REAL-TIME ARCHITECTURE DESIGN

- Target and host architecture
- Multithreading
- Sleeping
- Improving determinism
- Passing data between threads

- When implementing a system with the LabVIEW Real-Time Module, consider whether you need to use determinism.
- If your application only needs the embedded qualities of the LabVIEW Real-Time Module, including the reliability of the module, the ability to off-load processing, and headless black box design, you do not need to change your programming techniques.
- However, if your application needs to be deterministic, your application must either guarantee a response to an external event within a given time or meet deadlines cyclically and predictably.
  - When designing applications within real-time constraints, you must employ certain programming techniques to achieve determinism.
  - When programming a LabVIEW Real-Time Module application, you can decide how to establish communication between multiple tasks or threads without disrupting determinism.
- We will discuss how to design your application to achieve determinism.
REAL-TIME TERMINOLOGY FOR LABVIEW RT

• Determinism – A characteristic of a system that describes how reliably it can respond to events or perform operations within a given time limit.
  ○ High determinism is a characteristic of real-time systems and guarantees that your calculations and operations occur within a given time. Deterministic systems are predictable. This is an important characteristic for applications such as controls. In a control application, the control program measures inputs, makes calculations based on the inputs, and then outputs values that are a result of those calculations. Real-time systems can guarantee that the calculations finish on time, all of the time.

• Loop Cycle Time – The time taken to execute one cycle of a loop.
  ○ Many applications that require a real-time operating system are cyclical, such as a control application. The time between the start of each cycle, T, is called the loop cycle time (or sample period). 1/T is the loop rate or sample rate.

• Jitter – The variation of loop cycle time from the desired loop cycle time.
  ○ Even with real-time operating systems, the loop cycle time can vary between cycles. The amount of time that the loop cycle time varies from the desired time is called jitter. The maximum amount that a loop cycle time varies from the desired loop cycle time is called maximum jitter.

• Embedded –
  ○ The term embedded encompasses many application areas, but characterizes a computer system that is typically a component within a larger system. These embedded systems operate in a headless fashion. A headless system has no user interface, such as a keyboard, monitor, or mouse. In many cases, embedded applications operate within restrictions on the amount of RAM and other resources that you can use, as well as the amount of physical space the embedded application can occupy. Embedded hardware ranges from industrial computers like PXI/CompactPCI that sit within larger machine monitoring and control systems to systems running on a chip.
DEFINITION OF JITTER

• All Systems have some jitter, but the jitter is lower for real-time systems than for general purpose operating systems.
• The jitter associated with real-time system scan range within nanoseconds.
• General purpose operating systems have high or unbounded maximum jitter that may vary from time to time and is inconsistent.
REAL-TIME ARCHITECTURE -- the basic architecture of a well-designed real-time program is shown below

- The overall application program is divided into two parts:
  - The host program -- the host program contains the user interface.
  - And the target program -- the target program is divided into two parts: a) the time-critical loop, and b) the normal priority loop.
- The loops on the target program are contained within separate VIs. Any code that must execute deterministically is placed in the time-critical loop with all other code in the normal priority loop.
  - In most programs, the time-critical loop handles all control tasks and/or safety monitoring,
  - and the normal priority loop handles all communication and data logging.
The host program runs on the host computer and communicates with VIs running on the target computer. This communication may involve user interface information, data retrieval, data broadcast to other systems needing from the target program, and any other non-deterministic task that you may need.
TARGET PROGRAM

- Higher priority processes preempt lower priority ones

**Target Program**

| Normal Priority Loop | Time-Critical Loop |

Inter-Thread Communication

- Processes that must be deterministic are time-critical processes—all other processes should be set to a lower priority
- Multithreading allows you to set the priority of a process

- The target program consists of time-critical code and normal priority code. A subVI contains the time-critical code, and a second subVI contains the normal priority code. This architecture allows you to separate the portions of the program that must behave deterministically from the portions of the program that do not.

**Time Critical versus Low Priority Processes**

- Deterministic applications often perform a critical task iteratively, so that all iterations consume a measurably precise amount of processor time. Thus, deterministic applications are valuable not for their speed, but rather for their reliability in consistently responding to inputs and supplying outputs with little jitter.
- A common example of a deterministic application is a time-critical control loop, which gathers information about a physical system and responds to that information with precisely timed output. For example, consider the oil industry where thousands of feet of pipes are assembled daily. As two pipes are threaded together mechanically end-to-end, the torque required to twist together the pipes increases until the pipes are fully connected. Suppose the machine connecting the pipes uses a control loop to respond to an increase in resistance between the pipes by applying more torque.
- Furthermore, suppose that once a critical level of torque is attained, the control loop is triggered to terminate. Under these conditions, the loop must execute deterministically because lag in the software could result in severe damage to the pipes and other equipment.
- Multithreading expands the idea of multitasking.
WHAT IS MULTITHREADING?

- Extension of the idea of multitasking
  - Multitasking – Ability of the operating system to quickly switch between tasks giving the appearance of simultaneous execution of those tasks. Windows 95 and later rely on preemptive multitasking, where the OS can take control of the processor at any instant, regardless of the state of the application currently running. Preemptive multitasking guarantees better response to the user and higher data throughput. This minimizes the possibility of one application monopolizing the processor.
  - A task is generally an entire application, such as LabVIEW

- Multithreading extends the following multitasking capabilities into applications:
  - Subdivide specific operations within an application into individual threads,
  - Divide processing time among threads,
  - Enable assigning priorities.

- **Multithreading extends** the idea of multitasking into applications, so that specific operations within a single application can be subdivided into individual threads, each of which can run in parallel. Then the OS can divide processing time not only among different applications but also among each thread within the application.

- For example, in LabVIEW multithreaded program, the application might be divided into three threads – a user interface thread, a data acquisition thread, and an instrument control thread – each of which can be assigned a priority and operate independently. Thus, multithreaded applications can have multiple tasks progressing in parallel with other applications. The OS divides processing time on the different threads similar to the way it divides processing time among entire applications in an exclusively multitasking system.
ADVANTAGES OF MULTITHREADING

• Differentiates between time-critical and non-critical tasks
  ○ Example of time critical tasks: control loop; safety monitoring.
  ○ Example of non-critical tasks: communication; data logging.

• Real time performance needs an OS that can give scheduling priority to time-critical tasks.

• Multithreading is useful when you have parts of your code that must not be deterministic or if parts of your code rely on non-deterministic I/O. To help differentiate between deterministic tasks and non-deterministic tasks, you can evaluate whether a task is time critical or not. A control loop and safety monitoring are considered time critical because both need to execute on time, every time to ensure accuracy. Communication is not time critical because a person or computer may not respond on time, every time. Likewise, data logging is not time critical because an accurate time stamp can identify when the data is collected or calculated.

• What could happen to a time-critical process if a non-critical task was involved? Putting network communication tasks (non-critical task) inside the time-critical loop may harm determinism. For example, if time-critical code relies on responses from another PC over the network and if the other PC does not reply in time, the time-critical code may miss a deadline. To prevent missing deadlines, the threads can be broken up into time-critical tasks and tasks that are not time critical. In this manner, higher priority can be assigned to time-critical tasks to ensure that they always finish on time.

• Therefore, to have a genuine real-time OS, the ability to assign levelled priorities is important.
LabVIEW REAL-TIME SCHEDULING

- LabVIEW Real-Time Module uses a combination of round-robin (threads of equal priority receive equal time slices and it might take several turns for a thread to complete) and preemptive scheduling (higher priority thread immediately suspends execution of all lower priority threads).
- Each VI is assigned a priority.
  - Thread priority determines the execution of VIs, with higher priority threads preempting lower priority threads
  - Threads with equal priority use round robin scheduling
- The time-critical priority VI receives the processor resources necessary to complete the task and does not relinquish control of the processor until it cooperatively yields to the normal priority VI or until it completes the task.
- The time-critical priority VI gives processor control to other tasks only when:
  - Enters sleeping state -- Without sleep time built into the time-critical VI, all other operations on the system are unable to execute, causing the system to behave as though using preemptive scheduling.
  - Completes the operations.
  - The normal priority VI then runs until preempted by the time-critical VI.
SETTING PRIORITY LEVELS

- The priority levels available in LabVIEW Real-Time Scheduler include the following in order from the lowest to highest priority:
- Set only one VI to time-critical priority -- To set a VI as time-critical, select **File >> VI Properties >> Execution**. You also can right-click on the icon/connector on the front panel to access the **VI Properties** dialog box.

SLEEPING

- Because of the preemptive nature of time-critical VIs, they can monopolize processor resources. A time-critical VI might use all of the processor resources, not allowing lower priority VIs in the application to execute.
- You must build time-critical VIs that periodically yield or sleep, to allow lower priority tasks to execute without affecting the determinism of the time-critical code.
- Consider sleep mode as a programmatic tool that a VI can use to proactively remove itself from the LabVIEW and operating system scheduling mechanisms. Sleeping suspends the execution of a VI or a thread. By taking advantage of sleep mode, you can allow a lower priority VI to run by putting a higher priority VI to sleep.
- In the top graph, the time-critical thread starves the two normal threads, because no sleep has been placed in the time-critical thread.
To understand starvation, consider the following example: three processes compete for a resource. Because Process A has a time-critical priority (highest), it runs until it finishes using the resource, thus freeing the resource. At that point, another process may use the resource – in this situation, either Process B or Process C. If Process A is ready to run again, it takes control of the resource and runs until it is finished with the time-critical section. Again, one of the two processes may run. If Process A is not put to sleep long enough for both processes to run, the possibility that a lower priority process is never allowed to run can happen. This situation is called starvation.

At the start, Process A has the resource and Processes B and C wait for Process A to go into sleep mode so that they may use the resource. Then, Process A goes to sleep, and the next highest priority process is allowed to run. In this case, because Process B has higher priority than Process C, it can run before. When the sleep time of Process A is finished, Process A takes the resource back from B. If this situation continues indefinitely, Process C can be blocked from the resource and become starved.

In the next sleep cycle, Process A is sleeping long enough to allow Process B to run and sleep then run Process C. To prevent starvation of a lower priority process, the higher priority process must be put to sleep long enough to allow lower priority processes time to execute.
SLEEPING

The act of suspending the execution of a VI or thread

<table>
<thead>
<tr>
<th>Software-timed sleep</th>
<th>Hardware-timed sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the OS clock to control the rate of a software loop</td>
<td>Use hardware or processor clocks to control the rate of a software loop</td>
</tr>
</tbody>
</table>

- You can program a sleep mode in a VI by using the Wait function or by using a timed loop.
- When controlling the rate of a software loop by using the Wait Until Next ms Multiple function, users only can achieve rates in 1 ms multiples. This means you can either run your loop at full speed providing no sleep at all, or you can achieve loop rates of 1,000 Hz, 500 Hz, 333.3 Hz, 250 Hz, 200 Hz, and so on. However, if your controller has a Pentium 3 or 4 class processor, you can use the Wait function from the Real-Time Timing subpalette or the timed loop to achieve microsecond wait times, which adds to the available loop rates.
- Sleep can be provided using the data acquisition hardware on the PXI platforms. Use the DAQmx timing VIs to harness the clock on your data acquisition hardware.
SLEEPING AND TIME-CRITICAL PRIORITY

• The following features are unique to the LabVIEW Real-Time Module:
  ○ If any VI in a time-critical thread goes to sleep, then the entire thread goes to sleep
  ○ Do not use parallel loops in time-critical VIs, because multitasking is disabled
  ○ If parallel loops are needed to achieve multiple loop rates, do not use time-critical VIs – use loops instead

• The time-critical priority (highest) of the LabVIEW Real-Time Module threads have a unique characteristic that is different from normal LabVIEW scheduling. If any VI running in the time-critical priority (highest) thread goes to sleep, then the entire thread goes to sleep.

• Other VIs running on the thread are forced to sleep and cease executing until the original VI wakes up. This is only the case for the time-critical priority (highest) setting. Conversely, if two VIs (or two loops for that matter), are executing on the same thread (other than time-critical priority (highest)), and one of them goes to sleep, the other VIs on the same thread continue to execute. In other words, the execution system of the LabVIEW Real-Time Module does not schedule time-critical priority (highest) operations from parallel VIs or loops, when any one of them sleep in the same time-critical priority (highest) thread. All other priority threads in LabVIEW Real-Time, and all threads in normal LabVIEW, continue to schedule operations from parallel loops and/or VIs, in similar threads.

• Given the cooperative multitasking nature of scheduling multiple time-critical priority (highest) threads, it is recommended that only one VI and loop ever be used with the time-critical priority (highest) setting. This is the only way to receive a guarantee of deterministic execution.
  If more than one time-critical VI (or loop) is needed to achieve different loop rates, you can use a timed loop instead of a time-critical priority VI.
• When you understand priority levels, you must be able to separate your code into the time-critical components and the normal priority components. The boxes shown above are different components of a typical program.

ASSIGNMENT: Choose which components should be in the time-critical portion of the target program, in the normal priority portion of the target program and in the host program.

Target Time Critical: ...

Target Normal Priority: ...

Host: ...
• Choose appropriate hardware
• Avoid shared resources
• Avoid contiguous memory conflicts
• Avoid subVI overhead
• Disable non-essential options
• Use only one time-critical loop in an application

• The easiest way to improve the determinism is to choose a faster hardware platform. If you are unable to achieve a desired loop rate, first check your hardware to be sure it is capable of reaching the required rate.
• If your hardware rates are acceptable, you can improve the determinism of your application in software by avoiding shared resources, contiguous memory conflicts, and subVI overhead. Because the LabVIEW Real-Time Module Memory manager is a shared resource, using memory reduction techniques also helps to improve the determinism of an application.
• We have seen that it is suggested to use only one time-critical loop in one application because when a time-critical loop goes to sleep, the entire thread goes to sleep. Next we explain programming methods for improving determinism.
SHARED RESOURCES

• A shared resource in the LabVIEW Real-Time Module is anything that can only be used by one process at a time.

• LabVIEW RT shared resources include the following:
  ○ Global variables; real-time module memory manager; networking code (TCP/IP, UDP, VI server); shared variables; non-reentrant subVIs; semaphores; file I/O.

• Certain data structures, driver libraries, and variables only can be accessed serially, one process at a time. A simple example of a shared resource common to all programming languages is the global variable. You cannot access global variables simultaneously from multiple processes. Therefore, compilers automatically protect the global variables as a shared resource while one process needs to access it. Meanwhile, if a second process tries to access the global variable while it is protected, the second process must wait until the first process has finished with the global variable.

• Understanding shared resources and how to identify them it is an important skill when programming real-time applications.

• These operations are inherently non-deterministic. Never use them inside a time-critical priority loop if you are attempting to achieve real-time performance.
Before a process can begin using a shared resource, it must obtain a mutual exclusion (mutex)

Before Process 1 finishes, Process 2 proceeds

After Process 1 finishes, Process 2 proceeds

- For our purposes a shared resource is defined as a software object that can run only one thread at a time. In this example, assume both processes constantly must access the shared resource. Process 1 runs at normal priority, while Process 2 runs at time-critical priority.
- Normally, when a time-critical thread needs to execute, it preempts all other code running on the real-time target; however, a normal priority thread can block the time-critical thread if it has not released a mutex that the time-critical thread needs. This is known as a priority inversion because the real-time OS cannot allow the time-critical thread to preempt the normal priority thread, merely because of the mutex around a shared resource.
- A mutex is a mutual exclusion object that allows multiple threads to synchronize access to a shared resource. A mutex has two states: locked and unlocked. After a thread locks a mutex, other threads attempting to lock it will block. When the locking thread unlocks (releases) the mutex, one of the blocked threads acquires (locks) it and proceeds.
In this example, the shared resource is a global variable, which is shared by two VIs – one set to normal priority and one set to time-critical priority. The real-time OS uses a method called priority inheritance to resolve the priority inversion as quickly as possible, by doing the following:

- Allowing the lower priority thread to temporarily inherit the time-critical priority setting long enough to finish using the shared resource and to remove the protection.
- After the protection is removed, the lower priority thread resumes its original lower priority setting and is taken off of the processor.
- Now the time-critical priority thread is free to proceed and use the resource, that is, access the global variable.

A result of this priority inversion the task jitter in the time-critical priority thread is increased. However, the jitter induced by a protected global variable is small compared to the jitter induced by protecting the LabVIEW Memory Manager. Unlike accessing global variables, performing memory allocations is unbounded in time and can introduce a broad range of software jitter while parallel operations try to allocate blocks of memory in a wide variety of sizes. The larger the block of memory to be allocated, the longer the priority inheritance takes to resolve the priority inversion.
Configure a subVI for reentrant execution if unrelated parallel processes call the subVI – allows several instances of a subVI to be called simultaneously. **Reentrant subVIs** do NOT act like global variables.

- Sharing subVIs causes priority inversions the same way global variables do. When a subVI is set to subroutine priority, that subVI can be skipped within time-critical code to avoid software jitter that would have occurred from a priority inversion. However, if you run unrelated parallel processes that call the same subVI, you can configure the subVI for reentrant execution.
- A reentrant subVI establishes a separate data space in memory each time it is called. A reentrant subVI allows LabVIEW RT to call several instances of a particular subVI simultaneously. Because reentrant subVIs use their own data space, you cannot use them to share or communicate data between threads. You should only use reentrancy when you must simultaneously run two or more instances of a subVI within unrelated processes that do not need to share data within the reentrant subVI.
- To configure a subVI for reentrancy, select **VI Properties >> Execution**, and then select the **Reentrant Execution** checkbox.