

"Soft" Real-Time Applications Under Windows NTä

Duane Mattern
M.E.C.H.
2249 Zollinger Road
Columbus, OH 43221 USA
d.mattern @ ieee.org

ABSTRACT

The Microsoft® Windows NT™ operating system is inappropriate for "hard" real-time applications that require a deterministic response as might be required in a closed-loop control system where safety is an issue. There are third party development packages that extend NT so that it can be used in a "hard" real-time system. These tools cost several times more than the cost of the PC hardware and this is too expensive for most educational purposes. An example of how "soft" real-time applications can be run under Windows NT using only the Visual C compiler is presented in this article. The multimedia timer and the high performance counter are used in the example to provide a *user mode* "pseudo-interrupt". The example is an animated, second order, spring-mass-damper system with a frame time of 25 milliseconds. A circular buffer is used for data storage. The example program can be modified for other "soft" real-time applications and access to the source code is provided.

NOMENCLATURE

B	Mechanical Damping (Ns/m)
F	Applied Excitation Force, (N)
frametime	time period between simulation updates
ISR	Interrupt Service Routine
K	Spring Stiffness, (N/m)
M	Mass, (kg)
MSDOS	Microsoft Disk Operating System
overrun	Execution Timing Error in a Real-Time System
X	Displacement of mass, (m)

INTRODUCTION

Students learn many steps towards the design and implement of control systems. Typically it is the implementation of the control system that is most rewarding for the student because they can obtain a sense of the physics of the problem by interacting with real or simulated hardware. For some students the implementation completes the picture of the relationship between the physics and theory. Typically automatic control training includes real-time control problems using inexpensive PC computer hardware running on of the Microsoft operating systems. As MS-DOS is being phased out, Windows 95 & NT are now being used as the operating system in educational laboratory computer equipment. However, these operating systems have isolated the user from the hardware, making it more difficult for the user to write real-time software for these platforms.

This article is focused on an inexpensive approach for the user to write and implement "soft" real-time applications using Windows NT and the Visual C compiler. The application is a real-time simulation and animation of the displacement of the spring-mass-damper system shown in Figure 1. X is the displacement and K, M, B are the constants representing the physic characteristics of the spring, mass, and damper, respectively. F is an input forcing function. This article discusses why NT is ill-suited for hard real-time applications. Then an example of an inexpensive approach to write soft real-time applications is presented. This approach is suitable for variety of "soft" real-time applications.

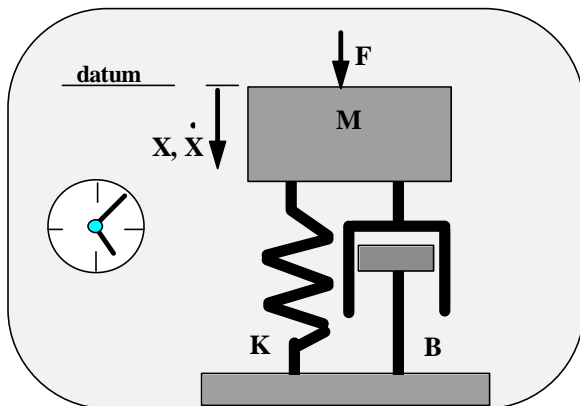


Figure 1 A spring-mass-damper system

WHAT IS REAL-TIME

In this article, I am interested in periodic sampled data and control systems and simulations with fixed update time intervals. I distinguish between “soft” real-time and “hard” real-time systems. From the list of frequently asked questions of the comp.realtime newsgroup, one definition of a “hard” real-time system is as follows:

“A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred.” (Gillies, 1998)

For a closed-loop feedback control system with a fixed sampling period, the required response must occur within the known time period. This time period is also known as the *frametime*. If the program does not perform the desired function within the allotted frametime, then a time period *overrun* is said to have occurred. Figure 2 presents a visual description of an overrun. The top diagram shows a task with a variable execution time and no frametime overrun. The bottom timing diagram shows a task that is interrupted and delayed. The delay causes the execution of the code to be completed in the *next* frame. At time $2T$ the overrun is detected when the task is called and found to be still active. The overrun will generate an exception in hard real-time systems so that an appropriate response is taken.

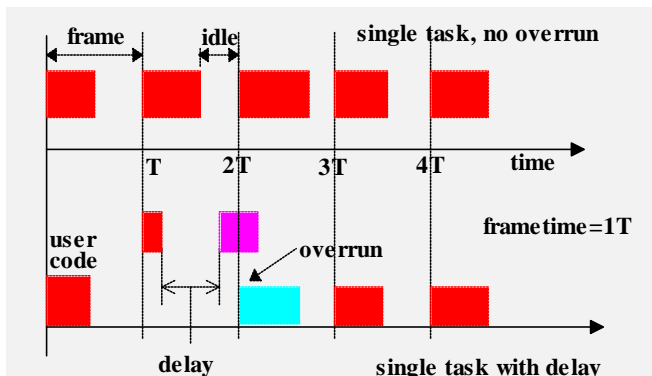


Figure 2 Timing diagram description of frametime overrun

A “soft” real-time system can be defined as a real-time system where an overrun is not classified as a fault, but rather as an annoyance that can be accommodated. An overrun in an animation would cause the motion to be nonphysical, but as long as it is known when it is nonphysical, false observations can be avoided. An overrun in a data acquisition system would violate the typical fixed sampling period, but if the time is accurately recorded when sampling occurs, interpolation can be used to provide an estimate of the missing sample time. If the overrun is so severe that multiple data points are missed, the data around the overrun can be avoided in any later analysis. In a hardware-in-the-loop, feedback control application, an overrun could lead to instability because a delay changes the control loop phase characteristics. Therefore, closed-loop control applications involving hardware that can be damaged, (as opposed to just a simulation), are classified as “hard” real-time systems should use an appropriate “hard” real-time solution to maintain the system safety.

USING WINDOWS NT FOR “HARD” REAL-TIME

With the MS-DOS OS, it was relatively inexpensive to implement real-time projects. David Bowlings 1989 article "Real-Time Modeling with MS-DOS", provided C language source code for a real-time animation of a spring-mass-damper system that could be modified for other real-time applications under MS-DOS. By modifying the timer interrupt vector address and the timer tick counter, a user can create their own interrupt service routine and the hardware was directly accessible under MSDOS. I have used this method on a 20MHz 386SX PC and was able to read/write to 8 analog channels at 2KHz.

The NT operating system provides a robust and stable operating environment, but it does so by insulating the user from the hardware. This makes NT an excellent development environment, but causes some difficulties if the user wants to run their own *user mode* code in real-time. User mode code can only obtain access to the hardware and the system timers through the system services. There are ways to access system services, but we will be dealing with the Win32 clients. Figure 3 shows how the user mode code is isolated from the hardware by layers of software.

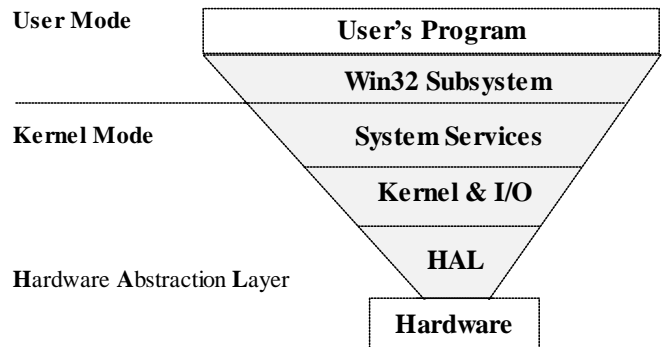


Figure 3 Isolation of user mode code from hardware

If the user wants to write their own interrupt service routine, this necessitates the purchase of the Microsoft Device Driver Developer’s Kit (DDK). The cost of the DDK (~\$500 US) is about half the cost of the computer hardware. Also NT interrupt service routines are restricted as to what the kernel mode code can access during the interrupt. To minimize the interrupt overhead, it is recommended that interrupts handle *only* their time critical components within the interrupt service routine and the less critical parts of the interrupts should be scheduled to run at a later time using a Deferred Procedure Call (DPC). The Deferred Procedure Calls are entered into a queue for later execution. Calls scheduled to this queue do not are not assigned a priority, so a DPC has to wait for any previously scheduled DPC in the queue. Discussion of the limitation of using Windows NT for hard real-time applications can be found in magazine articles by Timmerman and Monfret (1997), Bishop (1998), and Epplin (1998).

There are a number of third party extension for running hard real-time application under Windows NT. The products by Humusoft, Quanser and Xanalog all work in conjunction with the MathWork’s Simulink and start at \$1000 (U.S.). Other products handle more generic real-time applications, like the products from Imagination, LP_Elektronik, Radisys and VenturCom, which start at \$5000. The problem with these hard real-time NT solutions for the educational market is the expense. It is not cost effective to pay \$5000 for one copy of hard real-time software for education purposes when computer prices are under \$1500, not including I/O boards.

USING WINDOWS NT FOR “SOFT” REAL-TIME

It is possible to inexpensively run user developed, “soft” real-time NT applications for educational and training purposes. As with any low budget approach, this method is limited and should only be used when frametime overruns can be tolerated, if they occur. As a demonstration, I have recycled the structure of Bowling’s (1989) original code and rewritten his spring-mass-damper real-time simulation and animation for Windows NT. The demonstration has only a single real-time task and this task is based on a fixed frametime, periodic update. A simulation frametime of 25 milliseconds is used.

There are a number of minor things that you can do to reduce the chances of a frametime overrun. You can disconnect your network connection and avoid the use of other hardware, like a tape backup device or a hard disk, to avoid the associated hardware interrupts. Also, real-time code should be locked into physical memory to avoid any delays associated with disk swaps. Under Windows NT there is a VirtualLock function that locks a set of pages in RAM. However, according to Richter (1995),

“The system guarantees that the pages are locked in RAM only while a thread in your process is running. When the system preempts all the threads in your process, the system is free to unlock the pages and swap them to the physical storage in the paging file.”

Having a lot of memory and minimizing the use of other programs will decrease the likelihood that your program will be swapped out of physical memory. If a continuous looping task is run at the maximum priority as part of your real-time process, this will increase the likelihood that when using the VirtualLock function your real-time routines will remain in RAM. You could turn-off virtual memory, but this will have a drastic effect on your overall system when you use it for other applications. If a frame overrun does occur, you will be able to detect it as long as an accurate measure of the time is maintained. Now that the limitations have been stated, I have successfully run the demonstration at 25 ms under Windows NT on a 133MHz Pentium platform without a frametime overrun. Decreasing the frametime to 5 ms does cause overruns when doing other tasks, like minimizing the program window. Be aware that as you increase the process priority to “REALTIME_PRIORITY_CLASS” from “NORMAL”, you will have a gradual reduction in the keyboard and the mouse response on uni-processor systems.

A PSEUDO-INTERRUPT: THE MULTIMEDIA TIMER

The highest resolution, user mode timer under NT is the multimedia timer, with a minimum resolution of 1 millisecond. The multimedia timer is used to “approximate” an interrupt. I say “approximate”, because when the multimedia timer triggers it does not start the timer callback function immediately. It schedules an Asynchronous Procedure Call (APC) for the timer callback function. NT does not allow multiple callback functions associated with a specific timer to run at the same time. If an overrun occurs, the APCs just pile up in the queue and are resolved, in order, when time becomes available. In a normally real-time system with a periodic interrupt timer an overrun would be detected when an interrupt service routine is called while it is already active. This does not work with the multimedia timer callback function because of the APC queue. Instead I will use the systems high performance counter along with an alternative definition of what constitutes an overrun in order to provide a pseudo-interrupt service routine.

Previously an example overrun for a single task was shown in Figure 2. This figure shows the tasks starting at the beginning of the frame, after the timer interrupt is triggered, (less the task switching time). Using a multimedia timer for a “pseudo-interrupt”, the start of the callback function associated with a timer can vary because of the delay caused by waiting in the APC queue. Figure 4 presents a timing diagram for multimedia timer callback function to explain the need for an alternative definition of an overrun. The variable “ Δ ” defines the difference between the start of the timer callback function and the triggering of the kernel model timer interrupt. The variability with the starting time of the multimedia callback function must be taken into account when defining an overrun for this “pseudo-interrupt”.

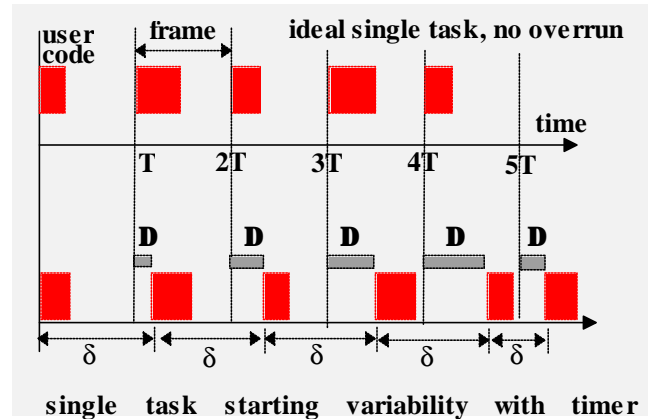


Figure 4 A timing diagram for a periodic callback function

Δ can not be measure because the trigger time of the kernel mode timer interrupt is not available. We can measure the time between the successive calls to the multimedia timer callback function, shown as “ δ ” in Figure 4, using the **QueryPerformanceCounter()** function. If we decide that maximum allowable difference between callback function start times, Δ , must be less than 25% of the frame time, then we can define an overrun as when δ is greater than $1.25 * T$. For a frametime of 25ms, if the time difference between callback functions is greater than 31.25ms, an overrun is declared. The demo program terminates when an overrun is detected. There are more sophisticated methods for defining and detecting an overrun with the multimedia timer, but this method will suffice for the demonstration program.

```
#include <mmsystem.h>
MMRESULT ISR; // MultiMedia Timer Event Identifier
TIMECAPS tc; // Declaration of Time Structure

//FUNCTION PROTOTYPE
void CALLBACK realtime_task(UINT Timer_ID,
    UINT msg,DWORD dwUser,DWORD dw1,DWORD dw2);

timeGetDevCaps(tc, sizeof(TIMECAPS));
timeBeginPeriod(tc->wPeriodMin);
ISR=timeSetEvent(25.0,realtime_task, 255, TIME_PERIODIC);
```

C Language code segment to setup a 25ms periodic multimedia timer and to define the callback function “realtime_task” (The linking of the program should include the winmm.lib library)

You may be tempted to use `QueryPerformanceFrequency()`, to measure the high performance clock frequency and to calculate the time of all future calls to the callback function using:

$$\text{counts} = 1^{\text{st}} \text{ count} + \text{FREQ} * \text{PERIOD} * k$$

where:

- 1st count is from the first call to `QueryPerformanceCounter`;
- FREQ is the result of `QueryPerformanceFrequency`;
- PERIOD is the real-time application framerate; and
- k is an index that is incremented when the callback is executed.

This should allow “Δ” to be calculated directly. The problem with this approach is that the clock frequency is not exact due to manufacturing tolerances. On one specific 133MHz machine `QueryPerformanceFrequency()` results in a value of 132940000. Using `QueryPerformanceCounter` and averaging over a long time period using a stop watch, the real value of this frequency was about 100000 Hz lower than indicated. This would cause drift in the above equations is used for an extended period.

This overrun definition assumes that the callback function start time does not drift relative to the ideal start time. This is a valid assumption because the multimedia timer is based on a kernel mode timer interrupt. This kernel mode interrupt maintains consistent time and thus the time when the APC is queued remains consistent. The variability, δ, is due to the amount of time required for the APC to be serviced. Figure 5 show the result of measuring the time variations, δ, for a 25ms console task on three PC's with no other applications running. The two spikes on the NT platforms were caused when the process window was minimized. Under Windows 95 the real-time priority locks out the mouse commands and prevents the window from responding to the minimization request. The mouse response on the uni-processor NT machine was poor, but on the dual processor machine, the mouse response was unaffected by the real-time priority.

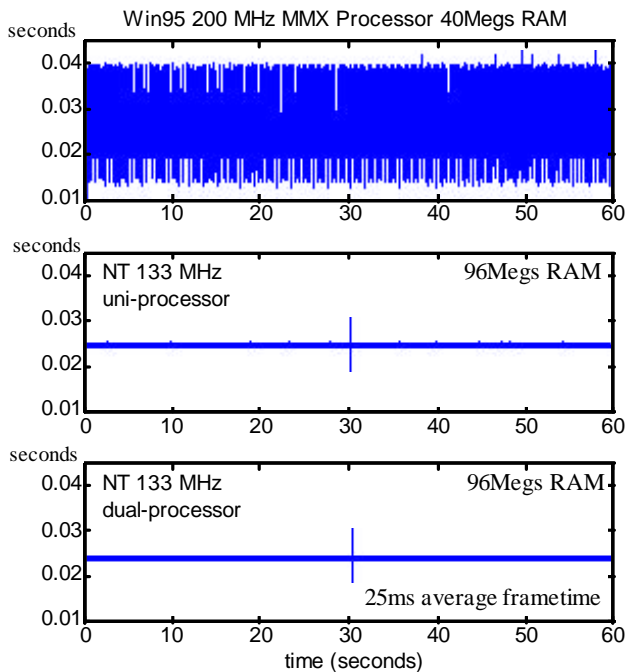


Figure 5 Measured frametime variations, “δ”, on 3 platforms

Figure 6 plots the measured time between calls to the multimedia callback function under Windows NT when the program is not otherwise disturbed. There is a ±1ms band around the desired frametime of 25ms with the multimedia timer setup for the minimum resolution of 1ms. Griffith (1998) developed a method to obtain a resolution on the order of one-tenth of a millisecond by combining the multimedia timer with the `SetWaitableTimer` function. This function allows the state of an event timer to be set to signaled with a resolution of 100 nanosecond. Using the multimedia timer with a period less than the required framerate, an event timer can be triggered from within the multimedia timer callback function that adjust for time variations between calls to the callback function. The `QueryPerformanceCounter` function is used to measure time variations between calls and the `pDueTime` parameter in the `SetWaitableTimer` function is used to reduce these variations. The user simulation code would be executed in an alertable loop, waiting for the event to occur using the `WaitForSingleObjectEx` Function. This method is not used in the demo program but it offers tighter control of the time period between functions calls and a way to get sampling periods that are not an integer multiple of one millisecond, (a 270 Hertz sampling rate for example).

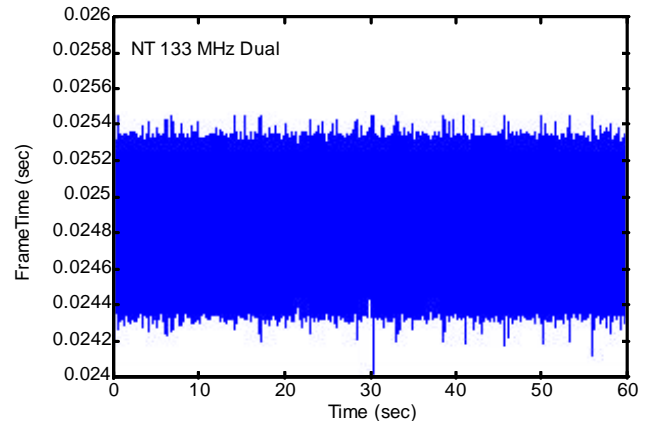


Figure 6 Measured frametime variations, “δ”, for a 25ms multimedia timer under NT when the system is not otherwise disturbed

THE "SOFT" REAL-TIME DEMO

The demo consists of a simulation of a spring-mass-damper system with a sinusoidal forcing function. Bowling's original simulation used the sinusoidal solution for a second order system in his simulation. I have replaced his solution with two first-order, linear ordinary differential equations in continuous state-space format as shown below:

$$\begin{aligned} \frac{dx}{dt} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

where dx/dt is the time derivative of x and “x” is the state vector. “u” is the input vector and “y” is the output vector. A,B,C, D are matrices of appropriate dimensions. A simple Euler integration scheme is used to obtain the position and velocity by integrating the state derivative vector containing the velocity and the acceleration. A continuous simulation with numerical integration was used so that the physical system parameters including the spring stiffness, K, damping, B, and mass, M, could be updated "on the fly". This allows the user to consider the response of systems with other parameters without having to stop the simulation..

The system does not check the parameters other than to bound the values. Note in the C language code segment below that the state update is done after the output calculations so that the output, y, at this time point is only affected by the input, u, at this time point through the direct feed-through terms in matrix D. Also note that the state variables have been implemented as static variables so that the values are maintained between function calls.

```

static double x[2] = {0.0, 0.0};

//          OUTPUT EQUATIONS
y[0] = C[0][0]*x[0] + C[0][1]*x[1] + D[0][0]*u;
y[1] = C[1][0]*x[0] + C[1][1]*x[1] + D[1][0]*u;
position = y[0];
velocity = y[1];

//          STATE UPDATE
A[1][0] = -K/M;
A[1][1] = -B/M;
B[0][1] = 1/M;
dx[0] = A[0][0]*x[0] + A[0][1]*x[1] + B[0][0]*u;
dx[1] = A[1][0]*x[0] + A[1][1]*x[1] + B[1][0]*u;
x[0] += dx[0]*frame_time; // Euler Integration
x[1] += dx[1]*frame_time;

```

“C” language code segment of a second order, single input state space system simulation and simple Euler integration

I have taken the structure that Bowling provided in his Dr. Dobbs Journal article and expanded it for a Windows environment. The code is broken up into the following ten functions:

- 1) WinMain()
- 2) Setup_user_background_task()
- 3) Setup_user_realtime_task()
- 4) Setup_realtime_task()
- 5) WindowProc(),
- 6) Dialog_Procedure()
- 7) Realtime_task()
- 8) Setdown_realtime_task()
- 9) Setdown_user_realtime_task()
- 10) Setdown_user_background_task()

The WinMain calls the other functions and provides the Windows message loop. Setup_user_background_task sets up and creates the display window animation. Setup_user_realtime_task initializes the user simulation parameters and variables. Setup_realtime_task sets up the process and thread priorities, the processor affinity for multiprocessor systems, and the timer. WindowProc handles the window messages and provides the animation by repainting the screen using global variables to obtain the simulation data. Version 1.0 of the demo program does not use double buffering for smooth animation. The "Setdown" tasks reverses whatever the "Setup" tasks initiated. Note that the "Setup" tasks could include the initialization of I/O hardware. Generally it is a good idea to configure I/O so that the outputs are "OFF" whenever the program is initially executed.

When the program is executed the multimedia timer is started but the simulation is initially disabled using the flag, SIM_ON = FALSE. This gives the user control over the running of the simulation without having to shutdown the timer. It also avoids a start-up quirk with the multimedia timer. Table 1 shows the results of using the high frequency counter to calculate the actual time during the startup of the multimedia timer configured as a periodic event timer with a period of 5 milliseconds and a minimum resolution of 1 ms.

Table 1 Start up of 5ms MultiMedia Timer

Sample Number	Simulation time (sec)	Counter time (sec)
0	0.000	0.0
1	0.005	0.000291
2	0.010	0.000453
3	0.015	0.000616
4	0.020	0.000780
5	0.025	0.000927
6	0.030	0.005131
7	0.035	0.010001
8	0.040	0.014859
9	0.045	0.020683

The multimedia timer appears to race ahead and then it settles into a normal update interval. This is probably due to an initial delay after the timer is first initiated and before the first APC calls are scheduled. The effect is only noticeable with sampling periods under 10ms. The data in Table 1 is also plotted in Figure 7. By starting the simulation with the logic flag SIM_ON equal to FALSE, you can avoid this startup problem with the multimedia timer.

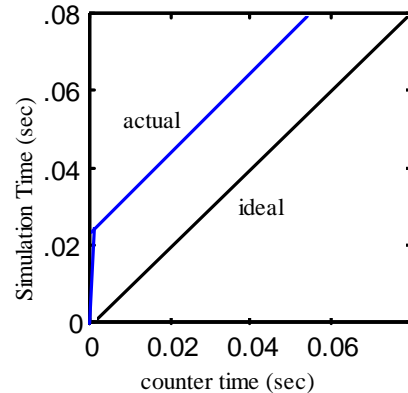


Figure 7 Counter Time vs Simulation Time using the Multimedia Timer as a Pseudo Interrupt

A global matrix buffer is defined for data acquisition. The pointer into this buffer is incremented at each call to the simulation. When the pointer reaches the end of the buffer, the pointer wraps-around, back to the beginning of the buffer. The high performance counter data is also recorded in the circular buffer. After "exit" is requested from the pull-down menu, the timer is canceled and the process priority is returned to NORMAL. Then, the Setdown_user_background_task function converts the recorded output the high performance counter to "seconds". This time data and recorded simulation variables (position, velocity, force) are written to the file named "timedata.txt" in the local directory prior to program termination. This ASCII file can be loaded into any graphing package to plot the recorded data.

The parameters in the spring-mass-damper system, (K, M, B), can be configured to modify the physical characteristics of the system. The frequency of the sinusoidal forcing function, F, can be modified over 0 to 10 Hertz.. There is noticeable peak amplitude in the motion near the system natural frequency and attenuation of the motion of the mass as the frequency increases beyond the natural frequency. The amplitude attenuation can more easily be seen in a plot of the displacement versus time.

There is a pull-down window to change the process priority. The bulk of the code for this dialog box was taken from the game "Moby Dick" from Donnelly (1996). NT has 32 priority levels for threads, but only 15 of the levels fixed, non-idle levels. There are 16 levels that the operating system can adjust up or down depending on the needs of the operating system and there is one idle thread. You can use Spy++ that comes with Visual C to check the priority of the process. For a more detailed discussion of Windows NT process and thread priorities, see the article in the MSDN Library (1996).

There is a pull-down window to change the process processor affinity on a dual processor system. The process processor affinity is the list of CPU's on which the process is allowed to run. A process processor affinity can be set from within the task manager as well, but you can not modify the processor affinity of "system" processes. Windows 95 is not a multiprocessor operating system, so this function is only valid under a multiprocessor version of NT.

THE DEMO RESULTS

The program is configured for a 25 ms update and circular buffer with a record length of length 2000 samples. This gives 50 seconds of running time before the buffer wraps around and starts to overwrite itself. The program has been executed with record length of 65536, (1638 seconds) but the file that is generated is large, (several megabytes). Also, the priority is initialized at "normal", so that the mouse can be easily accessed when the program first starts. This program has run for over an hour on a dual processor system while connecting to the Internet via an RAS connection, doing "chkdsk" and a backup on a SCSI tape drive, all without ever experiencing an overrun. This shows the inherent advantage of a multiprocessor system for this type of application. This demo is also useful as a simple tool to study the limitation with real-time user mode code under Windows NT.

CONCLUSION

In this paper I have explained some of the difficulties in using the Windows NT operating system for "hard" real-time applications. I then demonstrate a way to implement "soft" real-time applications in *user mode* under Windows NT that is appropriate for an inexpensive educational system. I show that by combining the "multimedia timer" and the "high performance counter" a pseudo-interrupt can be formed that is capable of detecting a sample time overrun. I then combine this pseudo interrupt with a circular buffer for data acquisition. A simple animation of the motion of a spring-mass-damper system is provided. The user has access to the system parameters via the user interface and can vary the spring stiffness, mass, damping, excitation frequency and excitation amplitude. This example serves as a template for others to write their own "soft" real-time applications under Windows NT, inexpensively, without having to resort to expensive 3rd party packages.

The executable program and source code for the Visual C/C++ example discussed here will be available on the Internet. Check www.gwlc.com/dmattern/ or send email to d.mattern@ieee.org for the current location as it is likely to change.

The original version of this program was written entirely in C. The program has been updated and these updates are included with the original version of the code on the website mentioned above. Some of the simple programs used to experiment with the multimedia and event callback functions are also provided.

The example presented here has been tested using Microsoft Visual C/C++ Version 4.2, 5.0 and Windows 95 and NT 4.0 on Pentium platforms. Alpha platforms and Windows 98 have not been test as of June, 1998.

REFERENCES

- Bishop B., "X86 RTOS availability follows hardware diversity", Jan. 1998, Vol 15, No. 1, Personal Engineering and Instrumentation News Magazine, www.pein.com, PECInc, Rye, NH, p34-39.
- Bowling, David, "Real-Time Modeling with MS-DOS", Dr. Dobbs Journal, Feb. 1989, p26.
- Cluster, H., "Inside Windows NT", Microsoft Press, ISBN 1-55615-481-X, 1993.
- Donnelly, P., Sept. 26, 1996, Source code example of a Windows based game, "Moby Windows v. 1.0", from the Microsoft Developer Network Library, Multimedia Article entitled, "Moving Your Game to Windows, Part I: Tools, Game Loop, Keyboard Input, and Timing.
- Epplin J., "Adapting Windows NT to Embedded Systems", June, 1998, Embedded Systems Programming Magazine, Miller Freeman, Inc., San Francisco, www.embedded.com, p44-61.
- Gillies, Donald, gillies@ee.ubc.ca, List of Frequently Asked Questions of the comp.realtime newsgroup.
- Griffith, L, 1998, personal email communications and discussions in the comp.realtime Internet newsgroup around Feb. 12, 1998.
- Horton, Ivor, "Beginning Visual C++5", Wrox Press, ISBN 1-861000-08-1, 1997.
- MSDN, 1995, Microsoft Developers Network Library: Operating Systems, "Real-Time Systems and Microsoft Windows NT".
- Richter, J., "Advanced Windows", Microsoft Press, ISBN 1-55615-677-4, 1995, p 195.
- Russinovich M., March 1, 1997, www.ntinternals.com/timer.htm, Inside Windows NT Hight Resolution Timers.
- Timmerman M., Monfret J-C., Windows NT as Real-Time OS?, Real-Time Magazine 97-2, www.realtime-info.be, p 6-13.
- Humusoft,s.r.o., www.humusoft.cz/rt/rt.htm
- Imagination, www.imagination.com Hyperkernel
- LP Elektronik, www.lp-elektronik.com LP-RTWin Toolkit
- Quanser Co., www.wincon.quanser.com WinCon 3.0
- Radisys, www.radisys.com/products/intime/ InTime
- VenturCom, www.vci.com/prod_serv/nt/rtx/index.html RTX - RealTime eXtensions
- Xanalog, www.xanalog.com: RT-Windows and REALoop

Webpages listed in references were known to be valid in June, 1998.