
NATIONAL
INSTRUMENTS™

Foundational Design Patterns for Moving Beyond One Loop

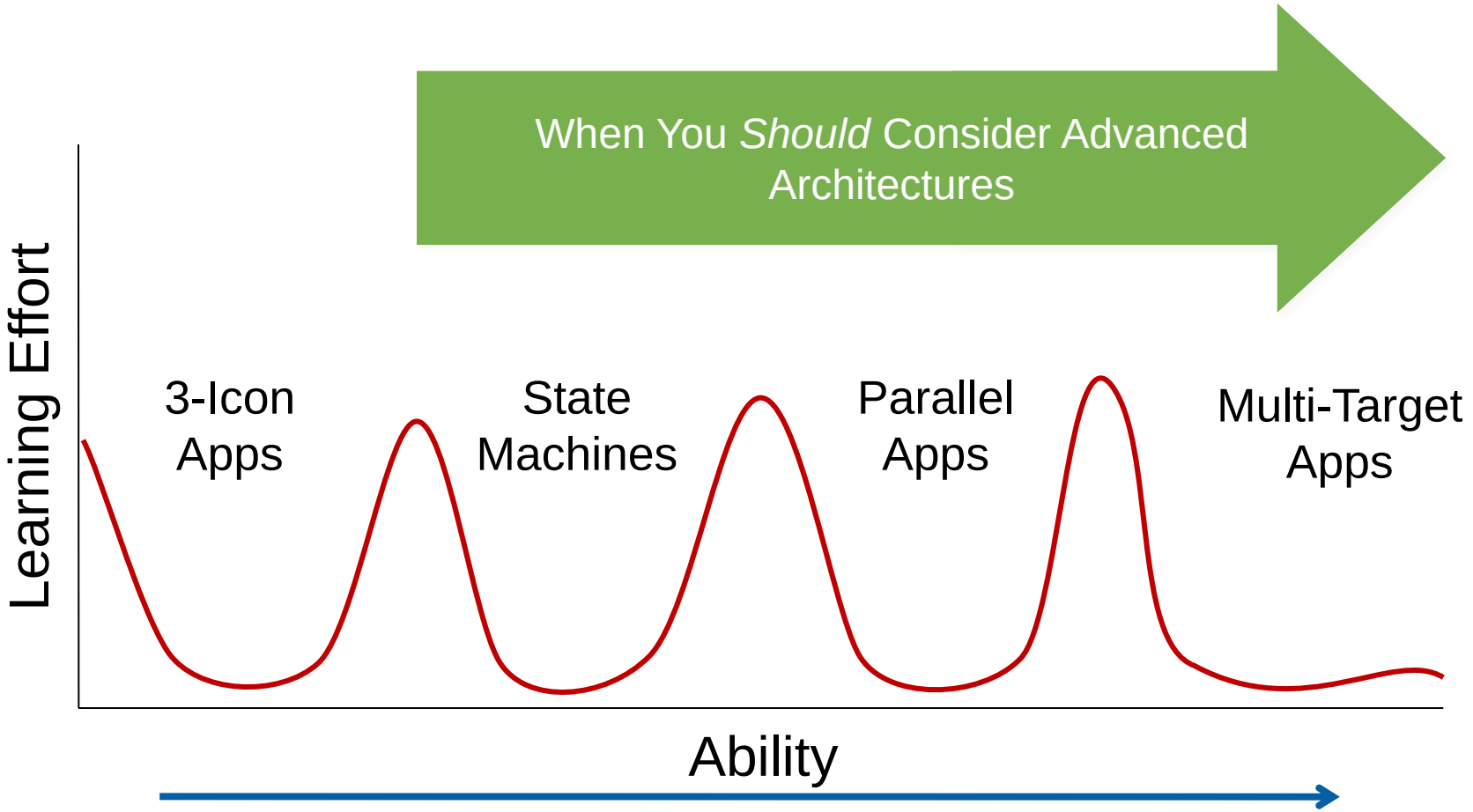
Steve Chiang
Technical Marketing Engineer

NATIONAL INSTRUMENTS

Agenda

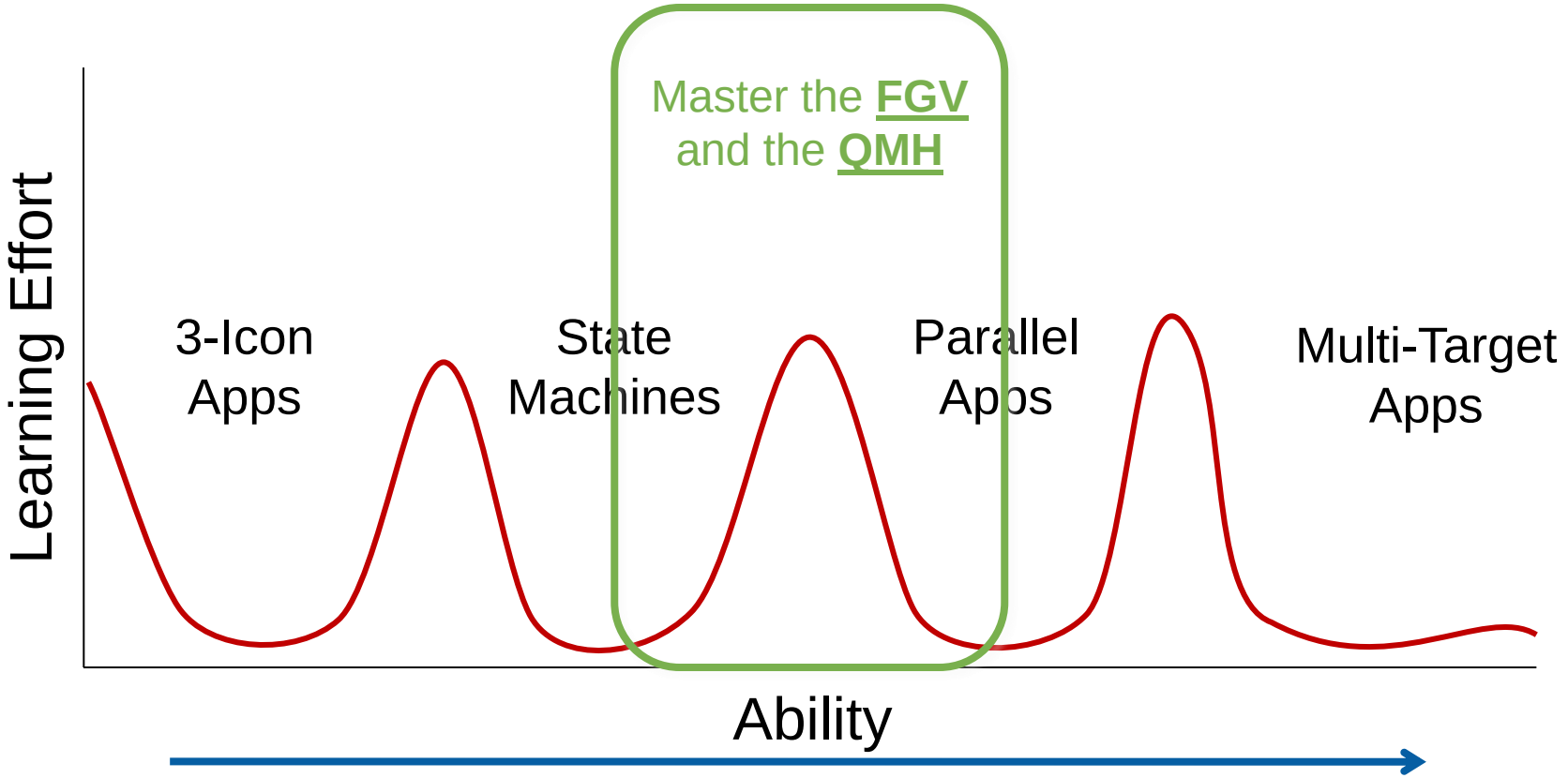
- Why move beyond one loop?
- What is a design pattern?
- Why learn communication mechanisms?
  - Functional global variables
  - Queued message handlers

Defining Advanced Applications & LabVIEW Ability

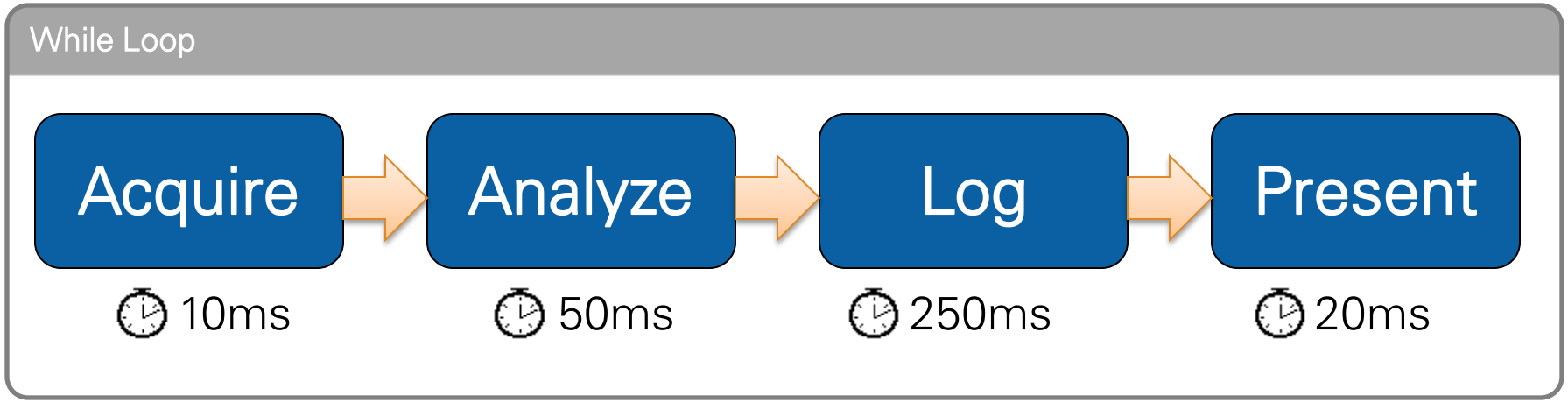


Master Key Design Patterns…

…and Understand Their Limitations

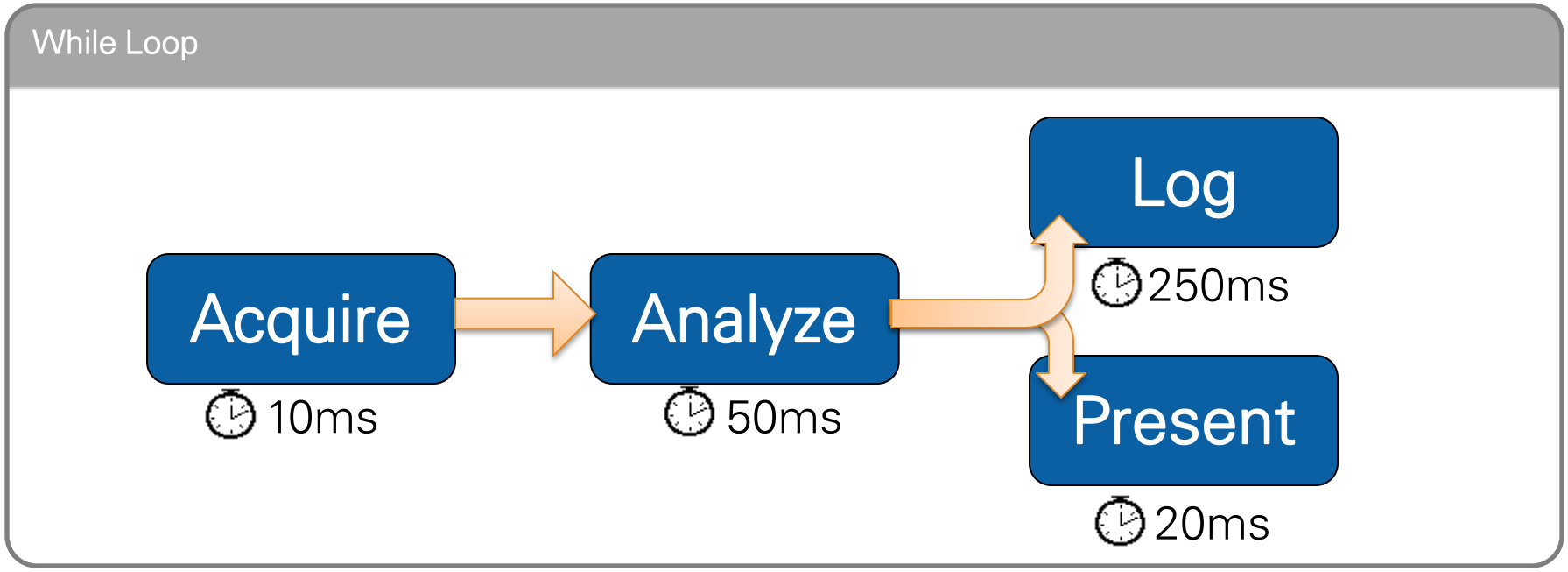


Doing Everything in One Loop Can Cause Problems



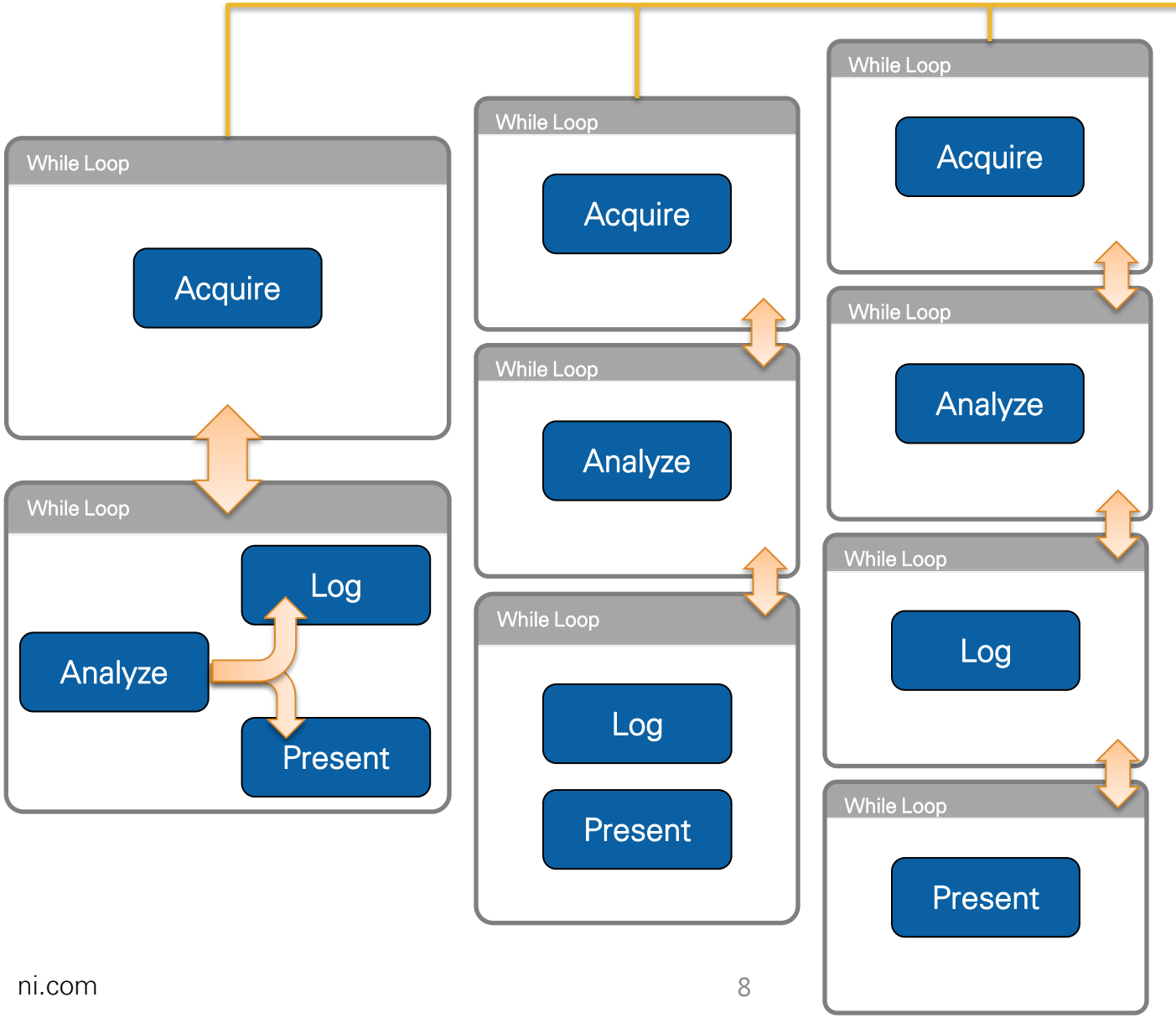
- One cycle takes at least 330 ms
- If the acquisition reads from a buffer, it may fill up
- User interface can only be updated every 330 ms

Doing Everything in One Loop Can Cause Problems



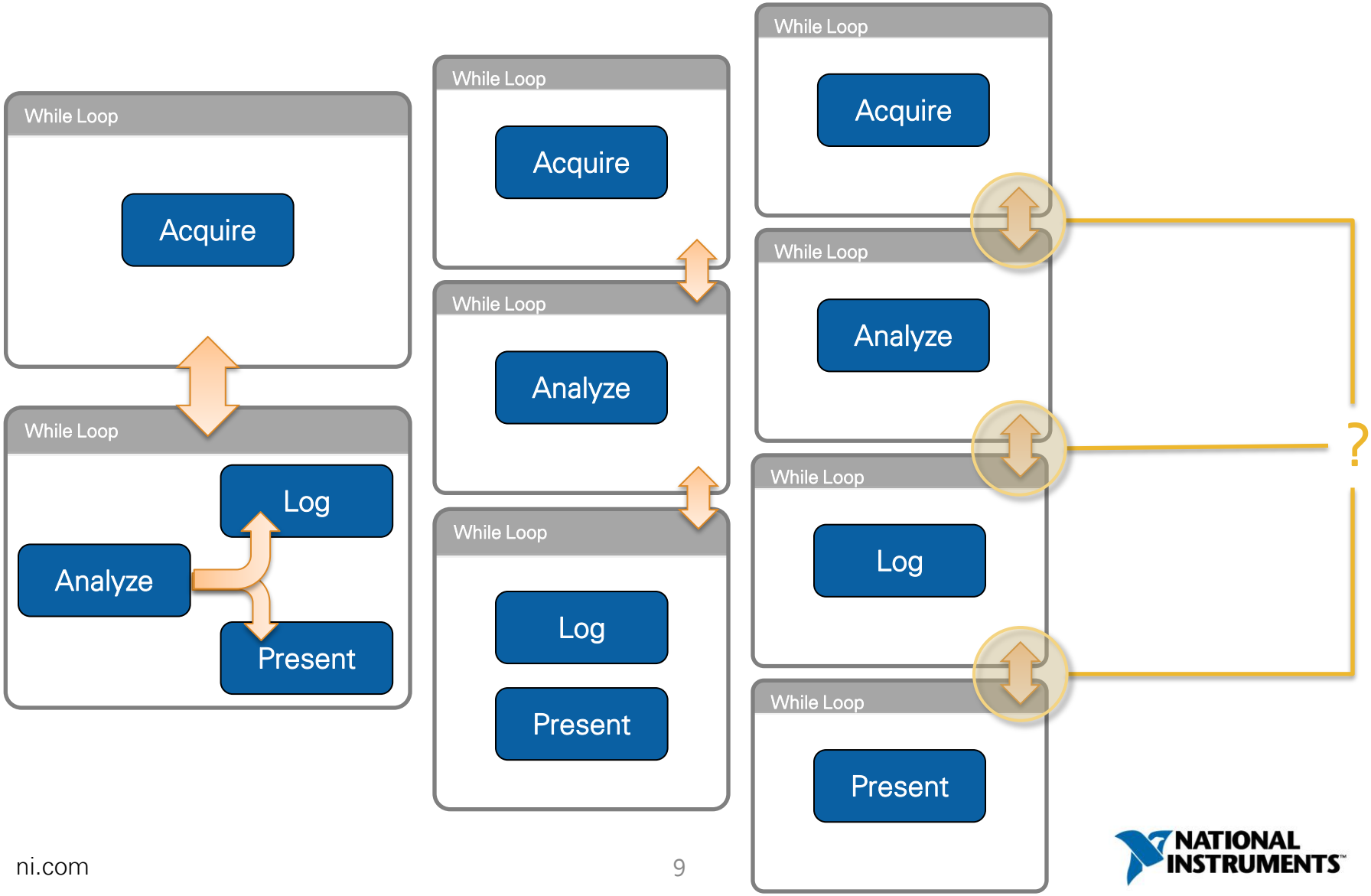
- One cycle still takes at least 310 ms
- If the acquisition reads from a buffer, it may fill up
- User interface can only be updated every 310 ms

How do we implement multiple loops?



NATIONAL INSTRUMENTS

How do we communicate between loops?



NATIONAL INSTRUMENTS

"Don't Re-invent the Wheel...

...or Worse, a Flat Tire."

-Head First Design Patterns

What is a design pattern and why use one?

- Definition:
  - A general reusable approach to a commonly occurring problem
  - Well-established, proven techniques
  - A formalized best practice
  - **Not** a finished design

- Design patterns:
  - Save time
  - Improve code longevity
  - Improve code readability
  - Simplify code maintenance

What is the difference between a design pattern and a framework?

Design Pattern

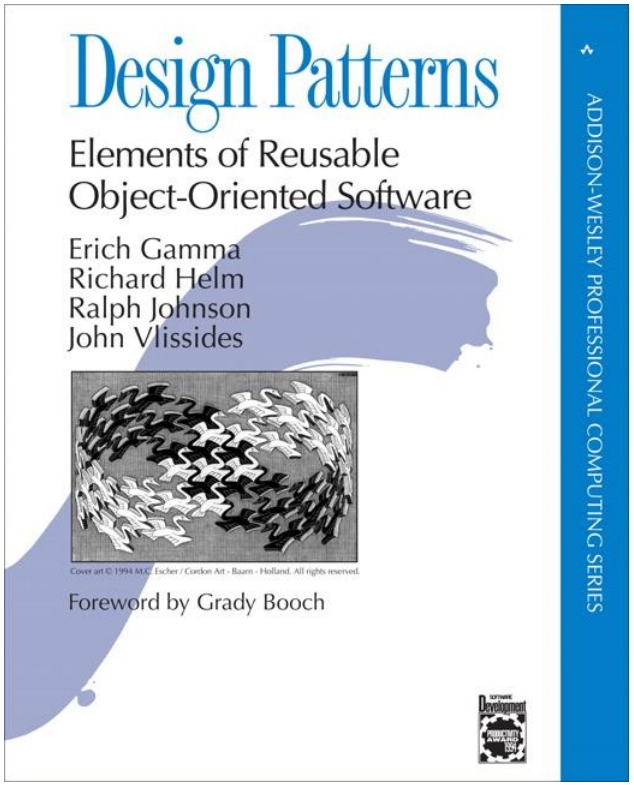
- Focuses on a specific problem
- Geared toward solo developers
- More foundational

Framework

- Focuses on larger applications
- Geared toward team development
- Often involve an architect

Design Patterns are **not** Specific to LabVIEW

- Gained popularity from the "Gang of Four" (GoF or Go4)
- Includes examples in C++, Smalltalk
- The LabVIEW community has adopted and extended several design patterns for use with LabVIEW
  - Examples:
    - Producer / Consumer
    - State Machine
    - Queued Message Handler
    - Factory Pattern
    - Singleton Pattern

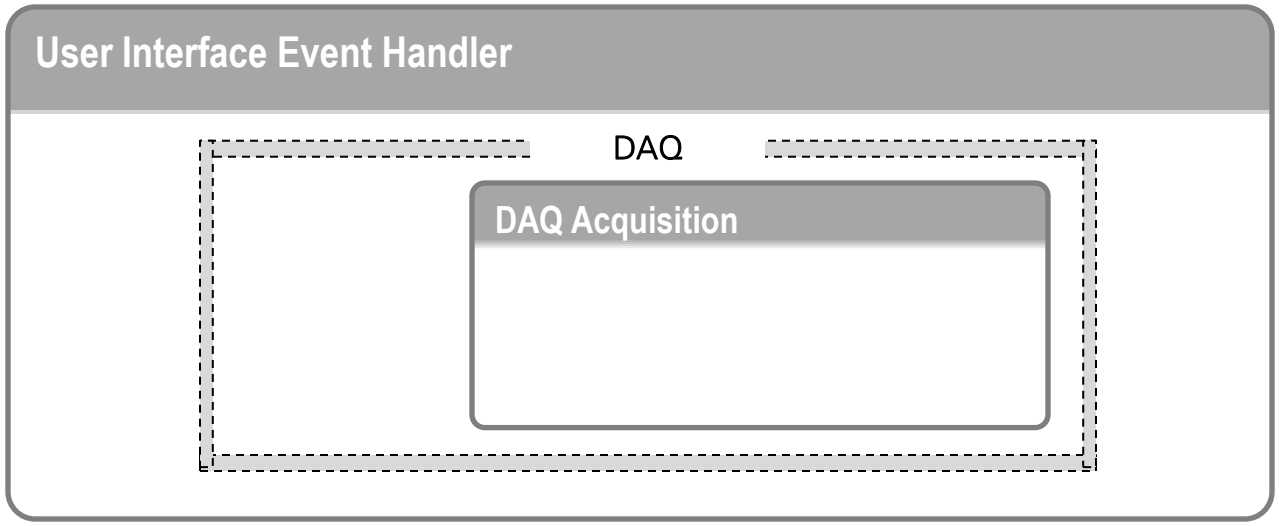


Risks Associated with a "Flat Tire"

(Poor Application Design)

- Nothing happens when you press "Stop"
- Poor error handling
- No consistent style
- Complicated maintenance and debugging
- Difficulty scaling application
- Tight coupling, poor cohesion

Why is Coupling Bad?



Execution: The user interface cannot respond to other input while acquiring data

Development: Modifying one process requires modifying the other

Extensibility: How do you add a data logging process?

Fragility: Can you fix a bug in one process without affecting another?

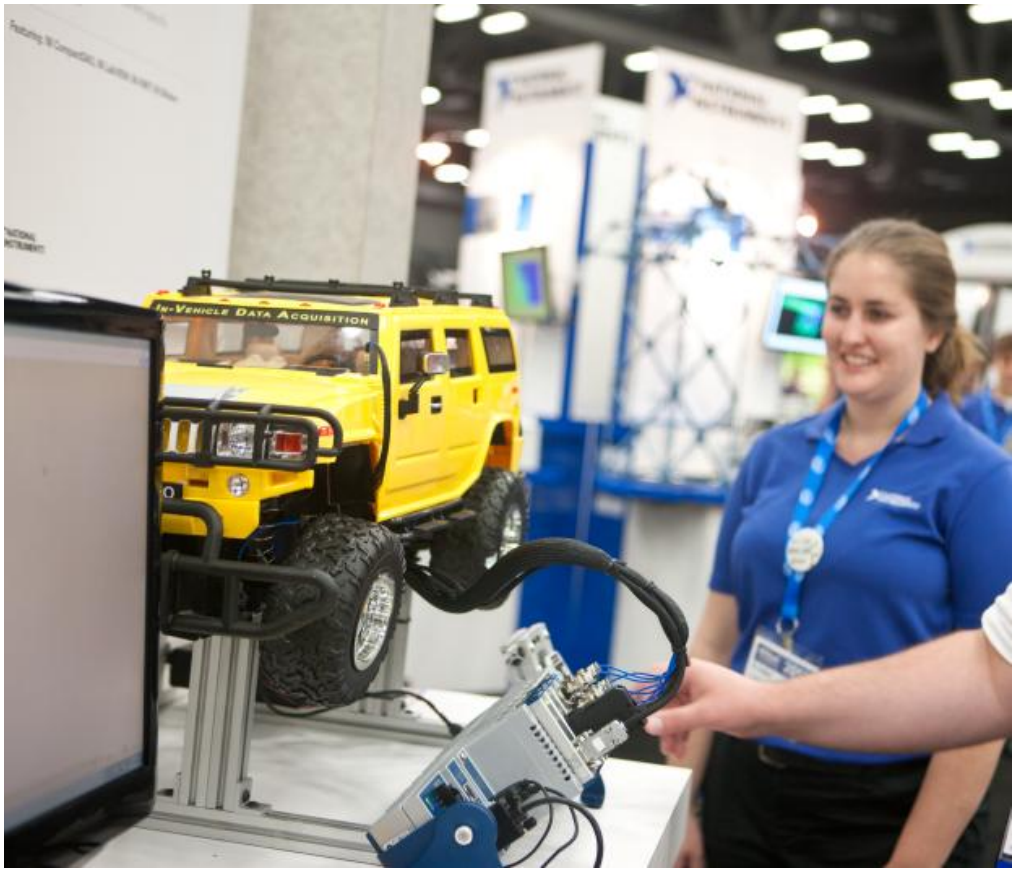
Maintenance: Can you test one piece of an application without testing its entirety?

Coupling is Often Accidental: Case Study

NIWeek 2011 Hummer Demo

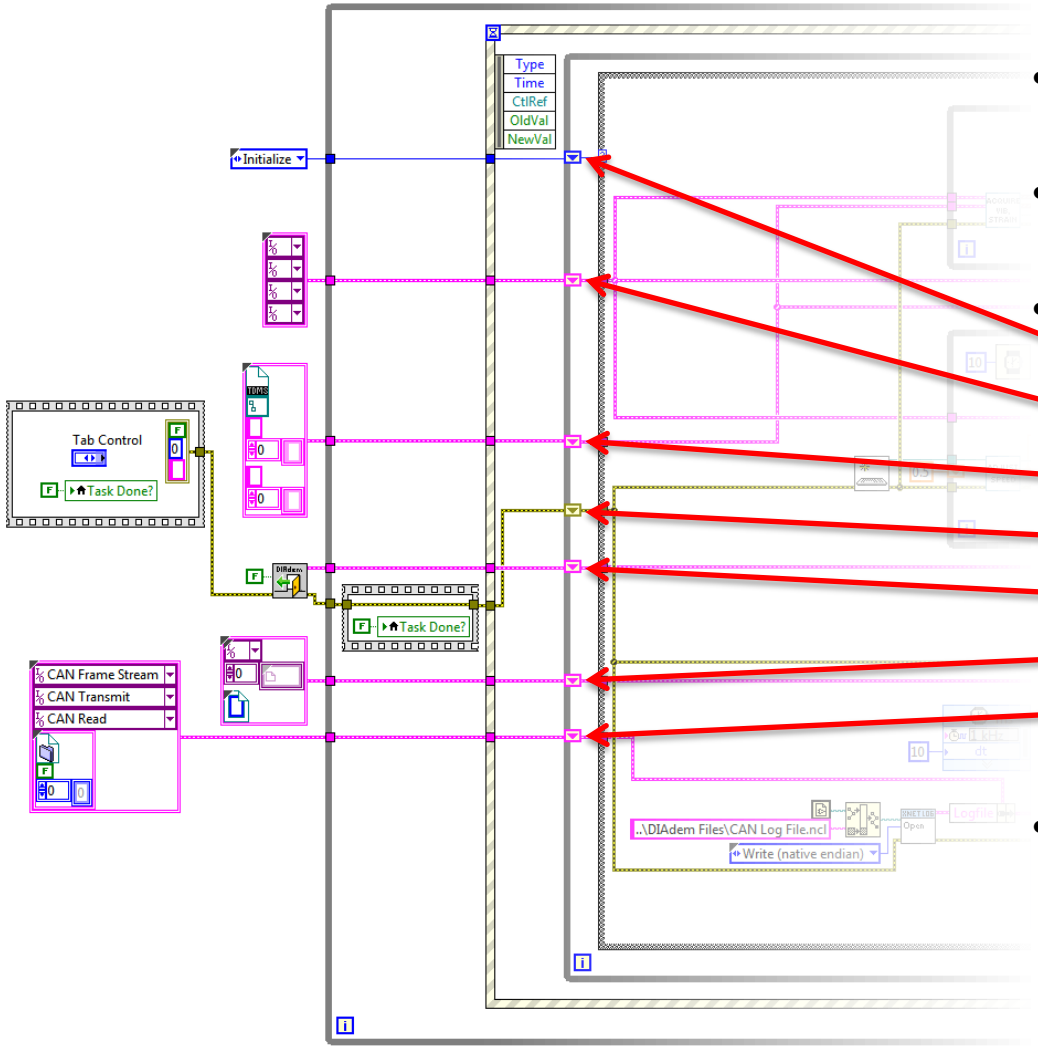
Initial Scope

- Demonstrate:
  - Data Acquisition
  - CAN Synchronization
- Selected Architecture:
  - State Machine



Coupling is Often Accidental: Case Study

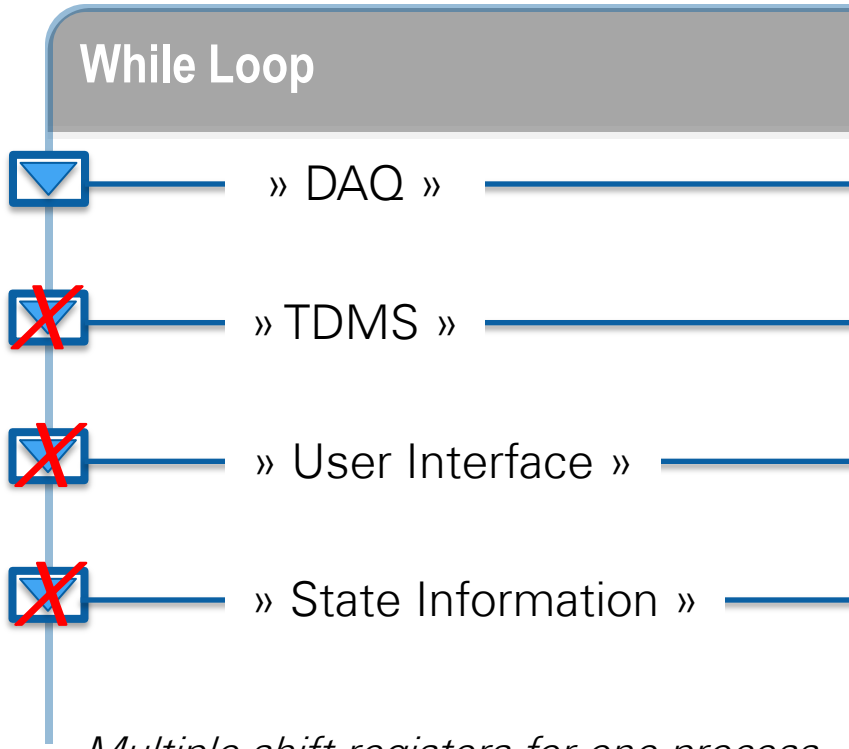
Final Demo Code



- Once you start…you can't stop
- You can't test independent actions
- There are 7 shift registers:
  - State
  - DAQ Settings
  - TDMS Settings
  - Errors
  - DIAdem References
  - IMAQ Settings
  - CAN Settings
- Whether used or not, every shift register goes across every case

Cohesion: Limiting Process Scope

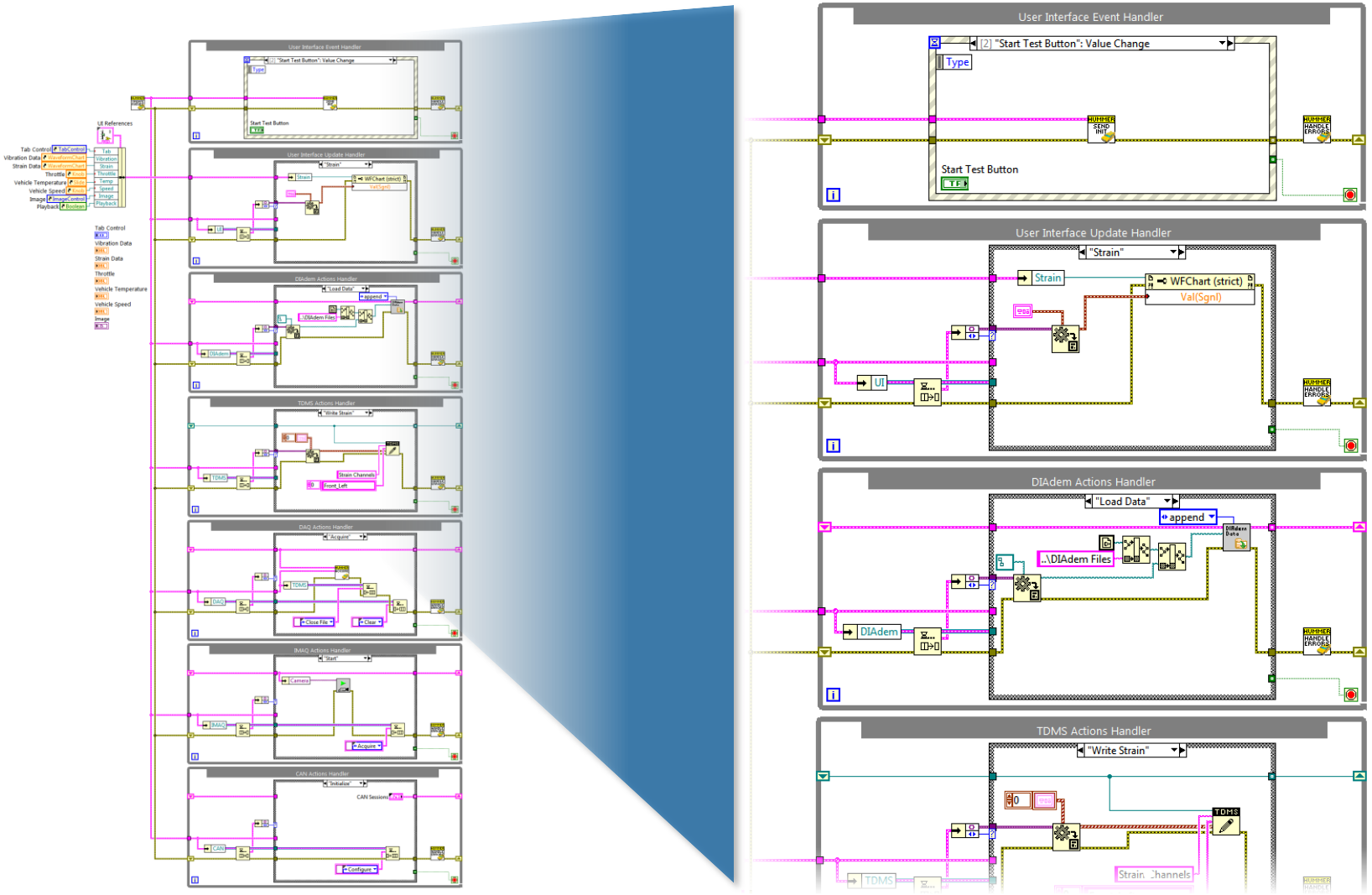
- Often, shift register defines scope of process
- Processes should be very cohesive
- Independent processes should be separate loops
- Good example: A drop-in Functional Global Variable (FGV)



Multiple shift registers for one process indicate over-coupling

The Rewritten Hummer Demo

Using a Queued Message Handler Architecture



How to De-Couple Independent Processes

Best Practices

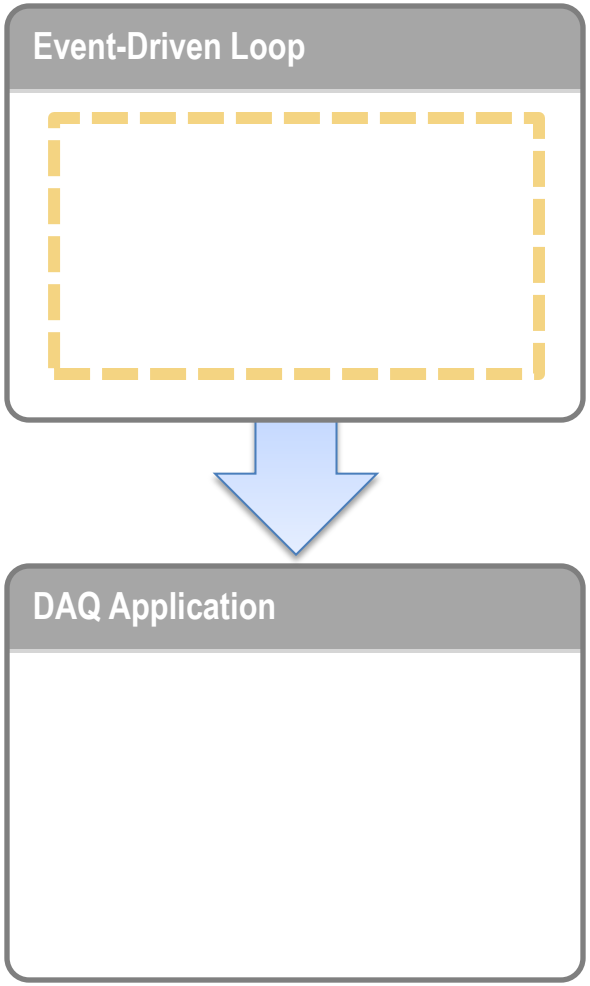
1. Identify data scope
2. Delegate actions to appropriate process
3. Avoid polling or using timeouts*

**except for code that communicates with hardware*

Considerations

1. How do you send commands?
2. How do you send data?
3. Which processes can communicate with each other?
4. How do you update the UI?

?

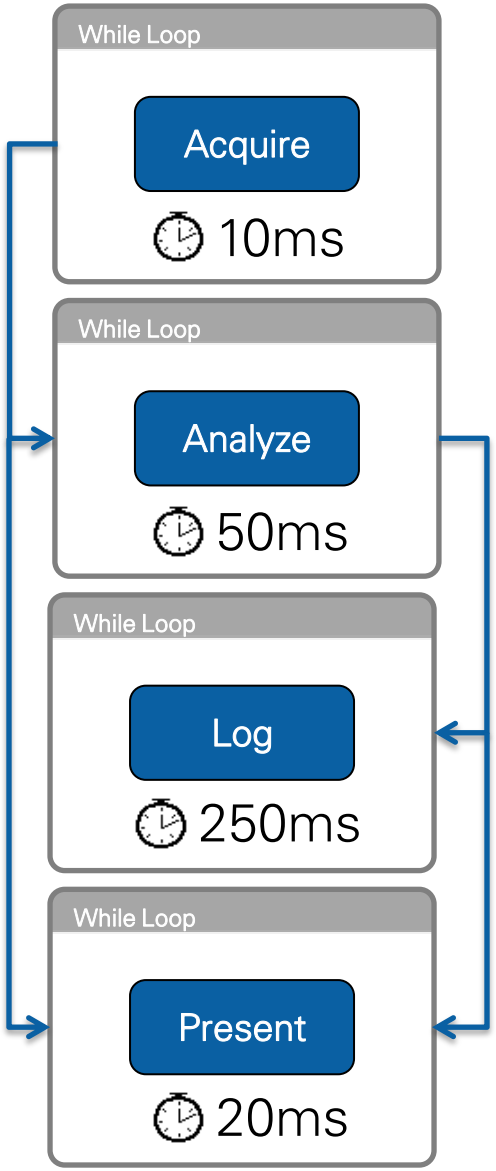


Inter-Process Communication

ensures tasks run asynchronously and efficiently

Goals …How?

- Loops are running independently
- User interface can be updated every 20 ms
- Acquisition runs every 10ms, helping to not overflow the buffer
- All while loops run entirely parallel with each other



Many Data Communication Options Exist in LabVIEW

In no particular order…

1. TCP and UDP
2. Network Streams
3. Shared Variables
4. DMAs
5. Web Services
6. Peer-to-Peer Streaming
7. Queues
8. Dynamic Events
9. Functional Global Variables
10. RT FIFOs
11. Datasocket
12. Local Variables
13. Programmatic Front Panel Interface
14. Target-scoped FIFOs
15. Notifiers
16. Simple TCP/IP Messaging
17. AMC
18. HTTP
19. FTP
20. Global variables

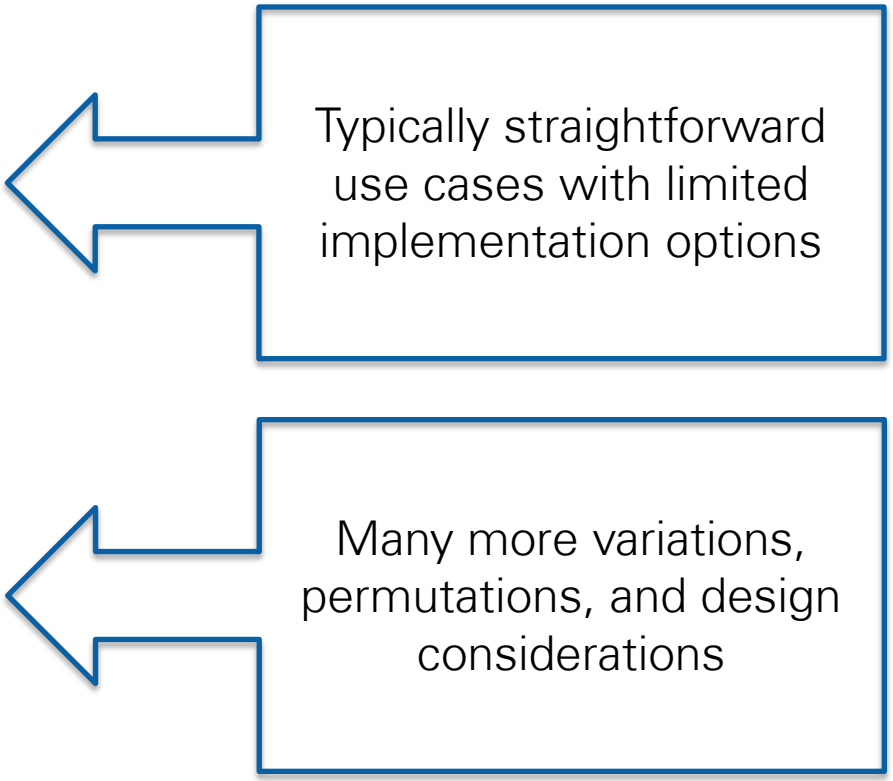
… just to name a few!

Tactic 1: Functional Global Variables

- What is a functional global variable (FGV)?
  - Does the FGV prevent race conditions?
- Is the FGV better than the global variable?

Inter-Process Communication

- Store Data
- Stream Data
- Send Message



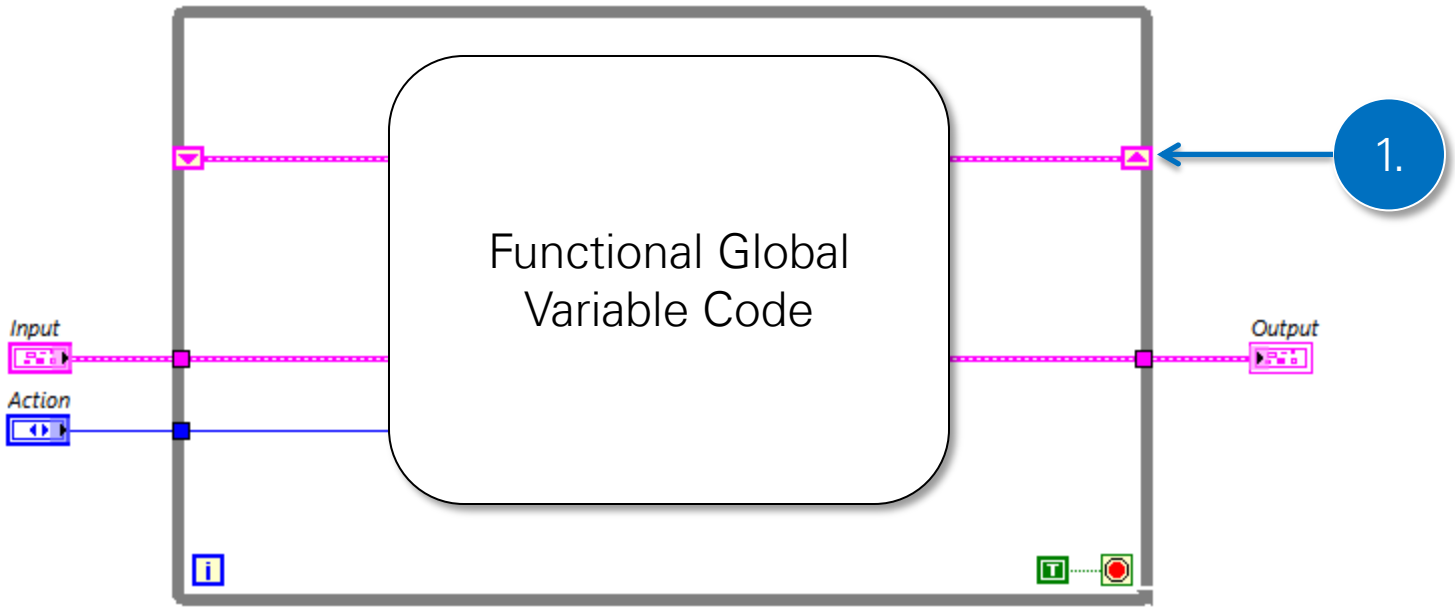
De-Facto Communication Choice: Locals / Globals

> *"The traditional, text-based programmer's first instinct is always to use local variables."*
> *– Brian Powell*

- Local and Global variables are simple and obvious
- Problem: Locals / Globals are **pure data storage**
- Text-based mindset isn't used to parallel programs, multithreading

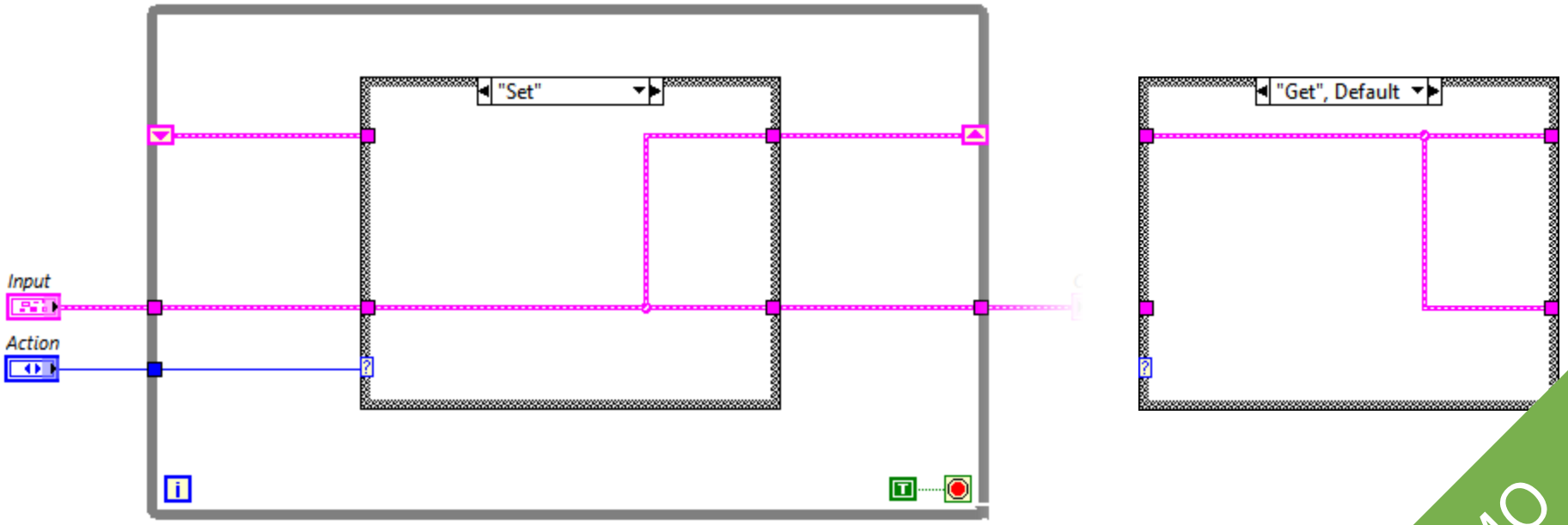
What is a Functional Global Variable?

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



What is a Functional Global Variable?

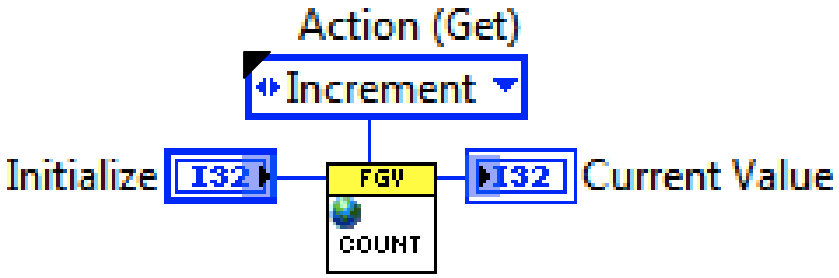
- A functional global variable usually has an action input parameter that specifies which task the VI performs
- The uninitialized shift register in a loop holds the result of the operation



DEMO

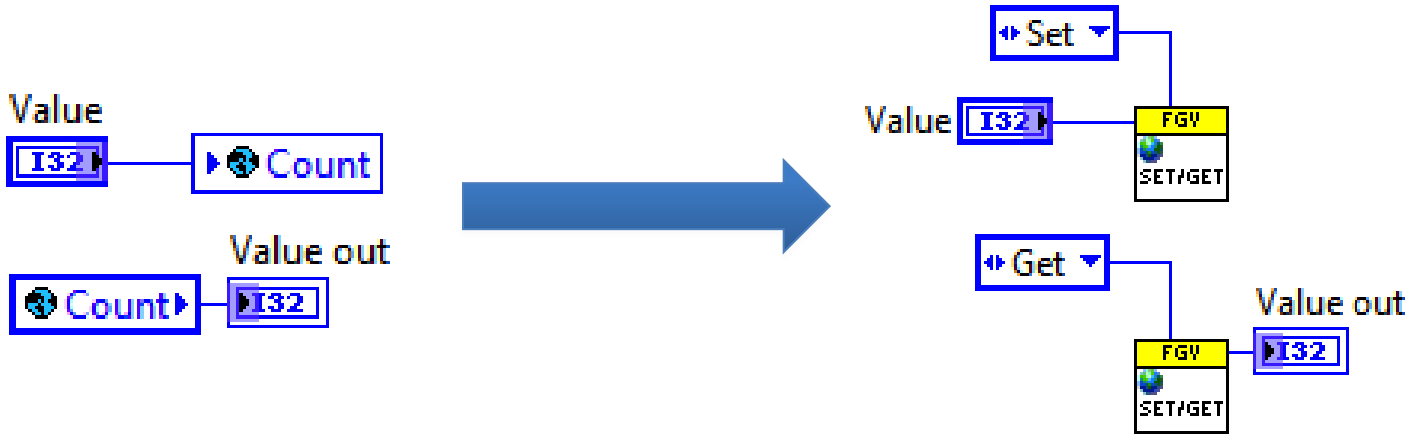
Best Practices for Documenting FGVs

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



Replacing Global Variables with FGVs

- This is a common initial use case
- Not necessarily a best practice for every application



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*
- *Adorn* data storage

Comparison of Options

Functional Global Variables

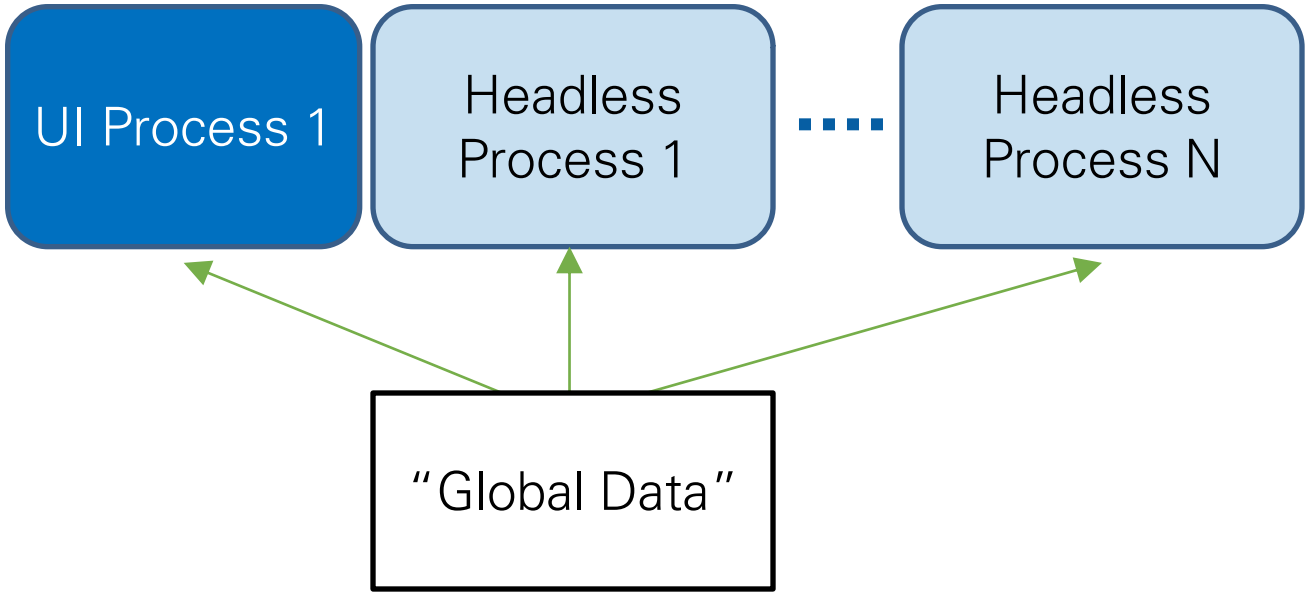
- Can behave like action engines
- Can prevent race conditions
- No copies of data created in memory
- Can handle error wires
- Take time to make

Global and Local Variables

- Cannot perform actions on data
- Can cause race conditions
- Create copies of data in memory
- Cannot handle error wires
- Drag and drop

Considerations for Storing Data

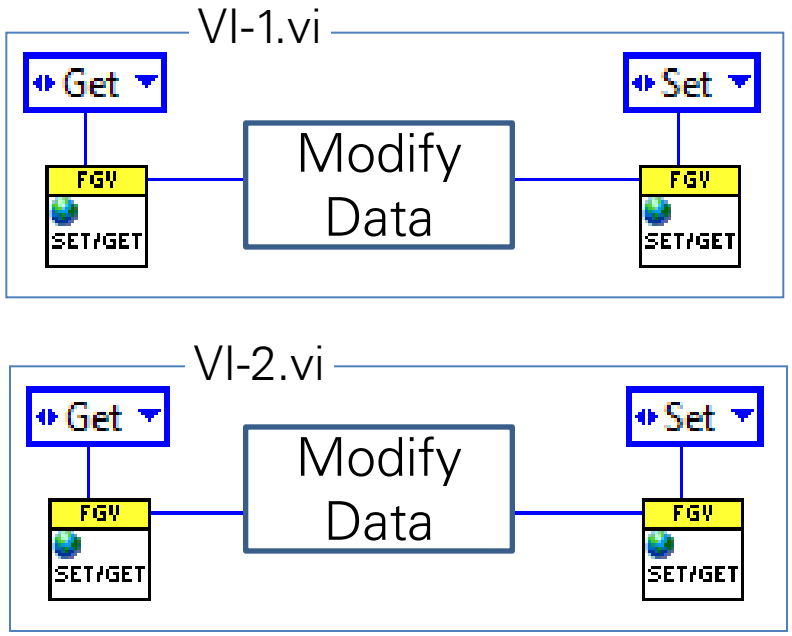
- Data is stored and made "globally" accessible
- Storage mechanism holds only the current value
- Other code modules can access the data as needed
- The potential for race conditions **must be considered**



Use Cases
Configuration data Slowly changing data Non-critical messages

Traditional FGVs Do Not 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?

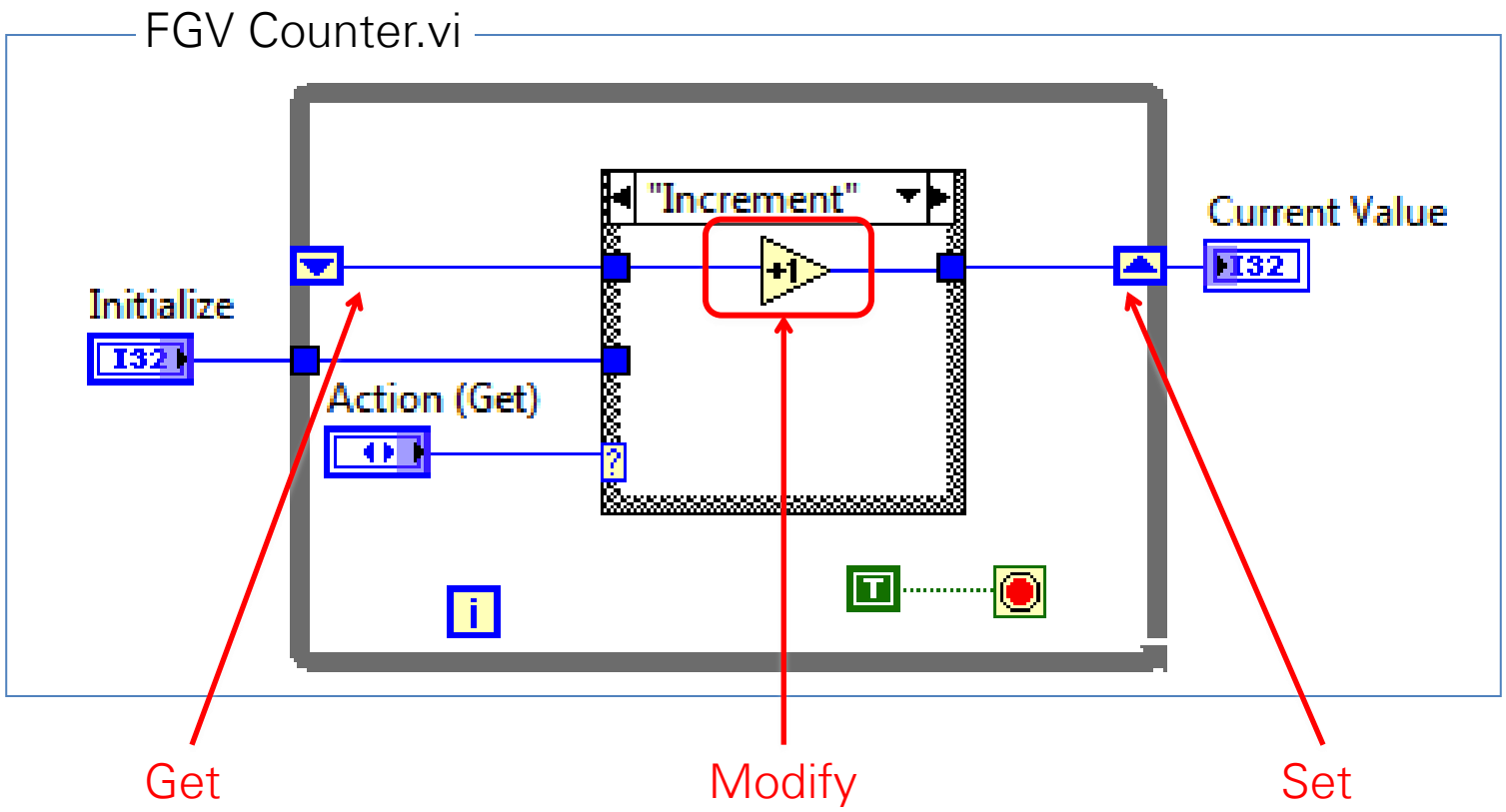
Use FGVs to Protect Critical Sections of Code

- 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

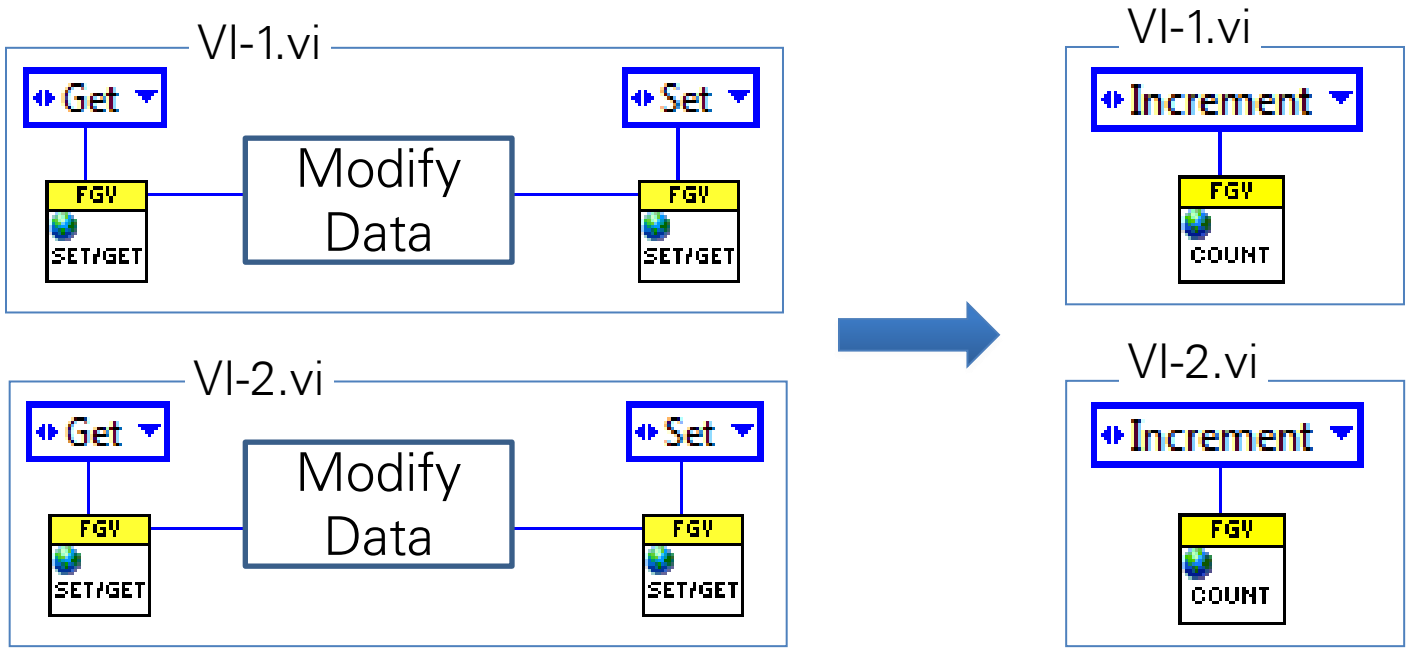
This is commonly called an Action Engine.
It is a special type of FGV.

FGV – Action Engine Protects Critical Sections of Code

- This action engine wraps the "get/modify/set" around the critical section of code



Action Engines Protect Critical Sections!



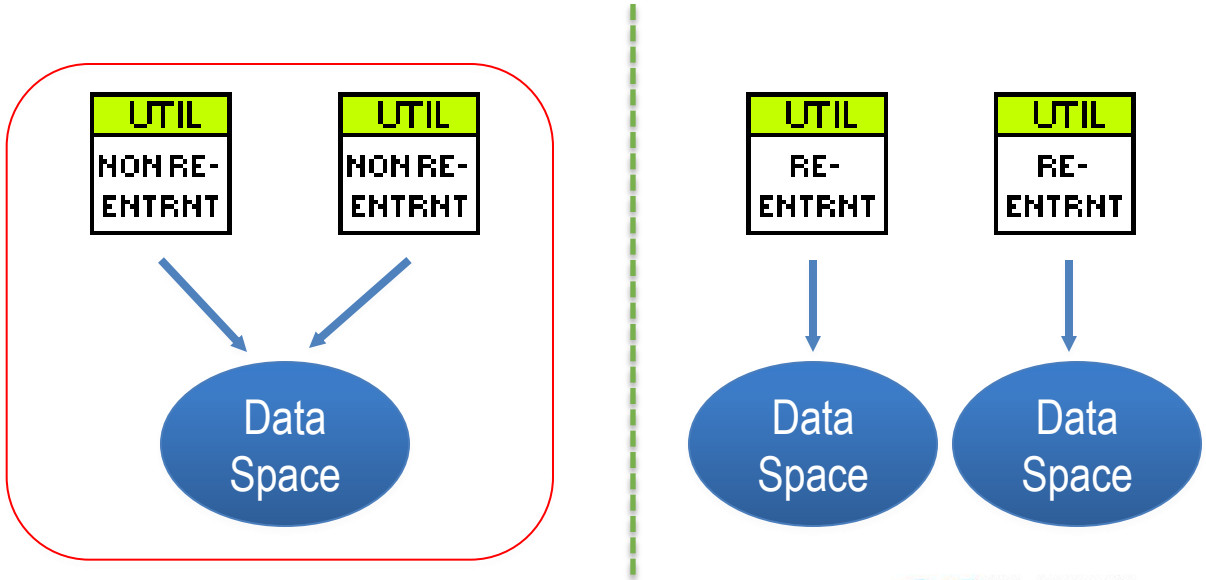
The FGV will block other instance from running until it has completed execution. Therefore, encapsulating the entire action does prevent the potential race condition.

DEMO

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



Various Inter-process Communication Methods Exist

- FGVs are one tool of many in the toolbox:

	Same target Same application instance	Same target, different application instances / 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

- When you need more than one (counter, timer, etc), investigate Data Value References (DVRs)

What Else Do I Need to Know?

- Store Data
- Stream Data

Typically straightforward use cases with limited implementation options

- Send Message

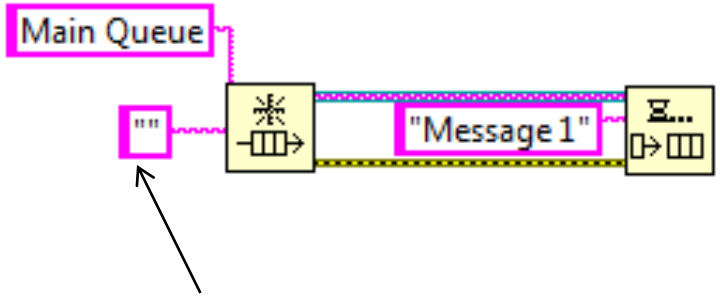
Many more variations, permutations, and design considerations

Tactic 2: Queued Message Handlers

- What are Queues?
- What is the Queued Message Handler (QMH)?
- Basic modifications to the QMH template

Queues 101: Inter-Process Communication

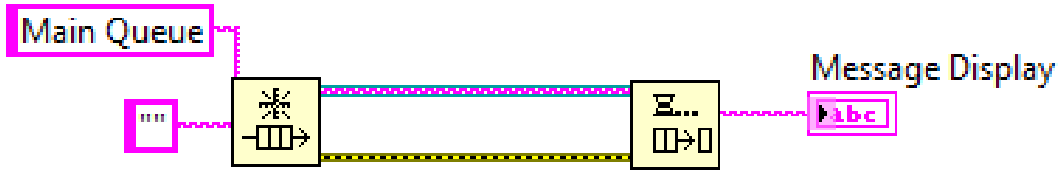
Adding Elements to the Queue



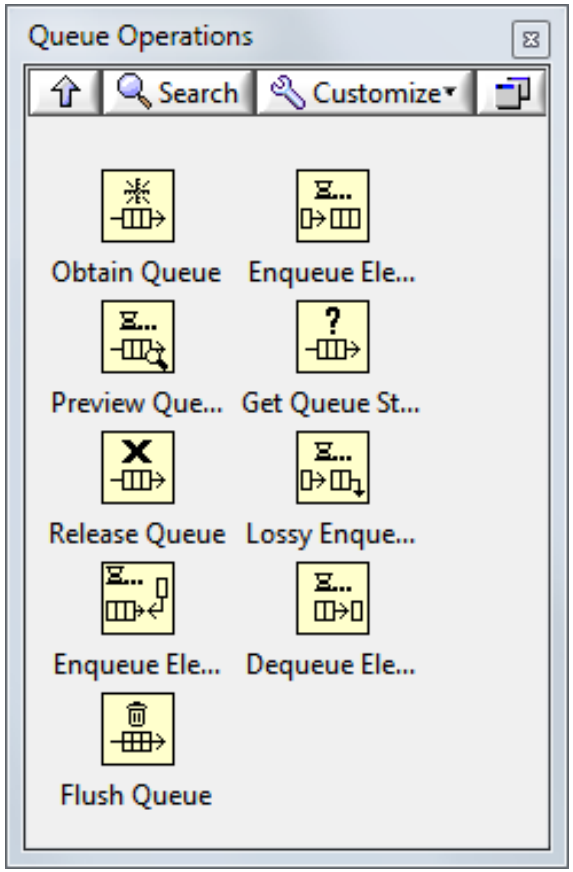
Select the data type the queue will hold

Reference to existing queue in memory

Dequeueing Elements



Dequeue will wait for data or time-out (defaults to -1)



NATIONAL INSTRUMENTS

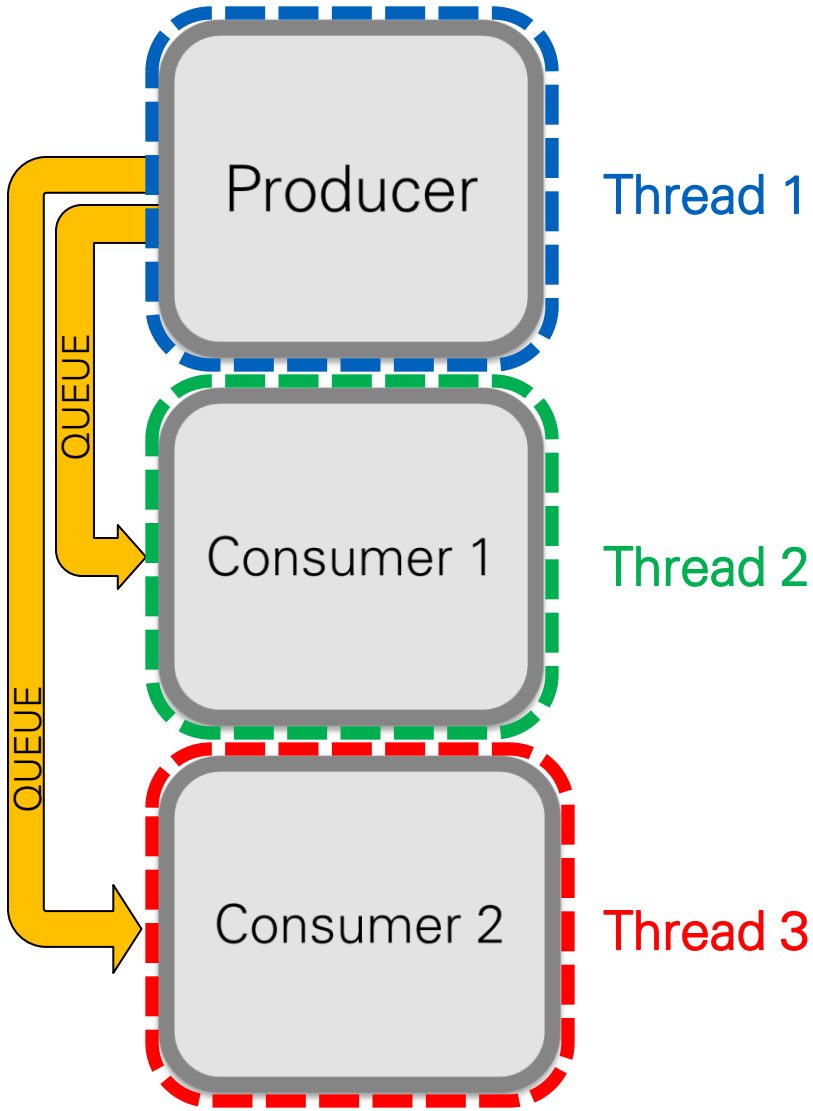
Producer Consumer Generalization

Best Practices

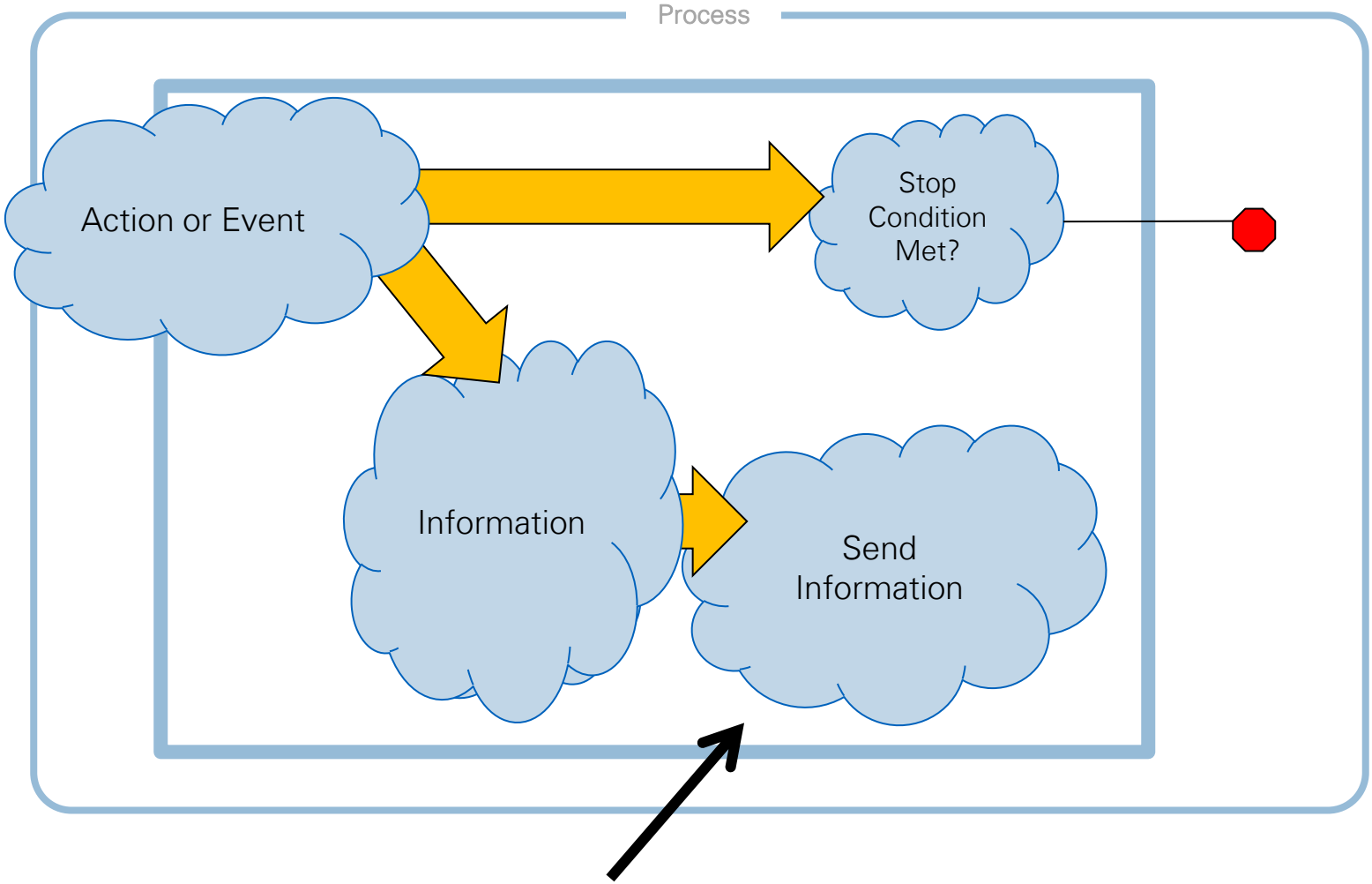
1. One consumer per queue
2. Keep at least one reference to a named queue available at any time
3. Consumers can be their own producers
4. Do not use variables

Considerations

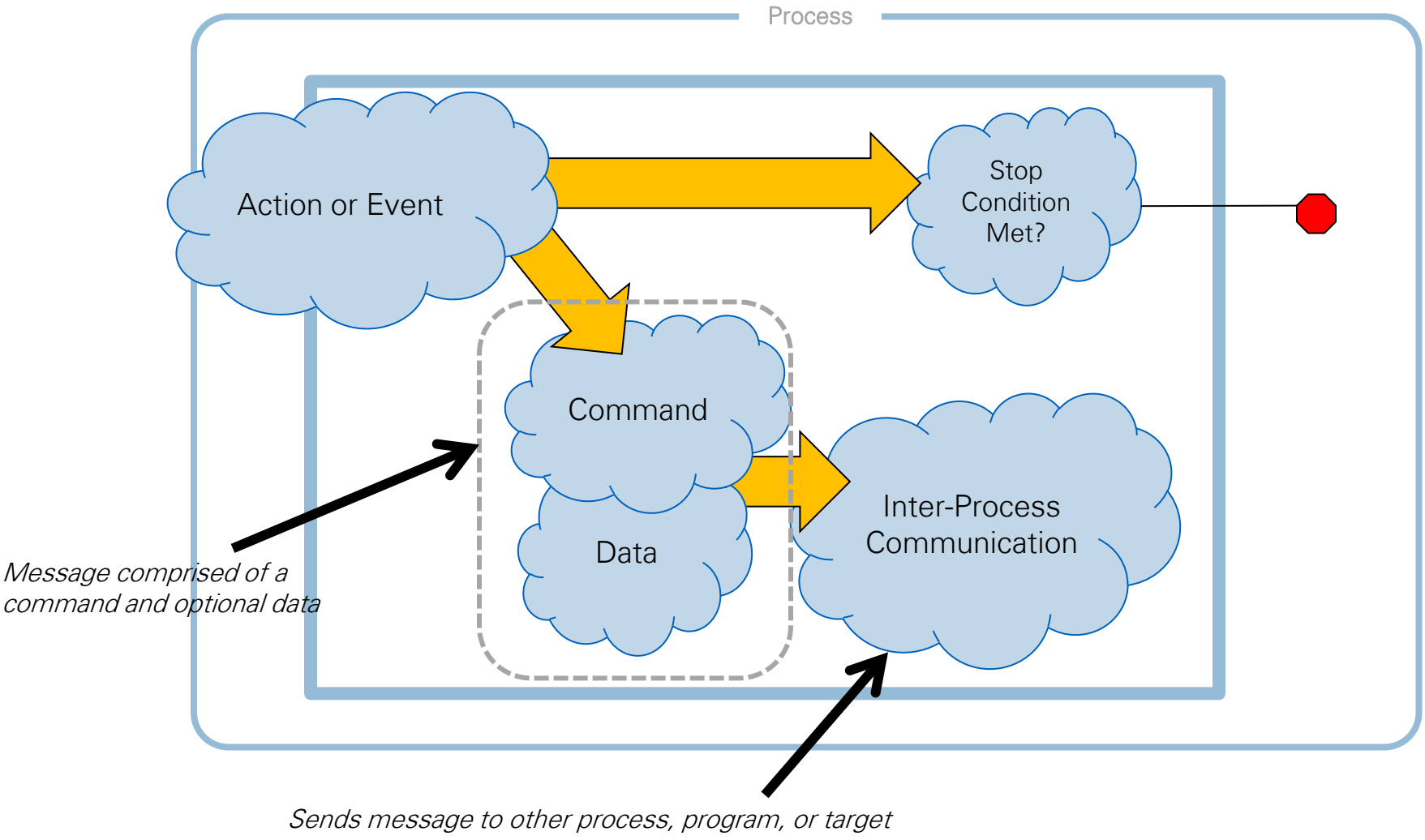
1. How do you stop all loops?
2. What data should the queue send?



Anatomy of a "Producer" Process



Anatomy of a Message Producer Process



Constructing a Message

Command

Enumerated constants (or strings) list all of the options.

Data

Variant allows data-type to vary. Different messages may require different data.

Examples:

Command	Data
Initialize UI	Cluster containing configuration data
Populate Menu	Array of strings to display in menu
Resize Display	Array of integers [Width, Height]
Load Subpanel	Reference of VI to Load
Insert Header	String
Stop	-

Sidebar: Enums versus Strings for Commands?

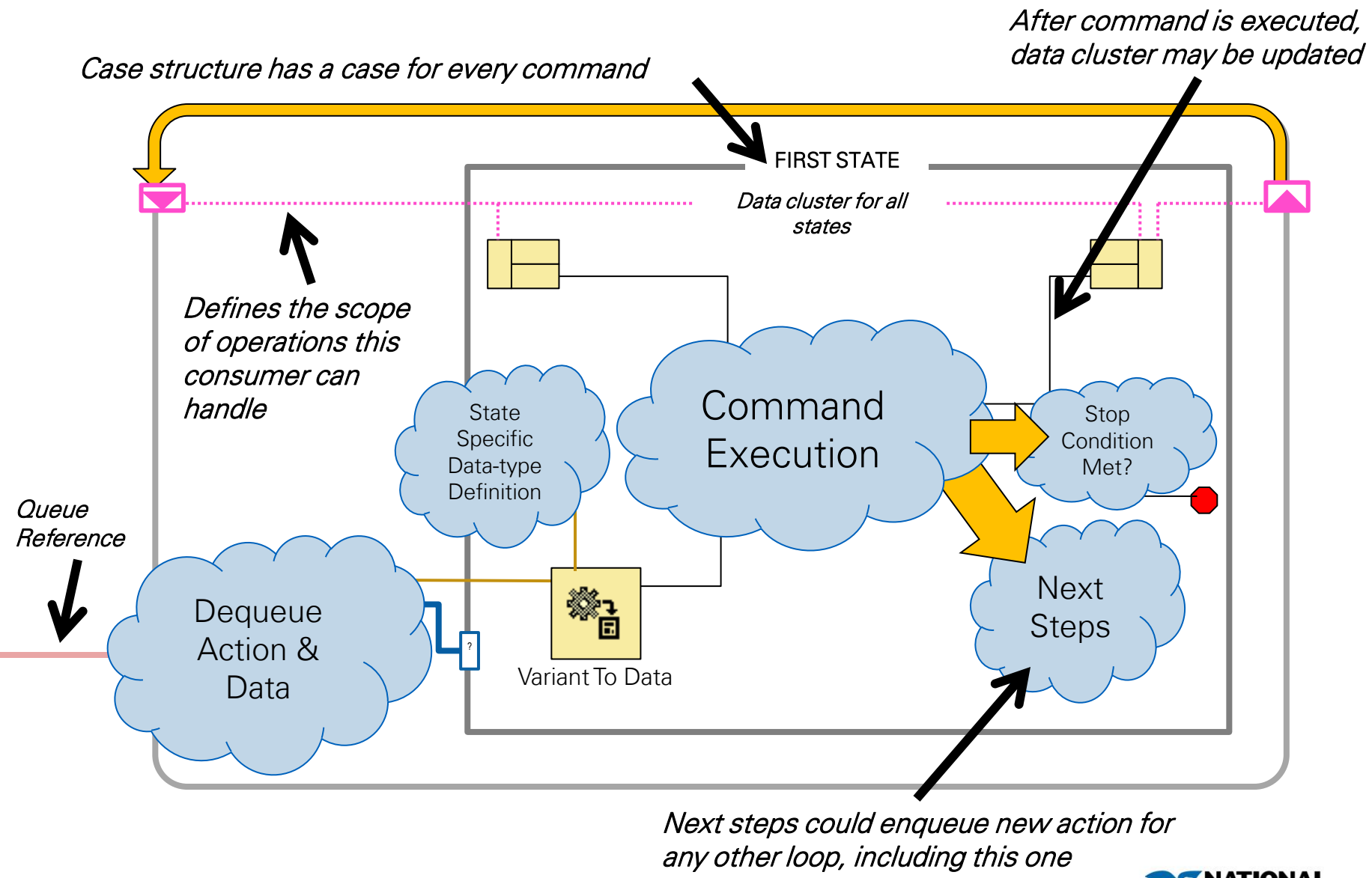
Enumerated Constant

- Safer to use – prevents misspellings
- Compiler checks for message validity
- Requires a copy for each API (e.g. "Initialize")
- "Add case for every value"

String

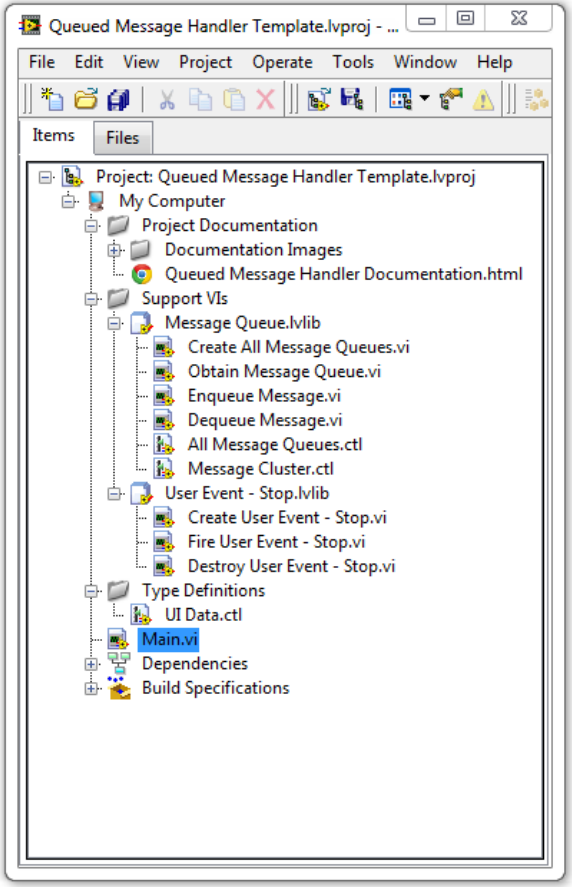
- Misspellings could lead to runtime errors
- Onus is on the developer to get messages correct
- Streamlines usage across multiple APIs
- Universal data type

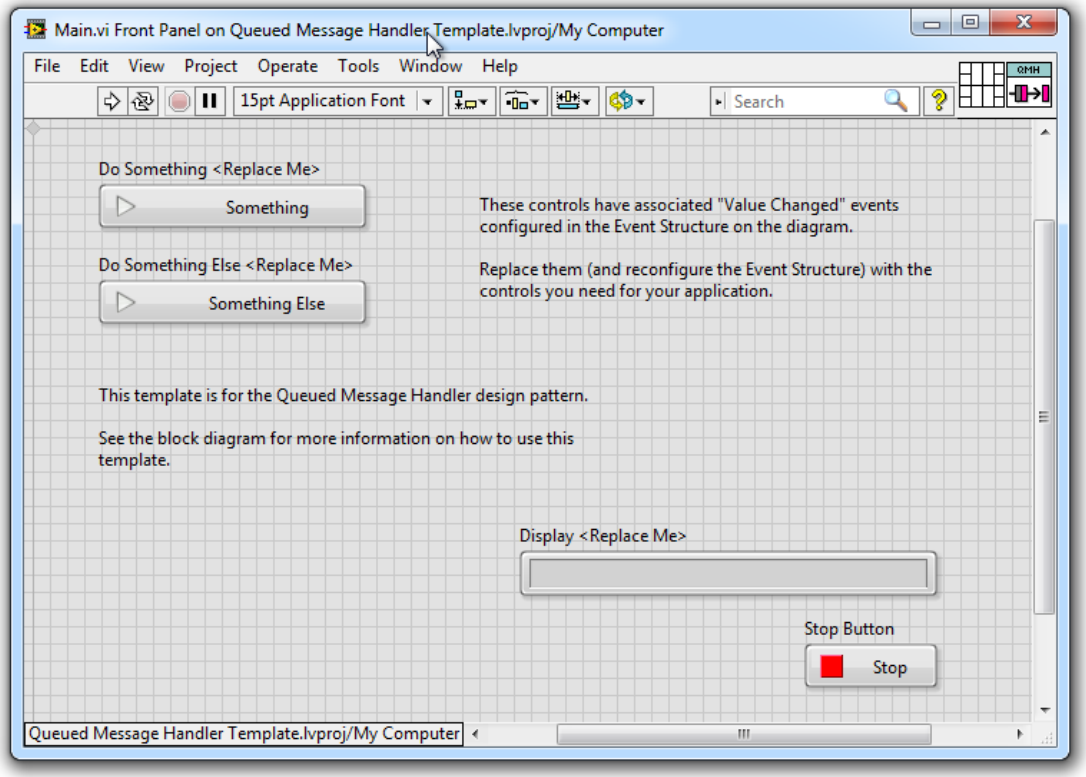
Queued Message Handler "Consumer" Process



Queued Message Handler

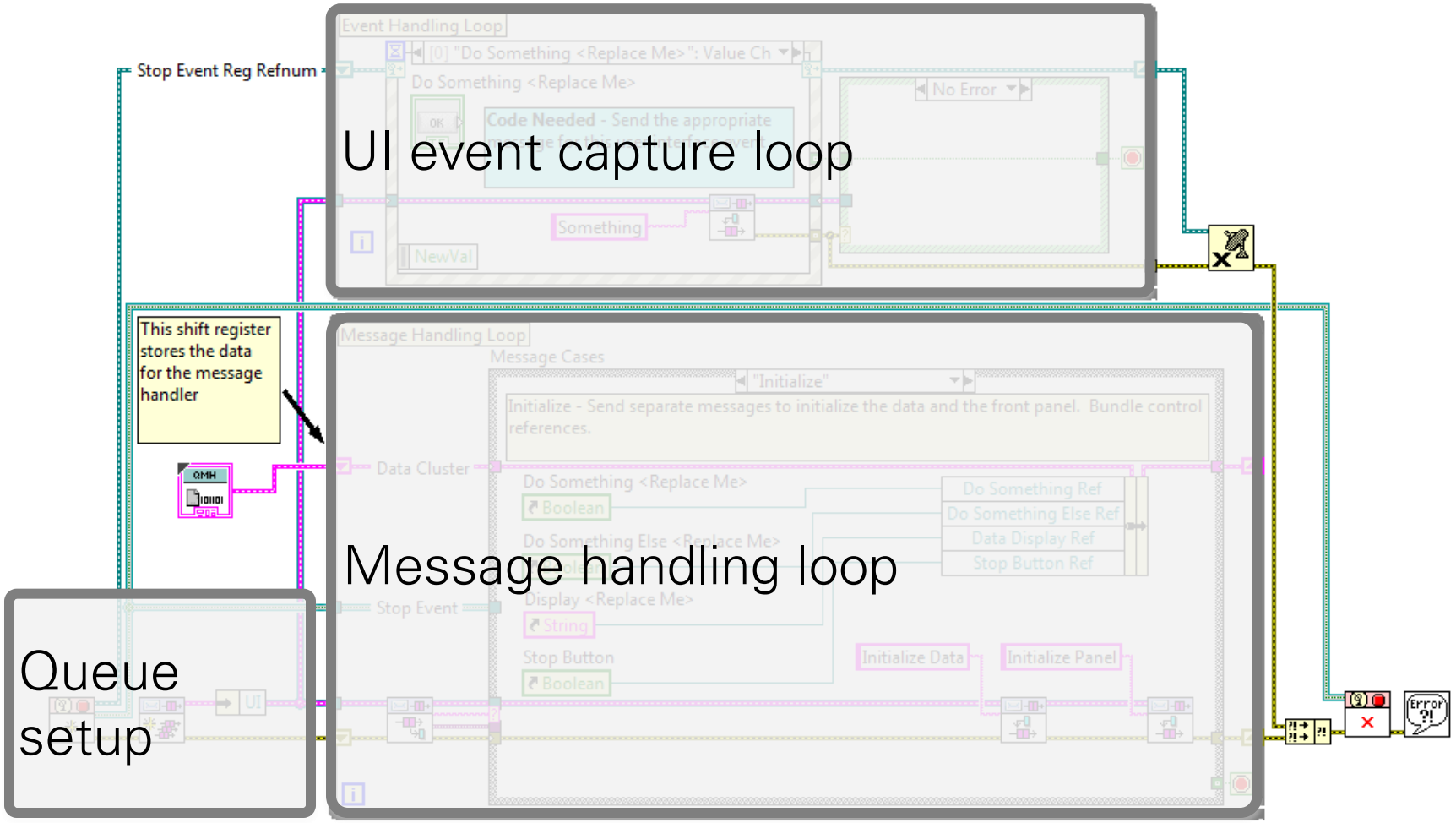
Template





Queued Message Handler

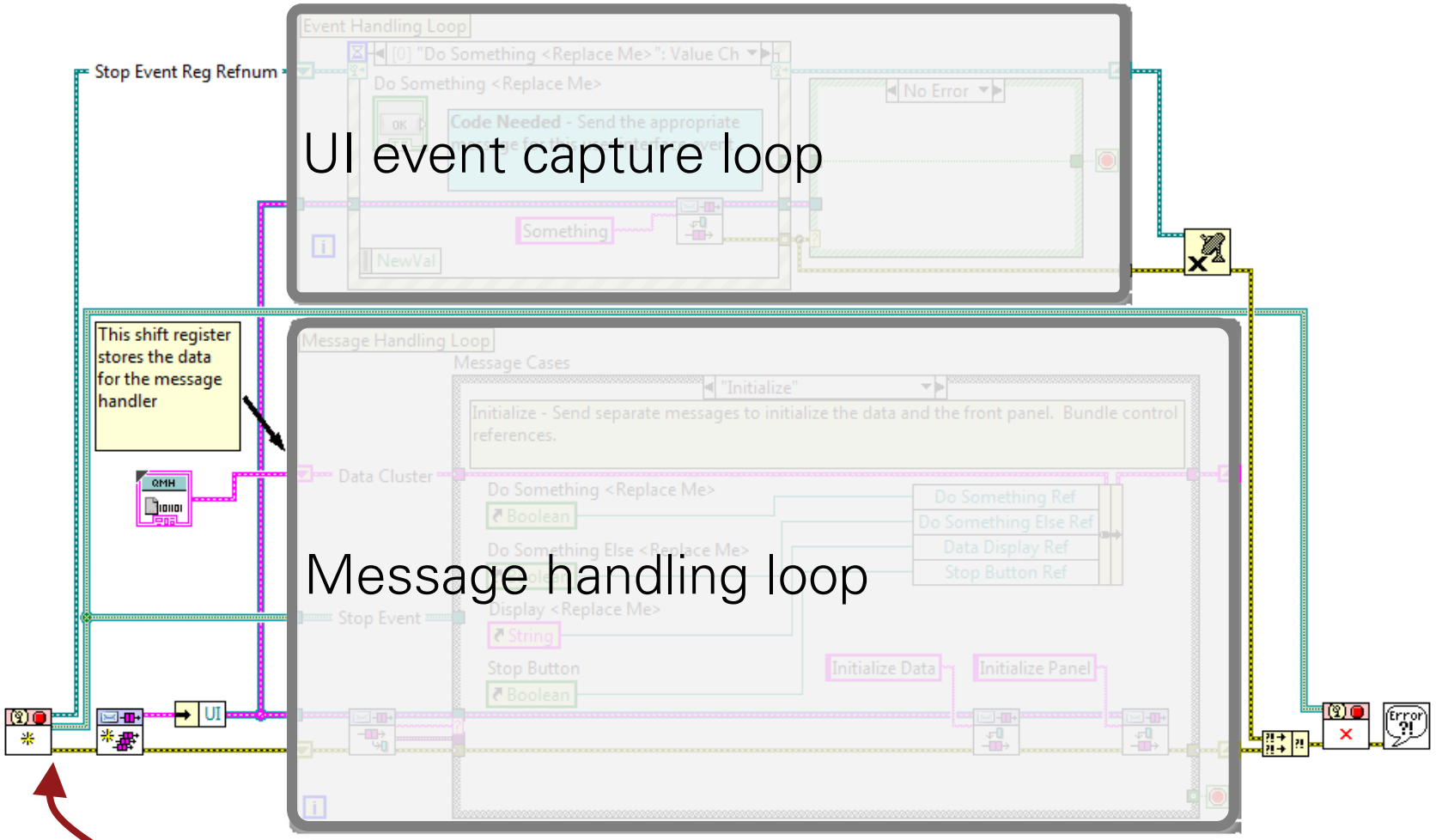
Template



NATIONAL INSTRUMENTS

Queued Message Handler

Template

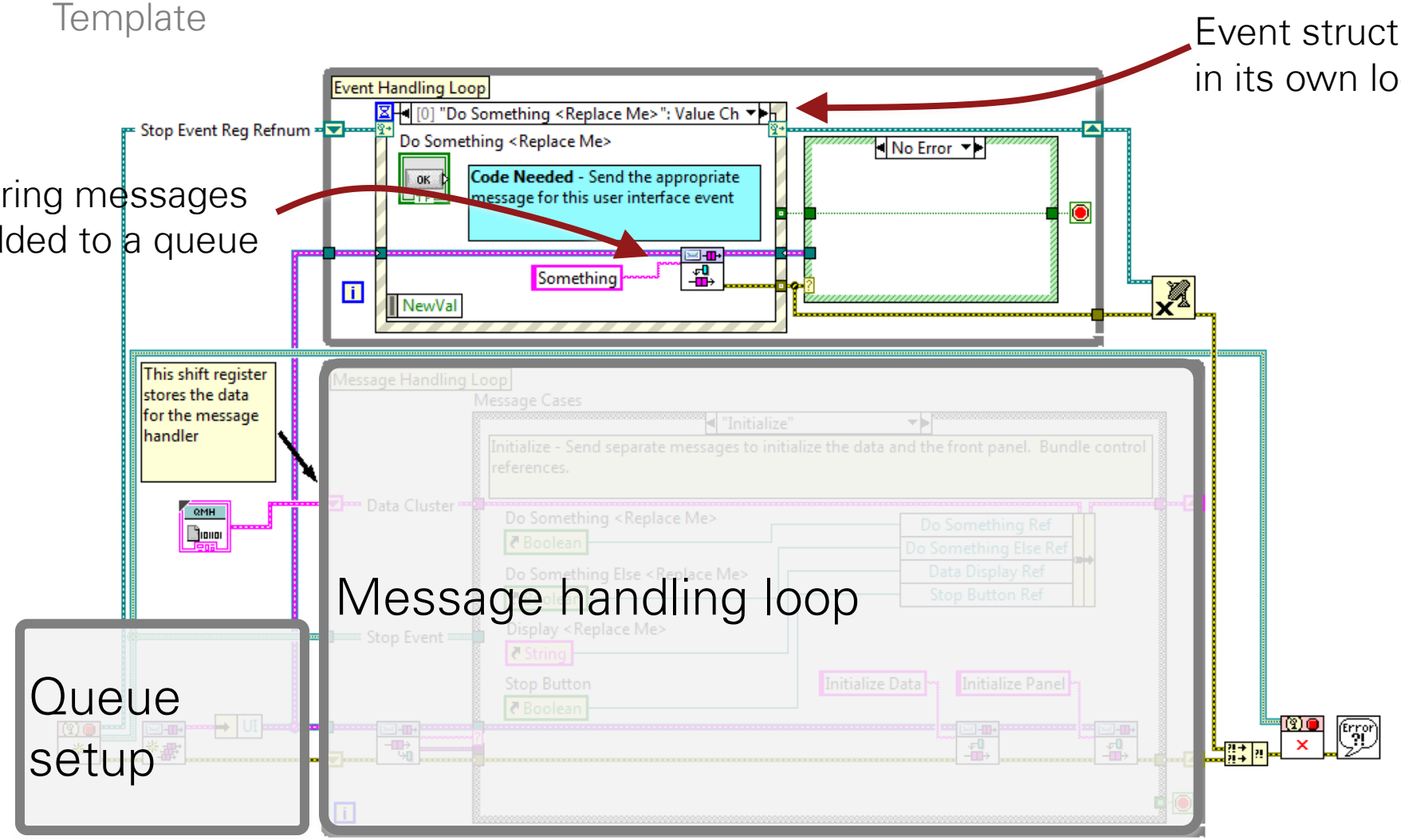


Create queue for message communication and user event to stop event loop

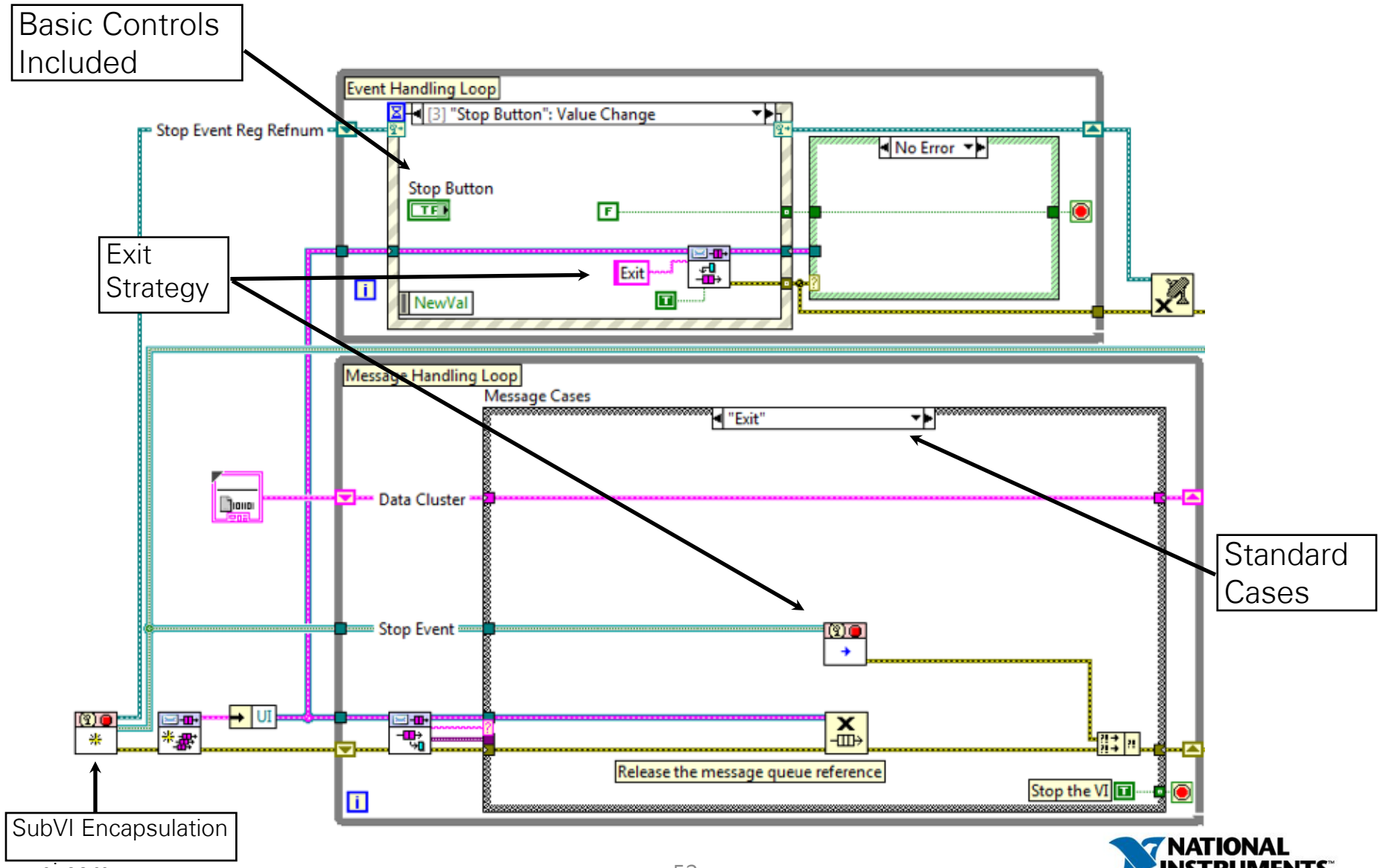
NATIONAL INSTRUMENTS

Queued Message Handler

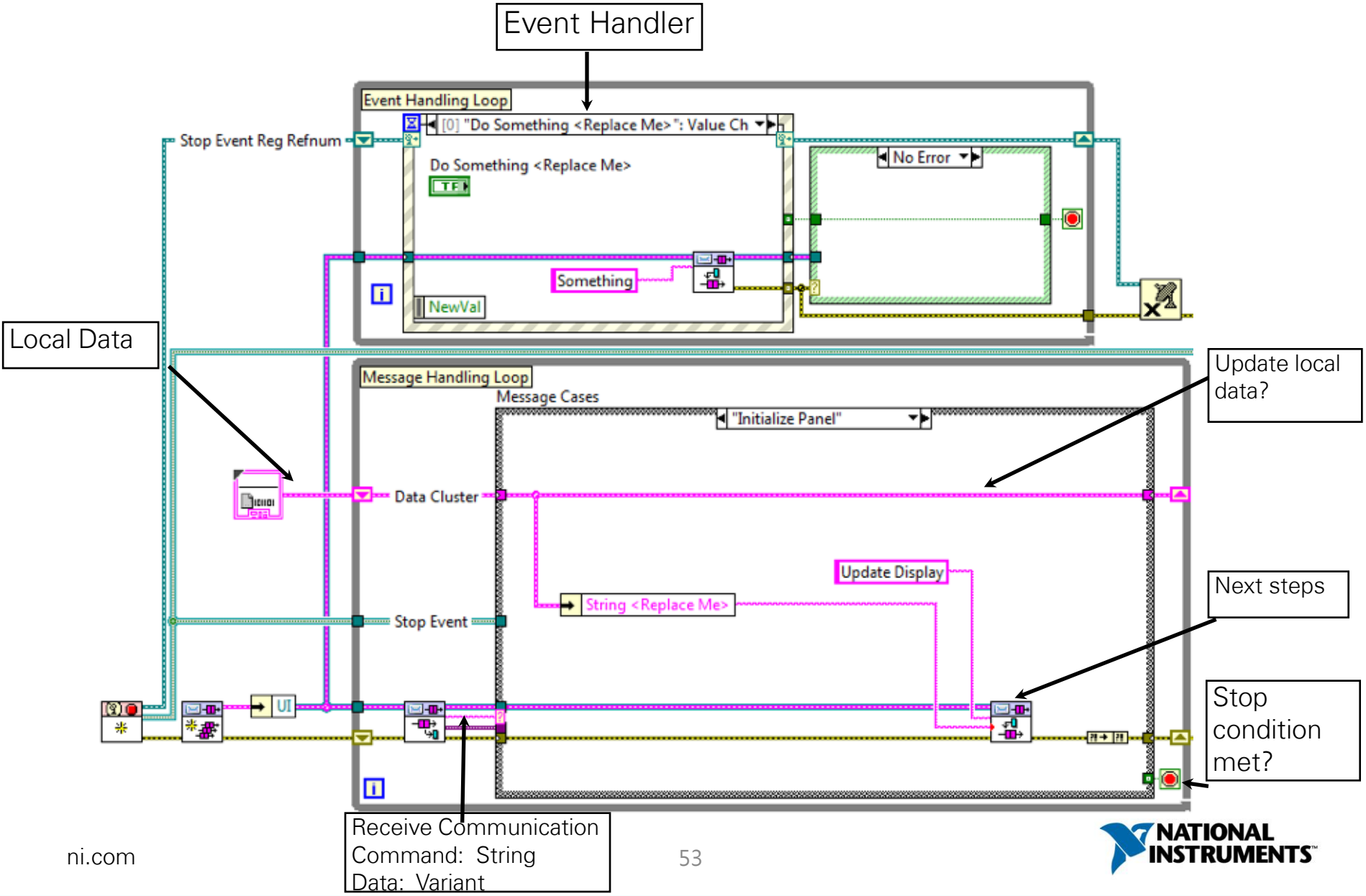
Template



QMH Extends Producer/Consumer



QMH Features



NATIONAL INSTRUMENTS

General Modifications to the QMH Template

- Add Additional Message Handlers
- Add Exit LabVIEW on Run Time to Destroy User Event
  - Exiting when on run-time should exit LabVIEW
- Add Timeout to Dequeue Messages
  - May want to timeout for regular Update Display execution
- Move Register User Event out of Create User Event
  - Forking this wire causes unpredictable results
- Remove Error Case Structure from Producer
  - Adds more real estate and declutters
  - Determine whether a use case exists for keeping this structure
- Determine an Interrupt Strategy
  - What if a process needs to disrupt the flow of tasks?
- …etc.

DEMO

Application-Specific Modifications to the QMH

- Official Login Process
  - Operator login and documentation
- Customized or Interactive Displays
  - Different ways of displaying different types of data
- Standardized Logging
  - Company or application specific file formatting and structure
- Cloud or mobile device publishing
  - Send acquired or analyzed results to a cloud server
- …etc.

Summary and Next Steps

- FGVs are an effective data storage mechanism
- Action engines prevent race conditions

- The QMH is really a starting point
  - It will get you really far, but…
  - It is frequently customized
  - It is frequently one component in a larger framework

- Next Steps:
  - Advanced Architectures in LabVIEW CustEd Course
  - History Probe for keeping track of actions:

    https://decibel.ni.com/content/docs/DOC-1106