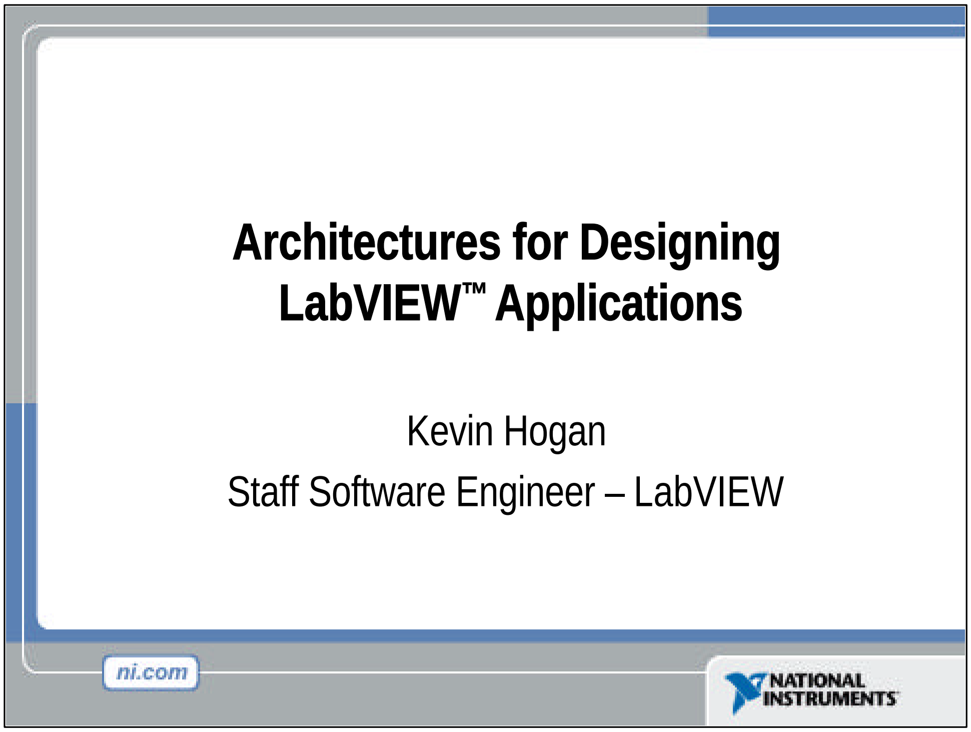
NATIONAL INSTRUMENTS™

ni.com

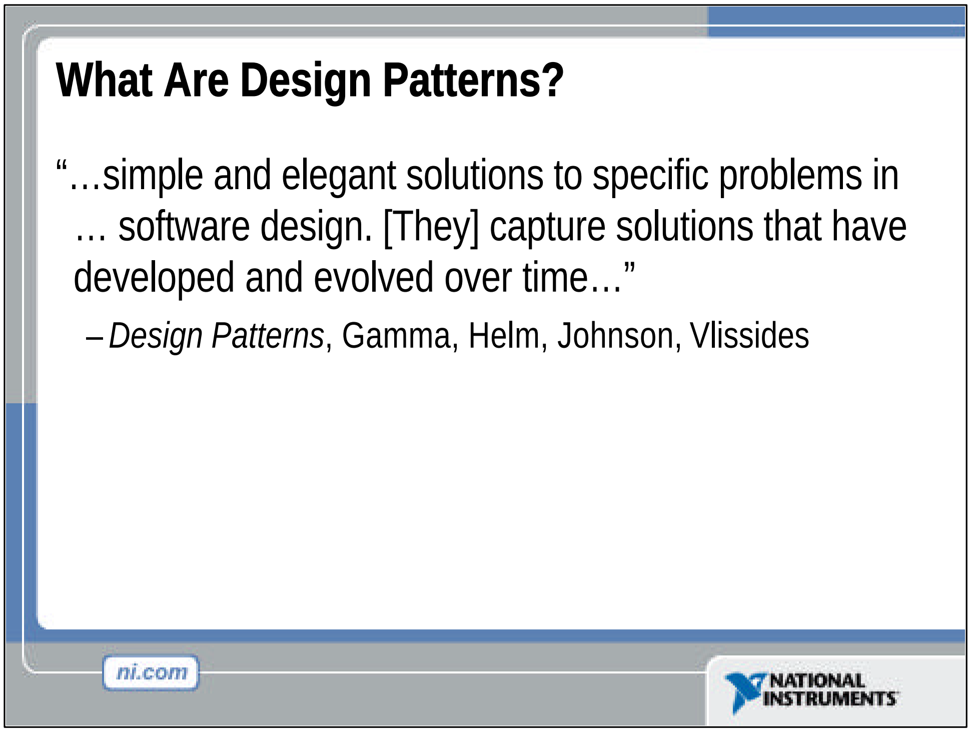


Architectures for Designing LabVIEW™ Applications

Kevin Hogan

Staff Software Engineer – LabVIEW

ni.com

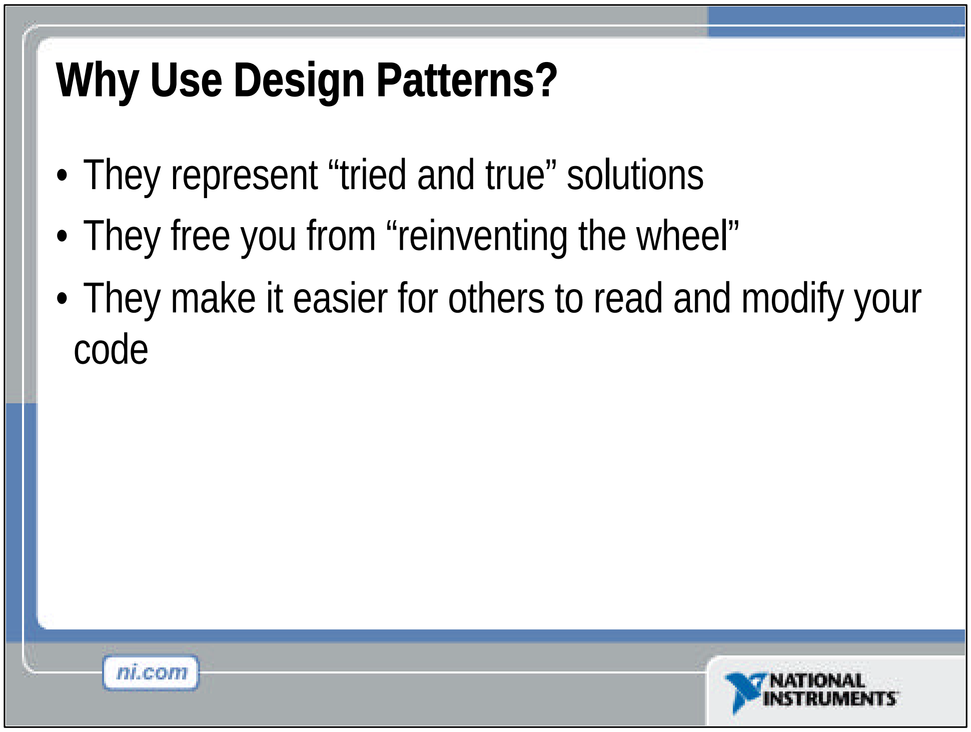


What Are Design Patterns?

"…simple and elegant solutions to specific problems in … software design. [They] capture solutions that have developed and evolved over time…"

– *Design Patterns*, Gamma, Helm, Johnson, Vlissides

ni.com

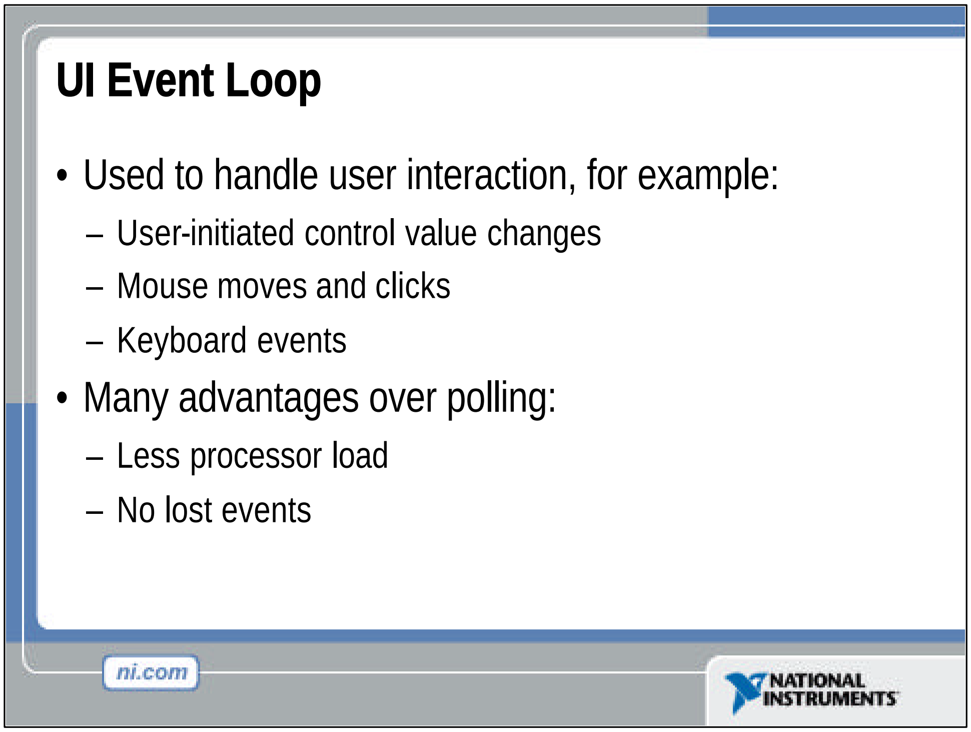


Why Use Design Patterns?

- They represent “tried and true” solutions
- They free you from “reinventing the wheel”
- They make it easier for others to read and modify your code

ni.com
NATIONAL INSTRUMENTS

Design patterns represent techniques that have proved themselves useful time and time again. They typically have evolved through the efforts of many developers and have been fine-tuned for simplicity, maintainability and readability. Furthermore, as a pattern gains acceptance, it becomes easier to recognize. This recognition alone helps other developers (and you!) to read and make changes to your code.

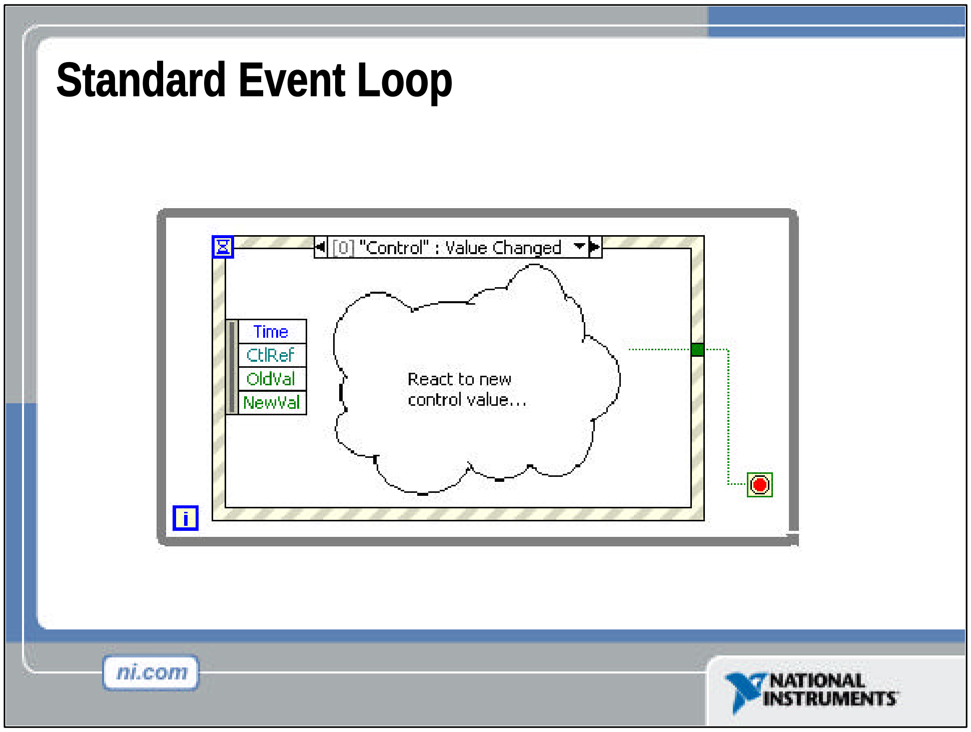


UI Event Loop

- Used to handle user interaction, for example:
  - User-initiated control value changes
  - Mouse moves and clicks
  - Keyboard events
- Many advantages over polling:
  - Less processor load
  - No lost events

ni.com
NATIONAL INSTRUMENTS

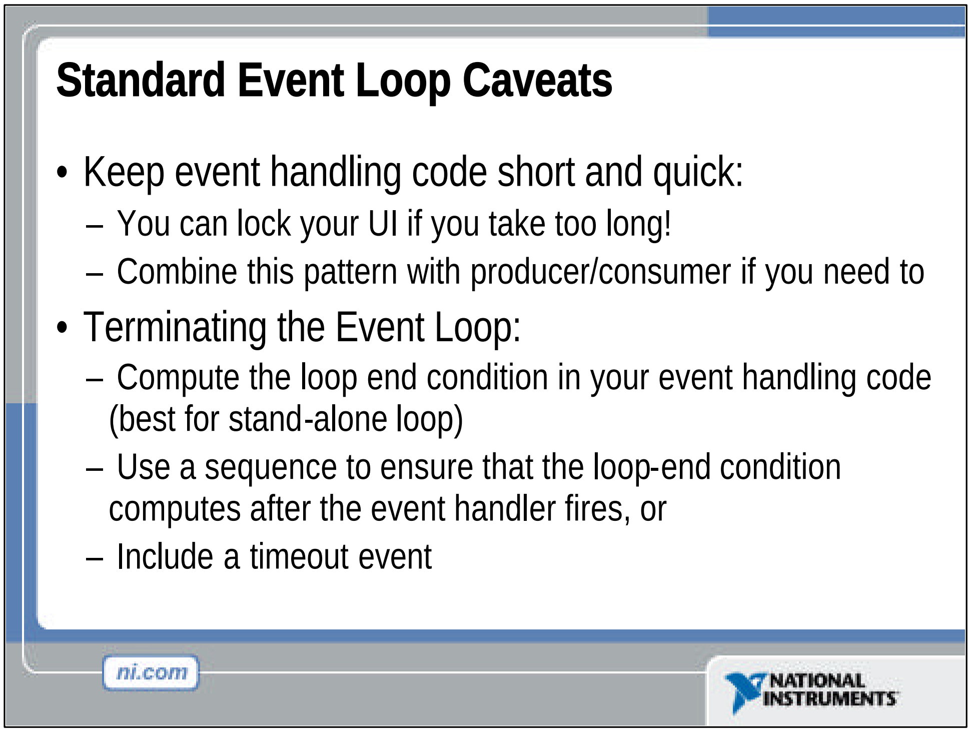
The UI event loop is a powerful and efficient method for handling user interaction with a LabVIEW program. It’s useful for detecting when a user changes a control’s value, moves or clicks the mouse, or presses a key. Because the event loop wakes up precisely when an event happens and sleeps in between, you don’t have to read control values over and over again (i.e. “polling”) in order to detect when a user presses a button. This will allow you to use the processor much less without risking loss of interactivity.



Standard Event Loop

The standard event loop consists of an event structure contained in a while loop. The event structure is configured to have one frame for each category of event you'd like to detect. Each frame contains "handling" code that executes immediately after an event occurs.

Note that it's important to make sure that the loop termination condition is computed *after* the handling code completes – one effective method is to compute the condition within the event structure. Alternatively, you could use a sequence structure to ensure that the event structure fires first, or simply add a timeout event to the structure.

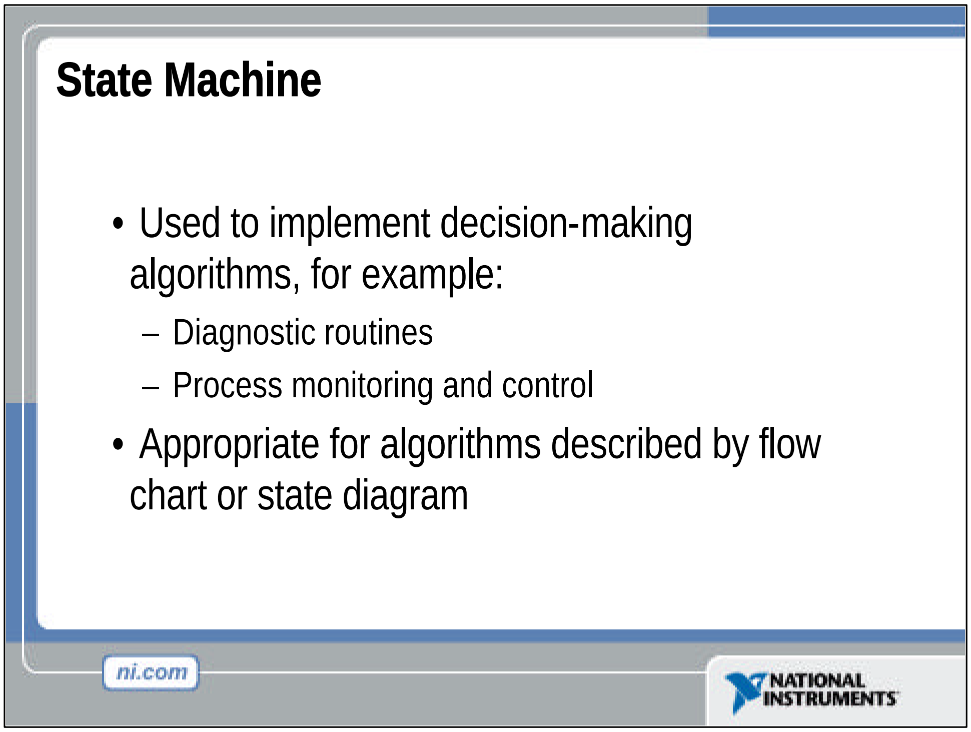


Standard Event Loop Caveats

- Keep event handling code short and quick:
  - You can lock your UI if you take too long!
  - Combine this pattern with producer/consumer if you need to
- Terminating the Event Loop:
  - Compute the loop end condition in your event handling code (best for stand-alone loop)
  - Use a sequence to ensure that the loop-end condition computes after the event handler fires, or
  - Include a timeout event

It’s easy to shoot yourself in the foot with the event loop, though. For instance, if an event structure is unable to handle an event that it is registered for (either because it’s not running, or because it’s busy handling another event), it will lock any UI elements that could produce that event. Effectively, this means that if you don’t handle events quickly, you could lock up your entire UI. This is easy to get around though – just do the bare minimum of work necessary to handle an event as it comes in (“synchronously”), such as graying out other controls that will be affected while this event is being handled. Any further processing that could delay the event loop should be handled elsewhere. One good method is to send the message via a queue to a separate parallel loop – we’ll see more of that with the “Producer/Consumer” pattern.

Another easy way to make trouble for yourself is to allow the loop termination condition to be computed *before* the event structure fires. This can cause the event loop to iterate one more time than you expected, and unless you have included a fairly short timeout event (200 ms or less), this can hang your VI on exit. To get around this, compute the loop end condition within all of your event handling code, use a sequence structure to ensure that the event code fires first, or include a timeout event.

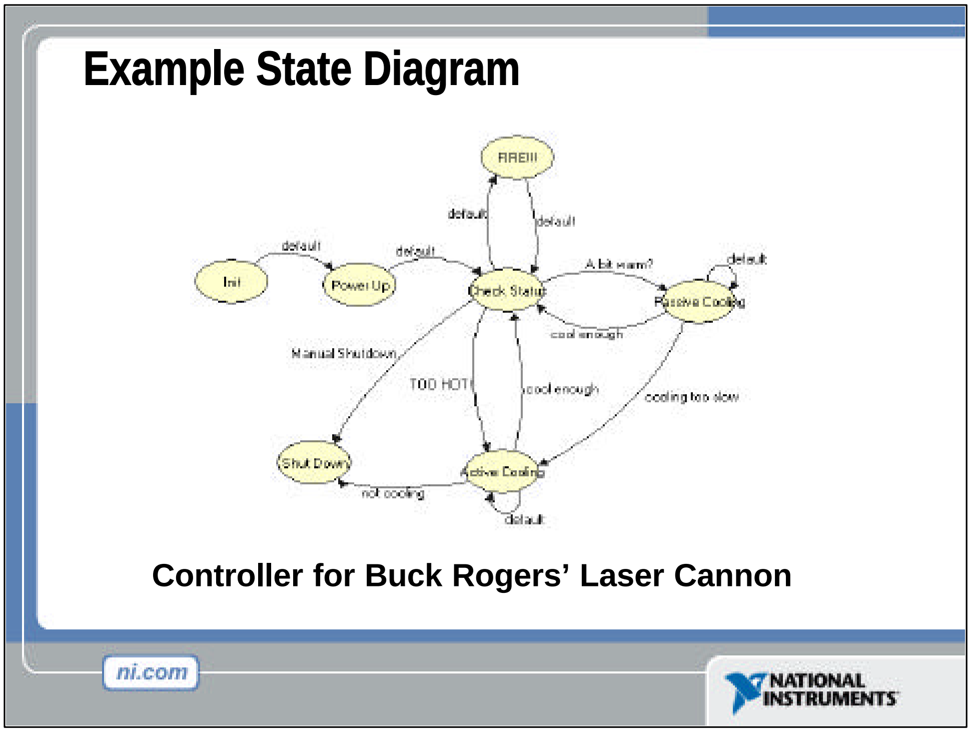


State Machine

- Used to implement decision-making algorithms, for example:
  - Diagnostic routines
  - Process monitoring and control
- Appropriate for algorithms described by flow chart or state diagram

The state machine pattern is one of the most widely recognized and highly useful design patterns for LabVIEW.

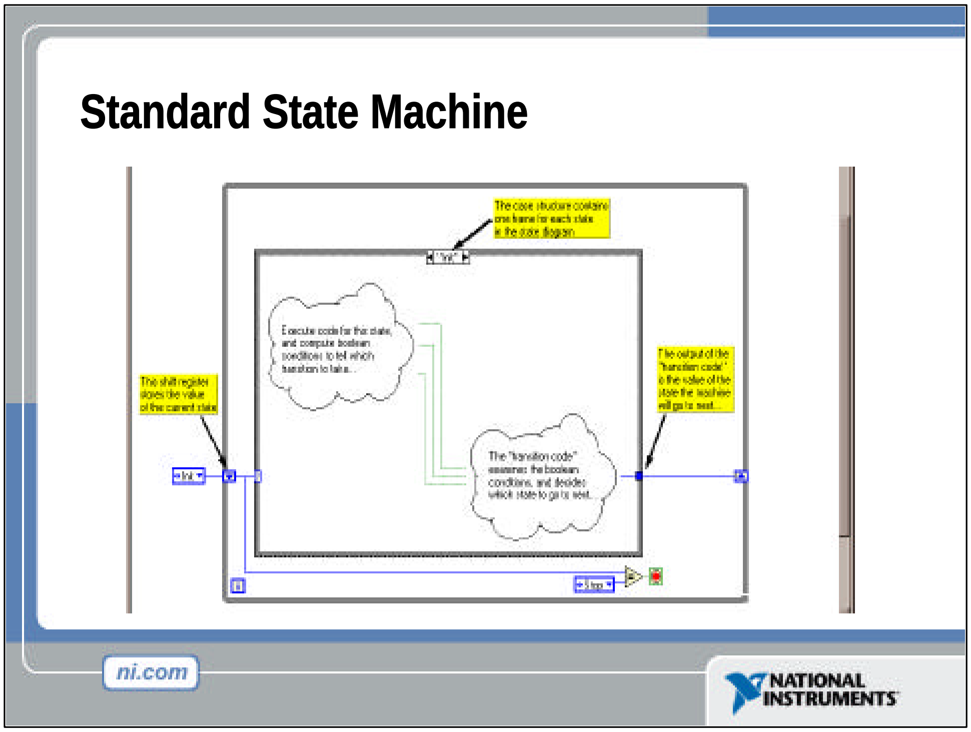
This pattern neatly implements any algorithm explicitly described by a state diagram (flow charts work, too). More precisely, it implements any algorithm described by a “Moore machine” – that is, a state machine which performs a specific action for each state in the diagram. (Contrast this with the “Mealy machine” which performs an action for each transition.) A state machine usually illustrates a moderately complex decision making algorithm, such as a diagnostic routine or a process monitor.



Example State Diagram

Controller for Buck Rogers’ Laser Cannon

Here’s an example diagram for a “Moore” machine that describes a control algorithm for a laser cannon – it should fire the cannon continuously without allowing it to get dangerously hot. Notice that in this diagram, the “states” (ovals) describe actions that are performed when the control process is in that state, whereas the “transitions” (arrows) simply describe when and how the process can move from one state to another.

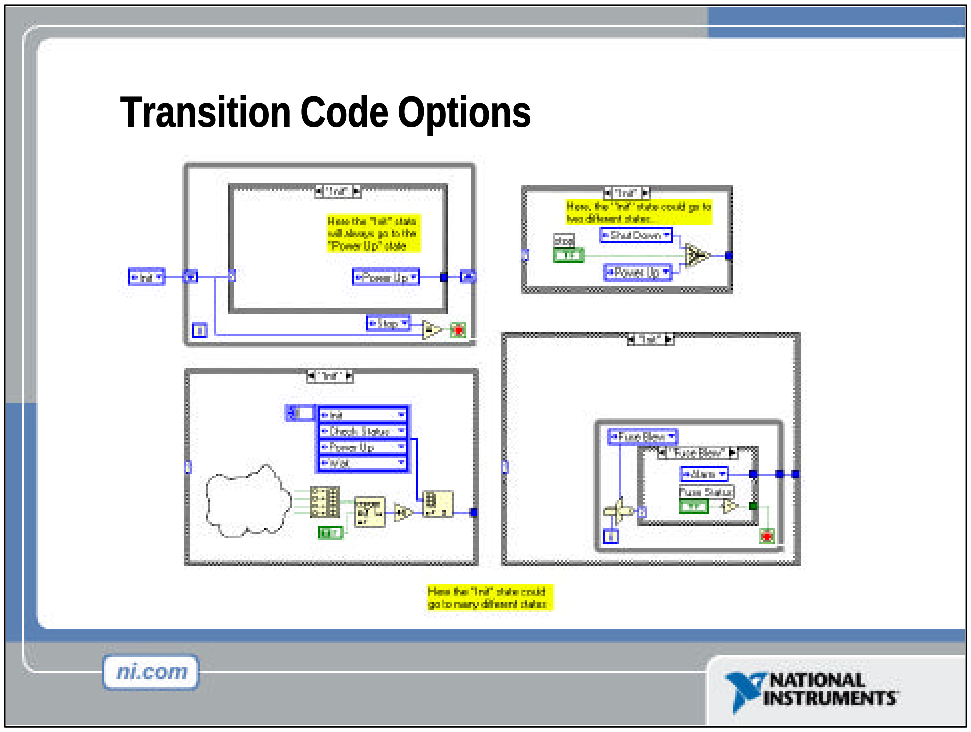


Standard State Machine

The case structure contains one frame for each state in the state diagram
"Init"
Execute code for this state, and compute boolean conditions to tell which transition to take...
The shift register stores the value of the current state
The output of the "transition code" is the value of the state the machine will go to next...
The "transition code" examines the boolean conditions, and decides which state to go to next...
Init
Stop
i
ni.com
NATIONAL INSTRUMENTS

The standard LabVIEW state machine consists of a large while loop, a shift register to remember the current state, and a case structure that holds separate code to run for each state. (If you use an enum to pass around the value for the current state, the pattern becomes much easier to read – and if you use a typedef, you only need to edit the typedef once in order to add another state to the machine)

Note that each frame in the case structure must contain code for deciding what state the process should go to next. We call this the “transition” code, and since there’s more than one standard way to implement that, we’ll leave it for the next slide.



Transition Code Options

Here the "Init" state will always go to the "Power Up" state

Here, the "Init" state could go to two different states...

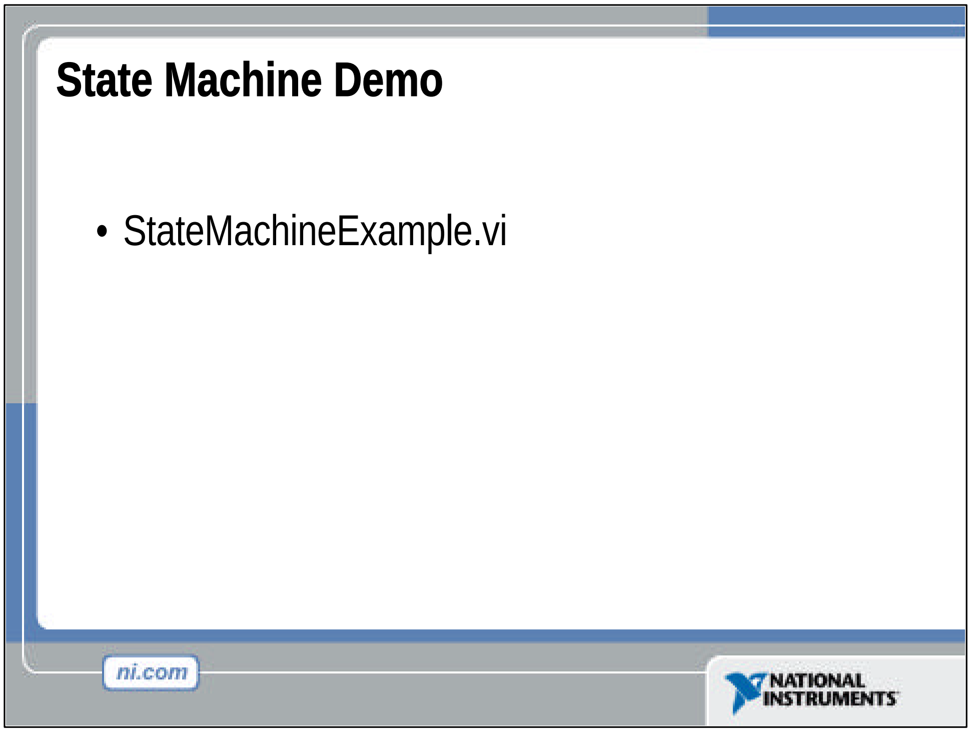
Here the "Init" state could go to many different states

Each of these case structures shows a different standard type of "transition code" – i.e. code that chooses the next state for the state machine.

The top left code clearly shows that the "Init" state has only one transition, and hence goes directly to the "Power Up" state without making any decision at all. The top right code switches between two possible transitions – the "Init" state will go to the "Shut Down" state if the "Stop" button has been pressed, but otherwise it goes to the "Power Up" state.

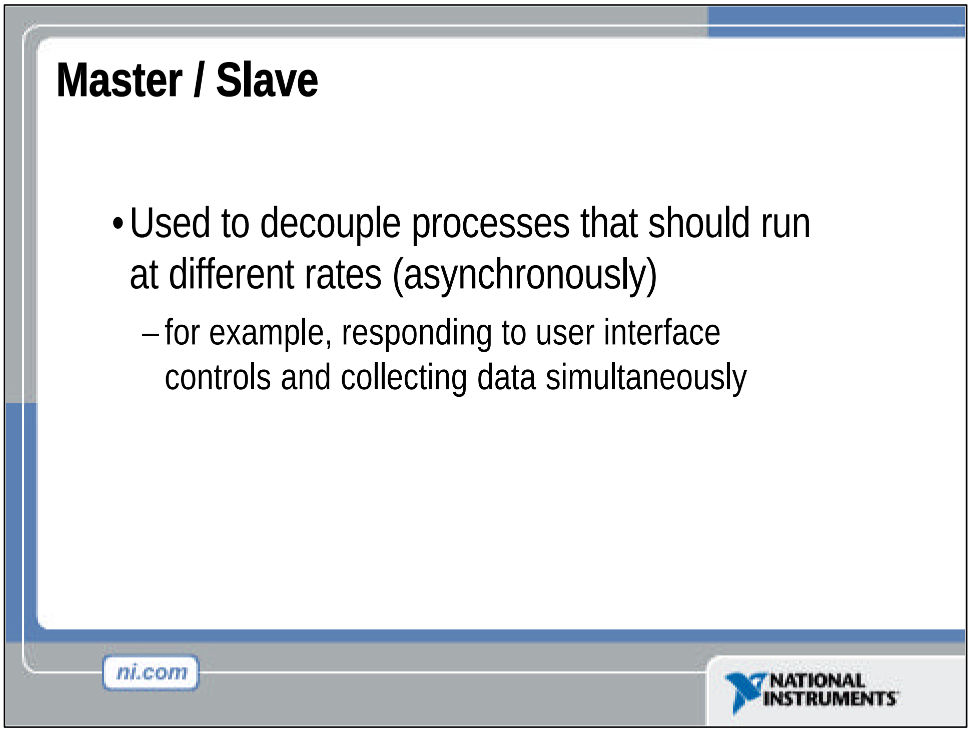
The code at the bottom left is a little more complicated, but still a common LabVIEW construct. The code for this state returns an array of boolean values, one for each transition we could take. Along with that array of boolean values is another array of enum constants specifying the new states where each possible transition could go. The index of the first "True" boolean in the array corresponds to the index of the new state in the array of enums… (Well, almost, but you can figure out the details yourself – just remember that the "Search Array" function will return –1 if the value is not found).

The code at the bottom right is functionally equivalent to the code to its left. It consists of a case structure embedded in a while loop. The case structure contains one diagram for each transition arrow that leaves the current state. Each of these diagrams has two outputs – a boolean value which specifies whether or not the transition should be taken, and an enumerated constant which specifies the state to which the transition goes. By using the loop index as input to the case structure, this code effectively runs through each transition diagram one by one, until it finds a diagram with a "TRUE" boolean output and then outputs the new state to which that transition goes. (Note that it is important to make sure the last transition always outputs a TRUE value). Though this code may appear slightly more complicated than the code to the left, it does offer the ability to add names to transitions by "casting" the output of the loop index to an enumerated type. This allows you to add "automatic documentation" to your transition code.



State Machine Demo

- StateMachineExample.vi



Master / Slave

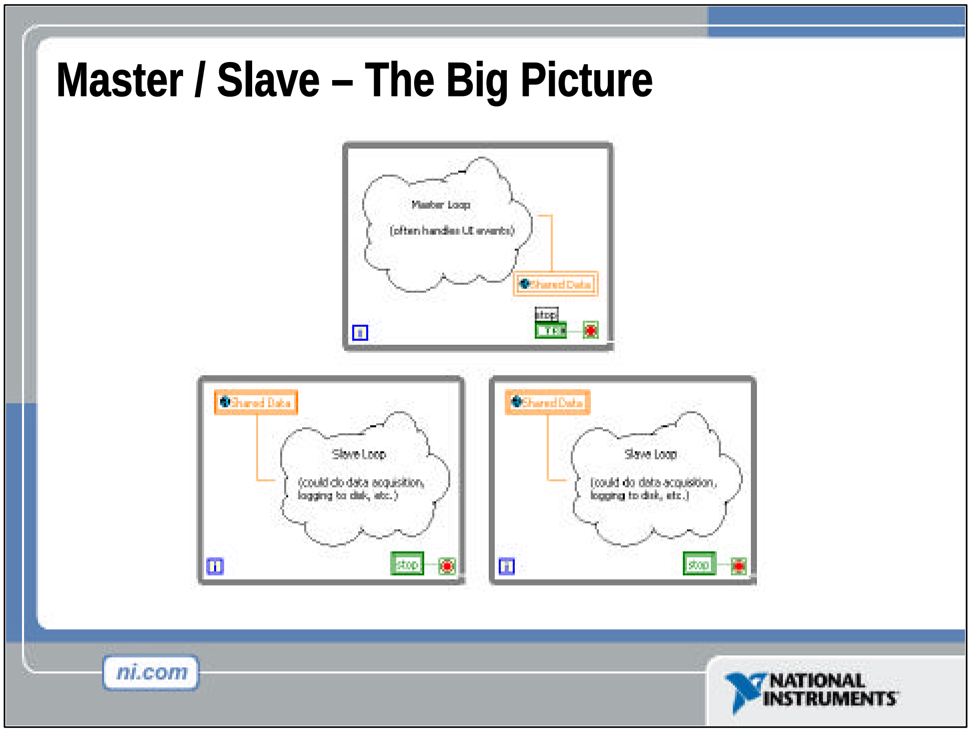
- Used to decouple processes that should run at different rates (asynchronously)
  - for example, responding to user interface controls and collecting data simultaneously

The Master/Slave pattern is generally useful when you have two or more processes that you expect to run continuously but at different rates.

For example, suppose you want to write an application that measures and logs a slowly changing voltage once every five seconds, acquires a waveform from a transmission line and displays it on a graph every 100 ms, and also provides a user interface that allows you to change parameters for each acquisition as well as providing the ability to bring up a print dialog to print various controls displayed on the panel. You could, conceivably, put both the voltage measurement and the waveform acquisition together in one big loop and only perform the voltage measurement on every 50th iteration of the loop. However, if the voltage measurement and log take longer to complete than the single acquisition and display of the waveform, then the next iteration of the waveform acquisition will be delayed because it cannot begin before all of the code in the previous iteration completes. Secondly, this architecture would make it difficult to change the rate at which waveforms were acquired without changing the rate of the voltage log.

The standard “design pattern” approach would be to put the acquisition processes in two separate loops, both of them driven by a master loop receiving input from the UI controls. This ensures that each acquisition process will not affect the other, and that any delays caused by the user interface (for example, bringing up a dialog), will not delay any iteration of the acquisition processes. The power of using this pattern is that it separates the data flow of your diagram into independent processes. However, in order for these to communicate, you will have to use some form of globally available, shared data. (e.g. locals, globals, occurrences, notifiers, and/or queues). This does break LabVIEW’s dataflow paradigm, leaves the door open for race conditions, and incurs a little more overhead than passing data by wire (acquiring mutexes, and so on).

The master loop could itself be a simple event loop or a state machine that incorporates event gathering as one of its states. Alternatively, the master loop could even start up a separate event loop that runs independently from it. This last pattern would probably be the best choice if the master loop is a state machine that gathers needs to gather UI event information in more than one state. Most importantly, you need to make sure you specify at the outset that only one loop will write to any given piece of shared data. Typically, the master loop writes to all shared data, and slave loops read the data.

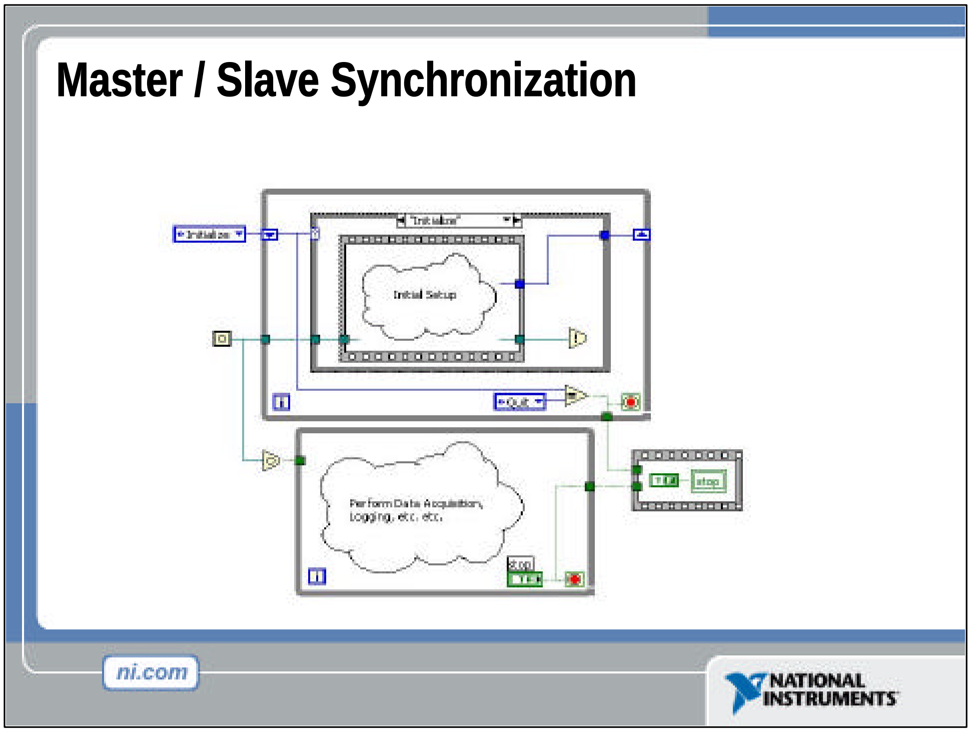


Master / Slave – The Big Picture

Master Loop
(often handles UI events)
Shared Data
stop
Shared Data
Slave Loop
(could do data acquisition, logging to disk, etc.)
stop
Shared Data
Slave Loop
(could do data acquisition, logging to disk, etc.)
stop
ni.com
NATIONAL INSTRUMENTS

The big picture – one master loop drives one or more slave loops, communicating via some shared data depot.

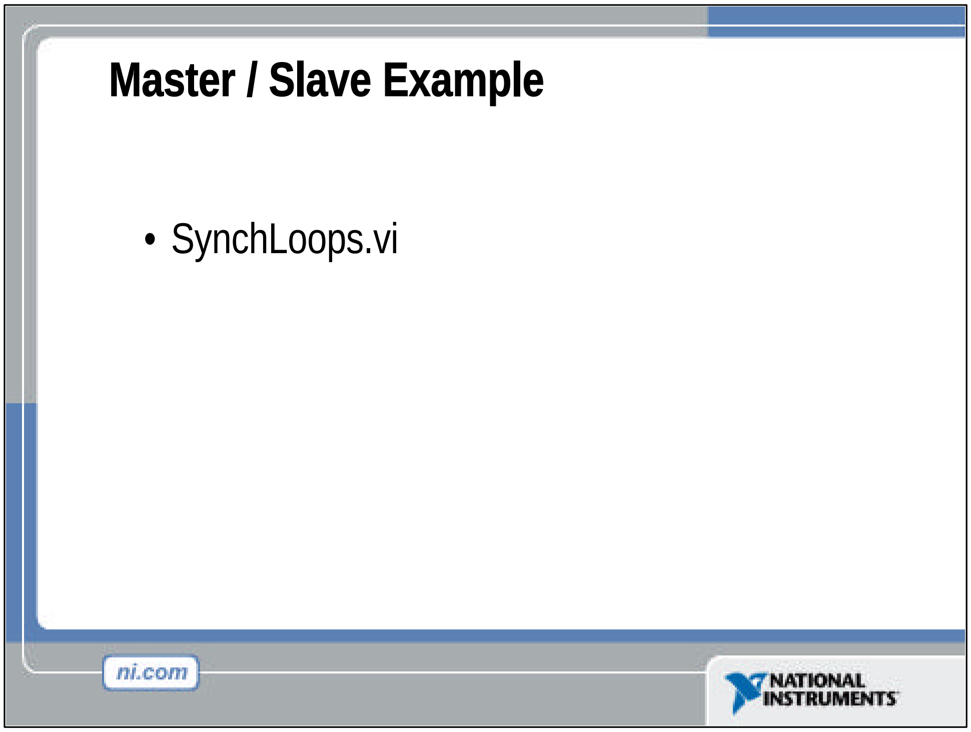
Variations on this include having loop A control loop B which controls loop C, and so on. You could also design “Peer loops” which communicate back and forth to each other, but you must make sure that two loops may not write to the same data depot at the same time. A good policy is to make sure that no more than one loop may write to any given piece of shared data.



Master / Slave Synchronization

Sometimes a Master/Slave application may require the slave loop to be halted while the master loop performs initialization or reset routines. This can often be done neatly by using occurrences.

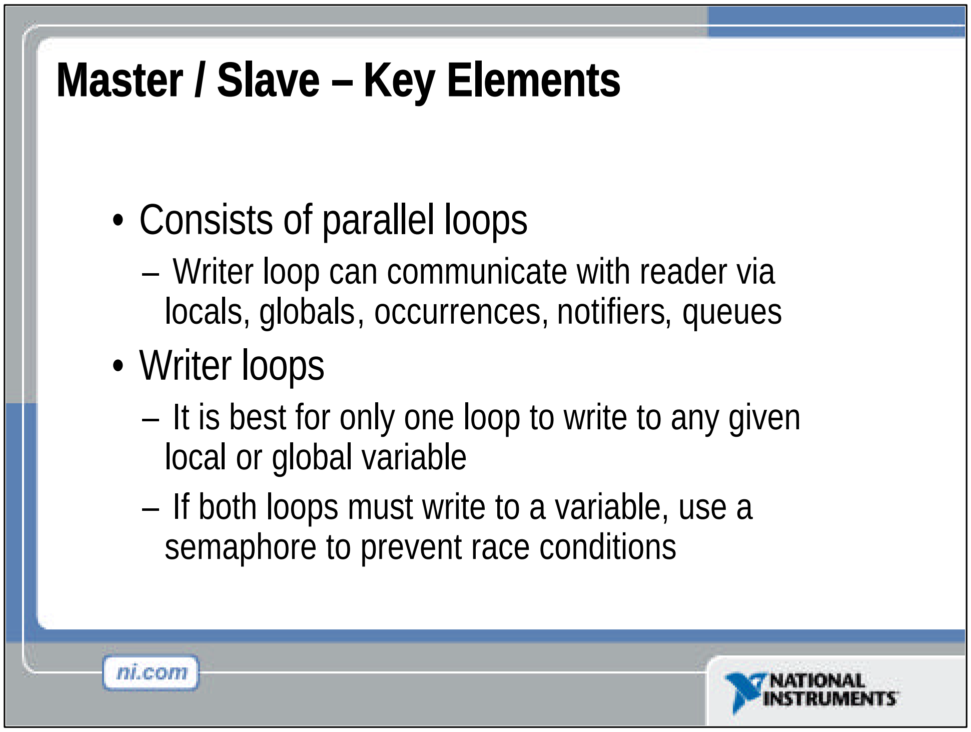
The above diagram shows an example that delays the start of the slave loop until an initialization routine is completed.



Master / Slave Example

- SynchLoops.vi

This example is slightly more complicated than the previous slide – it uses occurrences to not only delay the start of the slave loops, but also to turn them on or off during the execution of the program.



Master / Slave – Key Elements

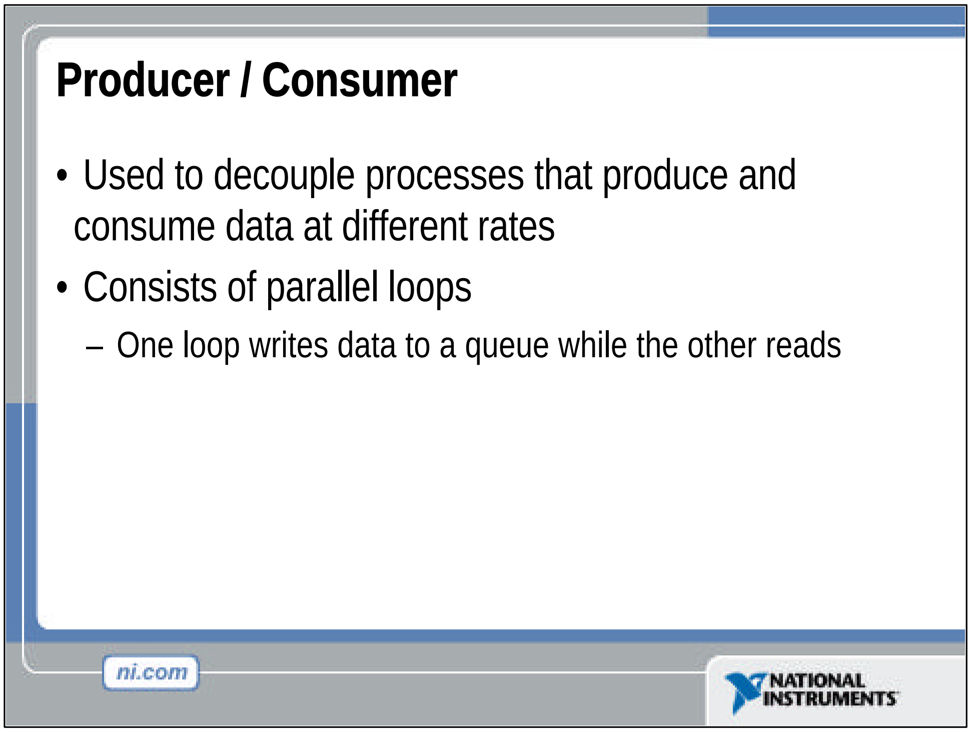
- Consists of parallel loops
  - Writer loop can communicate with reader via locals, globals, occurrences, notifiers, queues
- Writer loops
  - It is best for only one loop to write to any given local or global variable
  - If both loops must write to a variable, use a semaphore to prevent race conditions

ni.com

NATIONAL INSTRUMENTS

The Master/Slave pattern consists of multiple parallel loops, each of which may drive processes at different rates. Since communication of data between these processes breaks data flow, it must be done by writing to and reading from globally available data pools – i.e. locals, globals, occurrences, notifiers or queues.

Be careful, however, if more than one loop must write to a local or a global variable. If both loops try to write to the variable at the same time, there's no telling which value may ultimately get written! (This is known as a race condition.) To avoid this situation, place an "acquire/release semaphore" pair around any code that writes to the global to ensure that no one attempts to write to it at the same time.



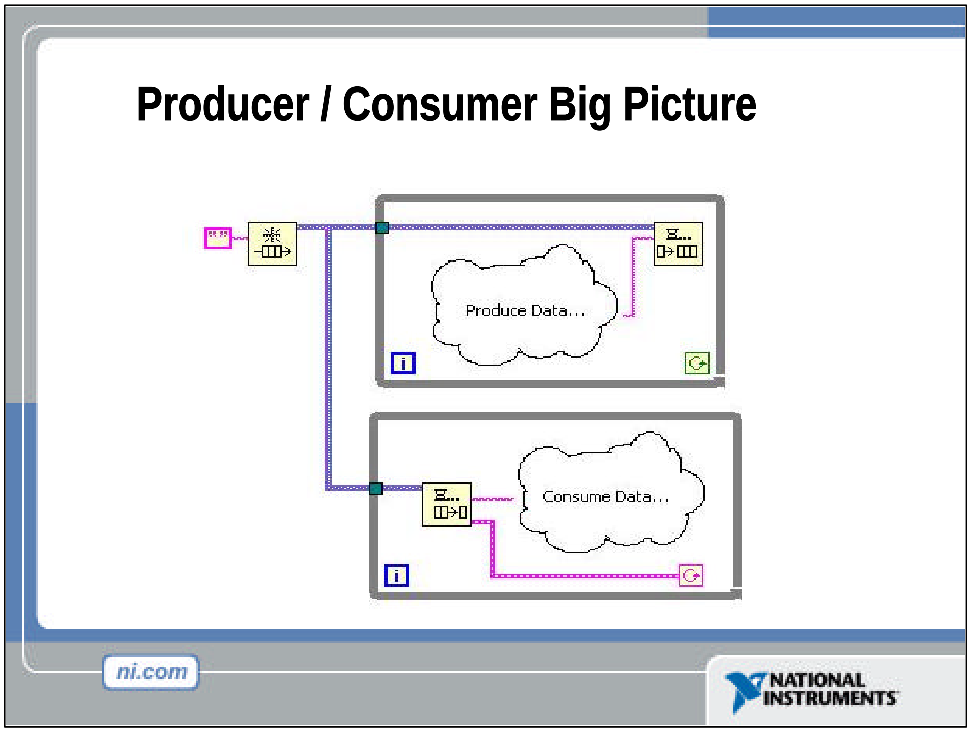
Producer / Consumer

- Used to decouple processes that produce and consume data at different rates
- Consists of parallel loops
  - One loop writes data to a queue while the other reads

ni.com

NATIONAL INSTRUMENTS

The Producer/Consumer pattern is really just a subclass of the Master/Slave pattern where the main communication between the loops is via queue. Using queues rather than globals to pass data provides a buffering effect, but if the data writer occasionally produces data faster than the reader can process it, data will not be lost. This can be especially useful for handling UI events that take a long time to complete (e.g. printing in response to a user).



Producer / Consumer Big Picture

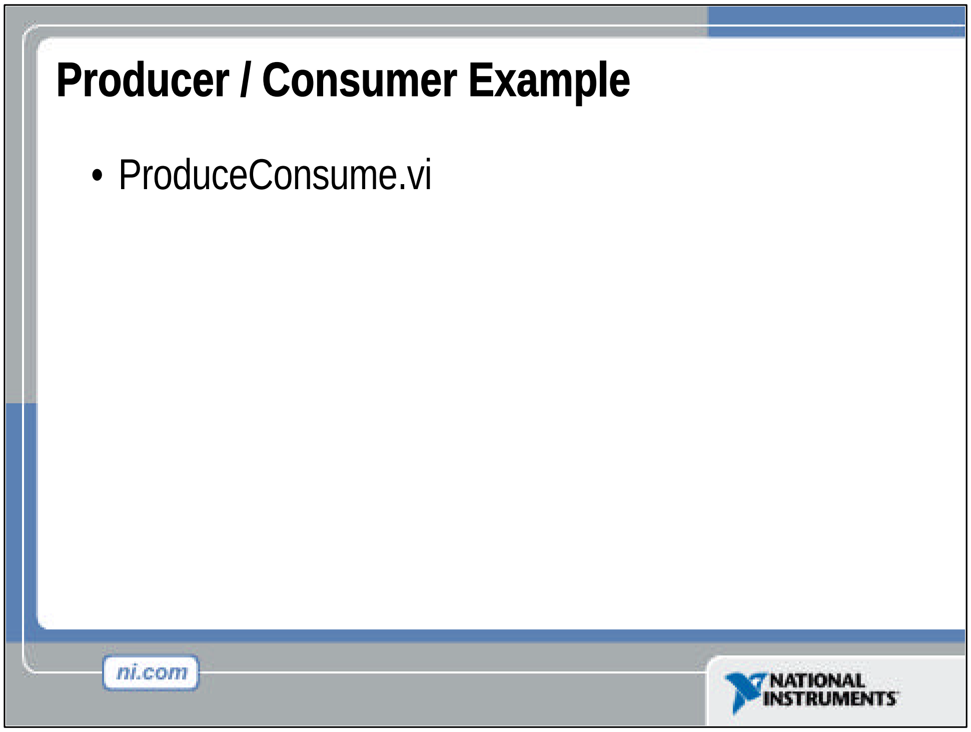
ni.com

NATIONAL INSTRUMENTS

One loop produces data (via computation, DAQ, and so on) and puts the data in the queue.

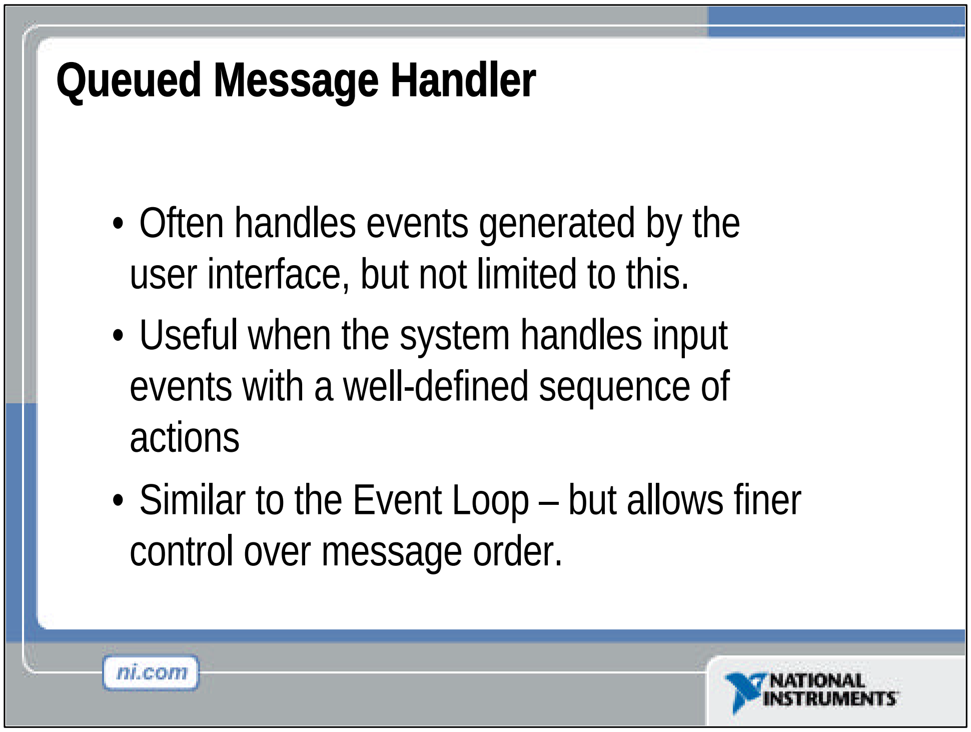
The other loop waits until there is data in the queue. Then it pulls the first element out of the queue and processes it.

It just doesn't get any simpler than this, folks!



Producer / Consumer Example

- ProduceConsume.vi



Queued Message Handler

- Often handles events generated by the user interface, but not limited to this.
- Useful when the system handles input events with a well-defined sequence of actions
- Similar to the Event Loop – but allows finer control over message order.

ni.com
NATIONAL INSTRUMENTS

Another popular pattern is the Queued Message Handler. (This pattern is often referred to as a “Queued State Machine”, but because it need not adhere strictly to a state diagram, I prefer to give it a different name to avoid confusion). This pattern has been discussed quite a bit in various issues of *LabVIEW Technical Resource* (LTR). It is most commonly used to implement code for a user interface, but you certainly don’t need to limit its use to this!

It is important to note that the behavior of the Message Handler is very similar to the event loop – various messages (events) are queued up, and they are handled one by one in the order that they exist in the queue. Furthermore, each sub-diagram in this pattern represents a handling routine, just like the Event Loop.

So what’s the difference? Why choose one of these over the other?

Here’s the scoop:

Event Loop –

+ simpler to configure

- handles events in FIFO (First In, First Out) order

- limited to UI messaging in LV 6.1

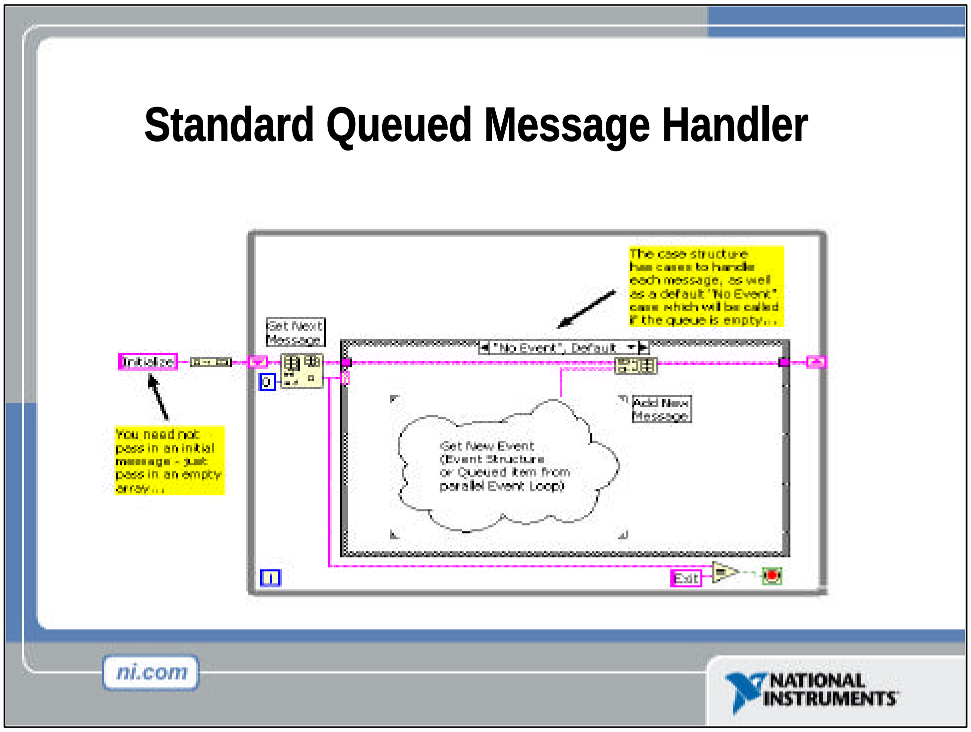
Queued Message Handler –

+ allows more control over order in which messages get handled (useful for error handling/recovery)

+ handles any user defined message

+ can be configured to work in parallel with Event Loop, handling messages from the UI asynchronously

- more work for initial setup



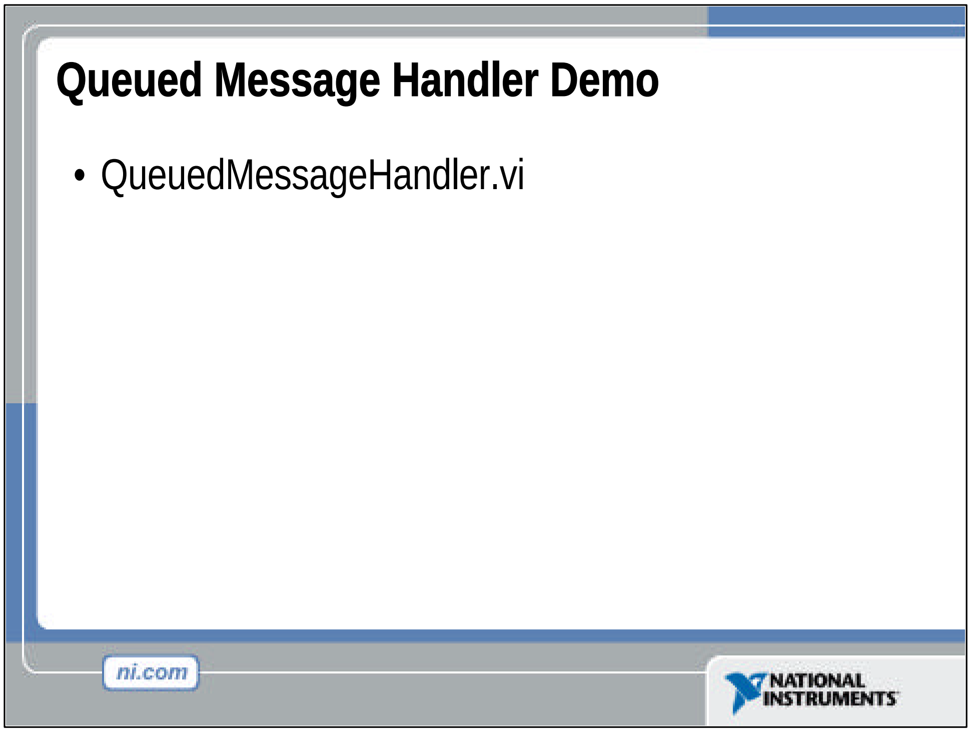
Standard Queued Message Handler

The case structure has cases to handle each message, as well as a default "No Event" case which will be called if the queue is empty…

You need not pass in an initial message - just pass in an empty array…

The Queued Message Handler consists of a large “while” loop, an internal “Case” structure, and a shift register on the while loop that holds the queued messages. (note that if you use a LabVIEW queue rather than an array of strings, you will not need the shift register). For each message that may be sent, the case structure contains one diagram with appropriate code to handle the message. The case structure may also have a default diagram which will queue up new events if the queue is empty.

Each iteration of the loop removes the top message from the queue and runs the proper handling diagram in the case structure. Handlers that have a “Default” or “No Event” frame will execute this code when the queue is empty. The loop terminates when the “Exit” message comes to the top of the queue.

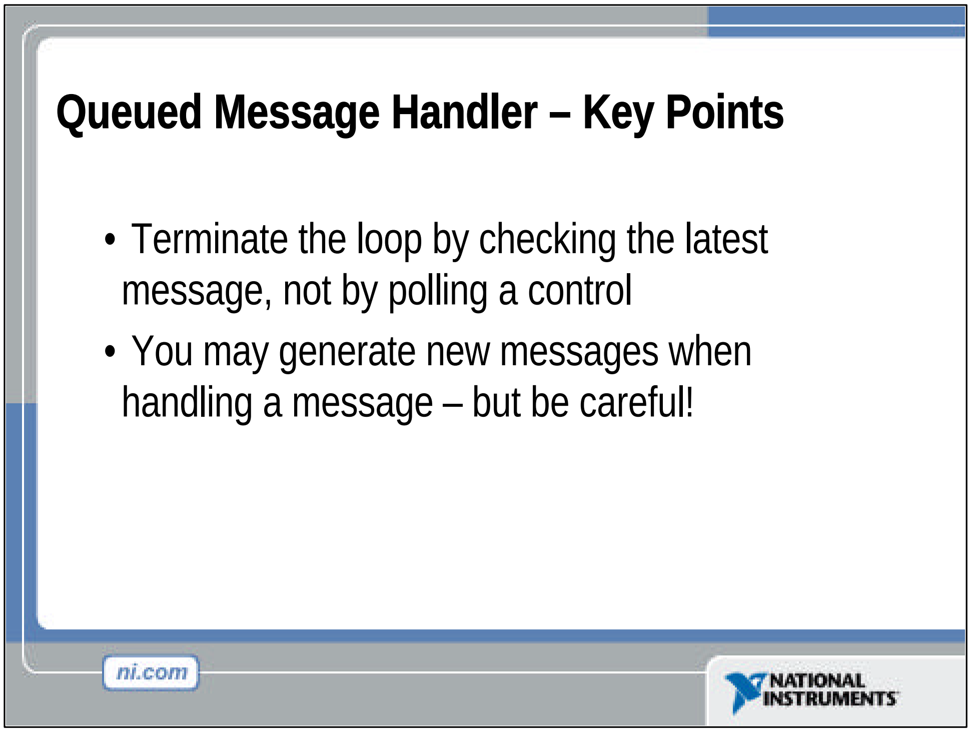


Queued Message Handler Demo

- QueuedMessageHandler.vi

ni.com

NATIONAL INSTRUMENTS



Queued Message Handler – Key Points

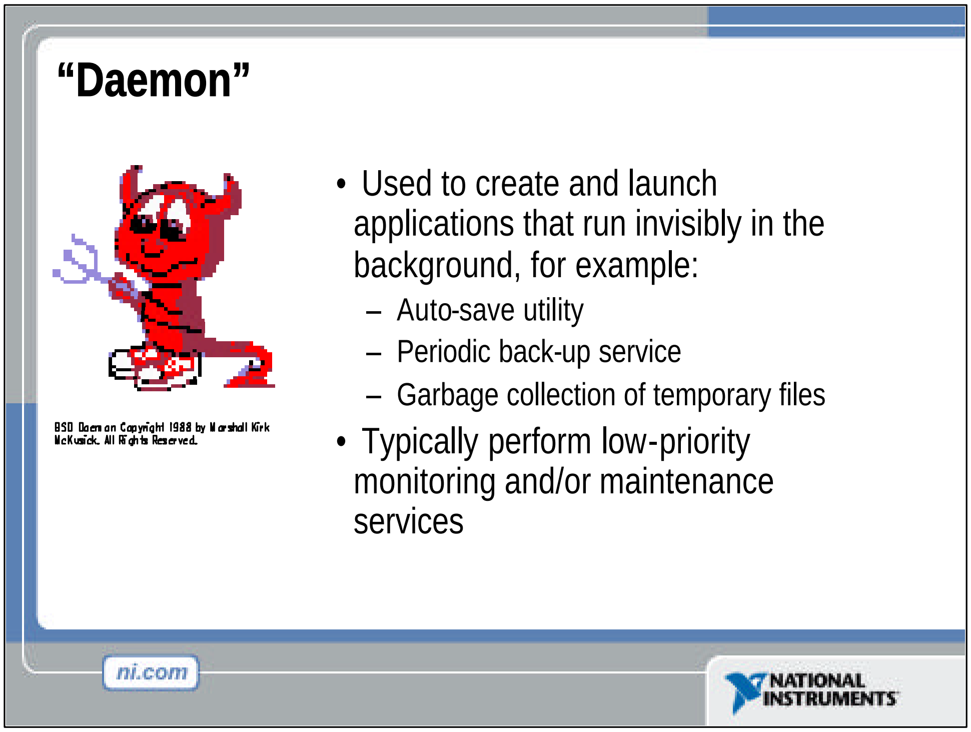
- Terminate the loop by checking the latest message, not by polling a control
- You may generate new messages when handling a message – but be careful!

ni.com

Two key points to remember when creating a Queued Message Handler:

First, the loop should be terminated by checking the latest message rather than polling a control. This architecture allows you to execute any necessary cleanup code (for example, shutting down parallel loops) before shutting down the main loop.

Second, you may generate new messages inside the handler code for a message. This gives you a lot of flexibility in designing event handling code, but it also makes it possible to generate an infinite cascade of messages which would effectively hang the user interface. A good rule of thumb might be to require that messages generated by a handler routine should never generate new messages of their own.



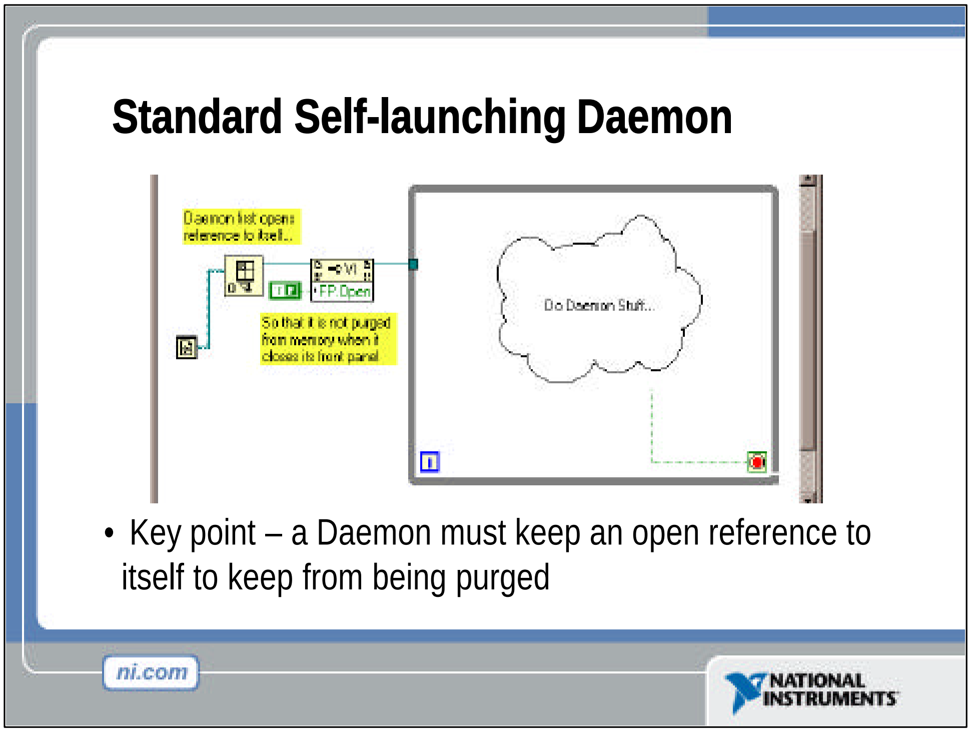
“Daemon”

BSD Daemon Copyright 1988 by Marshall Kirk McKusick. All Rights Reserved.

- Used to create and launch applications that run invisibly in the background, for example:
  - Auto-save utility
  - Periodic back-up service
  - Garbage collection of temporary files
- Typically perform low-priority monitoring and/or maintenance services

ni.com
NATIONAL INSTRUMENTS

The “Daemon” pattern is an especially powerful concept in LabVIEW – it allows you to run an invisible VI in the background to take care of routine monitoring and/or maintenance services. Combined with VI Server, the daemon pattern allows you to create some truly amazing applications.



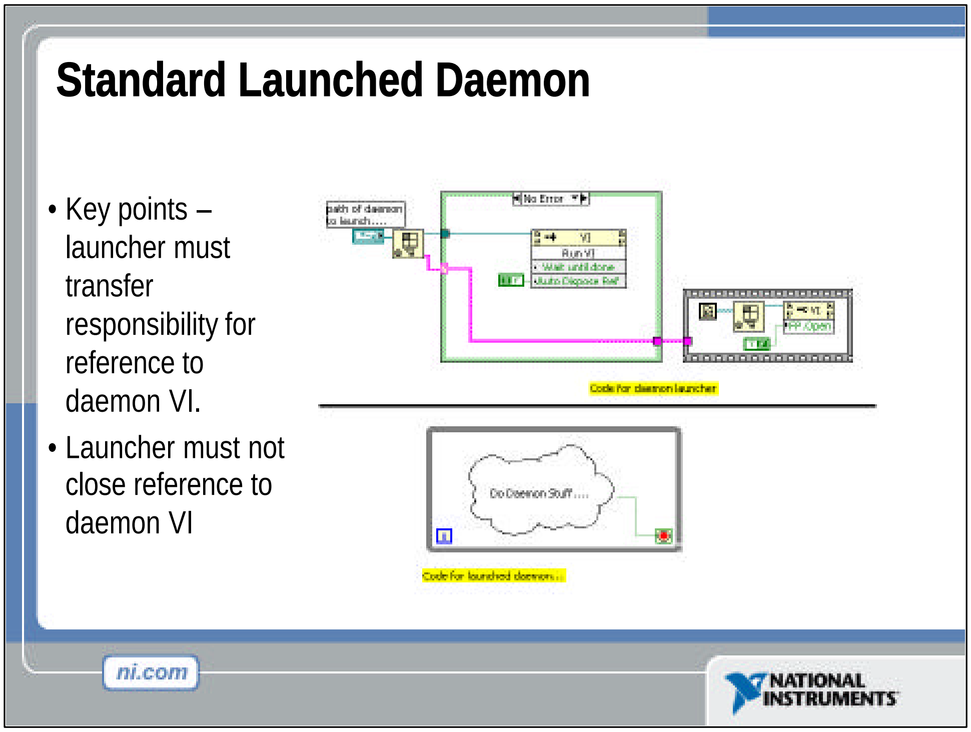
Standard Self-launching Daemon

Daemon first opens reference to itself...
VI
FP.Open
So that it is not purged from memory when it closes its front panel
Do Daemon Stuff...
ni.com
NATIONAL INSTRUMENTS

- Key point – a Daemon must keep an open reference to itself to keep from being purged

Here is some sample code to help you create self-launching daemons. Notice that the daemon obtains a reference to itself before closing its front panel – this prevents LabVIEW from removing it from memory before it has a chance to run the daemon code.

Note, however, that this daemon must briefly show its front panel while it obtains a self-reference on launch – this problem can be resolved by using a “launched” daemon pattern shown on the next slide.

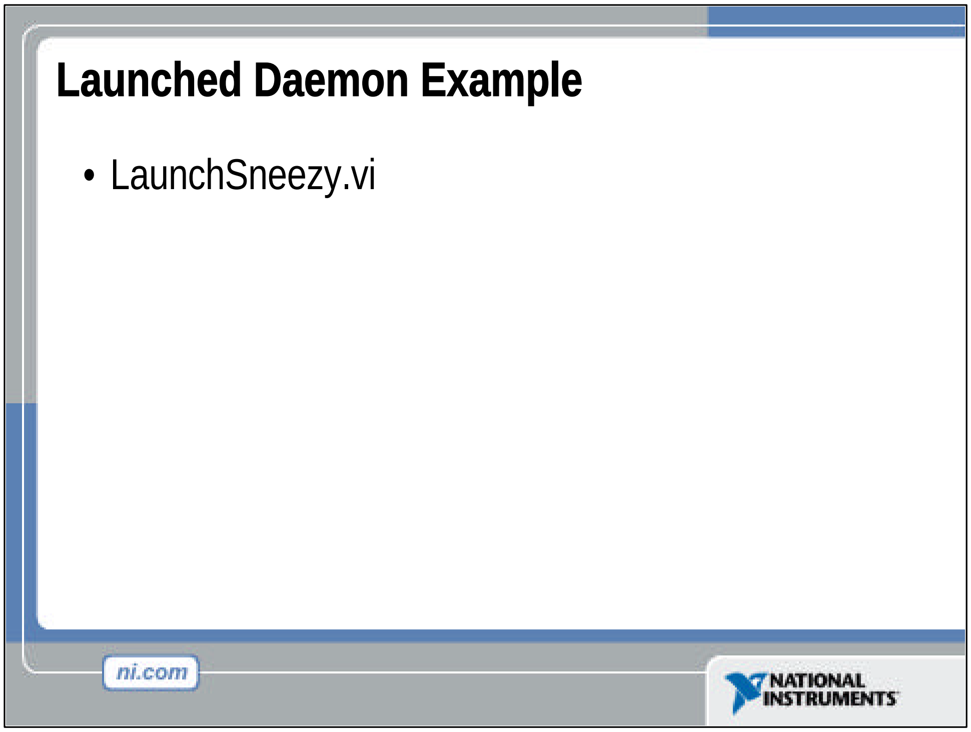


Standard Launched Daemon

- Key points – launcher must transfer responsibility for reference to daemon VI.
- Launcher must not close reference to daemon VI

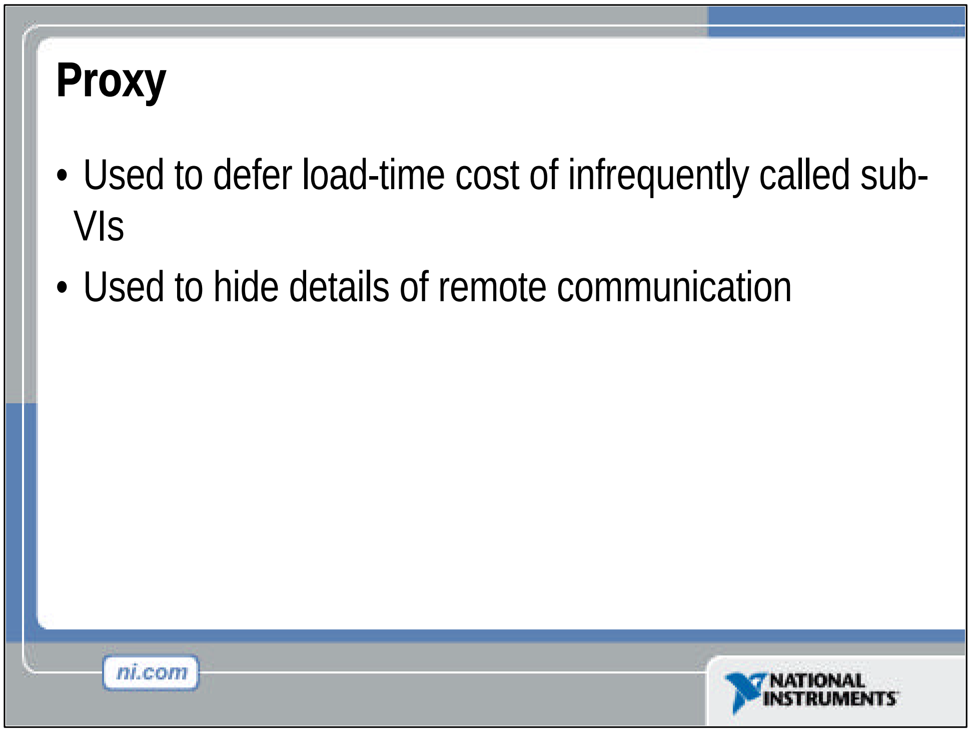
Here is some sample code to create diagrams for a daemon launcher and the launched daemon itself. Note that the daemon does not have to obtain a reference to itself – this is because the launcher obtained the reference and then just transferred responsibility for closing the reference to the daemon VI.

To transfer ownership of the daemon VI’s reference, set “Auto Dispose Ref” to true when you run the daemon. It is very important that the launcher does *not* close the reference to the daemon VI.



Launched Daemon Example

- LaunchSneezy.vi



Proxy

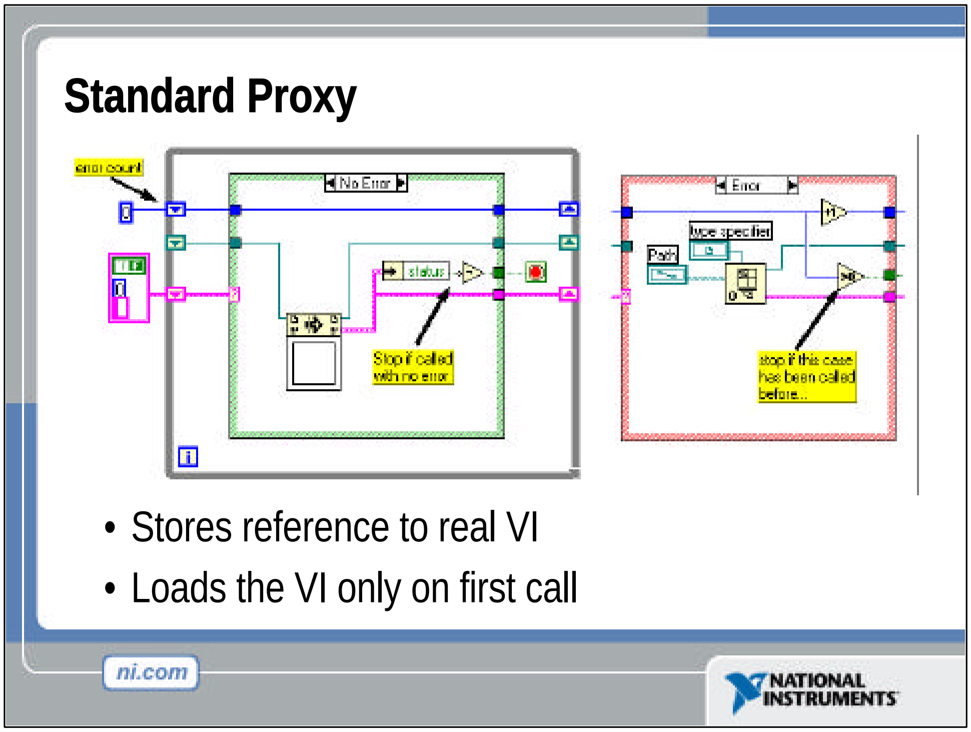
- Used to defer load-time cost of infrequently called sub-VIs
- Used to hide details of remote communication

ni.com

NATIONAL INSTRUMENTS

The Proxy pattern is generally used to defer loading of subVIs until they are needed by your program. This technique, sometimes called “lazy loading,” can dramatically decrease the initial time it takes to load your application. Note, however, that this has the drawback of incurring a slight delay the first time you call each deferred subVI.

In addition, the Proxy can be used to simplify the creation of a distributed application by hiding the details of the remote communication.



Standard Proxy

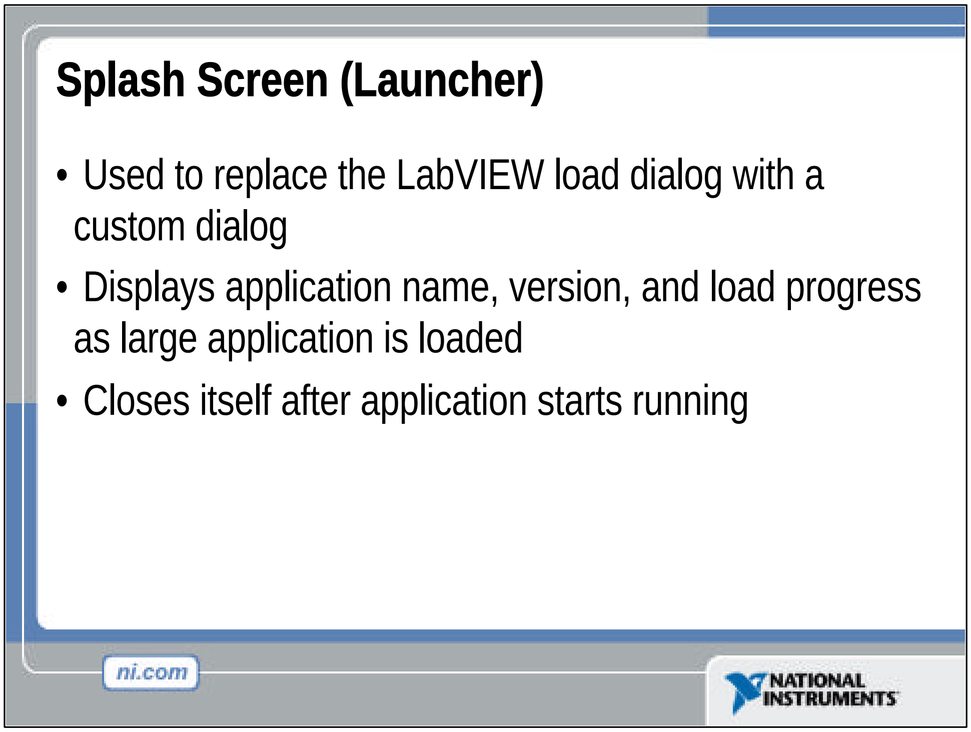
error count
No Error
status
Stop if called with no error
Error
type specifier
Path
stop if this case has been called before...

- Stores reference to real VI
- Loads the VI only on first call

A proxy VI stores a reference to the real VI – you can do this in LabVIEW by using an uninitialized shift register.

The proxy first attempts to call the real VI by reference. If this fails (usually meaning the reference is bad), the proxy attempts to obtain a valid reference to the VI and call it again.

Note that since the Proxy uses a Call By Reference node to call the subVI, you could easily re-configure it to call a subVI located on a separate machine in your network. This lets you move quickly and easily from a single machine application to a distributed app without having to modify your main application at all!

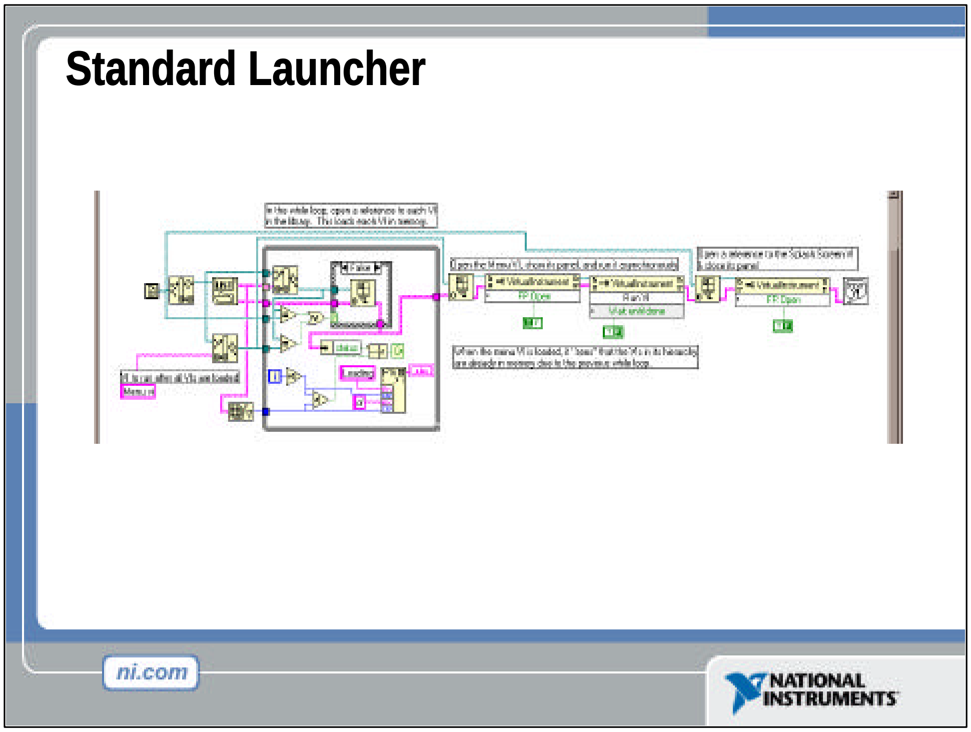


Splash Screen (Launcher)

- Used to replace the LabVIEW load dialog with a custom dialog
- Displays application name, version, and load progress as large application is loaded
- Closes itself after application starts running

ni.com

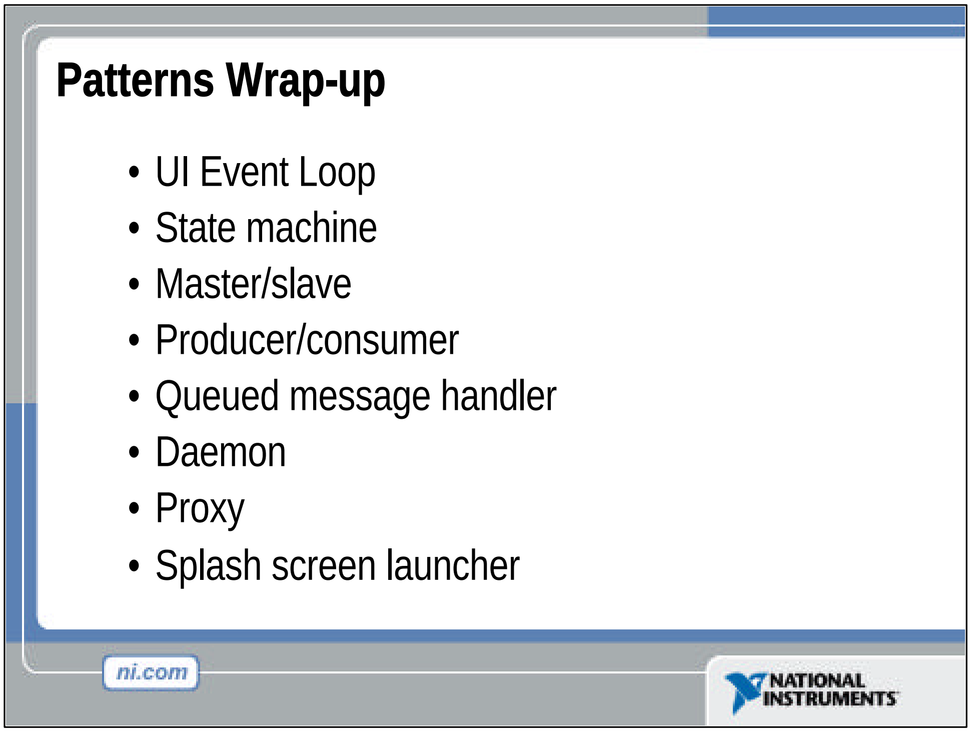
The Splash Screen (or Launcher) pattern allows you to replace LabVIEW’s load dialog with a custom window tailored specifically for your application. Like LabVIEW’s load dialog, it can be configured to display how much of the application has been loaded.



Standard Launcher

ni.com

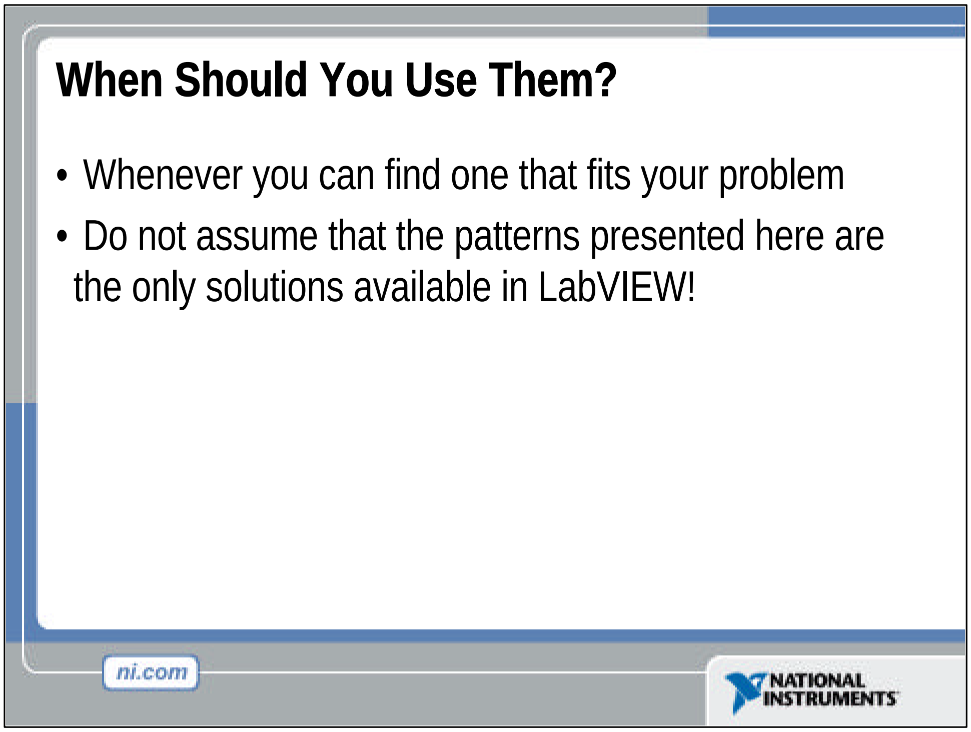
This diagram (taken from LabVIEW Technical Resource), is much less complicated than it looks. It assumes that all of the subVIs for your application are stored in one directory, and loads them all one by one. As each subVI is loaded, an indicator on the Launcher changes to show load progress. It is important that the top level VI is not loaded in this step because loading it would load all of the subVIs without updating the launch screen. The launcher then loads the top level VI. It opens the front panel of the top level VI so that the application is not purged from memory when the launcher closes. Finally, the launcher runs the application and, without waiting for the app to finish, closes its own front panel. As long as no other VI has a reference to the launcher, this will cause the launcher to exit memory.



Patterns Wrap-up

- UI Event Loop
- State machine
- Master/slave
- Producer/consumer
- Queued message handler
- Daemon
- Proxy
- Splash screen launcher

ni.com



When Should You Use Them?

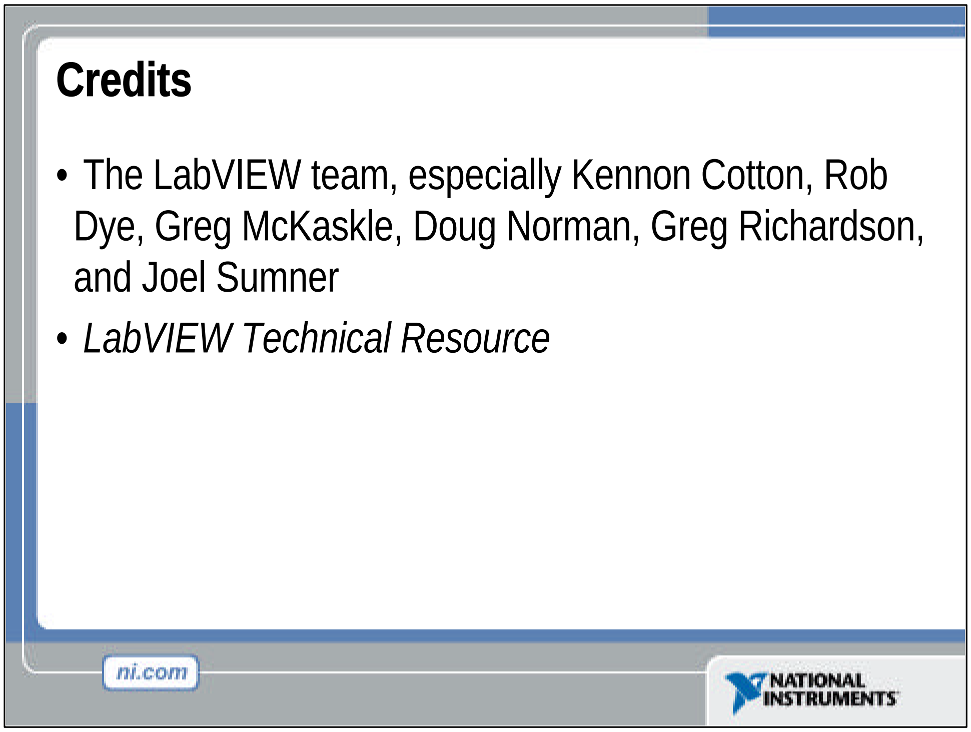
- Whenever you can find one that fits your problem
- Do not assume that the patterns presented here are the only solutions available in LabVIEW!

ni.com

NATIONAL INSTRUMENTS

When you are developing a new program in LabVIEW, it is a good idea to invest some time thinking about the high-level design first. If you do, you are more likely to find that your program can expand in a more manageable fashion. When you’re in this “design phase,” look for existing patterns that fit your application, and remember that you can base your design on a combination of patterns. (Don’t hesitate to modify a pattern if it doesn’t fit your problem perfectly.)

However, remember that what you can do with LabVIEW is limited only by your imagination – if you can’t find a pattern that fits you problem, make up a new one! (And submit it to the *LabVIEW Technical Resource* so that the rest of us can benefit.)



Credits

- The LabVIEW team, especially Kennon Cotton, Rob Dye, Greg McKaskle, Doug Norman, Greg Richardson, and Joel Sumner
- *LabVIEW Technical Resource*

ni.com
NATIONAL INSTRUMENTS