UiO : University of Oslo

FYS3240- 4240
Data acquisition & control

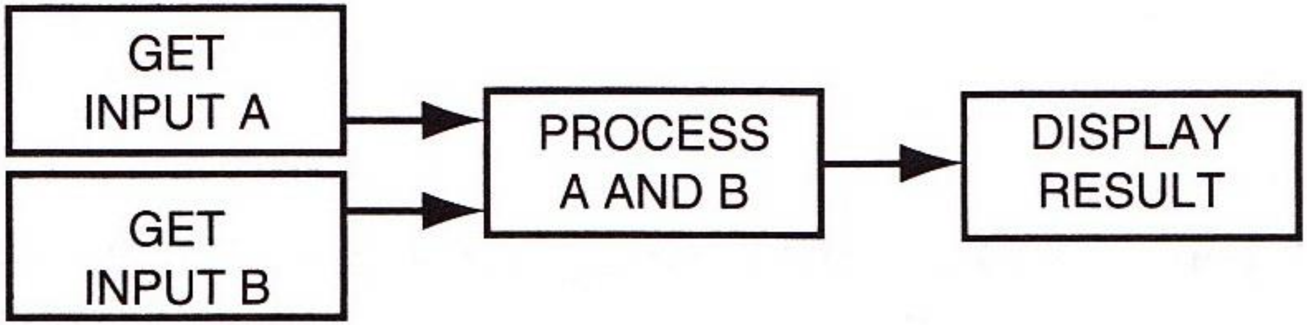
LabVIEW programming II

Spring 2019 – Lecture #3

Bekkeng 28.12.2018

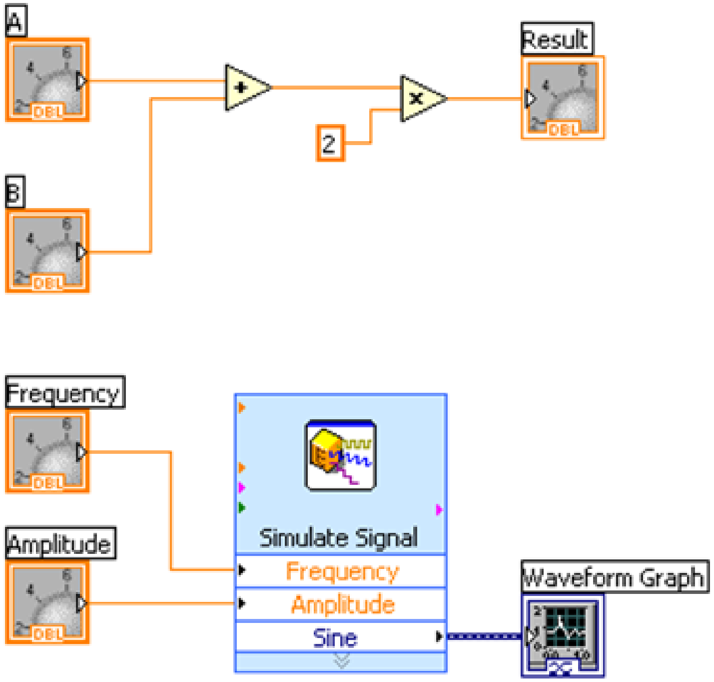
Dataflow programming

- With a dataflow model, nodes on a block diagram are connected to one another to express the logical execution flow
- When a block diagram node **receives <u>all required inputs</u>, it produces output data and passes that data to the next node** in the dataflow path. The movement of data through the nodes <u>determines the execution order of the functions on the block diagram</u>

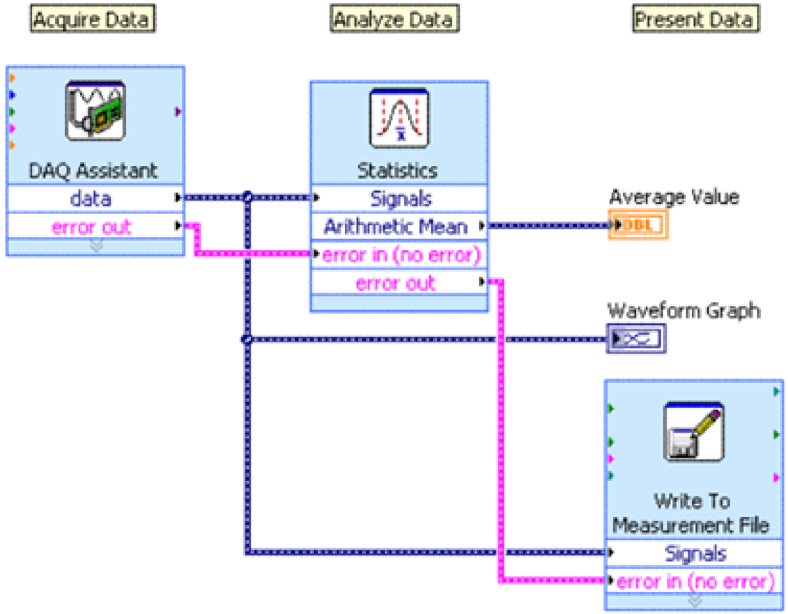


Dataflow Programming

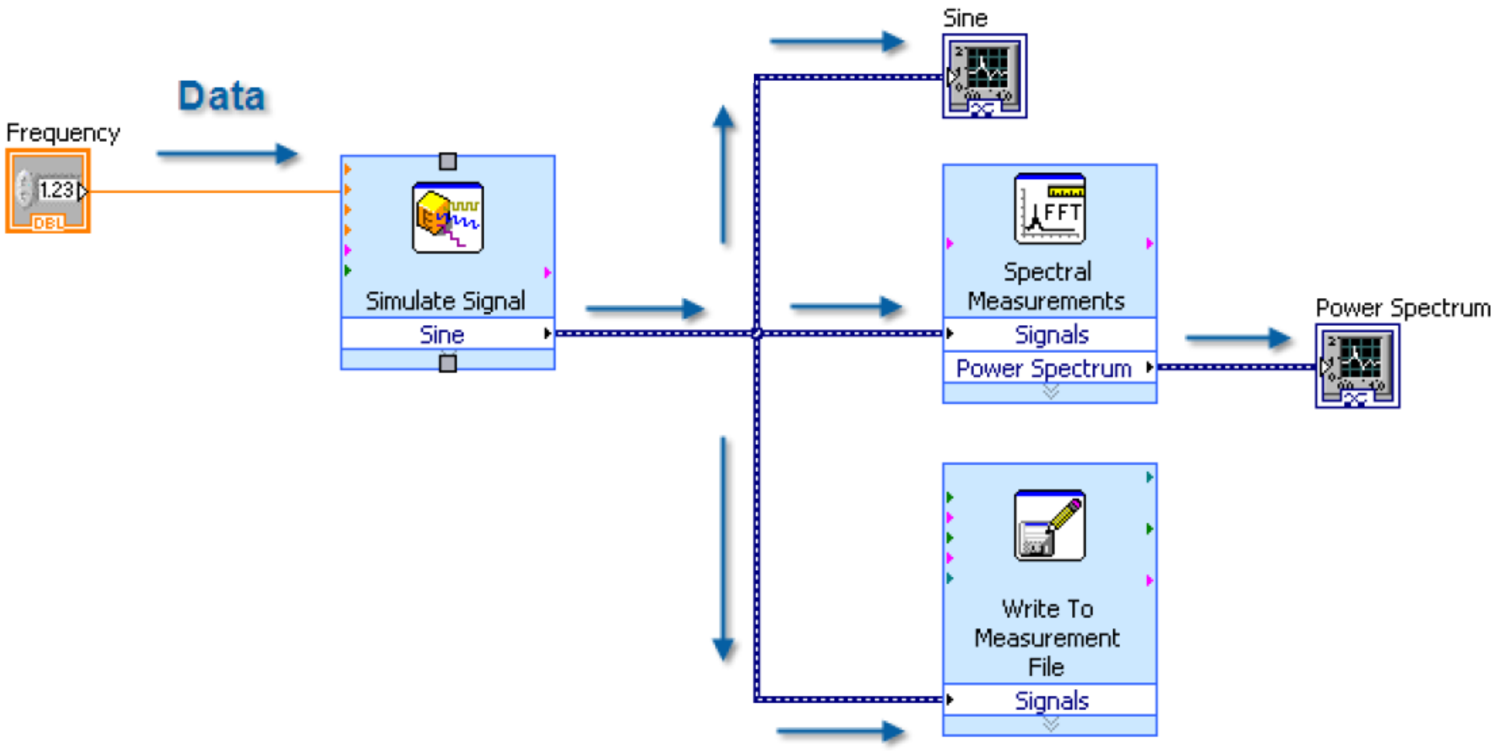
Which VI(s) will execute first?



Which VI will execute last?

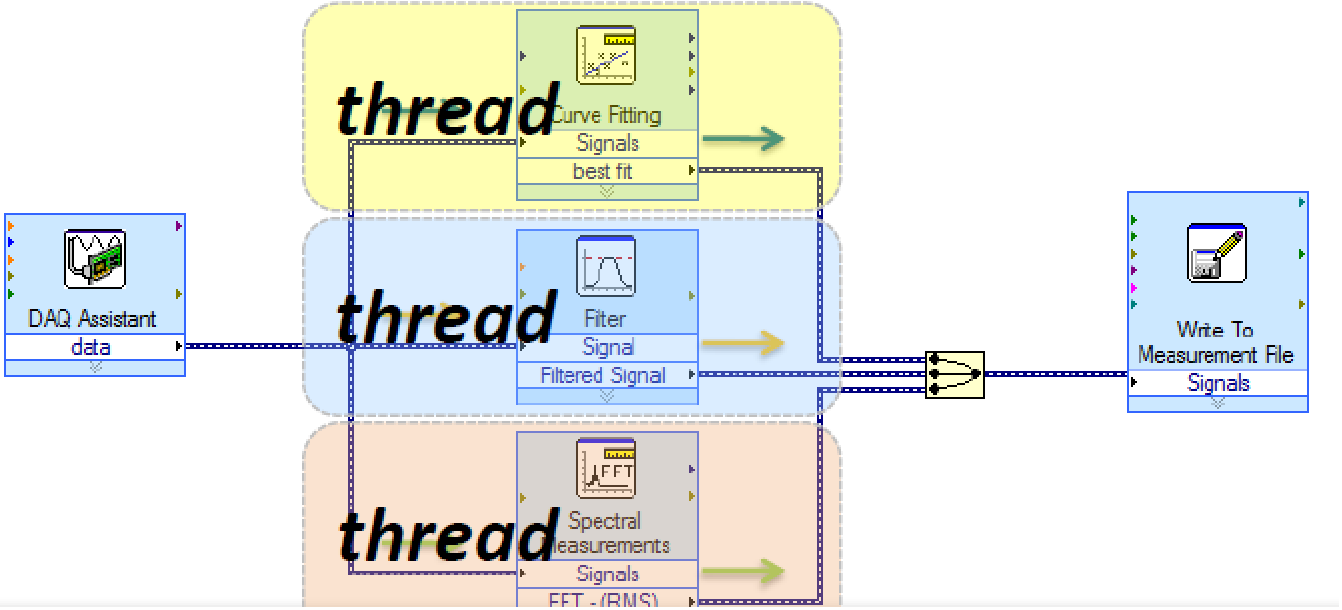


LabVIEW Graphical Programming – Dataflow

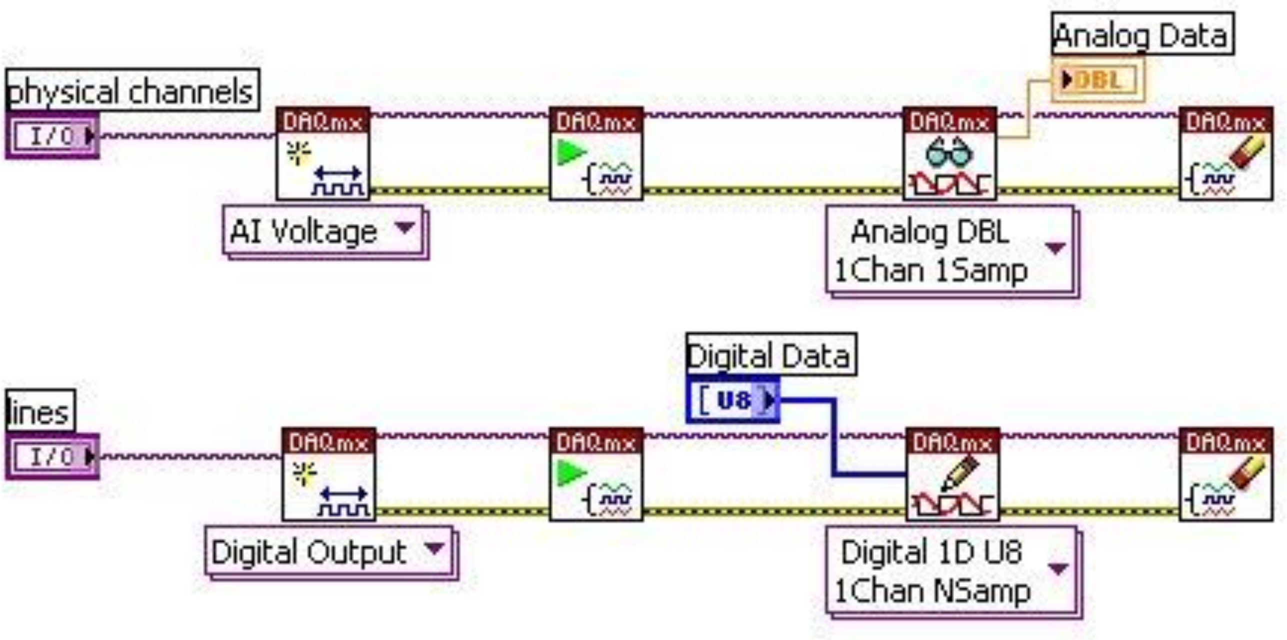


How LabVIEW Implements Multithreading

- **Parallel code paths on a block diagram can execute in unique threads**
- LabVIEW automatically divides each application into multiple execution threads (originally introduced in 1998 with LabVIEW 5.0)

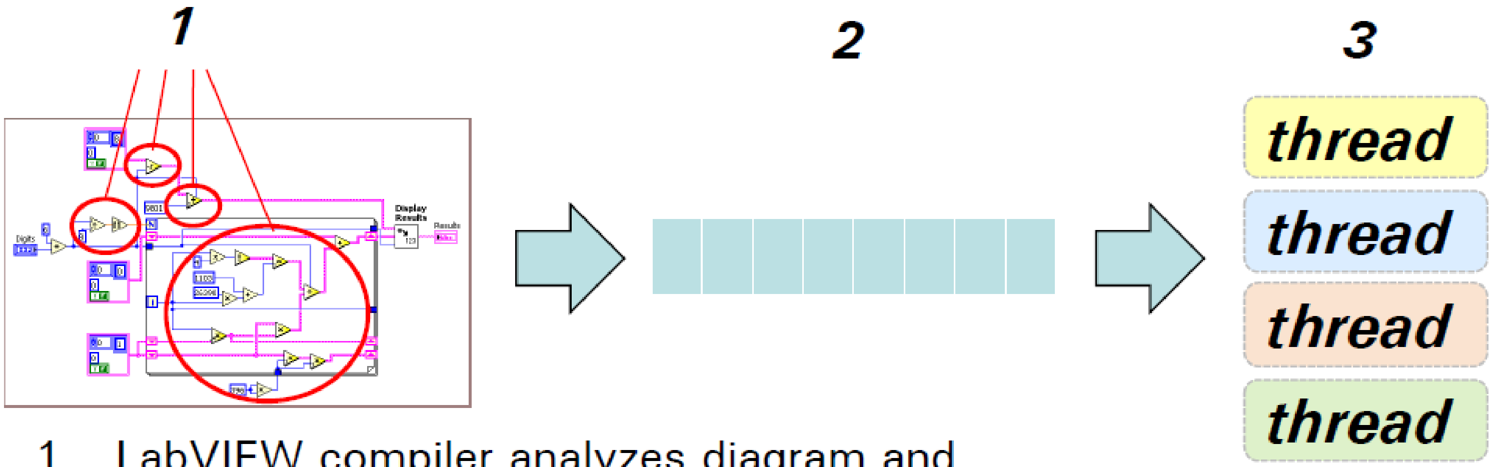


LabVIEW Example – Two separat threads



Two separate tasks that are not dependent on one another for data will run in parallel (two threads) without the need for any additional program code.

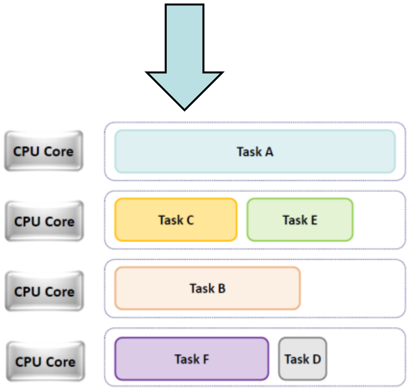
How LabVIEW Implements Multithreading



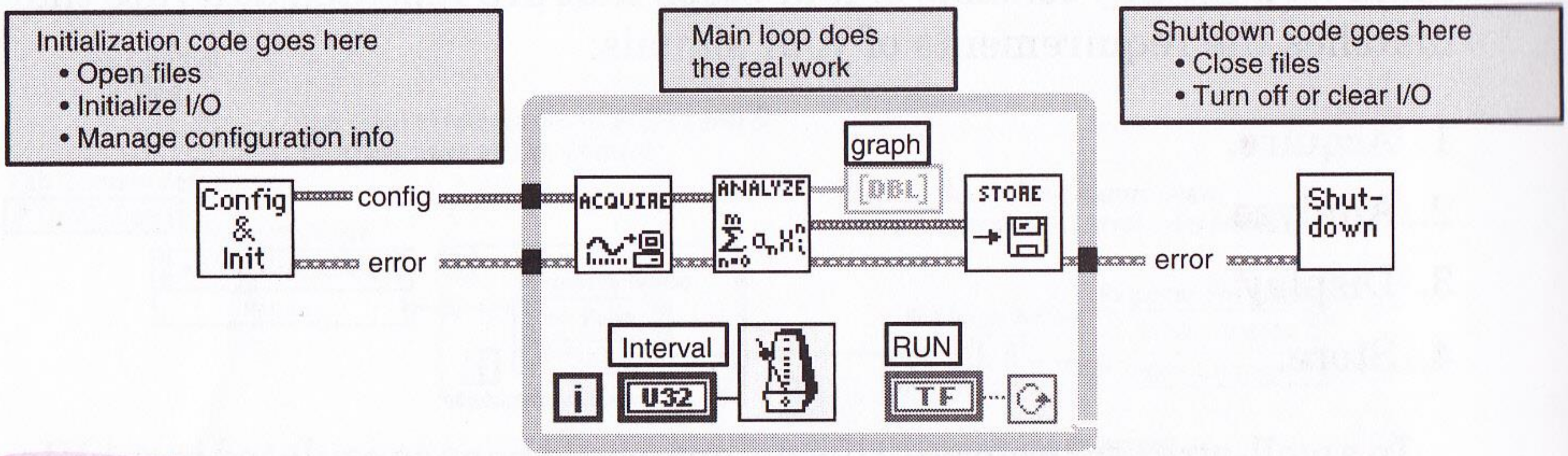
1. LabVIEW compiler analyzes diagram and assigns code pieces to “clumps”
2. Information about which pieces of code can run together are stored in a run queue
3. If block diagram contains enough parallelism, it will simultaneously execute in all system threads

…

# of threads scales based
on # of CPUs



Basic LabVIEW code architecture



Initialization code goes here
- Open files
- Initialize I/O
- Manage configuration info

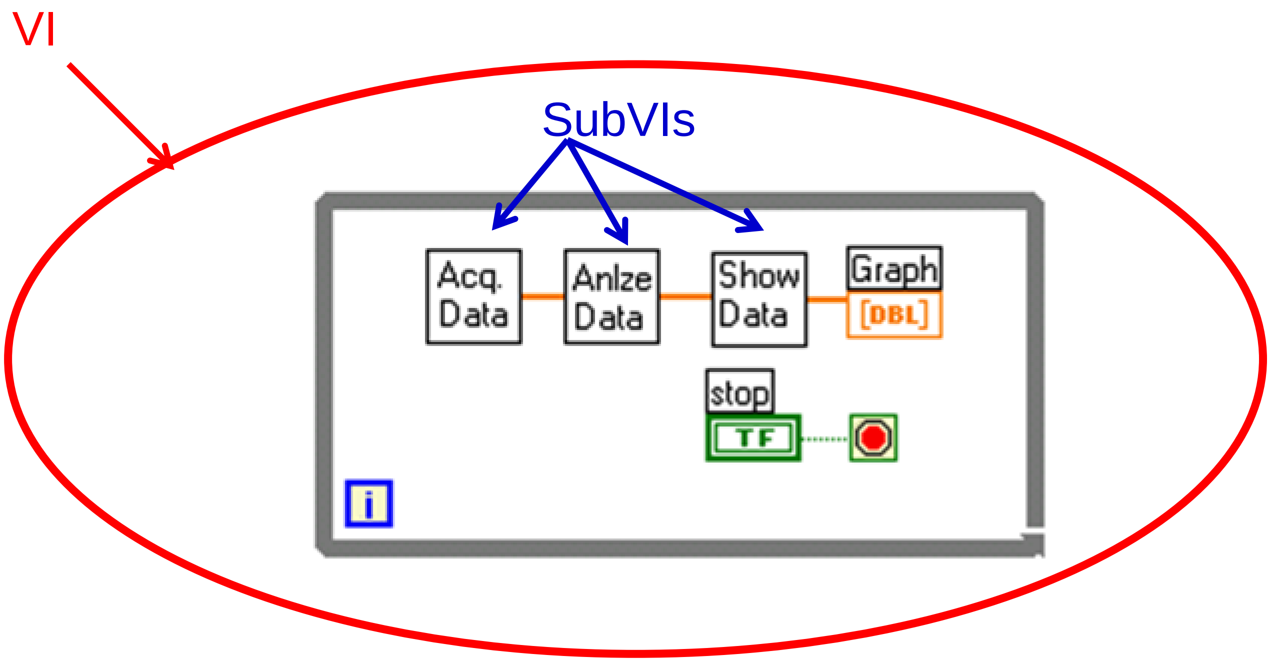
Main loop does the real work

Shutdown code goes here
- Close files
- Turn off or clear I/O

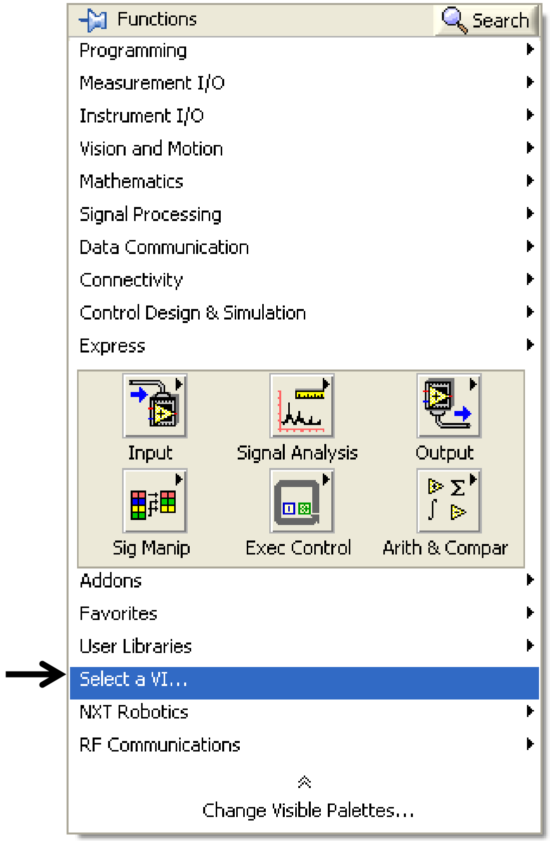
Figure 3.25 Dataflow instead of Sequence structure enhances readability. It has the same functionality as Figure 3.24. Note that the connections between tasks (subVIs) are not optional: they force the order of execution.

SubVIs

- A LabVIEW program is called a Virtual Instrument (VI)
- A subVI is a VI used in a block diagram of another VI
- SubVIs makes the code more readable, scalable, and maintainable
- How to create sub VIs:
  - Create an Icon with I/O connections for the VI

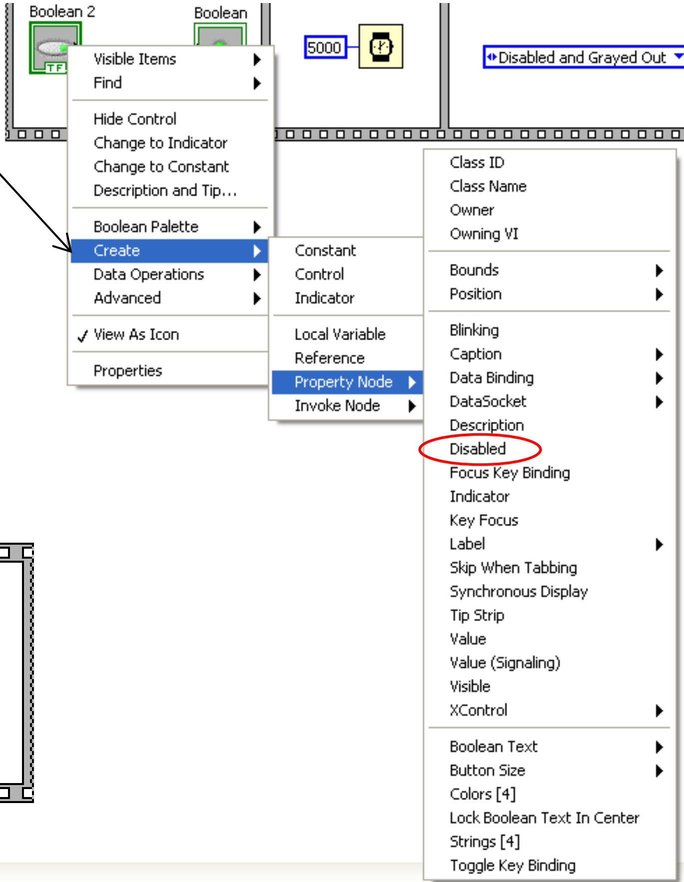


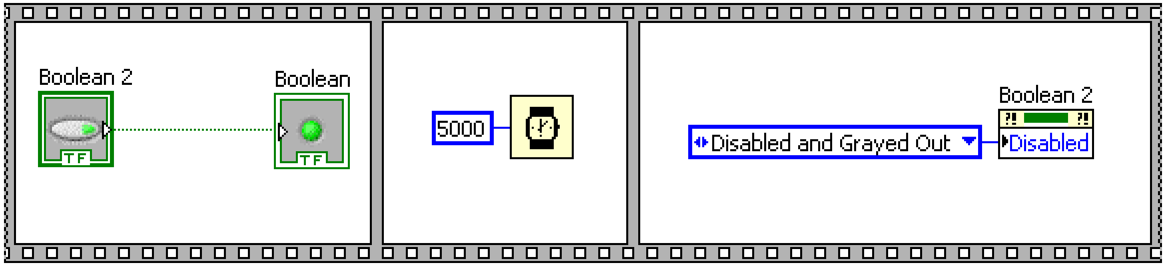
Add SubVI:



Property Nodes

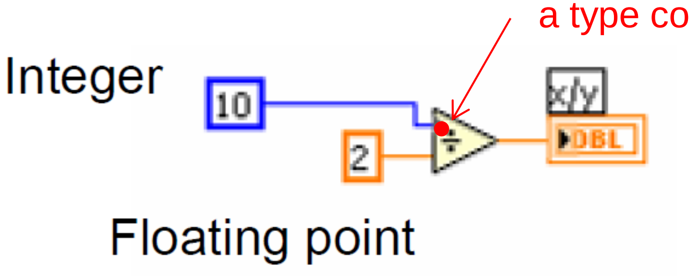
- Used to manipulate the appearance and behavior of the user interface (Front Panel controls and indicators)
- Limit the number of property nodes due to performance degradation
- Right click the control/indicator to <u>create</u>





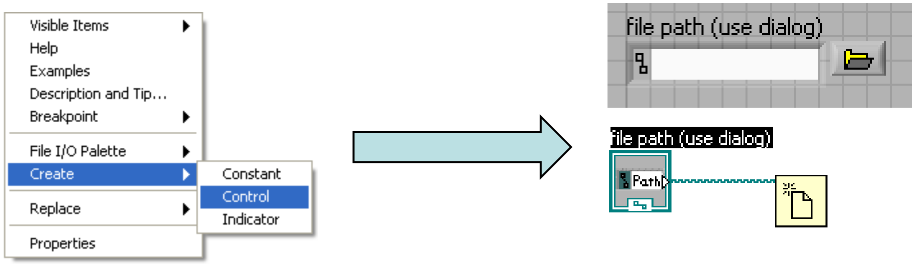
Type conversion

- LabVIEW will convert data types as it sees appropriate



Coercion dot (means that a type conversion is done)

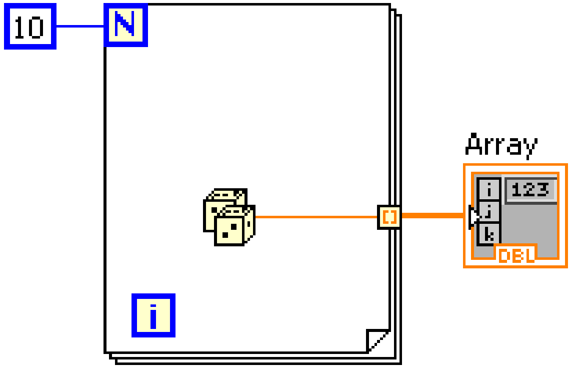
- Avoiding coercions (represented by a red dot) can speed up the application
- To select/change representation, right click the numeric on the block diagram and select **Representation**
- Right clicking the I/O of a block diagram icon and select create will create the proper data type



Arrays

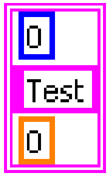
- Can be multidimensional.
- Must have the same data type for each element

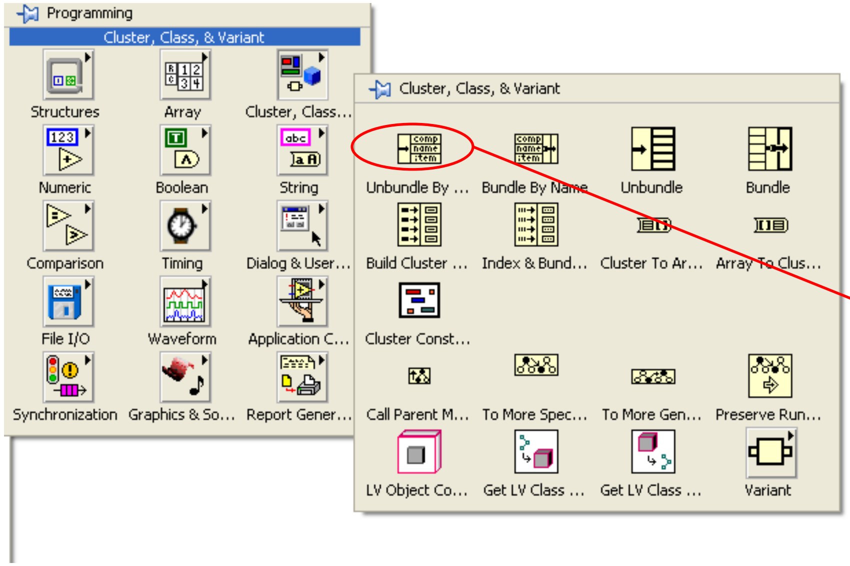
Simple method to create an array:

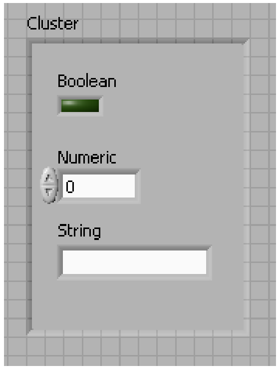


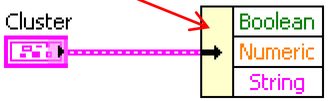
Clusters

- Used to <u>group related data</u>
  - Reduce the number of terminals (I/O) required on a SubVI
  - Minimize the number of wires on the diagram
- The elements can be of different data types
- Can not contain a mixture of controls and indicators

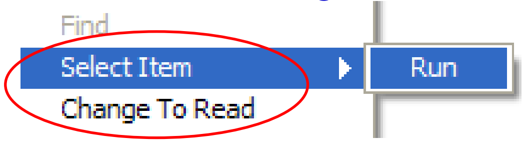




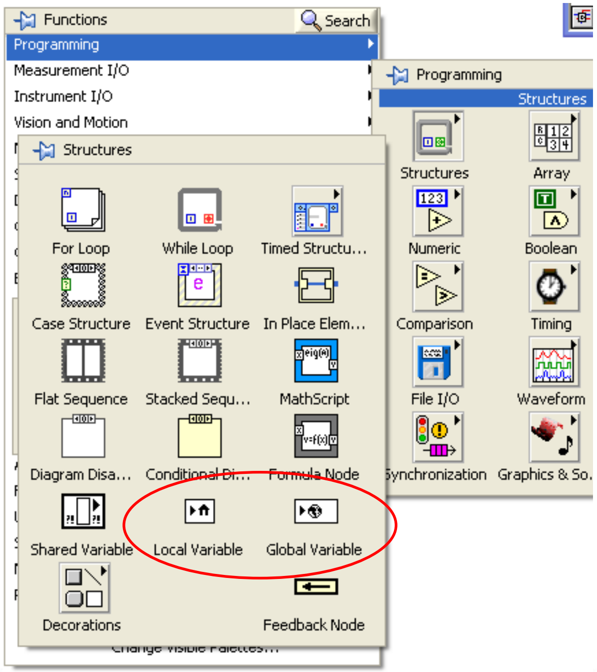




Local variables: Right-click:



Local & Global variables



- **Minimize the use** (especially global variables)
  - **Use wires when possible**
  - **Each local variable creates a copy of the data!**
- Global variables can create race conditions!
  - When two or more events can occur in any order, but they need to occur in a particular order
  - The data dependency (dataflow) in LabVIEW generally prevents race conditions, but global variables provides a way to violate the strict dataflow
  - Use global variables only when no other good options!

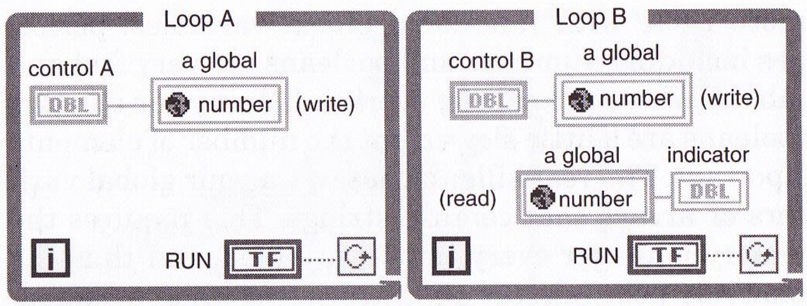
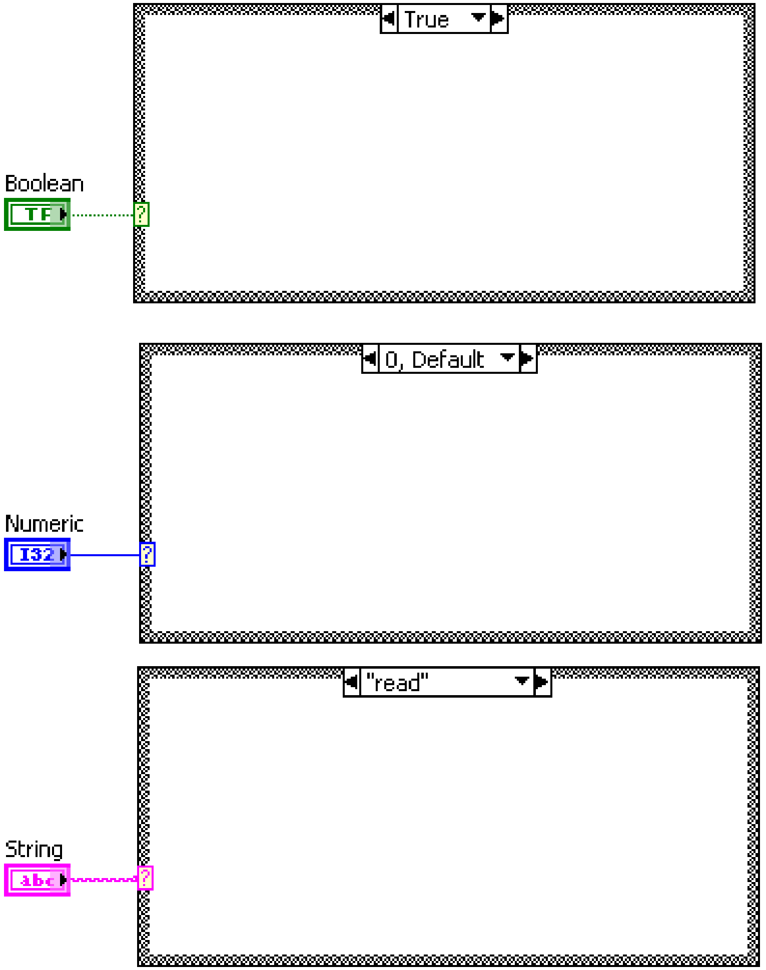
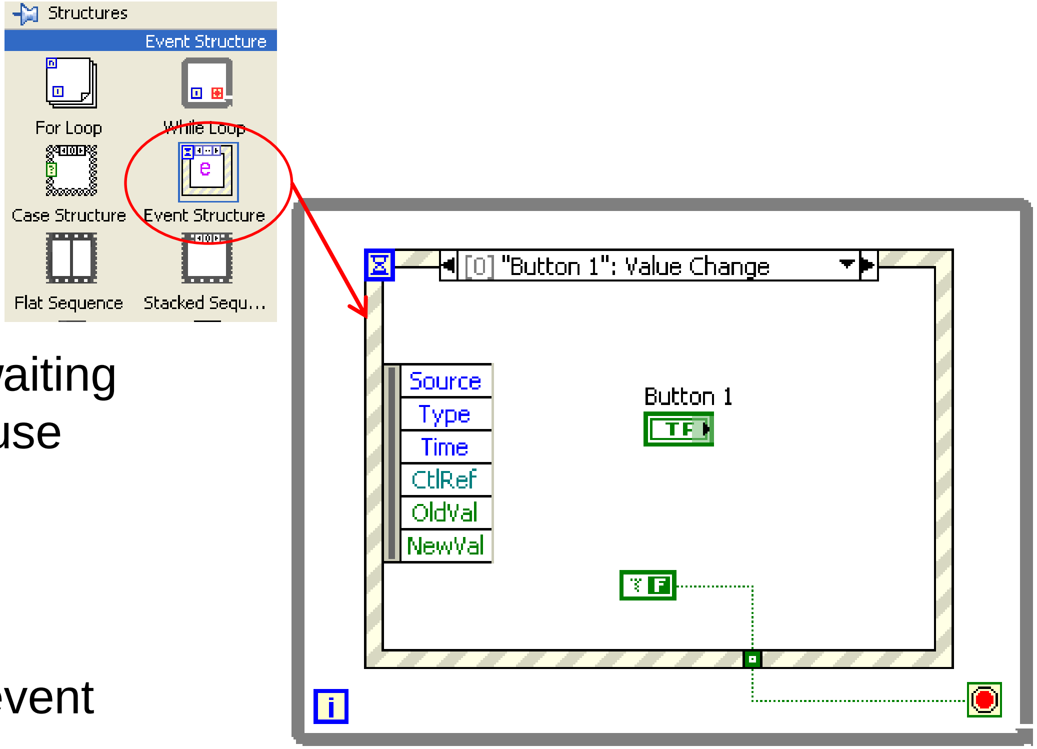


Figure 3.13 Race conditions are a hazard associated with all global variables. The global gets written in two places. Which value will it contain when it's read?

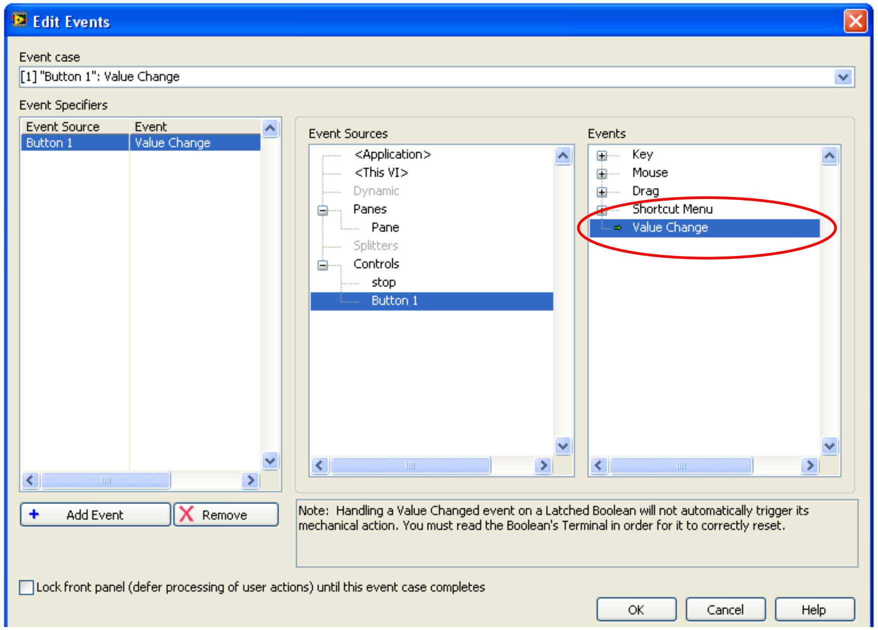
Case structure



Event Structure



- To <u>limit the CPU usage </u>while waiting for **user interface events** (mouse clicks, key pressed etc.)
  - **Avoids polling!**
- Detects all events!
- Do minimal processing inside event structures!
- How it works:
  - Operating system broadcasts system events (mouse click, keyboard, etc.) to applications
  - Registered <u>events are captured by event structure and executes appropriate case</u>
  - Event structure enqueues events that occur while it's busy



Edit Events

Event case
[1] "Button 1": Value Change

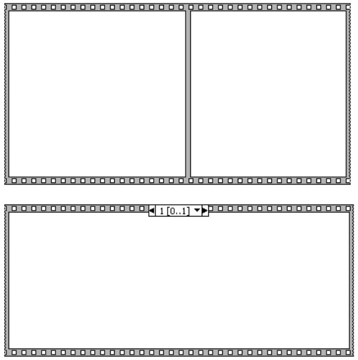
Event Specifiers

Event Source	Event
Button 1	Value Change

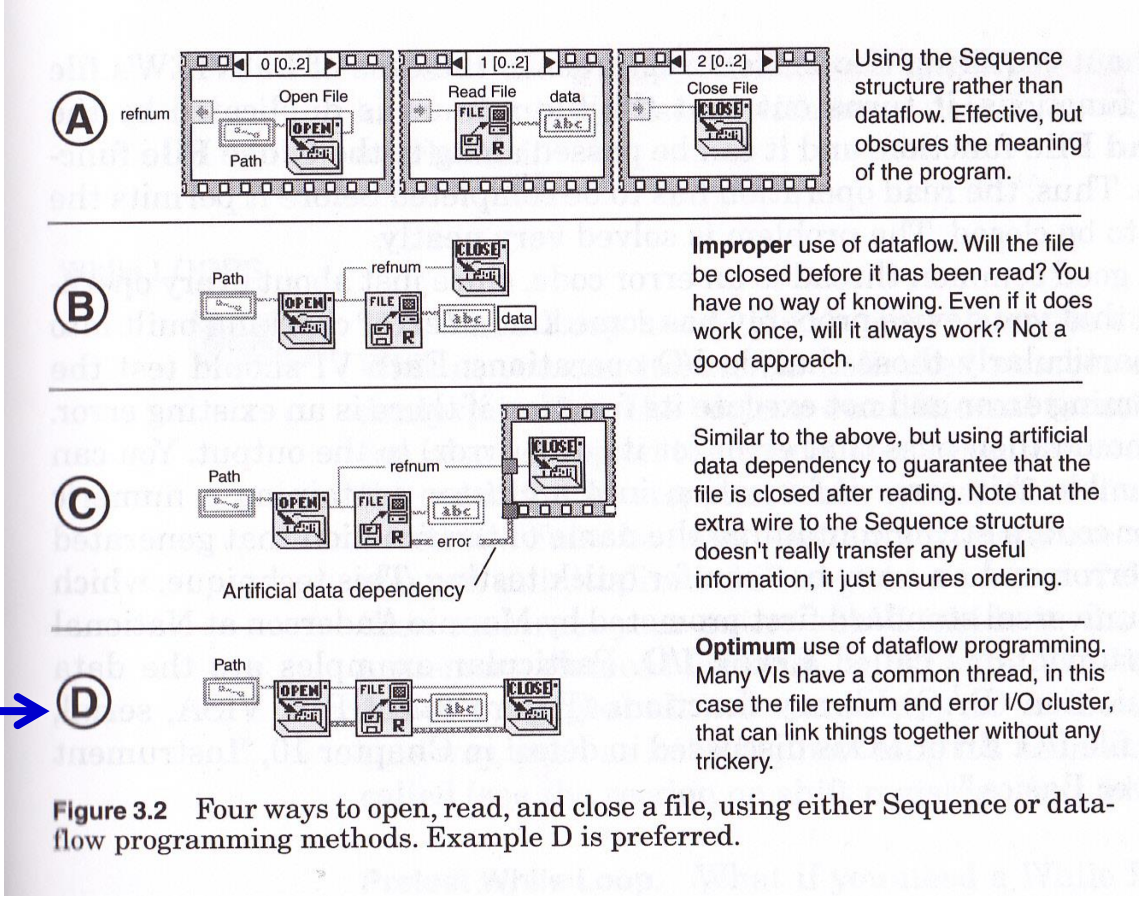
Note: Handling a Value Changed event on a Latched Boolean will not automatically trigger its mechanical action. You must read the Boolean's Terminal in order for it to correctly reset.

☐ Lock front panel (defer processing of user actions) until this event case completes

Sequence structure



- Can be used to enforce the order of execution
- Use dataflow programming (data input dependence) to control the dataflow!



Using the Sequence structure rather than dataflow. Effective, but obscures the meaning of the program.

Improper use of dataflow. Will the file be closed before it has been read? You have no way of knowing. Even if it does work once, will it always work? Not a good approach.

Similar to the above, but using artificial data dependency to guarantee that the file is closed after reading. Note that the extra wire to the Sequence structure doesn't really transfer any useful information, it just ensures ordering.

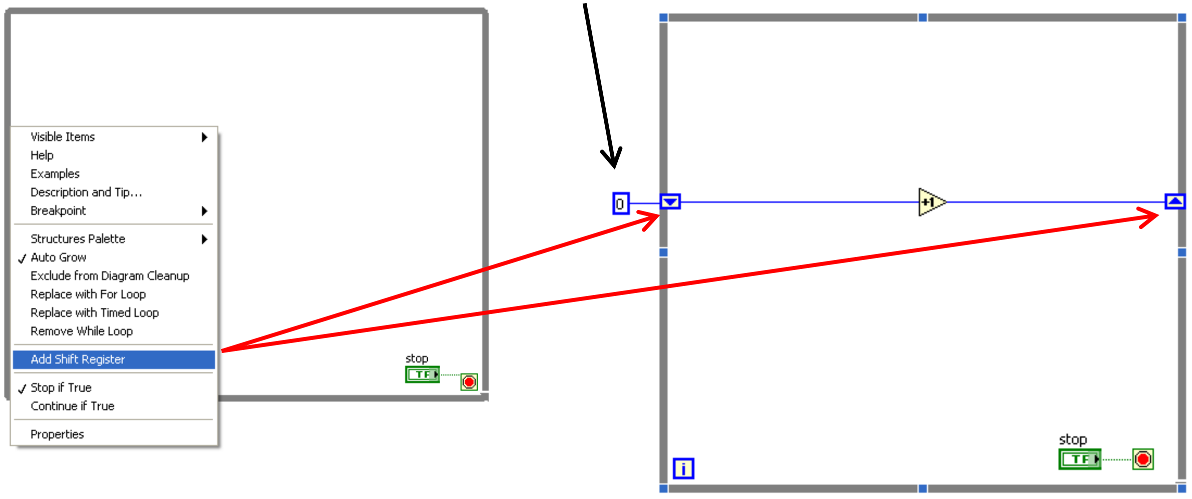
Optimum use of dataflow programming. Many VIs have a common thread, in this case the file refnum and error I/O cluster, that can link things together without any trickery.

Figure 3.2 Four ways to open, read, and close a file, using either Sequence or dataflow programming methods. Example D is preferred.

Program like this!

Shift registers

- Memory elements available in For Loops and While Loops
- Transfer values from completion of one loop iteration to the beginning of the next
- Initialize the shift registers (unless you want to create a Functional Global)

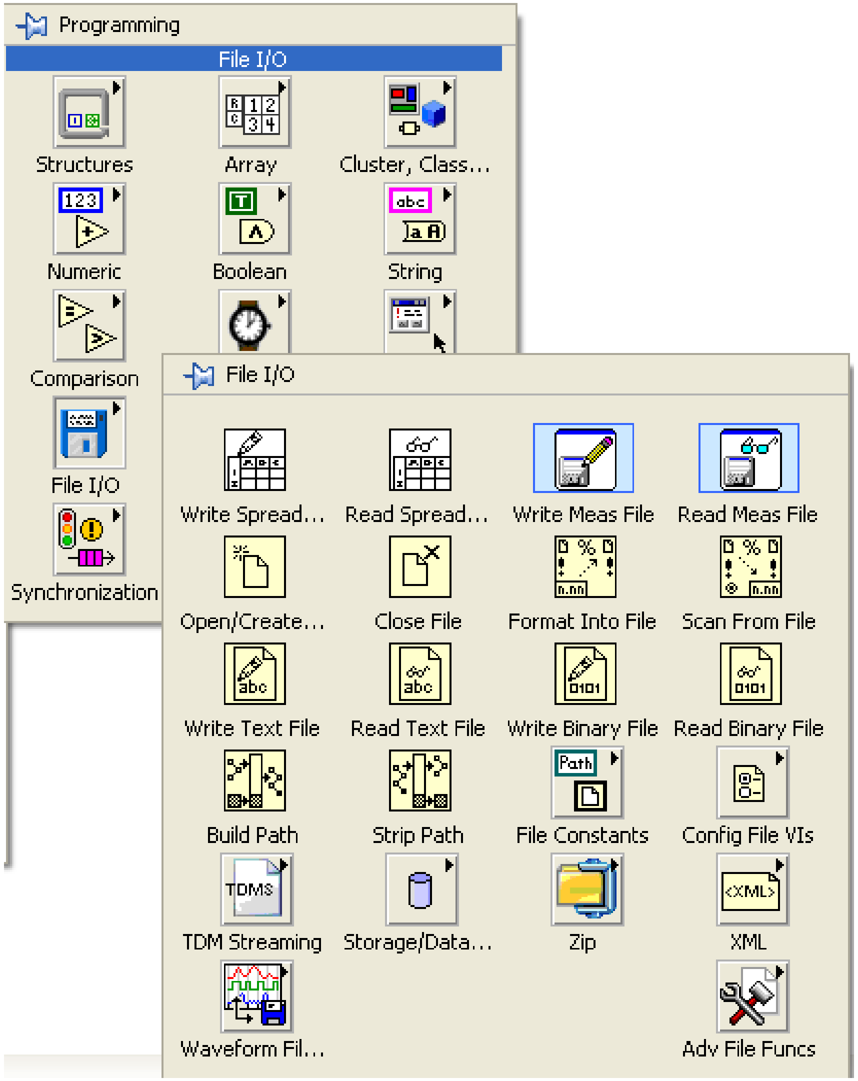


Right-click the loop:

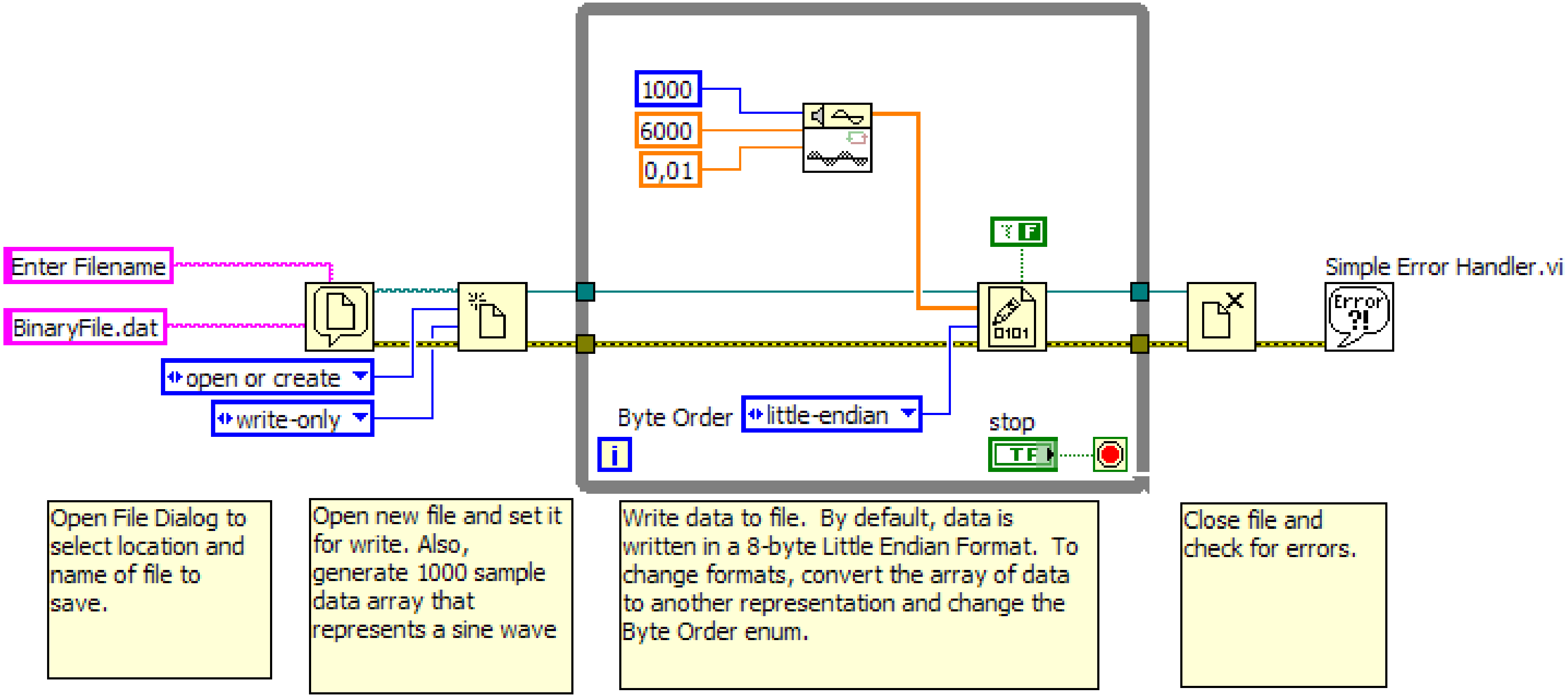
Initialization!

File I/O

- File Types supported in LabVIEW
  - ASCII
  - Binary
  - TDMS
  - Config File
  - Spreadsheet
  - AVI
  - XML



File I/O – Write binary file example



Open File Dialog to select location and name of file to save.

Open new file and set it for write. Also, generate 1000 sample data array that represents a sine wave

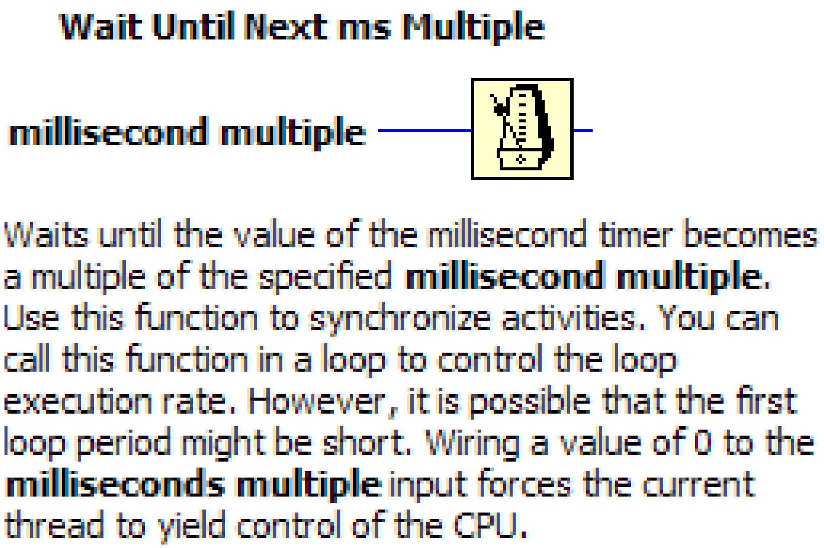
Write data to file. By default, data is written in a 8-byte Little Endian Format. To change formats, convert the array of data to another representation and change the Byte Order enum.

Close file and check for errors.

Software Timing I

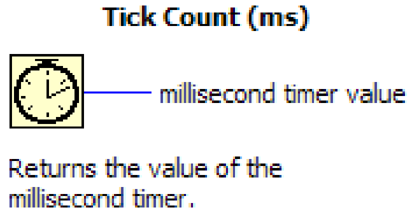
- The functions **Wait Until Next ms Multiple**, **Wait (ms)** and **Tick Count (ms)** attempts to resolve <u>milliseconds</u> on a PC within the limitations of the operating system (such as Windows) and the computer clock.

- If you need better resolution or accuracy (determinism) you have to use another hardware solution.



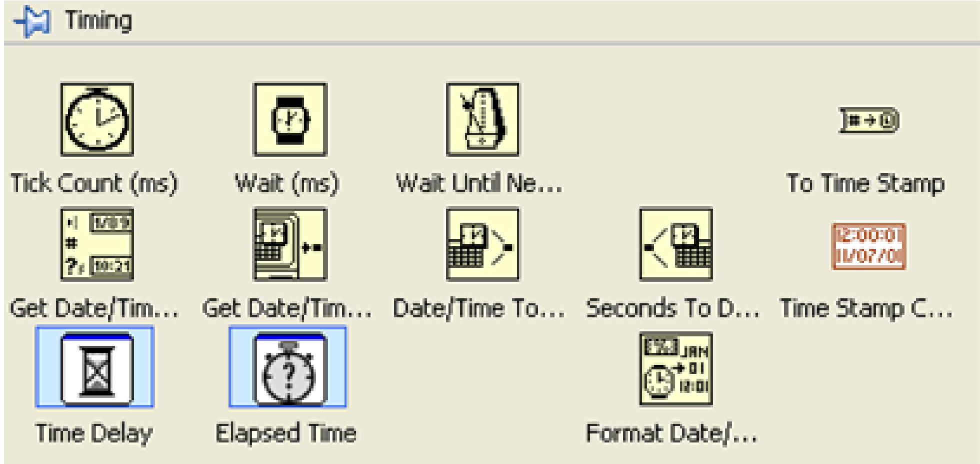
Wait Until Next ms Multiple

Waits until the value of the millisecond timer becomes a multiple of the specified **millisecond multiple.** Use this function to synchronize activities. You can call this function in a loop to control the loop execution rate. However, it is possible that the first loop period might be short. Wiring a value of 0 to the **milliseconds multiple** input forces the current thread to yield control of the CPU.



Tick Count (ms)

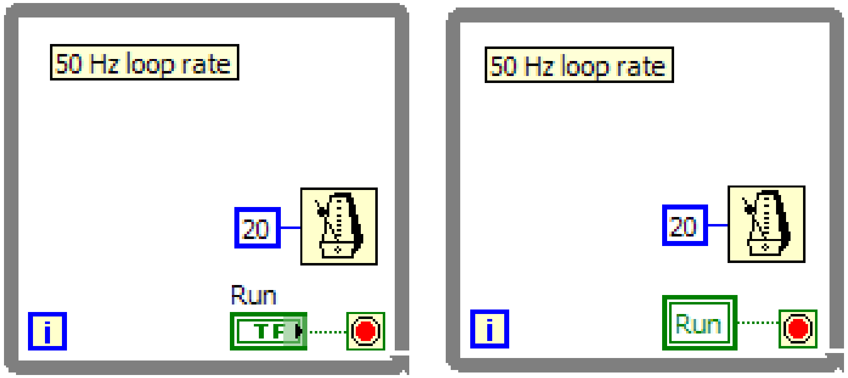
Returns the value of the millisecond timer.



Software Timing II

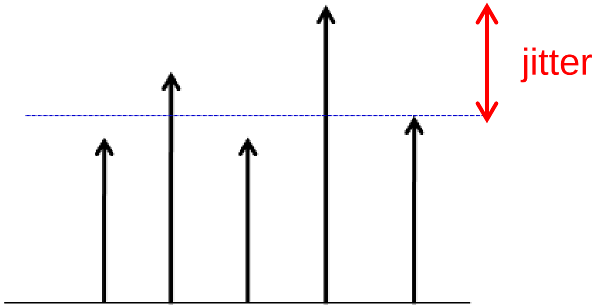
- To make a while loop run at nice regular intervals add the **Wait Until Next ms Multiple**
  - always use the **Wait Until Next ms Multiple** (or another timer) in a loop to avoid using unnecessary CPU power.
  - without any “wait” a while loop will run as fast as possible ...
- Two loops can be software synchronized using the **Wait Until Next ms Multiple** in both loops.

- To prioritize execution of different parallel loops use Wait functions to slow down lower priority loops in the application.



Software Timing III

- If you use software timer functions to control a loop, then you can expect differences in the time interval between each iteration (jitter) of the loop, depending on what other processes are running on the computer at that instant.
  - If you have several windows open at the same time and you are switching between different windows during your data acquisition, then you can expect a lot of overhead on the Central Processing Unit (CPU), which might slow down the loop that is performing the data acquisition.
  - In DAQ applications you should use hardware timing instead of software timing, if possible.



Error handling example

- Propagate the **error cluster** through every SubVI

Report errors using dialog/and or log files

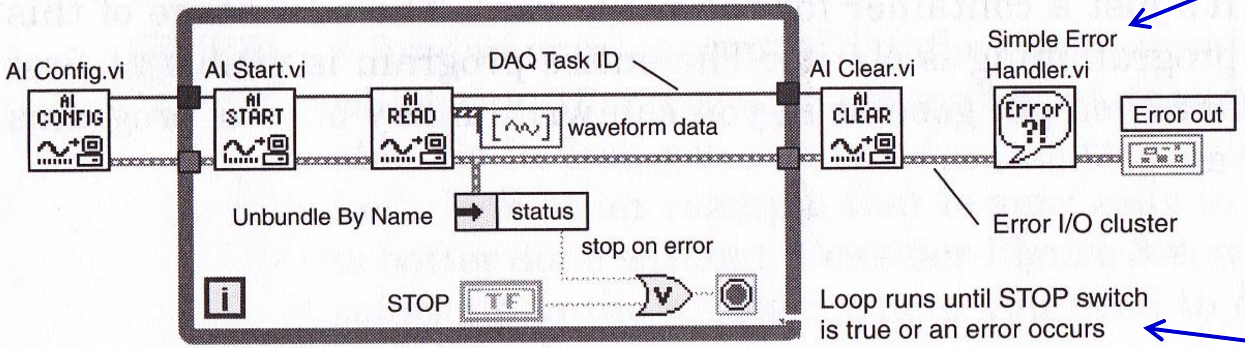
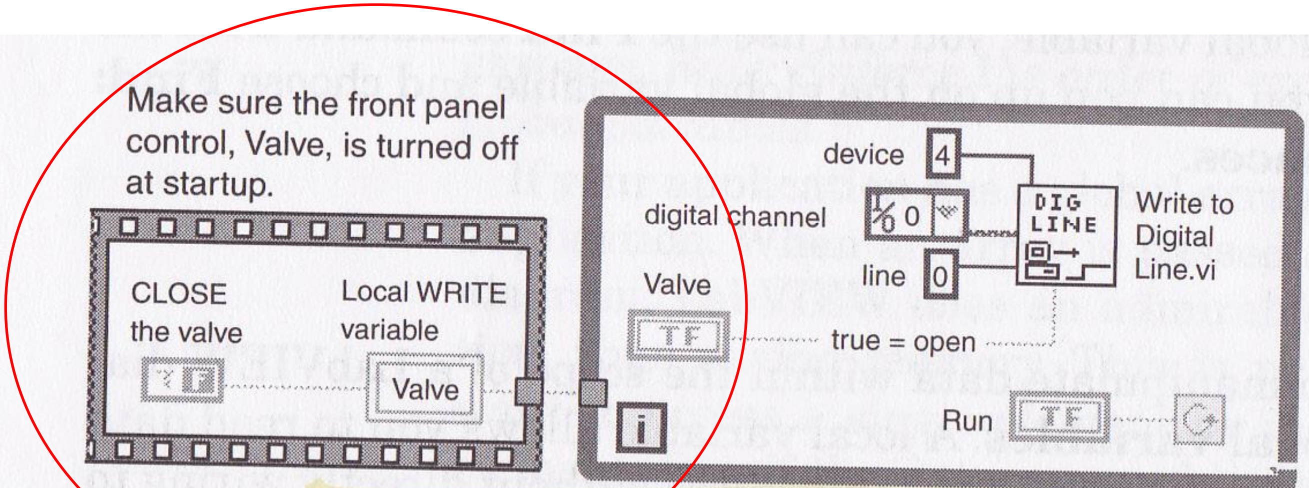


Figure 3.3 A cluster containing error information is passed through all the important subVIs in this program that use the DAQ library. The While Loop stops if an error is detected, and the user ultimately sees the source of the error displayed by the Simple Error Handler VI. Note the clean appearance of this style of programming.

Loop terminates if an error occur

Loop initialization

- Important to preset the controls to a correct initial value at startup of the program
- A sequence structure can be used, see illustration below
  - If a state machine is implemented initialization states are used.



Make sure the front panel control, Valve, is turned off at startup.

Figure 3.14 Local variables are a convenient way to initialize front panel controls. The Sequence structure guarantees that the initialization is completed before the While Loop starts.

Parallel loops – simplest case

- Sometimes no data need to be exchanged between loops (independent loops)
  - e.g. different sensor signals to be logged at two different rates

- Parallel loops can be stopped using local variables

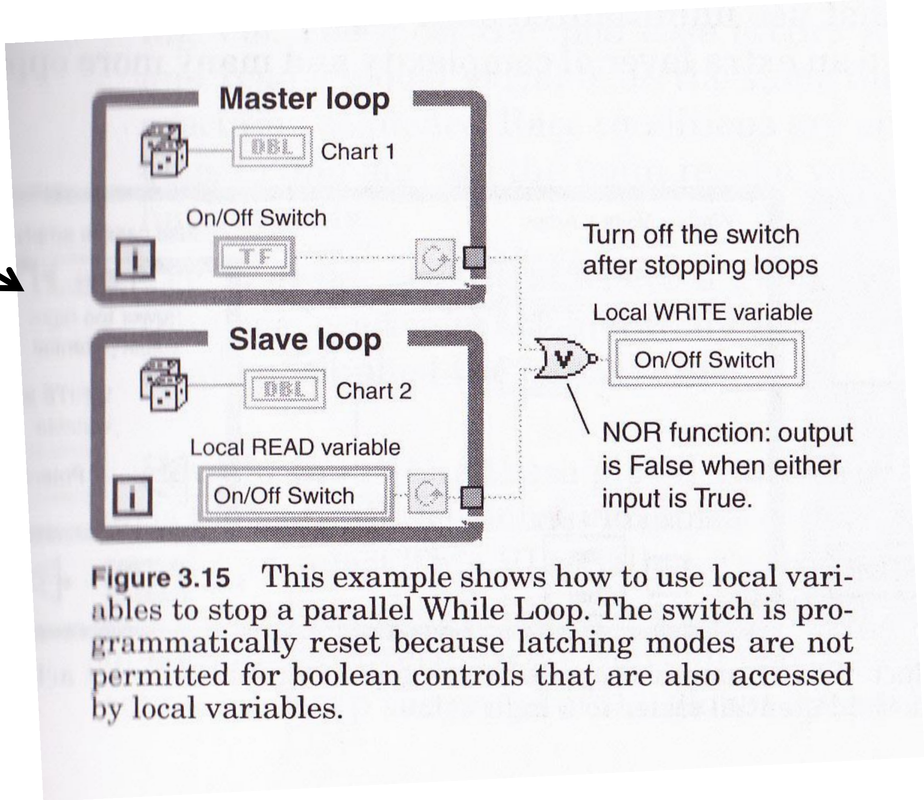
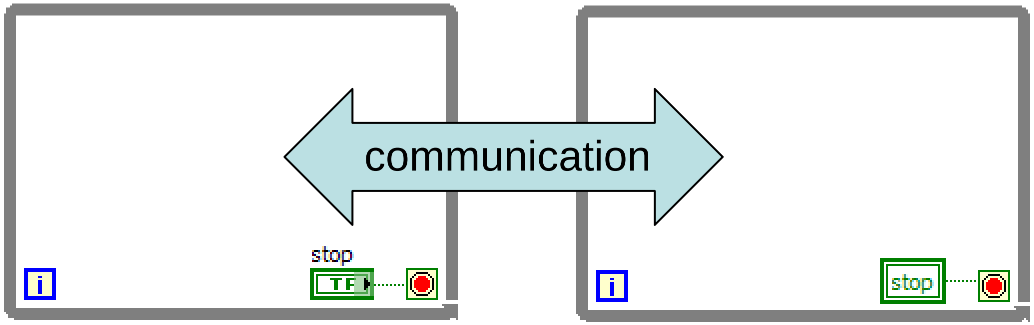


Figure 3.15 This example shows how to use local variables to stop a parallel While Loop. The switch is programmatically reset because latching modes are not permitted for boolean controls that are also accessed by local variables.

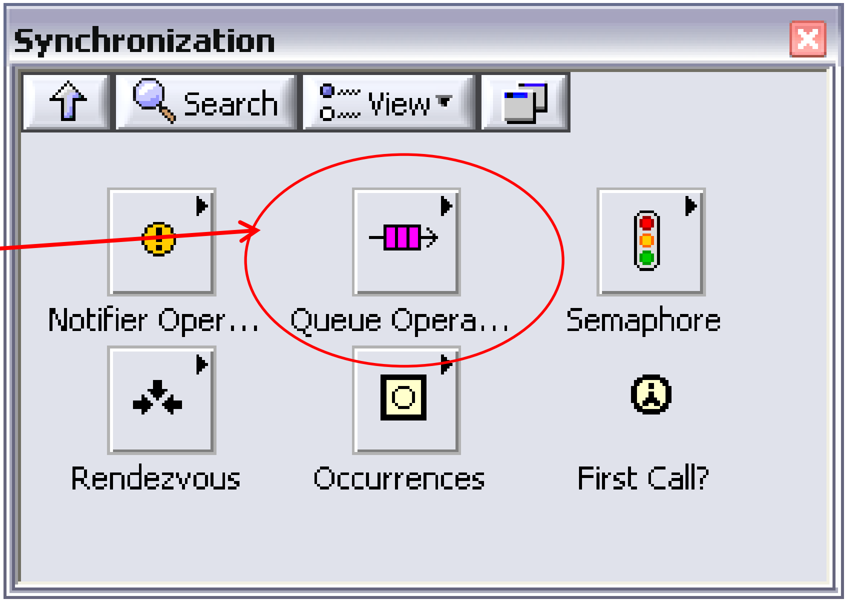
Design Patterns for loop communication

- Master-Slave pattern
- Client – Server pattern
- Producer / Consumer pattern


communication

Loop Communication mechanisms

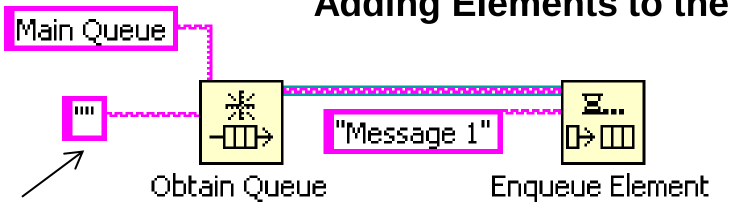
- Variables
- Occurrences
- Notifier
- **Queues**
- Semaphores
- Rendezvous



Queues

- Used for synchronization and data transfer between loops
- Data are stored in a FIFO buffer, and the useful queue depth is limited (only) by the computer's RAM
  - **No data are lost**
- A read (dequeue) from the queue is destructive
  - Data can only be read by one consumer loop (without a new enqueue)
- Different queues must have unique names!

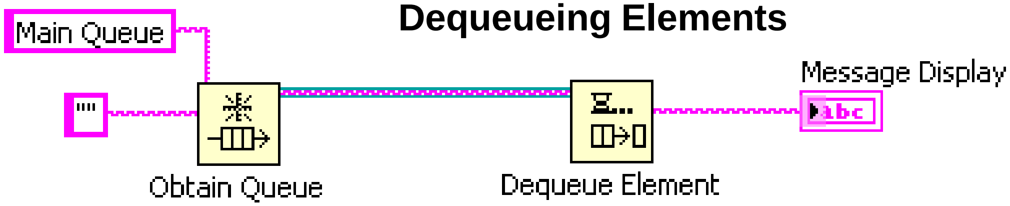
Adding Elements to the Queue



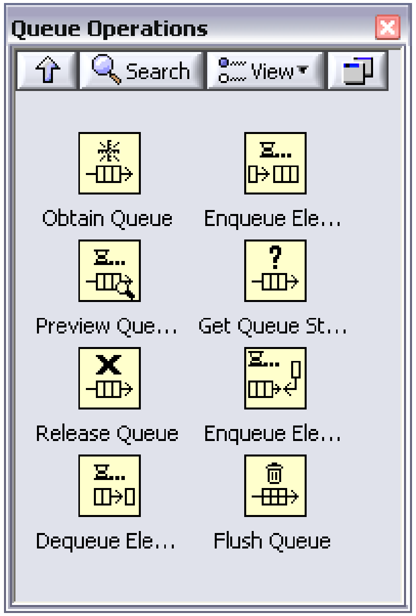
Select the data type the queue will hold

Reference to existing queue in memory

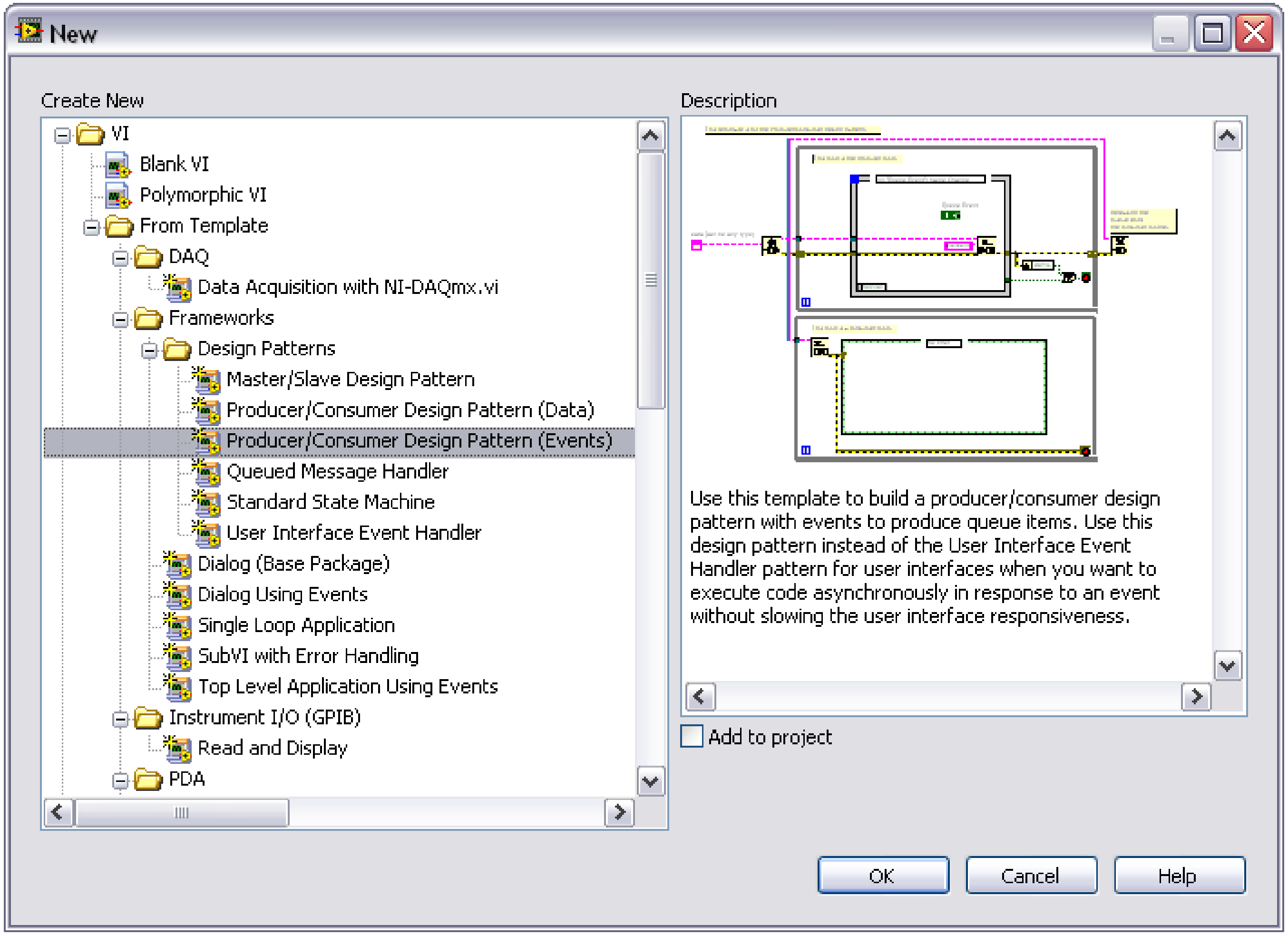
Dequeueing Elements



Dequeue will wait for data or time-out



Design patterns (templates)

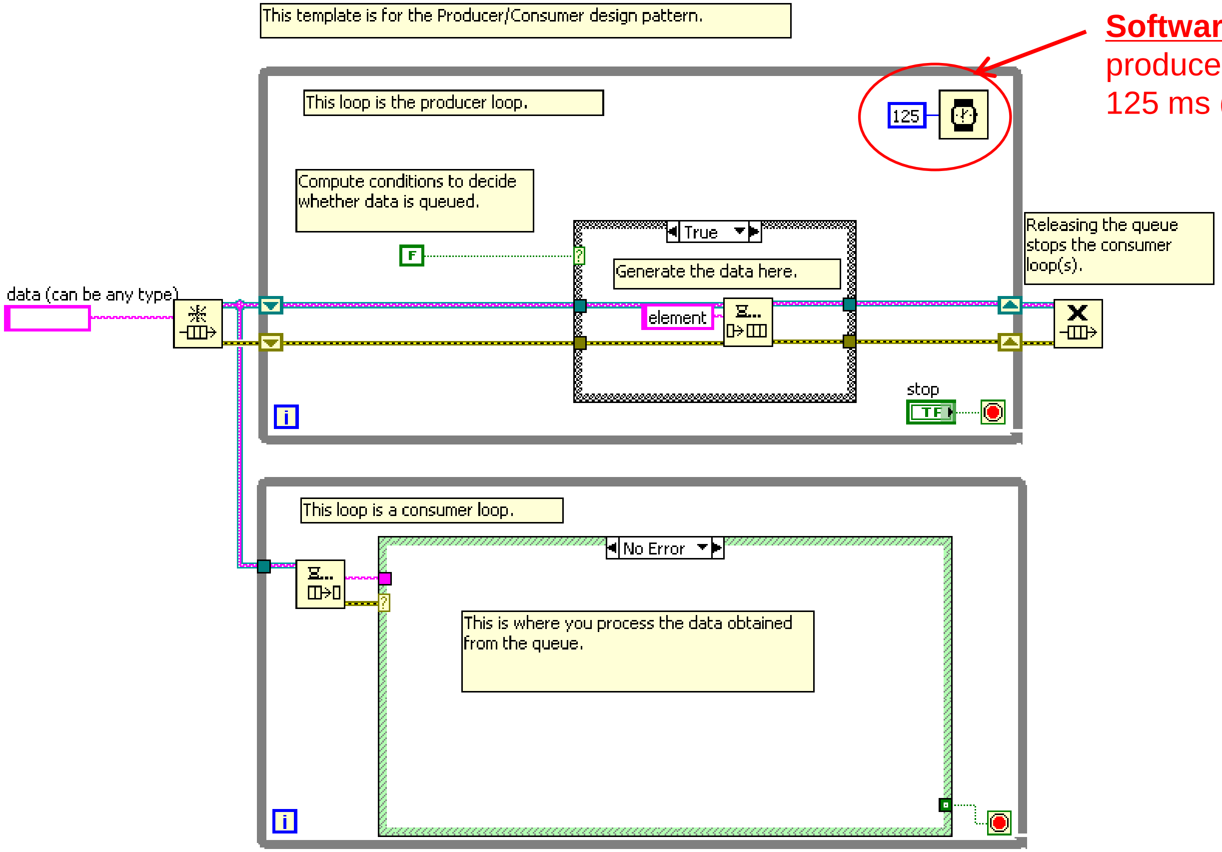


Use this template to build a producer/consumer design pattern with events to produce queue items. Use this design pattern instead of the User Interface Event Handler pattern for user interfaces when you want to execute code asynchronously in response to an event without slowing the user interface responsiveness.

http://zone.ni.com/devzone/cda/tut/p/id/7605

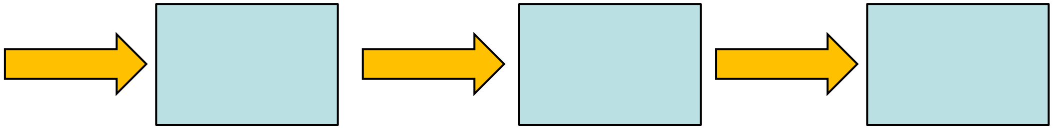
Producer – consumer

- **Queues** are used for loop communications in multi-loop programs, to execute code in parallel and at different rates
- The queues buffer data in a FIFO structure in PC RAM



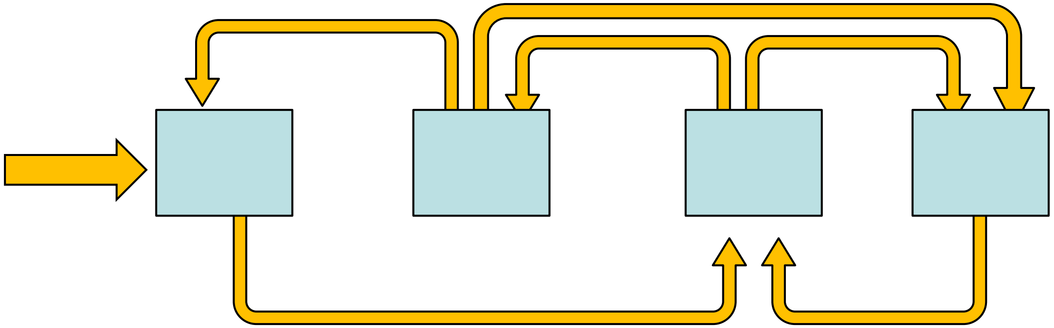
Software timing - the producer loop will run every 125 ms (8 Hz)

State machines - background



Static Sequence

Known order of execution

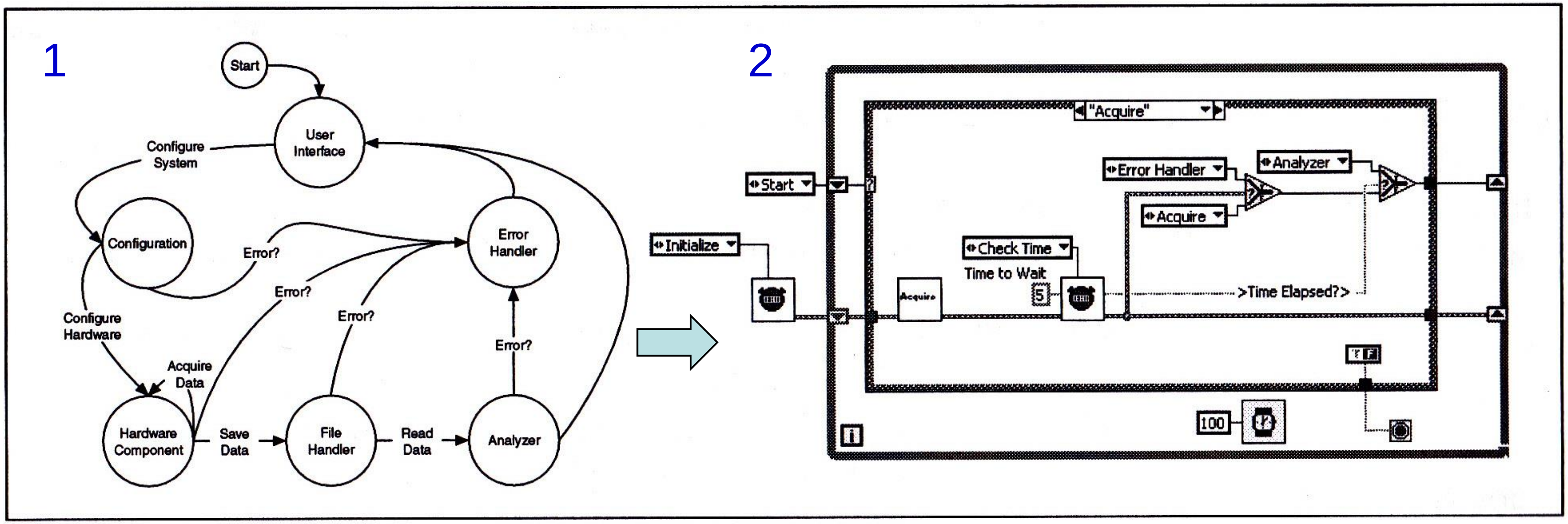


Dynamic Sequence

Distinct states can operate in a programmatically determined sequence

State machine design

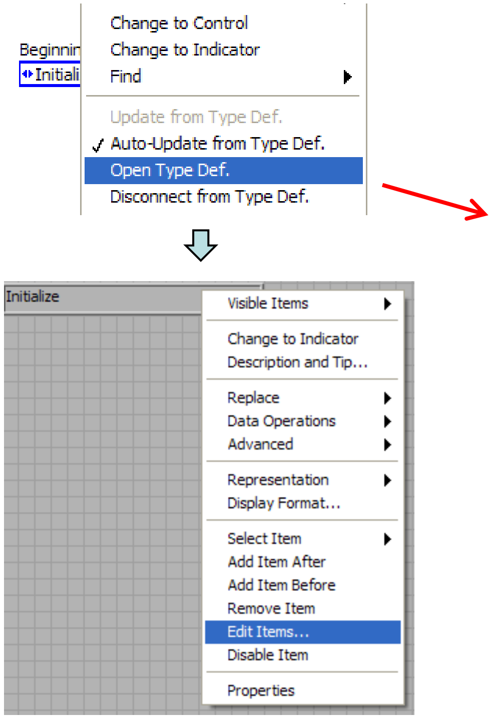
1. Draw the **state diagram**
2. Translate the state diagram into **LabVIEW code**

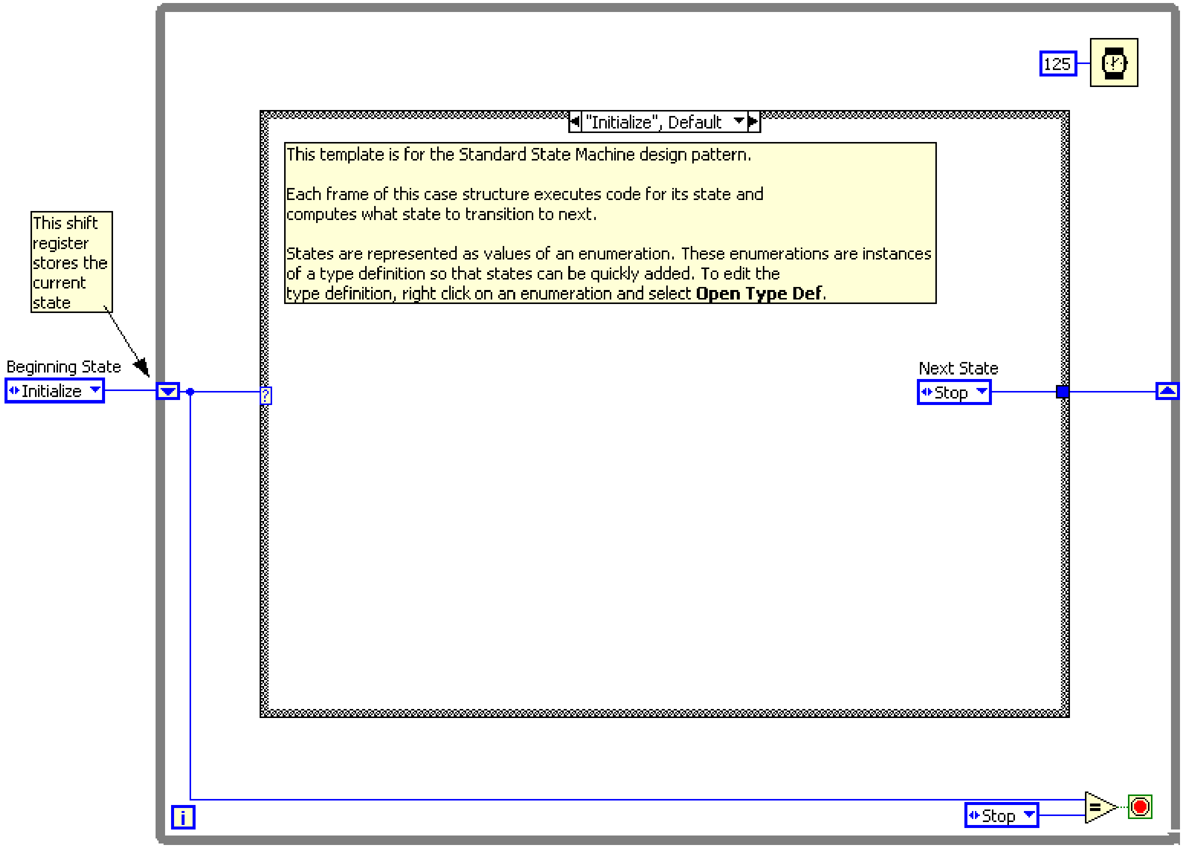


Standard State machines in LabVIEW

- Case structure inside of a While loop
- Each case is a state
- Current state has decision-making code that determines next state
- Use enumerated constants (called **typedefs)** to pass value of next state to shift registers

Edit/add/Remove states:





This shift register stores the current state

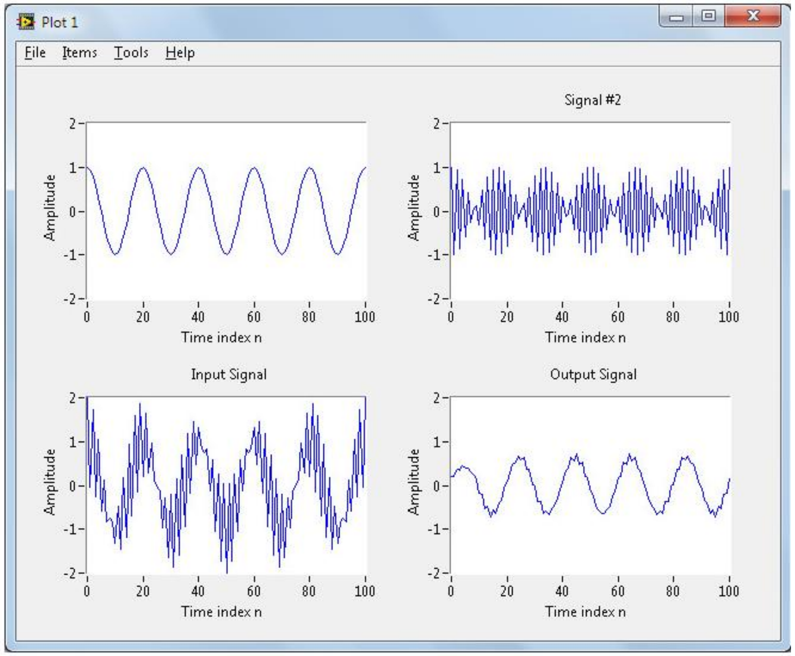
This template is for the Standard State Machine design pattern.

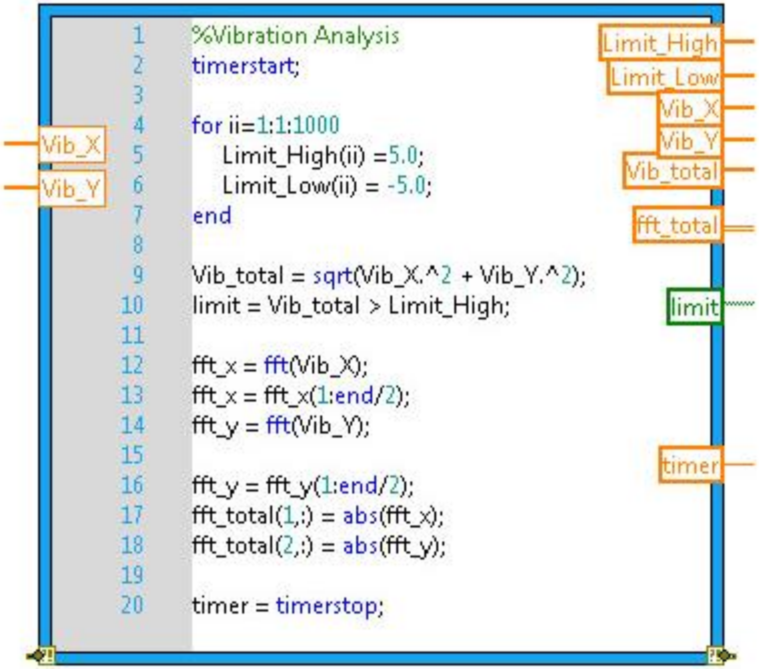
Each frame of this case structure executes code for its state and computes what state to transition to next.

States are represented as values of an enumeration. These enumerations are instances of a type definition so that states can be quickly added. To edit the type definition, right click on an enumeration and select **Open Type Def**.

MathScript

- Adds math-oriented, textual programming to the LabVIEW graphical development environment
- General compatibility with widely used **.m file script** syntax (not all Matlab functions are supported)
- Reuse Matlab .m files
- Very useful for algorithm development
  - – compact code for matrix operations etc
- Available also for RT-targets





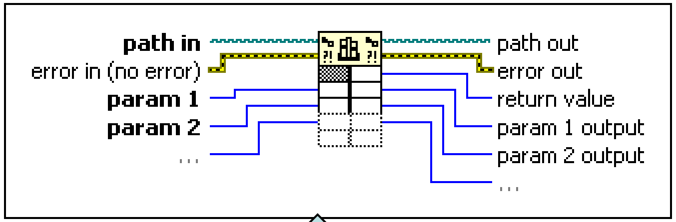
```
%Vibration Analysis
timerstart;

for ii=1:1:1000
    Limit_High(ii) =5.0;
    Limit_Low(ii) = -5.0;
end

Vib_total = sqrt(Vib_X.^2 + Vib_Y.^2);
limit = Vib_total > Limit_High;

fft_x = fft(Vib_X);
fft_x = fft_x(1:end/2);
fft_y = fft(Vib_Y);

fft_y = fft_y(1:end/2);
fft_total(1,:) = abs(fft_x);
fft_total(2,:) = abs(fft_y);

timer = timerstop;
```

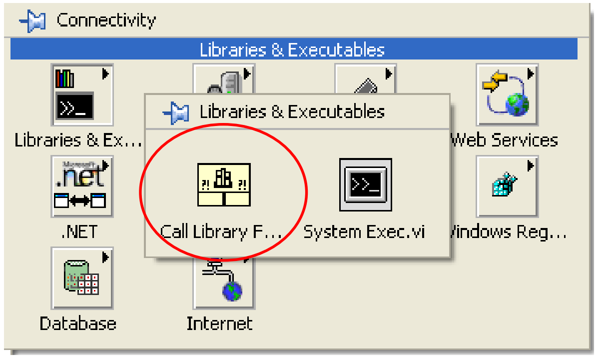
Figure 11. MathScript Node Places Your Custom .m File Code Inline with Graphical G Code

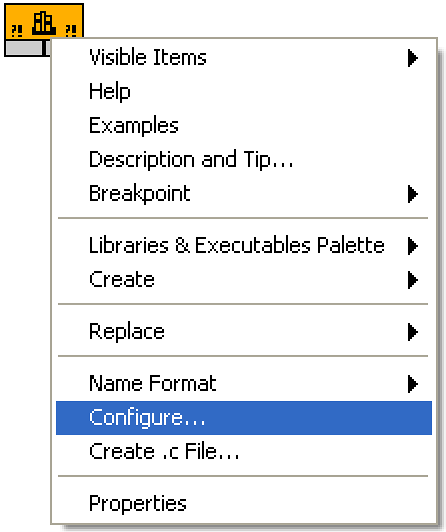
Extra

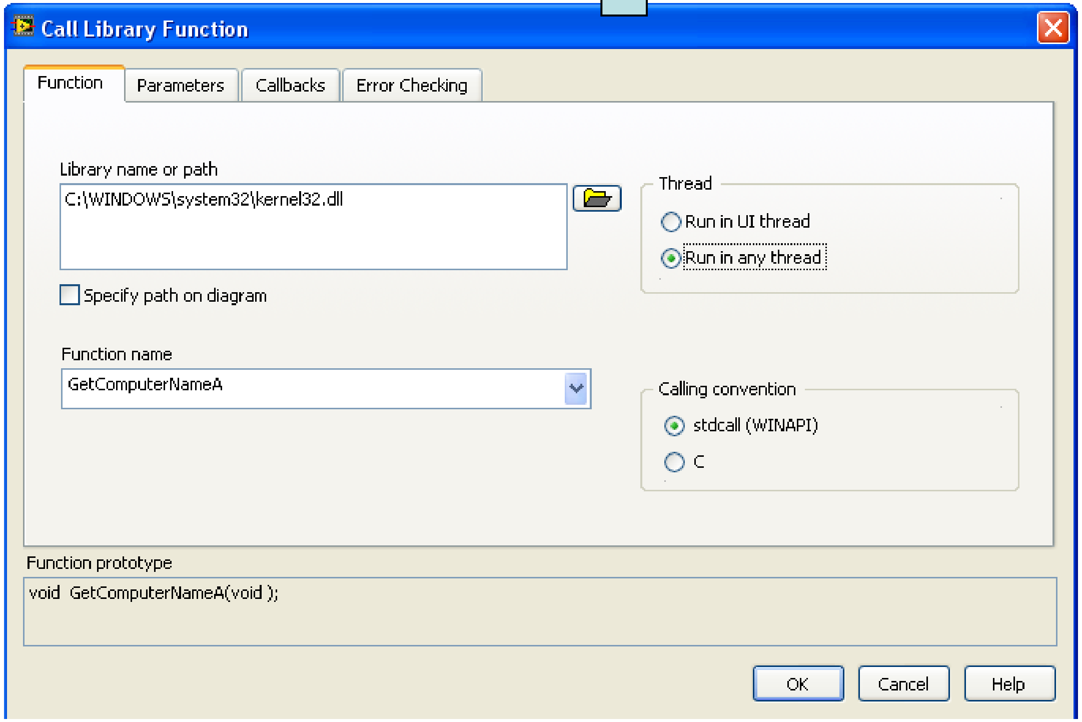
Connectivity – Using DLLs in LabVIEW

- Call Library Function
  - To use DLLs in LabVIEW
  - Almost any compiler can generate a DLL









Library name or path: C:\WINDOWS\system32\kernel32.dll

☐ Specify path on diagram

Function name: GetComputerNameA

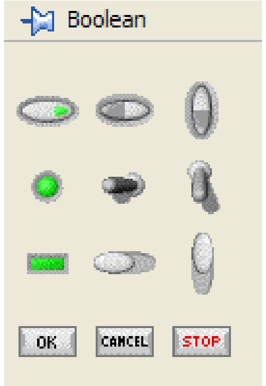
Thread:
○ Run in UI thread
● Run in any thread

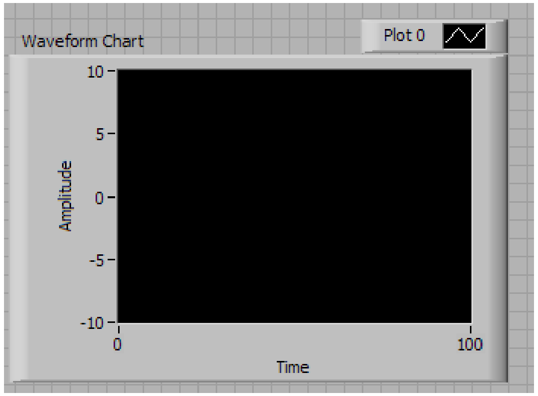
Calling convention:
● stdcall (WINAPI)
○ C

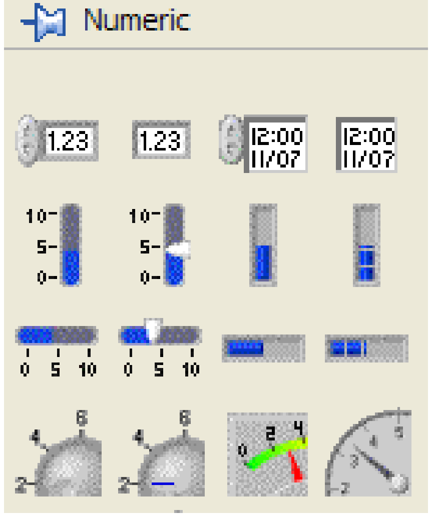
Function prototype: void GetComputerNameA(void);

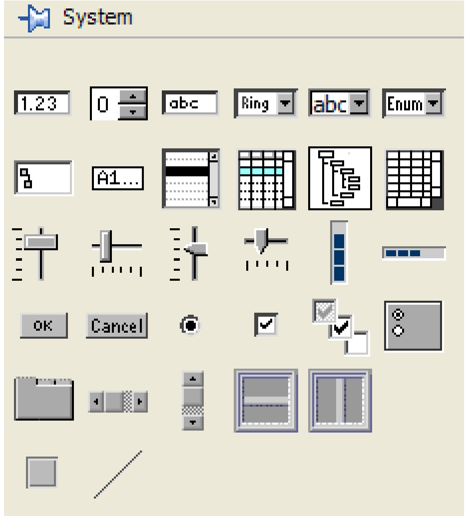
Visualization

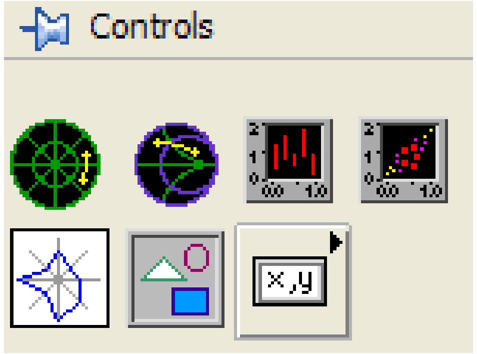
- Displaying data can require considerable computer resources
- Improving display performance:
    - using smaller graphs and images
    - display fewer data points (down sampling)
    - less frequent display updates











Building an application …

- Chose a design architecture (design pattern)
- Start with a paper design ….
  - draw block diagrams
  - draw flow charts/state diagrams
- Prototype the user interface
  - helps defining necessary controls and indicators
- Divide-and-conqure
  - break the problem(s) into manageable blocks
  - make SubVIs for each function
  - put the entire design together

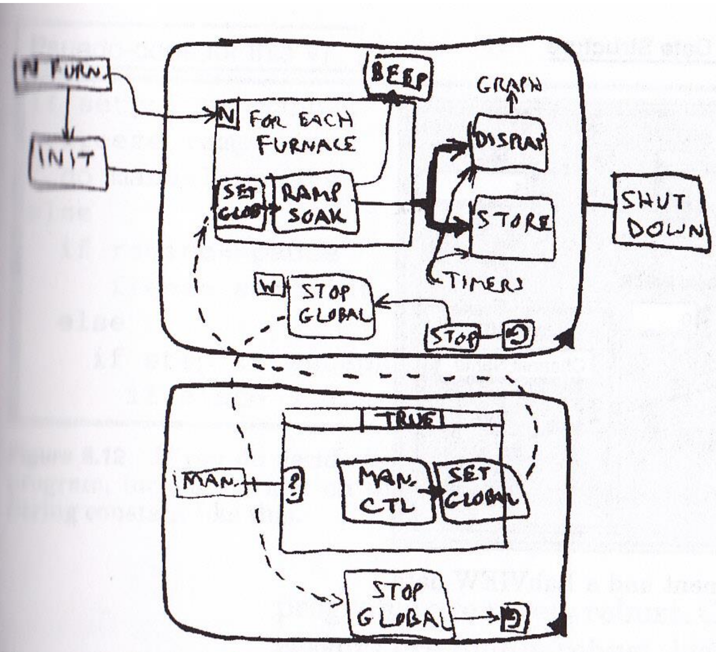


Figure 8.9 Sketch for the main VI used in VBL. Not very detailed at this point, but it sure gets the point across.