the more flexible DVR library.
- Create the constructor and destructor.
- Create a method (VI) for each case that was in the FGV.



Data Value Reference (DVR) - Library

- Reference acts as a pointer to the data
- Create unlimited instances
- Easily expand the library



Using a DVR Library

DEMO

DVR Library Design Issues

- Easily add new methods (VIs) to the library as needed.
- Create a library the has a similar look and feel to native APIs (Queues, Notifiers, Semaphores)
- Identify the owner of the library who will update and maintain the library.
- Anyone with Core 1 & Core 2 understanding can use the DVR library.

Add a Method to the DVR Library

DEMO

Cool Stuff with DVRs and Classes

DEMO

Summary Slide

- Use Action Engines
- Use FGVs for Resource Storage
- Learn about DVRs
- Learn about other techniques for messaging.

What Else Do I need to Know?

- Store Data
- Stream Data

Typically straightforward use cases with limited implementation options

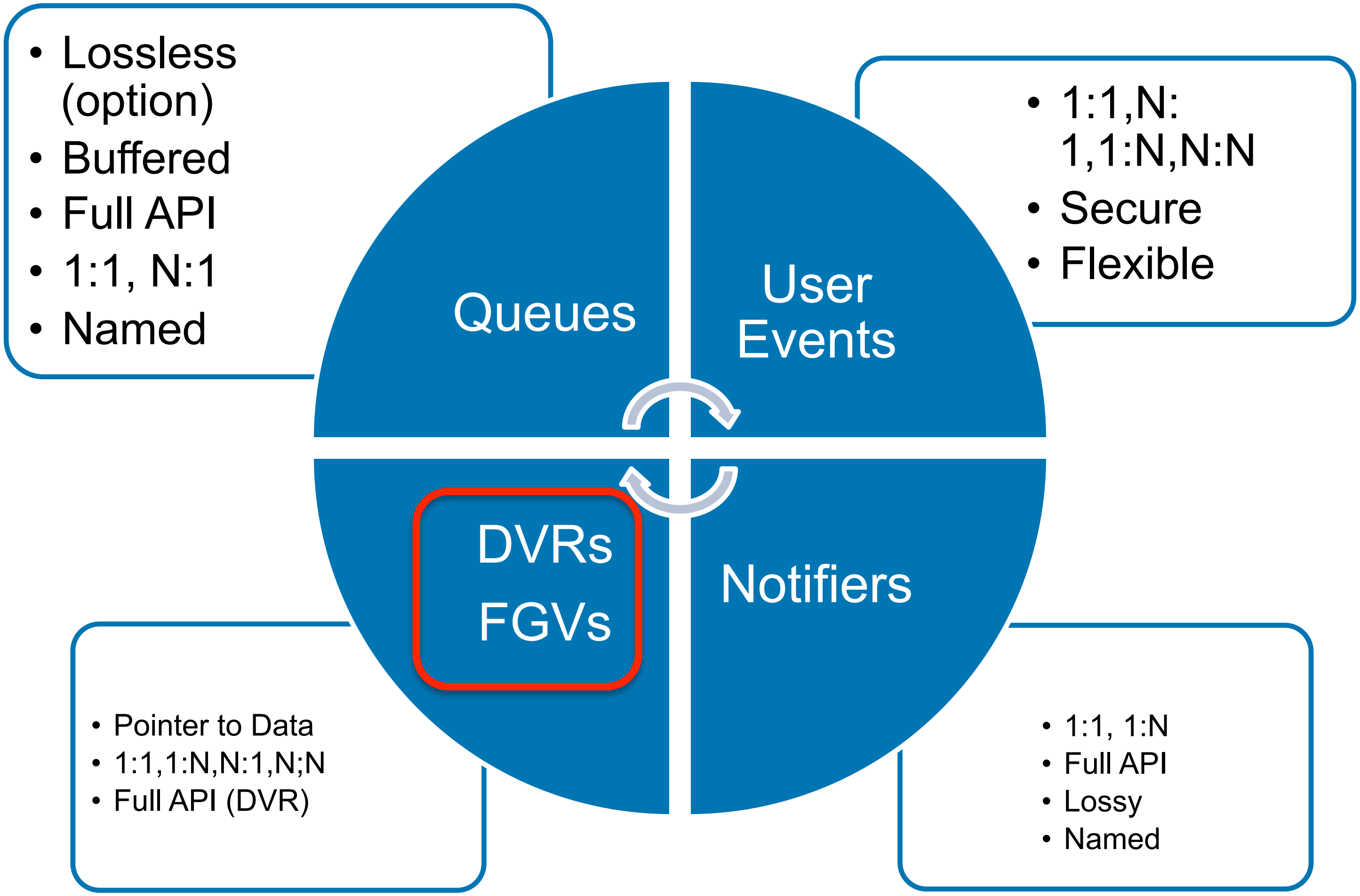
- Send Message

Many more variations, permutations, and design considerations

Various Inter-process Communication Methods

	Same target Same application instance	Same target, different application instances OR Different targets on network
Storing - Current Value	• Single-process shared variables • Local and global variables • FGV, SEQ, DVR • CVT • Notifiers (Get Notifier)	• Network-published shared variables (single-element) • CCC
Sending Message	• Queues (N:1) • User events (N:N) • Notifiers (1:N) • User Events	• TCP, UDP • Network Streams (1:1) • AMC (N:1) • STM (1:1)
Streaming	• Queues	• Network Streams • TCP

Foundational APIs for Storing & Messaging



- Lossless (option)
- Buffered
- Full API
- 1:1, N:1
- Named

- 1:1,N:1,1:N,N:N
- Secure
- Flexible

- Pointer to Data
- 1:1,1:N,N:1,N;N
- Full API (DVR)

- 1:1, 1:N
- Full API
- Lossy
- Named

Thank You!