Design of a High Resolution Soft Real-Time Timer under a Win32 Operating

JANNO GROBLER AND DERRICK KOURIE
University of Pretoria

This article defines a real-time timer, and shows that a Win32 PC operating system does not provide a timer that fits this definition. Without a real-time timer, a true hard real-time system cannot be implemented. Not withstanding this, some companies, including Microsoft itself, provide real-time extensions to the Win32 PC platform, enabling the development of real-time implementations. Bearing this in mind, a solution is presented in which a workaround is found for the shortcomings of the timers provided Win32 PC platform, without looking to real-time extensions as the solution. Also considered is the question of whether the workaround offered here is likely to be compatible with expected advances in future Central Processing Unit (CPU) technologies.

Categories and Subject Descriptors: D.4 [**Software**]: Operating Systems - *Performance*; B.8 [**Hardware**]: Performance and Reliability- *Performance analysis and Design Aids*; C.1 [**Computer Systems Organization**]: Processor Architectures; C.4 [**Computer Systems Organization]** Performance of Systems
General Terms: Design, Experimentation, Reliability, Performance
Additional Key Words and Phrases: real-time implementations, timers, events, hyper-threading

1. INTRODUCTION

A real-time system typically consists of a set (or sets) of operations that have to be executed at periodic intervals of predictable size. The correctness of these computations relies on the logical correctness of the operations as well as on the time it takes to execute these operations [Wikipedia 2005]. Since the execution time of an operation plays such an important role, a real-time system may be regarded as a system that is dependant on accurate and predictable time measurement.

To measure the duration of the periodic intervals, an accurate or high precision timer is required. Such a timer may also be referred to as a real-time timer. The timer may be based on either a hardware- or a software source. The hardware source is typically a device external to the PC that generates signals at fixed interval. However, the hardware timer is beyond the scope of this article. Section 2 provides more information on the real-time timer.

This article aims to show that the software-based timers provided on the Win32 PC platform are in adequate for a true real-time implementation. That being said, real-time extensions for the Win32 platform are available from Microsoft and from other third party companies. Windows XP Embedded and Windows CE are Win32 implementations that were developed with real-time in mind. These extensions and operating systems are beyond the scope of this article since the article focuses on whether a workaround may be found for the in inadequacies of the Win32 PC platform timers. Results that illustrate these shortcomings are presented in section 3.

The PC based Win32 platform was not developed with true real-time as a feature [Newcomer 2000]. The results of this are applicable to Windows NT, Windows 2000 and Windows XP. These operating systems were designed as general purpose or networking platforms [Timmerman et al 1997].

The remainder of this article consists of the following: Section 0 defines a timer, and focuses on the characteristics of a real-time timer. Section 3 describes different Win32 timers. Section 4 presents a solution as a workaround to the shortcomings of the Win32 timers. Section 5 shows how emerging technologies fits into the solution found in section 4 and the final section presents the conclusions to be drawn from work to date and points to future work in this field.

2. REAL-TIME SYSTEMS AND TIMERS

2.1 Hard- and Soft Real-time

Wikipedia [2005] defines a real-time operation as follows:

> 'An operation within a larger dynamic system is called a real-time operation if the combined reaction- and operation-time of a task is shorter than the maximum delay that is allowed, in view of circumstances outside the operation. The task must also occur before the system to be controlled becomes unstable.'

Timmerman et al [1997] define a real-time system as one that '…responds in a timely predictable way to unpredictable external stimuli arrivals.' The latter distinguish between hard- and soft real-time, based on the consequent instability that arises when deviations occur in the maximum allowable delay referred to by the Wikipedia definition. A hard real-

Author Addresses:
J. Grobler, Department of Computer Science, University of Pretoria, Pretoria, 0002, South Africa; jannogrobler@gmail.com.
DG Kourie, Department of Computer Science, University of Pretoria, 0002, South Africa; dkourie@cs.up.ac.za.

time system does not allow a task to exceed the maximum allowable delay. It is assumed that if this happens, then a system failure has occurred [Banachowski 2003]. Furthermore, the cost of such failure is regarded as ‘infinitely” high. On the other hand, a soft real-time system is tolerant of a measure of deviation from the maximum allowable delay. Deviations may cause some measure of system degradation such as lower performance, and this may worsen as the deviation rises [Banachowski 2003].

However, soft real-time has practical applications. Before a hard real-time system is deployed in its operational environment, it has to be thoroughly tested. Often a real-time system (for example, the weapons computer on a fighter aircraft [Gill 2001]) may have catastrophic results should it fail unexpectedly. A soft real-time equivalent is consequently used to validate such systems, where the operational environment for the real-time system under test is simulated. In this environment, deviations from the maximum allowable delay are less important; the focus of the validation is more on the functionality of the real-time system. However, since the soft real-time system’s performance degrades as the extent of deviation rises, unpredictable results may occur if the timing mechanisms are not predictable. Thus, to test the hard real-time system’s reliability, the soft real-time deviation has to be constrained within acceptable limits.

Henceforth unqualified reference to real-time will be construed to refer to soft real-time.

2.2 Timers and Intervals

To ensure that each task in a real-time system (whether hard or soft) is completed within the maximum allowable delay, time in such systems is usually segmented into consecutive intervals of equal size. A timer is used to signal the end of each such interval. The MSDN [April 2005] defines a timer as ‘… an internal routine that repeatedly measures a specified interval, in milliseconds.’ A timer may therefore be thought of as a clock whose ticks mark the end of an interval and the start of the next.

In a hard real-time environment, the system needs to execute its required functionality within the duration of an interval [Barabanov 1997]. An example of a system with such deterministic requirements is the avionics on an aircraft [Gill 2001], typically the flight computer. The flight computer needs to have accurate information at all times about where the aircraft is. It will therefore communicate with the Global Positioning System (GPS), determine the aircraft’s position and then carry out necessary operations on this information. If this functionality does not complete within the desired interval, a system failure occurs. The timer ticks that indicate the interval’s duration therefore have to be extremely accurate. Moreover, the system needs to know when the interval has elapsed. The timer is said to fire an event when the interval has elapsed, at which point the system executes the next required set of computations.

A good measure of the accuracy of an interval’s duration is the maximum allowable difference between of the actual interval duration and the desired interval duration. If, for example, an interval of τ ms is required, a real-time system expects that the interval duration be as close to τms as possible, otherwise a system failure may be the result. The maximum deviation is the difference between τ and the actual duration.

The interval duration of a real-time system is characterized in terms of the number of timer events per second – i.e. in terms of the frequency of the interval (measured in Hertz). For example, an interval of 20ms equates to 50 intervals per second, or 50Hz. It is also essential that the timer does not drop intervals, as the system expects to be executing at every interval, without exceptions.

Resource reservation is an important consideration in a real-time environment [Abeni 2002], as the timer also requires access to available resources, most notably the processor. The processor has to ensure that computations are executed within the desired interval and completes before the end of the interval. Accordingly, the timer has to use as little processor time as possible, to allow the system to use the maximum amount of processor capacity for other computations.

2.3 Types of Timers

There are two types of timers: loop timers and waitable timers. A loop timer repeatedly references a system-provided counter (or time stamp) to determine when the required interval duration has expired. For example, modern personal computer hardware provides a counter on the Advanced Configuration and Power Interface (ACPI). This counter is incremented every processor clock cycle, making it independent from processor speed. It is thus very fine-grained, and ought to serve as the basis for a very accurate timer. However, no other functionality is provided. In particular, no mechanism is provided to fire an event at a certain timestamp value. The counter has to be polled externally to determine the current time timestamp. A loop timer would typically read such a counter in a continuous loop and use that information as a basis for determining when an interval has expired. Such a loop timer is indeed very accurate. However, it uses all available processing time, leaving no time for a real-time system to execute the required computations on the same processor. It is therefore only of practical value in a multiprocessor context.

In a real-time context that relies on a single processor, it is essential for timer to be waitable. A waitable timer fires events at the end of every interval, but only uses the processor when the event occurs. The system is thus free to use the processor between timer events. This means that a waitable timer is either interrupt driven or event driven. In the case of an interrupt-driven timer, a hardware device measures an interval and generates an interrupt when it elapses. In an event driven timer, the operating system measures the interval and sets an event when the interval elapses.

Hard real-time applications rely on interrupt-driven waitable timers. Indeed, even when the software belonging to such applications is to be tested on a desktop machine, it is common practice to attach an external hardware device to the desktop that provides an interrupt-driven waitable timer to the software under test. This generally provides a timing mechanism to the system under test that testers regard as sufficiently accurate.

Below, a soft real-time timer is proposed – one that is sufficiently accurate to replace the external timer hardware currently needed when testing real-time systems on desktops. The context in which this soft real-time timer was needed, dictates that it should have the following properties:

a) It should be a waitable timer.
b) It should be an accurate and predictable timer, which deviates maximally by ±500µs from *any* specified interval duration of 1ms or more. Thus, the actual interval duration may be at most 500µs less than and at most 500µs more than the desired interval duration, irrespective of what the desired interval duration is. The result will be that the interval is predictable – the actual interval duration is always within 500 µs of the duration specified.
c) It should provide a timing mechanism that is periodic.
d) It should generate an event to indicate that the interval has elapsed.
e) It should use minimal system resources. Minimal in this context will be defined as on average less than 10%.
f) Finally, it should be executable in a Win32 context.

In seeking such a timer, timer mechanisms available in the Win32 SDK were investigated. The findings are reported in the next section.

3. WIN32 TIMER MECHANISMS

Win32 SDK provides a number of mechanisms that may be used to measure timer intervals. In this section results are provided of tests undertaken to explore the accuracy of these mechanisms. They indicate the extent to which these timing mechanisms conform to the requirements of a soft real-time timer as discussed in the previous section. All the timers discussed are waitable as defined previously. Therefore they all use minimal system resources, thus conforming to part a) and e) of the soft real-time timer definition in 2.3.

In this section, each timer's conformity to parts b) through d) is discussed. In each case, the timer was required to produce a sequence of 1ms intervals. In the sections to follow, the figures represent data measured over a period of approximately 15 seconds or less. This is to make the charts more readable. However, the timers exhibit the same overall behaviour over longer periods.

3.1 System Timer

The Win32 API provides a mechanism for creating a so-called system timer that is supposed to trigger a timer event after a specified time interval. The mechanism is based on the Windows system time. When Windows XP boots up, the current time of the real-time clock (or the RTC) is taken as the immediate system time. From this point onwards, the system time is updated every time a clock interrupt is received. This system time is used to determine the duration of intervals in the created system timer.

3.1.1 1kHz Interval System Timer

Such a system timer was created and specified to fire timer events at 1ms intervals. Figure 1 illustrates the *actual* interval duration produced by the system timer over a period of approximately 15 seconds. As can be seen, the timer never provides the required 1ms second interval. Instead it provides intervals of approximately 15.6ms, reaching a maximum interval of 16.336ms and a minimum interval of 14.528ms. The timer's maximum deviation is thus 16.336ms-1ms = 15.336ms. It is clearly incapable of generating the required soft real-time interval, thus failing in part b) of the required soft real-time timer in 2.3. Nevertheless, the timer is periodic, thus satisfying part c) and it generates an event when the interval elapses, thus satisfying part d). Note also that the observed deviations of such a systems timer could well improve if required to fire after an interval duration of say 20ms. However, its performance as measured above was enough to disqualify it for use as a soft real-time timer.

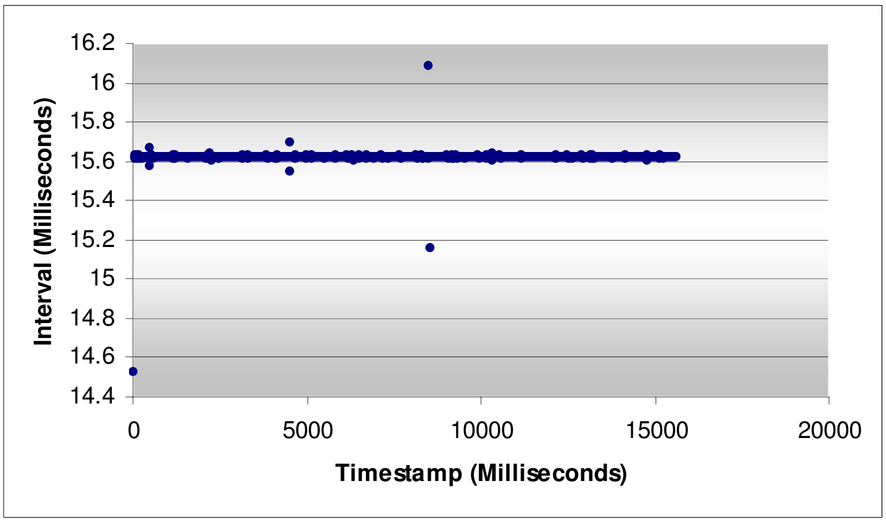

Figure. 1. *System Timer 1kHz Frequency*

The overriding reason for the degraded performance is that the periodic clock interrupt is used to update the system time. Thus, only when this periodic interrupt is raised does the Windows operating system check whether the timer's interval has elapsed. On most systems, the interval of this periodic clock interrupt is 10ms. It has been known to be 15ms [Peng 2002]. As a result, the system timer cannot meet the requirements of a soft real-time timer that is to accurately measure out 1ms intervals.

3.2 Multimedia Timer

The multimedia timer was developed by Microsoft to provide the greatest resolution possible for the hardware platform, in light of the shortcomings of the system timer, as exemplified in 3.1. The multimedia timer was developed primarily for use in multimedia applications, such as video and audio playback. These multimedia applications are examples of soft real-time systems and require that the underlying system provide sufficient periodic scheduling time with enforced quality guarantees, such as a fixed execution interval [Lin 1998]. Therefore, the system timer inaccuracies render it unsuitable for use in, for example, video playback applications.

Initializing the multimedia timer places the entire system in a high-performance state. In other words, the interval of the interrupt at which the system time is updated is dropped to 2ms [Peng 2002]. The multimedia timer is periodic, therefore part c) is satisfied. The timer generates an event when the interval has completed, therefore part d) is satisfied.

3.2.1 50Hz Multimedia Timer

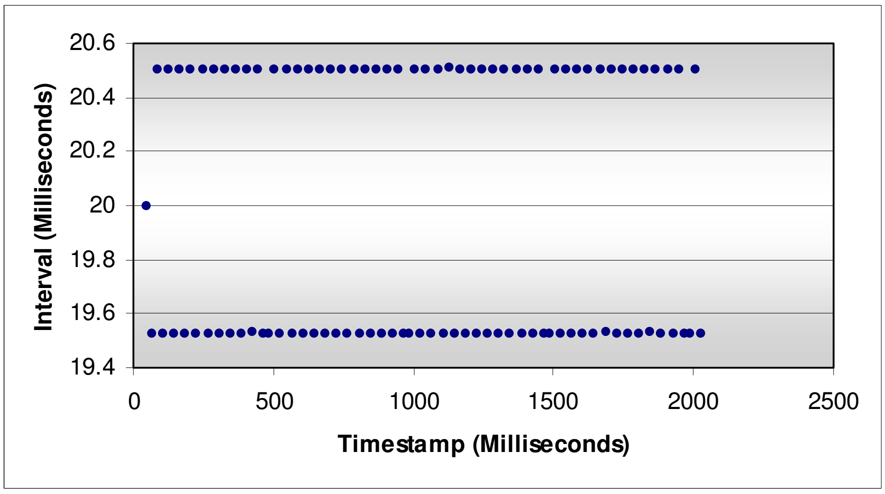

Figure 2. *Multimedia Timer 50hz Frequency*

Figure 2 illustrates the interval duration of the multimedia timer over a period of 2 seconds with an interval size of 20ms (frequency 50Hz). Looking at the figure the timer stays within the range 19.5ms and 20.5ms. Observed over a 60 second period, with the maximum interval was recorded at 20.51ms and the minimum at 19.527ms.

Therefore, the timer exhibits a maximum deviation of 510μs, which is close to the 500μs resolution that was required for the soft real-time timer.

3.2.2 1kHz Interval Multimedia Timer

Figure 3 illustrates the interval duration of the multimedia timer over a period of approximately 32 seconds where the interval frequency was set to 1KHz (i.e. 1ms). Looking at the figure the timer is seen to stay within the range 1ms and 2ms. Over a period of 60 seconds the maximum interval recorded was 1.966ms and the minimum interval 0.926ms. The timer thus exhibits a maximum deviation of 0.966μs, which is well outside the 500μs deviation defined in b). Although the multimedia timer exhibits good results at bigger intervals, it does not fully conform to part b) of the required soft real-time timer.

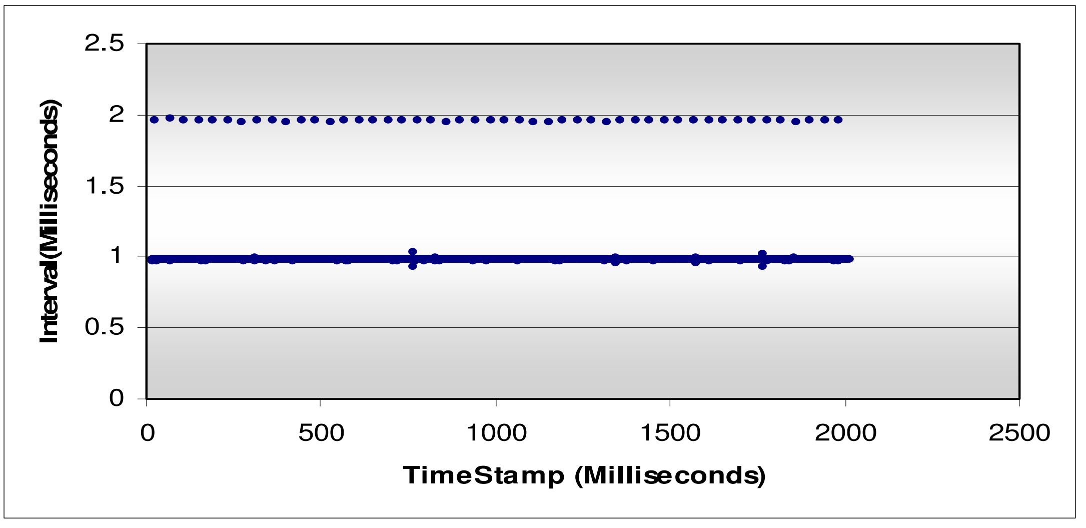

Figure. 3. *Multimedia Timer 1kHz Frequency*

3.2.3 The 1ms resolution

Microsoft tests have determined that lowering the interval at which the system time is updated to 2ms (as was done when the multimedia timer was introduced) has a negligible effect on processor usage. However, they report that the overall system performance is greatly reduced when a resolution of 1ms is used [Peng 2002]. It was for this reason that the clock update interval for the multimedia timer was kept at 2ms. The timer implementation in 3.3 below illustrates this.

3.3 The Multimedia Sleep timer

As stated in 3.2, when initializing the multimedia timer, the effects ripple through to all the timing mechanisms in the system. The Win32 API provides the functionality to suspend the processing of thread for a specified interval of time. This API function is called *"Sleep"*. Without the multimedia timer, this sleep interval exhibits the same maximum deviation as the system timer in 3.1. The multimedia timer, however, allows the sleep mechanism to reduce its maximum deviation in line with the more fine-grained resolution provided by the multimedia timer itself.

To illustrate this, the algorithm in Figure 4 was implemented:

```
Initialize the multimedia timer
For (duration of test)
    Sleep for the desired time
    Trigger the timer event
End for
```

Figure. 4. *Sleep Timer algorithm*

The results of this algorithm are in line with that of the multimedia timer, but it was found that at a frequency of 1KHz, the sleep timer provides a more or less consistent 2ms interval, as can be seen in Figure 5.

The maximum interval recorded over a period of 60 seconds was 2.024ms and the minimum interval recorded 1.883ms. Therefore an interval of 2ms with a maximum deviation of approximately 120 µs is achieved. The average interval size over the 60 seconds period was 1.95ms. The processor usage is below 2% due to the wait that is induced by the sleep functionality. It thus conforms to requirement e) in section 2.3. Nevertheless, it is readily apparent that the sleep timer does not conform fully to software real-time requirements, as part b) of the real-time requirement is not met.

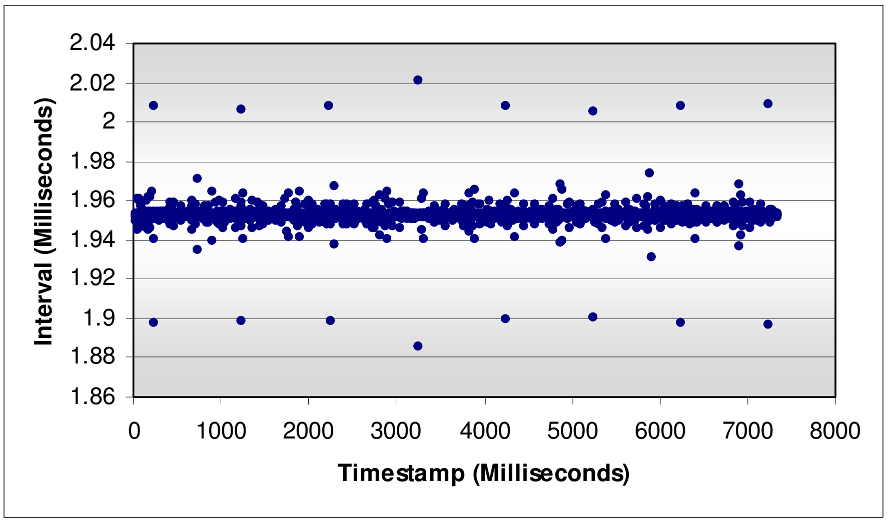

Figure. 5. *Sleep Timer 1kHz Interval*

4. THREAD INDUCED WAITABLE TIMER (TIW TIMER)

As stated in 2.3, modern hardware supports an ACPI clock counter, which provides a mechanism to compute accurate timestamps. The loop timer can be used to accurately measure a specified interval by reading this clock, but as was noted in section 2.3, all available CPU time is used in the process. To develop a timer that would improve on the Win32 timers of section 3, a mechanism needs to be found where this ACPI clock counter may be polled to determine timestamps without using all available processing time. In other words, a method is sought to use the ACPI clock counter as a basis for a waitable timer.

To achieve this, a “waiting” period has to be enforced to prevent the timer from taking up all the available CPU processing time, in contrast to a loop timer. The waiting period has to be small enough for a 1ms interval timer to be derived that has a maximum deviation of 500µs. A solution to this seemingly contradictory set of objectives presented itself, based on the use of multiple threads. In fact, it turns out that two threads suffice to build the required timer.

According to the empirical evidence of section 3.3, if the multimedia timer is initialized and the sleep function instructs a thread to sleep for an interval of approximately 2ms, then the thread sleeps for an average of 1.95ms instead. This observation suggests a way of improving the accuracy of a 2ms interval timer, without making excessive demands on the CPU.

As stated, two threads are necessary which we will refer to as thread A and thread B. Both threads read the ACPI clock counter to determine how much time has elapsed. Only one of these threads is active at a given time. The timing between the two threads is shown in Figure 6. Thread A will wait in a loop polling the ACPI clock counter until an interval of 1ms has elapsed after which the timer event is issued followed immediately by the 2ms sleep instruction. At this point thread B is made active and waits in a loop polling the ACPI clock counter for an interval of 1ms, followed by the timer event and 2ms sleep instruction as was the case with thread A. Since thread A would only have been sleeping for 1ms by this time, 1ms from the timer event issued by thread B has to elapse before thread A resumes. In this time, neither thread is executing; therefore a waiting period is induced.

By the time that thread B has waited in the loop for 1ms, thread A has approximately 1ms of sleeping time left. Therefore when thread A completes its 2ms sleeping time, approximately 0.95ms (1.95ms – 1ms) has elapsed since the timer event issued by thread B. When thread A resumes, it waits in a loop polling the ACPI clock counter for on average approximately 50µs (1000µs – 950µs) until 1ms has elapsed since thread B started its 2ms sleep interval. This period is the α period in Figure 6.

Therefore, from this point, thread B will resume execution after on average 0.95ms, resulting in another α period, processed by thread B in this case. After another approximate 0.95ms thread A resumes and so on.

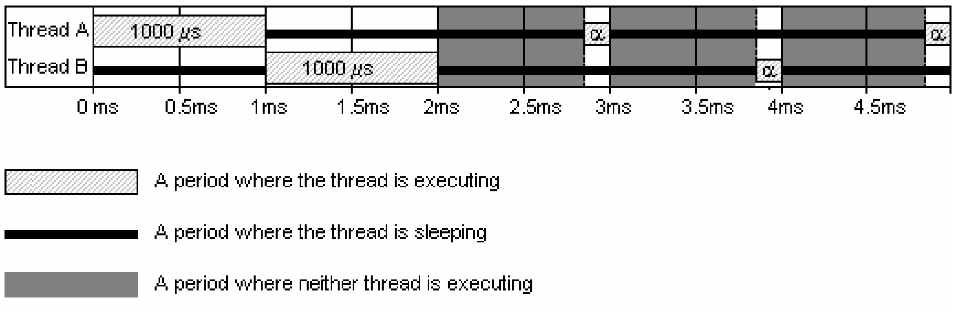
Thread A
Thread B
1000 µs
1000 µs
α
α
α
0 ms
0.5ms
1ms
1.5ms
2ms
2.5ms
3ms
3.5ms
4ms
4.5ms
A period where the thread is executing
A period where the thread is sleeping
A period where neither thread is executing

Figure. 6. *Thread Induced Waitable Timer Timing diagram*

Our prior results already suggest that this wait in a loop where the ACPI clock counter is polled, will endure on average for about 50µs, which does not seem too severe on the CPU. Furthermore, the data in Figure 5 suggests that there will occasionally be intervals that endure for slightly longer than the required 2ms. However, they are well within the allowed tolerance of 500µs. Should this be the case, the active thread at this point will issue the timer event immediately and enter the 2ms sleep interval.

The top-level algorithm of the timer is given in Figure 7.

```
Initialize thread A
Initialize thread B
Save the first timestamp from the ACPI Timer
Start thread A
Start thread B
While the timer is running
    Sleep for 2 ms
Stop thread A
Stop thread B
```

Figure. 7. *Thread Induced Waitable Timer: High level logic*

The first timestamp is taken from the ACPI clock counter. Thread A is started first and will start measuring the initial 1ms interval immediately. Thread B will be started as well, but will wait for thread A to finish the initial 1ms interval before commencing its execution. While these two threads are executing the main application thread will wait in a loop while the timer is active. This loop is necessary to ensure that the main thread does not exit until the timer is terminated. When the timer is terminated, thread A and B has to be terminated as well.

However, such a setup – i.e. where each thread operates entirely independently of the other – would be subject to time drift. Instead of such independent functioning, the readings taken from the ACPI clock counter should be stored in variables that are globally available to both threads. (Reads and writes to these variables should off course be in critical sections of code, protected by mutual exclusion mechanisms that prevent simultaneous access.) Call these variables $\sigma_{previous}$ and $\sigma_{current}$. The first timestamp in Figure 7 is saved as $\sigma_{previous}$. The interval at any point in time is given by $\sigma_{current}-\sigma_{previous}$. A thread induced waitable timer for a 1ms interval can thus be built by starting off a thread (thread A), waiting for 1ms, recording the ACPI clock counter value as $\sigma_{previous}$, and starting off a second thread (thread B). When each thread is started, both will execute the algorithm given in Figure 8.

```
While the timer is running
    Wait to enter the critical section
          // Initial value of $\sigma_{previous}$ read just before thread 2 starts
          // Both threads execute the algorithm below
          For (duration of the test)
             Sleep for 2ms
             $\sigma_{current}$ := Timestamp from ACPI timer
             While ($\sigma_{current}$ - $\sigma_{previous}$ <1000µs)
                        $\sigma_{current}$ := Timestamp from ACPI timer
             $\sigma_{previous}$ := $\sigma_{current}$
             Fire the timer event
           End for
    Leave the critical section
```

Figure. 8. *Thread Induced Waitable Timer: Thread logic*

The results of this algorithm on a frequency of 1KHz, is illustrated in Figure 9 and shows the performance over a period of 3 seconds. Over a period of 60 seconds the maximum interval recorded was 1.05ms and the minimum interval recorded 0.999ms. Therefore an interval with a maximum deviation of approximately 50 µs is achieved. The processor usage varies between 0% and 13.33%. The reason for the difference in processing is due to the fact that the amount of work done by the threads in the α period varies due to the performance of the *Sleep* instruction (refer to 3.3). The average processor utilization was 5.32%

Therefore:

- The timer is waitable (Part a))
- The timer has a maximum deviation of 50µs (Part b))
- The timer has a periodic interval (Part c))
- An event can be triggered when the interval elapses (Part d))
- The timer uses on average less than 10% of the processor's available capacity (Part e)).

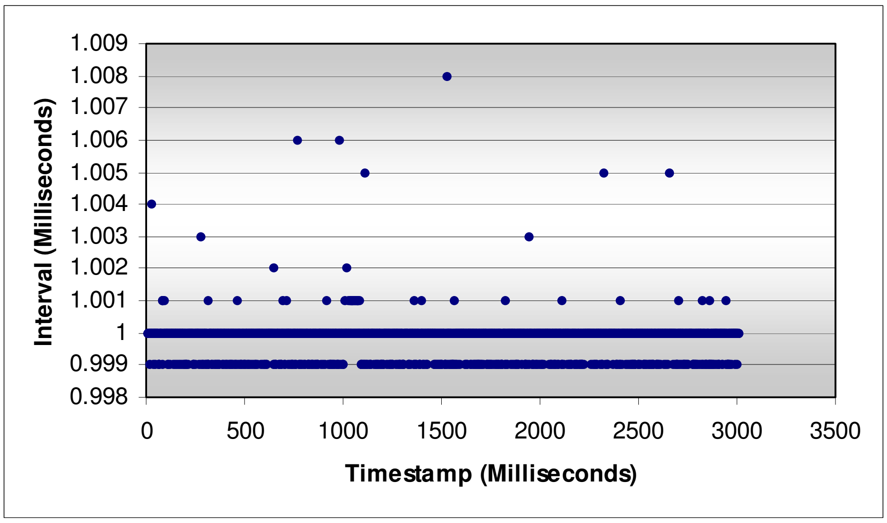

Figure. 9. *Thread Induced Waitable Timer 1kHz Interval*

This thread-induced waitable timer will be referred to as a TIW timer.

5. EMERGING TECHNOLOGIES

The results above do not take timer load into account. The whole idea behind the development of a timer is to trigger the operations that have to be completed at periodic intervals. The triggered operations have priority when it comes to the processor usage.

Although the TIW timer provides good performance, the fact that it uses an average of 5.32% of the processor capacity means that, theoretically, the real-time computational functionality has on average only roughly 95% of the processor's capacity at its disposal. Untimely scheduling of the concurrent threads of the TIW timer may hamper the real-time accuracy of the solution [Chu 1997]. However, it is the fact that the timer is a multi-threaded application means that its design is well positioned to take advantage of impending hardware technology improvements to better support threads.

Intel recently introduced just such improvements in hyper-threading technology on their Pentium 4 processor. The technology enables multi-threaded applications to execute threads in parallel. In the past, threads were split into multiple streams in order to enable multiple physical processors to execute them. This technology basically enables multi-threaded software to simultaneously execute its threads [Intel 2004]. In a hyper-threading enabled processor, certain sections outside of the main execution resources are duplicated; typically the sections that store the architectural state of the processor. This allows the processor to be seen as two logical processors by the host operating system. This allows the operating system to schedule two threads or processes simultaneously [Tian 2003]. A thread executing on a processor does not necessarily use all of the execution resources available on the processor. Hyper-Threading allows the processor to use these unused resources to execute a second thread [Wikipedia 2005].

Since the TIW timer threads are mutually exclusive, only one will be occupying the CPU at any time. This will allow another thread to be scheduled using idle CPU resources. Therefore, this thread may be spawned within the timer, and may be given the responsibility for the execution of the desired operations.

When one of the timer threads wants to execute, and the thread executing the desired operations is still running, the TIW timer threads may be executed at the same time thanks to hyper-threading, boosting the viability of the TIW timer.

The TIW timer provides a mechanism to implement a timer using software, without being too concerned about the underlying hardware. This suggests that the mechanism will be easily portable to processors that might be developed in the near future.

With demands on new hardware developments as they are at the moment, it seems as if multi-threaded applications could benefit even more. The following are some of the demands on new hardware systems:

- Greater business productivity
- Increase in the number of transactions processed
- Larger Workloads [Intel 2004].

To achieve the above, the processors do and will have to continue supporting the execution of multiple processes at once. In other words, multi-threaded applications will be used more and more and the TIW timer will still be a viable solution with new processor developments.

Microsoft and Intel have jointly developed a new timer called the High Performance Event Timer (HPET). This timer was designed specifically to measure 1ms intervals, without excessive deviation. Tests on the HPET by Microsoft engineers have determined that the HPET improves accuracy and system performance [Peng 2002]. When the HPET becomes widely available, all Win32 APIs will be ported and the underlying Windows code extended to take advantage of the new timer.

In future, the TIW timer may be extended to read the timestamp from the HPET rather than the ACPI clock counter, for greater accuracy.

6. CONCLUSION AND FUTURE WORK

As shown in section 3, the Win32 timers are unable to provide constant intervals that comply with a maximum deviation requirement of 500μs or less. However, a workaround has been found in the form of the thread induced waitable timer (TIW timer) discussed in section 4.

The TIW timer is capable of providing a constant interval with a maximum deviation recorded on a 1ms interval of 50 μs while consuming on average less than 10% of the available processing power (refer to section 4). The TIW timer is waitable and periodically fires an event when the specified interval elapses. Thus the TIW timer conforms to our requirements for a soft real-time timer given in section 2.3.

Future studies on the TIW timer is aimed at the following:

- determining the effect of CPU load on the TIW timer;
- determining the number of TIW timer instances that may be run reliably together on a single computer;
- quantifying the performance of the TIW timer at different frequencies;
- comparing the performance of the TIW timer on the Win32 platform and on a POSIX platform; and
- comparing the performance of the TIW timer and the HPET.

Preliminary studies have indicated that the TIW timer may be extended to provide an interval of as little as 500μs by using 4 threads instead of 2. The maximum deviation is still in the order of 50μs and the processor usage is still on average less the 10%. Future studies will determine if yet smaller intervals may be achieved by increasing the number of threads, possibly at the cost of consuming a larger percentage of the processor time.

The TIW timer is a flexible algorithm that conforms to the definition of a soft real-time timer. It provides a timer event at the specified frequency on the Win32 PC platform. The TIW timer is also positioned to take advantage of new processor technologies such as hyper-threading and the high performance event timer.

REFERENCES

Lin Chih-han, Chu Hao-hua, Nahrstedt Klara. 1998. 2ND USENIX WINDOWS NT SYMPOSIUM, Seattle August 1998. *A Soft Real-time Scheduling Server on the Windows NT*

Chu Hao-hua, Nahrstedt Klara. 1997. EUROPEAN WORKSHOP ON INTERACTIVE DISTRIBUTED MULTIMEDIA SYSTEMS AND TELECOMMUNICATION SERVICES, Darmstadt, Germany September 1997. *A Soft Real Time Scheduling Server in UNIX Operating System*"

Banachowski Scott, Brandt Scott. 2003. INTERNATIONAL WORKSHOP ON PARALLEL AND DISTRIBUTED REAL-TIME SYSTEMS. *Better Real-time Response for Time-share Scheduling*. (WPDRTS 2003), pp. 124–131, 2003.

Tian X, Chen Y.-K, Girkar M., Ge S., Lienhart R., Shah S. 2003. INT'L PARALLEL AND DISTRIBUTED PROCESSING SYMP. *Exploring the Use of Hyper-Threading Technology for Multimedia Applications with Intel OpenMP Compile.r*

Gill Chris, Levine David, Schmidt Douglas C. 2001. REAL-TIME SYSTEMS, THE INTERNATIONAL JOURNAL OF TIME-CRITICAL COMPUTING SYSTEMS. *The Design and Performance of a Real-Time CORBA Scheduling Service.* Special issue on Real-Time Middleware, Kluwer Academic Publishers, Volume 20, Number 2, March 2001.

Barbanov M. 1997. M.Sc THESIS, New Mexico Institute of Mining and Technoloy. *A Linux-Based Real-Time Operating System*

Abeni Luca, Lipari Giuseppe. 2002. PROCEEDINGS OF THE FOURTH REAL-TIME LINUX WORKSHOP, BOSTON, MA. *Implementing Resource Reservations in Linux*

Peng Jeff.T. 2002. WINDOWS HARDWARE AND DRIVERS CENTER. *Guidelines For Providing Multimedia Timer Support.* http://www.microsoft.com/whdc/system/CEC/mm-timer.mspx.

MICROSOFT DEVELOPERS NETWORK April 2005. *What is a Timer?* http://msdn.microsoft.com

Timmerman Martin, Monfret Jean-Christophe. 2002. DEDICATED SYSTEMS MAGAZINE 1997. *Windows NT a Real Time OS?* http://www.omimo.be/magazine/97q2/winntasrtos.htm
Intel. 2004. HARDWARE DESIGN. *Hyper-Threading Technology.* http://www.intel.com/technology/hyperthread 2004
Newcomer Joseph M. 2000. THE CODE PROJECT. *Time, the simplest thing*. http://www.codeproject.com/system/simpletime.asp
WIKIPEDIA 2005. *Real-Time.* http://en.wikipedia.org/wiki/real-time
WIKIPEDIA 2005. *Hyper-Threading*. http://en.wikipedia.org/wiki/Hyper-threading