Timing is Everything…

- Why all loops should be throttled down.
- How regular is the timing on my loop?
- How regular does it have to be?

A LabVIEW program is automatically split up into multiple threads. These threads share the limited resources of the cRIO and they all demand their turn in the OS queue. It is bad practice to have any loops running without some sort of throttle to keep them from hogging all the CPU time. Unrestrained loops cause jerky response to the driver controls. That's also one reason why some user programs cause the ftp program that transfers new code onto the cRIO to time out, because the ftp program has a lower priority than your user code and your code can suck up all the available CPU time before the ftp program is allowed a shot.

Wait

This competition between tasks for CPU resources also means that your loop with its' time delay set to 10 or 100ms doesn't really run at the rate you specified. In Figure 1: Wait Timing Function shows a minimal loop restricted by a Wait function. All the loop does is graph the elapsed time between each loop. The red graph line is the actual time for each individual loop and the green line is a moving average of the time over 25 loops.

It's noticeable that *Wait* won't run any faster than the specified 10ms, but has a lot more variance as to when it might run after the required 10ms are up. This puts it at the whim of all the other threads contesting and queued for service by the cRIO CPU. All the other threads suffer equally, and this isn't a problem. It's the most efficient way to schedule the myriad tasks for both your program and the OS. In this example this particular loop has an average that varies all around 11.5 or 12ms. Non-critical tasks should almost always be slowed down by the *Wait* function to help the CPU distributed the compute load equally.

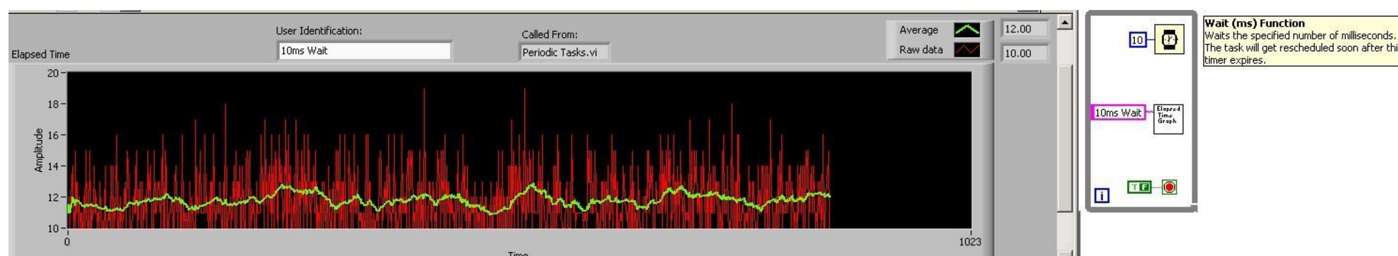

Figure 1: Wait Timing Function

Wait Until Next Multiple

By contrast, *Wait Until Next Multiple* is one step up from *Wait,* because it gives us an average loop time that is very close to the specified 10ms even though the actual loops may still vary quite a bit. This is good for tasks that work best when run at very regular intervals, for instance a PID loop. This can be made more precise by raising the vi execution priority, so it doesn't have to share so much with all your other program tasks that are slowing it down irregularly, however, changing priorities especially for whole vi's can be damaging to your overall program execution.

Figure 2: *Wait Until Next Multiple* shows the average is on or very close to the specified 10ms.

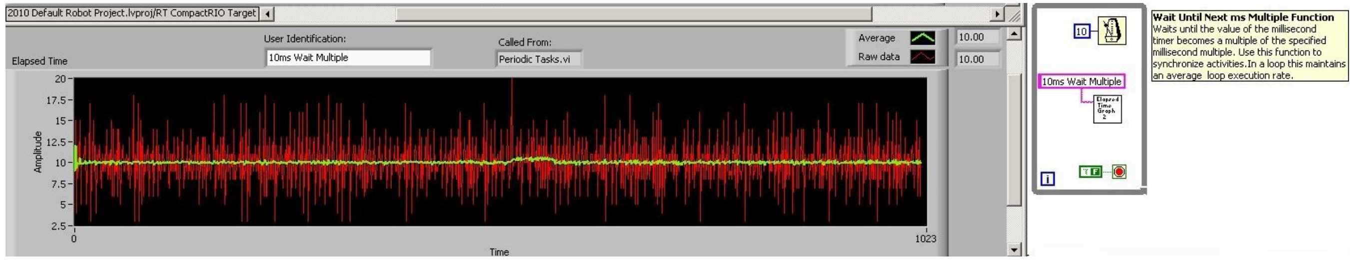

Figure 2: Wait Until Next Multiple

Timed Structures

One more step up from these approximate timings are the Timed Structures pallete found in *Programming->Structures->Timed Structures*. These give you precise synchronized timing that is built into a special while loop or sequence. These timed structures do their own task scheduling designed to work for you without having to raise the task priority, however it will also allow you to explicitly set just the priority of the Timed Loop, rather than using VI Properties to change the priority of a whole vi. Figure 3: *Timed Loop* demonstrates just how accurate the Timed Loop can be in comparison to the *Wait*'s used in Figures 1&2. None of the loops are off by so much as a sub-millisecond. A *Timed Sequence* is also available.

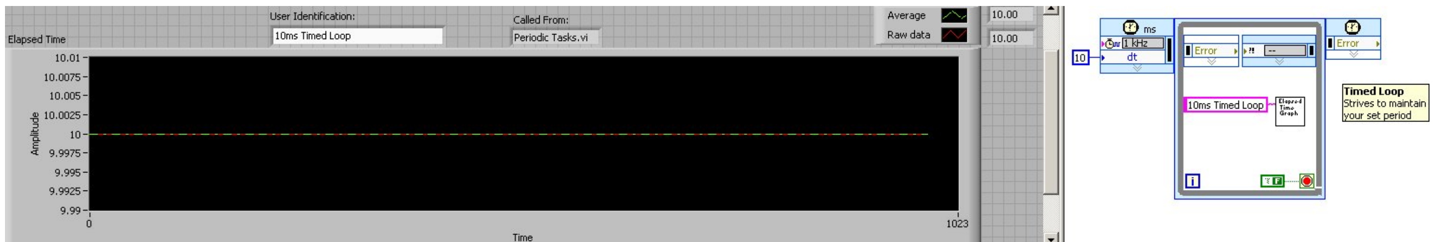

Figure 3: Timed Loop

A side note: Overriding the default priorities can be dangerous, so use this rarely if at all and exercise great care. Enough tools are provided that it is simply not necessary. Do not place your Timed Loop at a higher priority than critical OS tasks. If you have too many Timed Loops or loops that execute too fast while running at a high priority they will lock out lesser tasks. For instance, a single Timed Loop set for 1ms and a higher priority will lockout your driver controls or make response very erratic, because the Teleop loop will never get time on the CPU.