Design and Implementation of a High Resolution Soft Real-Time Timer

by

Johannes Petrus Grobler

Submitted in fulfilment of the requirements for the degree
Magister Scientia
in the Faculty of Engineering, Built Environment and Information Technology
University of Pretoria
Pretoria
February 2006

ABSTRACT **7**

CHAPTER 1 .... **8**

1.1 What is a Timer? *8*
1.2 Timer Resolution *9*
1.3 Real Time *10*
1.4 Sub-Classification of Timers *11*
1.4.1 One-Shot Timers 12
1.4.2 Periodic Timers 12
1.4.3 Waitable Timers 12
1.4.4 Duration Measurement Timers 13
1.5 Analysis Methods *13*
1.6 Dissertation Layout *14*

CHAPTER 2 .... **15**

2.1 Current Hardware Based Timers *15*
2.1.1 8254 Programmable Interrupt Timer (PIT) 15
2.1.2 Real Time Clock (RTC) 16
2.1.3 Advanced Programmable Interrupt Controller (APIC) 16
2.2 High Resolution Hardware Counters *17*
2.2.1 Hardware Timestamp Counters 17
2.2.2 Loop Timer 18
2.2.2.1 Algorithm 18
2.2.2.2 Results 19
2.2.2.2.1 Loop Timer Performance (1 kHz) 19
2.2.2.2.2 Loop Timer Performance (50 Hz) 19
2.2.2.2.3 Loop Timer Performance (1 Hz) 20
2.2.2.2.4 CPU usage 20
2.2.2.3 Conclusion 21
2.3 WIN32 Timers *21*
2.3.1 System Timer 21
2.3.1.1 1 kHz Interval System Timer 22
2.3.1.2 50 Hz Interval System Timer 23
2.3.1.3 1 Hz Interval System Timer 23
2.3.1.4 Conclusion 24
2.3.2 Multimedia Timer 24
2.3.2.1 1 kHz Interval Multimedia Timer 26
2.3.2.2 50 Hz Multimedia Timer 27
2.3.2.3 1 Hz Interval Multimedia Timer 27
2.3.2.4 CPU Usage 28
2.3.2.5 Conclusion 28
2.4 POSIX Timer *29*
2.4.1 BSD Timers 29
2.4.1.1 BSD Timers (1 kHz Frequency) 30
2.4.1.2 BSD Timers (50 Hz Frequency) 30
2.4.1.3 BSD Timers (1Hz Frequency) 31
2.4.2 Conclusion 31
2.5 External Timers *31*
2.5.1 1 kHz Synchronisation Signal 32
2.5.2 50 Hz Synchronisation Signal 33
2.5.3 1 Hz Synchronisation Signal 33
2.5.4 Conclusion 34
2.6 Problem Statement *34*

CHAPTER 3 .... **36**

3.1 Development Programming Language *36*
3.2 Application Programming Interfaces (APIs) *36*

3.3 Operating Systems *37*
3.4 Process Priority *37*

CHAPTER 4 .... **39**

4.1 Timestamp Calculator *39*
4.1.1 Query Functions 39
4.1.2 Timestamp Calculation 41
4.1.2.1 Precision 41
4.1.2.2 Process of timestamp calculation 41
4.1.3 Class Description 42
4.2 Rejected Solutions *43*
4.2.1 WIN32 Sleep Timer (Non Multimedia) 43
4.2.1.1 Algorithm 43
4.2.1.2 Results 44
4.2.1.2.1 1 ms interval (1 kHz) 44
4.2.1.2.2 20 ms interval (50 Hz) 44
4.2.1.2.3 1 second interval (1Hz) 45
4.2.1.3 Conclusion 45
4.2.2 WIN32 Sleep Timer (Multimedia Timer) 46
4.2.2.1 Algorithm 46
4.2.2.2 Results 46
4.2.2.2.1 1 ms interval (1kHz) 46
4.2.2.2.2 20 ms interval (50 Hz) 47
4.2.2.2.3 1 second interval (1Hz) 48
4.2.2.3 CPU usage 48
4.2.2.4 Conclusion 49
4.2.3 Critical Section Timer 49
4.3 Thread Induced Waitable Timer *50*
4.3.1 Design 52
4.3.2 Algorithm 59
4.3.3 Results 59
4.3.3.1 TIW timer (1kHz Interval) 60
4.3.3.2 TIW timer (50 Hz Interval) 61
4.3.3.3 TIW timer (1Hz Interval) 62
4.3.4 An alternative TIW timer 63
4.3.4.1 TIW timer Alternative (50 Hz Frequency) 64
4.3.4.2 TIW timer Alternative (1Hz Frequency) 65
4.3.4.3 TIW timer vs TIW timer Alternative 66
4.4 Comparison *72*
4.4.1 WIN32 Timers 72
4.4.1.1 System Timer 72
4.4.1.1.1 1 kHz System Timer Comparison 73
4.4.1.1.2 50 Hz System Timer Comparison 73
4.4.1.1.3 1 Hz System Timer Comparison 74
4.4.1.2 Multimedia Timer 74
4.4.1.2.1 1 kHz Multimedia Timer Comparison 75
4.4.1.2.2 50 Hz Multimedia Timer Comparison 75
4.4.1.2.3 1 Hz Multimedia Timer Comparison 76
4.4.2 POSIX Timer 77
4.4.2.1 WIN32 TIW timer vs. POSIX Timer. 77
4.4.2.2 WIN32 TIW timer vs. POSIX TIW timer. 79
4.5 Benchmark (External Timer) *80*
4.5.1 1 kHz TIW timer vs. External Timer. 80
4.6 Timer Performance *82*

CHAPTER 5 .... **84**

5.1 Load Testing *84*
5.1.1 Loading on a single timer 84
5.1.1.1 Reasonable Load 84

5.1.1.2 Over Load ... 85
5.1.1.3 Conclusion ... 86
5.1.2 Multiple Timers ... 87
5.1.2.1 1 kHz, Two TIW timers running ... 87
5.1.2.2 1 kHz six TIW timers Running ... 88
5.1.2.3 1 kHz Five TIW timers Running ... 90
5.1.2.4 Conclusion ... 91
5.1.3 Other Multimedia Applications running with the TIW timer ... 91
5.1.3.1 1 kHz TIW timer with other Multimedia Application ... 91
5.1.3.2 50 Hz TIW timer with other Multimedia Application ... 92
5.1.3.3 1 Hz TIW timer with other Multimedia Application ... 93
5.1.3.4 Conclusion ... 93
5.2 Period Adjustment ... 94
5.2.1 4 Threads, 500 ?s Interval (2 kHz) ... 94
5.2.2 8 Threads, 250 ?s Interval (4 kHz) ... 96
5.2.3 16 Threads, 125 ?s Interval (8 kHz) ... 97
5.2.4 5 Threads, 400?s Interval (2.5 kHz) ... 98
5.2.5 Conclusion ... 99
5.3 Number of Threads ... 100
5.4 A real world application ... 103

CHAPTER 6 ... 106

REFERENCES ... 109

List of Figures

Figure 1-1: Timer Illustration ... 8
Figure 1-2: Countdown Timer Depiction ... 9
Figure 1-3: Margin of Error ... 10
Figure 1-4: Duration Measurement Timer ... 13
Figure 2-1: Loop Timer Performance (1 kHz) ... 19
Figure 2-2: Loop Timer Performance (50 Hz) ... 20
Figure 2-3: Loop Timer Performance (1 Hz) ... 20
Figure 2-4: Loop Timer CPU usage ... 21
Figure 2-5: 1 kHz System Timer ... 22
Figure 2-6: 50 Hz System Timer ... 23
Figure 2-7: 1 Hz System Timer ... 23
Figure 2-8: System Time Depiction ... 25
Figure 2-9: Ideal System Timer Depiction ... 25
Figure 2-10: Actual Multi-Media Timer Depiction ... 26
Figure 2-11: Multimedia Timer 1 kHz Frequency ... 26
Figure 2-12: Multimedia Timer 50 hz Frequency ... 27
Figure 2-13: Multimedia Timer 1 Hz Frequency ... 28
Figure 2-14: Multimedia Timer CPU Usage ... 28
Figure 2-15: BSD Timer (1 kHz) ... 30
Figure 2-16: BSD Timer (50 Hz) ... 30
Figure 2-17: BSD Timers (1 Hz) ... 31
Figure 2-18: 1 kHz Synchronisation Box ... 32
Figure 2-19: 50 Hz Synchronisation Box ... 33
Figure 2-20: 1 Hz Synchronisation Box ... 33
Figure 3-1: API Definition ... 36
Figure 4-1: Sleep Timer (Non Multimedia) (1 kHz) ... 44
Figure 4-2: Sleep Timer (Non Multimedia) (50 Hz) ... 45
Figure 4-3: Sleep Timer (Non Multimedia) (1 Hz) ... 45
Figure 4-4: Sleep Timer (Multimedia) (1 kHz) ... 47
Figure 4-5: Sleep Timer (Multimedia) (50 Hz) ... 47
Figure 4-6: Sleep Timer (Multimedia) (1 Hz) ... 48
Figure 4-7: Sleep Timer (Multimedia) CPU usage ... 48
Figure 4-8: Critical Section Timer CPU usage ... 49

Figure 4-9: Critical Section Timer ... 50
Figure 4-10: Thread Induced Waitable Timer Timing diagram ... 51
Figure 4-11: TIW timer Class Diagram ... 58
Figure 4-12: TIW timer (1 kHz Interval) ... 60
Figure 4-13: 1 kHz TIW timer CPU usage ... 61
Figure 4-14: TIW timer 1 kHz Worst Case ... 61
Figure 4-15: TIW timer (50 Hz Interval) ... 62
Figure 4-16: 50 Hz TIW timer CPU usage ... 62
Figure 4-17: TIW timer (1 Hz Interval) ... 63
Figure 4-18: 1 Hz TIW timer CPU usage ... 63
Figure 4-19: TIW timer Alternative (50 Hz Interval) ... 64
Figure 4-20: 50 Hz TIW timer Alternative CPU usage ... 65
Figure 4-21: TIW timer Alternative (1 Hz Interval) ... 65
Figure 4-22: 1 Hz TIW timer Alternative CPU usage ... 66
Figure 4-23: Normal TIW timer 500 Hz ... 67
Figure 4-24: 500 Hz Normal TIW timer CPU Usage ... 67
Figure 4-25: Alternative TIW timer (500 Hz) ... 68
Figure 4-26: 500 Hz Alternative TIW timer CPU usage ... 68
Figure 4-27: 333.33 Hz Normal TIW timer ... 69
Figure 4-28: 333.33 Hz Normal TIW timer CPU usage ... 69
Figure 4-29: 333.33 Hz Alternative TIW timer ... 69
Figure 4-30: 333.33 Hz Alternative TIW timer CPU usage ... 70
Figure 4-31: Normal TIW timer (250 Hz) ... 71
Figure 4-32: Normal TIW timer (250 Hz) ... 71
Figure 4-33: Alternative TIW timer (250 Hz) ... 71
Figure 4-34: Alternative TIW timer (250 Hz) CPU usage ... 72
Figure 4-35: System Timer vs. TIW timer (1 kHz) ... 73
Figure 4-36: System Timer vs. TIW timer and TIW timer Alternative (50 Hz) ... 74
Figure 4-37: System Timer vs. TIW timer and TIW timer Alternative (1 Hz) ... 74
Figure 4-38: Multimedia Timer vs. TIW timer (1kHz) ... 75
Figure 4-39: Multimedia Timer vs. TIW timer (50Hz) ... 76
Figure 4-40: Multimedia Timer vs. TIW timer (1Hz) ... 76
Figure 4-41: 1kHz TIW timer vs POSIX Timer ... 77
Figure 4-42: 50Hz TIW timer vs. External Timer ... 78
Figure 4-43: 1Hz TIW timer vs. External Timer ... 78
Figure 4-44: 1 kHz POSIX TIW timer ... 79
Figure 4-45: 50Hz POSIX TIW timer ... 79
Figure 4-46: 1 Hz POSIX TIW timer ... 80
Figure 4-47: 1 kHz TIW timer vs. External Timer ... 81
Figure 4-48: 50 Hz TIW timer vs. External Timer ... 81
Figure 4-49: 1 Hz TIW timer vs. External Timer ... 82
Figure 5-1: 1kHz Reasonable Load ... 85
Figure 5-2: TIW timer Reasonable Load CPU usage ... 85
Figure 5-3: 1kHz Overload ... 86
Figure 5-4: 1kHz Overload CPU Usage ... 86
Figure 5-5: Timer A 1 kHz Test ... 87
Figure 5-6: Timer A CPU Usage ... 88
Figure 5-7: Timer B 1 kHz Test ... 88
Figure 5-8: Timer B CPU usage ... 88
Figure 5-9: Best Timer with 6 Timers Running ... 89
Figure 5-10: Worst Timer with 6 Timers Running ... 89
Figure 5-11: Best Timer with 5 Timers Running ... 90
Figure 5-12: Worst Timer with 5 Timers Running ... 90
Figure 5-13: 1 kHz other Multimedia Application ... 92
Figure 5-14: 1 kHz TIW timer with other Multimedia application CPU Usage ... 92
Figure 5-15: 50 Hz other Multimedia Application ... 93
Figure 5-16: 1 Hz other Multimedia Application ... 93
Figure 5-17: 2 kHz TIW timer ... 94
Figure 5-18: 2 kHz TIW timer – Worst Case ... 95

Figure 5-19: TIW timer 2 kHz CPU usage 95
Figure 5-20: 4 kHz TIW timer 96
Figure 5-21: TIW timer 4 kHz CPU usage 96
Figure 5-22: 8 kHz TIW timer 97
Figure 5-23: TIW timer 8 kHz CPU usage 98
Figure 5-24: 2.5kHz TIW timer 98
Figure 5-25: TIW timer 2.5kHz CPU usage 99
Figure 5-26: 2 kHz TIW timer Timing Diagram 100
Figure 5-27: 2.5 kHz TIW timer Timing Diagram 100
Figure 5-28: 4kHz TIW timer Timing Diagram 101
Figure 5-29: 2kHz TIW timer Timing Diagram – Wrong number of threads 102
Figure 5-30: 2 kHz TIW timer, 5 threads CPU usage 102
Figure 5-31: 2kHz TIW timer Timing Diagram – Wrong number of threads 103
Figure 5-32: 3 kHz TIW timer 103
Figure 5-33: TIW Timer 1 hour: Final Timestamp 104
Figure 5-34: TIW Timer 1 kHz: One Hour 105

List of Tables

Table 1-1: Timer Classification 12
Table 2-1: Timer Resolution Comparison 35
Table 4-1: Timer Calculator Precision 41
Table 4-2: System Timer vs. TIW timer 73
Table 4-3: Multimedia Timer vs. TIW timer 75
Table 4-4: POSIX Timer vs TIW timer 77
Table 4-5: External Timer vs. TIW timer 80

Abstract

There are several timing mechanisms on presently available commercial operating systems. Two operating system platforms that immediately come to mind are the Microsoft Windows environment (the WIN32 platform) and its UNIX-based counterpart, Linux (with its POSIX standard). The timing mechanisms under these operating systems are adequate for use in conventional multimedia applications currently run on these platforms. However, the requirements of such applications are not as stringent within a real-time environment.

The goal of this dissertation was to determine if it would be possible to find a workaround for applications where current timing mechanisms in the WIN32 and POSIX environments do not meet the requirements of real-time. Before a proposed workaround is presented, a clarification is given as to what is meant by the notion of a timer. Attention is also given to the fact that its accuracy is quantified in terms of its resolution. It is acknowledged that real-time extensions to both the Windows and Linux operating systems exist. However, it was decided to find a solution without such assistance.

Real-time is also defined and sub-classified into hard- and soft real-time, differentiating environments that have precise constraints (hard real-time) on timing as opposed to environments where demands on accuracy and efficiency are less stringent (soft real-time). The timer that was ultimately implemented had to conform to the latter form of real-time.

This dissertation therefore aims to provide a solution in a soft real-time environment. The current timing mechanisms are discussed and their performance is quantified. Their deficiency in measuring a reliable periodic interval of 1 ms is highlighted. From this qualification of timers stems the requirements for the soft real-timer timer. The areas in which improvement is sought are stated.

The design and implementation of a soft real-time timer that meets these requirements is presented and its performance at various frequencies is quantified. A comparison is given between the timer and the existing timing mechanisms as well as comparison between its implementation under both Windows and Linux. Additionally, the viability of the proposed timer compared to a proven hard real-time timer is presented.

Finally it is recognised that a timer would not be useful if it was not effective in a practical environment. Consequently, the timer's performance under the same load that it would experience in a practical soft real-time environment is investigated as well. The dissertation concludes with a discussion on the compatibility of this timer with expected advances in future Central Processing Unit (CPU) technologies.

Supervisor: Prof D G Kourie

Department of Computer Science

Magister Scientia

Chapter 1

Background

In this chapter, introductory material on the broad theme of this dissertation is given: designing and implementing a high-resolution soft real-time timer. Various key notions are explained. Thus, in section 1.1, the notion of a timer is explained. Since the value of a timer is dependent on its resolution, and since the study is in quest of a high-resolution timer, section 1.2 encapsulates the meaning of timer resolution. The next area of concern in the study is the notion of real-time, which is explained in section 1.3. Many different types of timers can, in principle, be used to measure real-time. Section 1.4 surveys these. Section 1.5 then introduces the analysis method that will be employed in this study. Finally, section 1.6 points the reader ahead to the remainder of the work, indicating the themes of chapters.

1.1 What is a Timer?

In everyday day life, time is measured using a clock, and the same principle applies to computer systems. Computer systems apply a number of mechanisms to keep track of time, be they clocks on the hardware itself or software time mechanisms that are based on these clocks – specifically on the operating system clock [Peng 2002] (refer to section 1.2). The operating system running on the computer system uses the operating system clock to determine processor usage and when timers should fire [Yoav et al. 2003]. This raises the question: *"What, precisely, is a timer?"*

According to the MSDN [MSDN 2003], a timer is *"…an internal routine that repeatedly measures a specified interval, in milliseconds."* It is a mechanism that allows software events to be scheduled efficiently [Li et al. 2003].

An application may require time to be segmented into consecutive intervals of equal size. Specified events may be executed within each segment. For example, on an aircraft, 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) periodically at a constant interval, determine the aircraft's position and then carry out necessary operations based on this information. The flight computer 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 flight computer executes the next required set of computations.

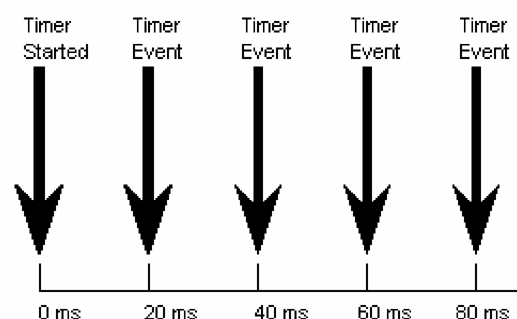

Figure 1-1: Timer Illustration

Therefore, a timer may be thought of as a clock that measures a constant interval repeatedly, firing an event after each interval has elapsed. This is illustrated in Figure

1-1. The figure shows the operation of a timer over a period of 80 ms. Every 20 ms, an event is fired to which the application using the timer may react.

A further example is an application that displays a countdown in intervals of one second. The countdown, for the sake of argument, is 10 seconds. The application will require a timer that fires an event every second, upon which the application displays *"x SECONDS"* (where x is the number of seconds remaining), until 10 seconds have been counted down. This is depicted in Figure 1-2.

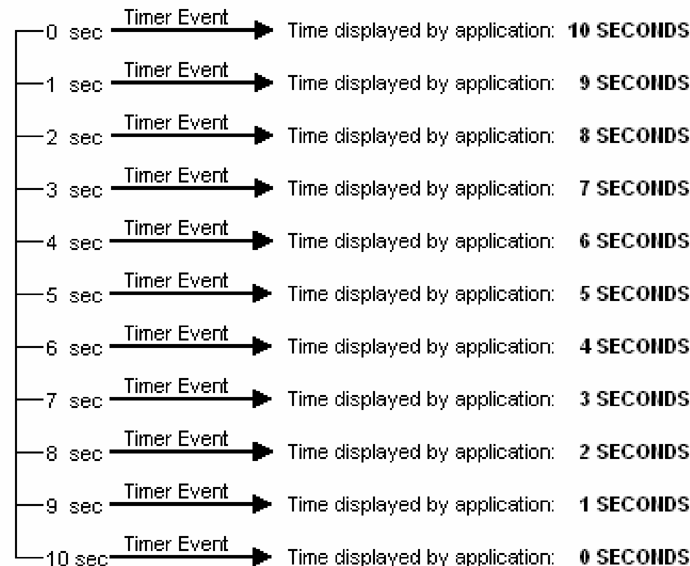

Figure 1-2: Countdown Timer Depiction

The reason these timers are based on the operating system clock is to reduce the overhead involved in processing a separate interrupt for every timer that is created [Yoav et al. 2003]. However, this mechanism of measuring time has its drawbacks, as will be illustrated in section 1.2 and section 2.3.2.

1.2 Timer Resolution

According to [Yoav et al. 2003], a computer system employs the service of two clocks – one is hardware based and the other is governed by the operating system. The frequency of the operating system clock is not predetermined as is the case with the hardware clock. Instead, the decision as to which frequency is deemed most suitable is made during the operating system's design.

The most common frequency of the operating system clock in use at present is 100Hz. This is the frequency used in the Linux, BSD, Solaris and WIN32 operating systems [Yoav et al. 2003]. In other words, 100 clock ticks per second (or 100Hz) are registered and processed or one clock tick every 10 ms. Therefore in these systems, the smallest interval of time that may be "accurately" measured by the operating system clock is 10 ms. As explained in section 2.3.1, this interval under the WIN32 operating system is actually 10 – 15 ms.

The notion of accuracy needs some explanation and qualification. As with any physical system, measurement is inevitably subject to a degree of inaccuracy – i.e. it is seldom 100% accurate. The above claim that 10 ms is the smallest interval of time that may be accurately measured by the operating clock systems under discussion, should be seen in this context. In general, there is a need to characterize a timer

clock's accuracy. This is done in terms of two quantities: the margin of error that is manifested, and the resulting timer clock's resolution.

The margin of error may be explained as referring to the maximum extent to which the actual occurrence of a timer event deviates from the desired time at which the timer event should take place [Lamie 2003]. Figure 1-3, shows a time-line where a timer event should be fired every 10 ms. Instead, two actual timer events are shown below the time-line. In the first case, the event takes place 10 ?s after it should have, and in the second case it takes place 10 ?s before it should have.

This means that the first actual time interval depicted is 10.010 ms, or 10 ?s longer than it should be. The duration of the second actual time interval is 9.990 ms. This is 10 ?s shorter than it should be, if the next timer event should be 20 ms from the start. However, if the next timer event should be 10ms after the *last actual timer event* then it is 20 ?s shorter than it should have been.

For the purposes of this document, the desired time for the next timer event will be deemed to be a constant amount of time after the last timer event was supposed to have happened. Under this assumption then, the margin of error observed in Figure 1-3 is between -10 ?s and +10 ?s.

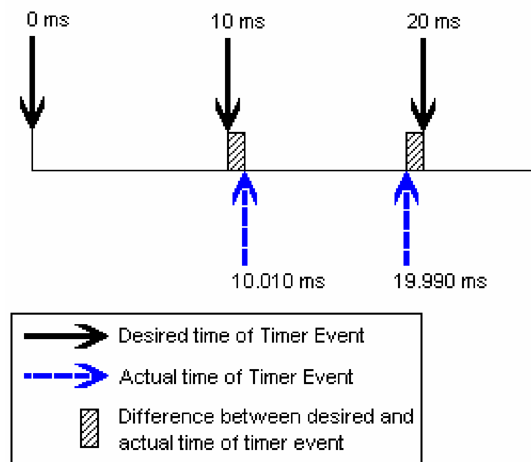

Figure 1-3: Margin of Error

The resolution of the timer will accordingly be denoted as *desired interval ? margin of error*[1]. In the above example, the timer has a resolution of 10 ms ? 10 ?s.

1.3 Real Time

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 is determined by the logical correctness of the operations as well as by the time it takes to execute these operations [Gopalan 2001]. Since the execution time of an operation plays such an important role, a real-time system may be regarded as a system that is dependent on accurate and predictable time measurement. Therefore, when a real-time system is designed and implemented, the reliability of both the hardware and the [1]software of the system have to be guaranteed [Barr 1999].

[1] Note that this is a practical and not a statistical notation. The *margin of error* is an indication of the worst possible error in interval size and not a representation of the standard deviation. This entails that the actual interval may for example exceed the *desired interval* by the *margin of error* and never be less than the *desired interval.*

Janno Grobler
jannogrobler@gmail.com
M.Sc Computer Science
University of Pretoria

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 computer (refer to section 2.5)

The PC based Win32 platform was not developed with hard real-time as a feature [Newcomer 2000]. This applies to Windows NT, Windows 2000 and Windows XP. These operating systems were designed as general purpose or networking platforms [Timmerman et al. 2002]. 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 dissertation, as the aim is to investigate the Win32 and POSIX standard timers and endeavour to find a solution to their inadequacies (more on this section 3.3). Results that illustrate these shortcomings are presented in chapter 2.

[Timmerman et al. 2002] define a real-time system as one that '…responds in a timely predictable way to unpredictable external stimuli arrivals' and also distinguishes between hard- and soft real-time.

A hard real-time system does not allow a task to exceed the maximum allowable delay. In other words, if an operation should complete within a certain time, a hard real-time system expects it to meet its deadline. It is assumed that if this does not happen, then a system failure has occurred [Barabanov 1997]. Furthermore, the cost of such failure is regarded as 'infinitely" high [Barr 1999].

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. An example of this is a video-conferencing system, where although it is preferable that every frame of the video is displayed, it is acceptable if a frame or two is occasionally missed [Barabanov 1997].

Soft real-time systems are often used in conjunction with hard real-time. 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 et al. 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. An example of this is covered by [Gill et al. 2001] in their article "The Design and Performance of a Real-Time CORBA Scheduling Service".

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.

1.4 Sub-Classification of Timers

There are various types of timers whose use is dictated by the application at hand. This could be either in a hard or soft real-time context. A single application can create multiple timers that measure different interval durations. These timers could either measure a single interval or measure intervals periodically [Friesen 2001]. Of course

this is completely dependant on the design and function of the software. Here, a classification of such timers is given.

However, before this sub-classification is presented, it is necessary to clarify the meaning of the concept of a "timestamp":

"A timestamp is a representation of the amount of time elapsed since a pre-determined moment in time" [Perkins 2003].

1.4.1 One-Shot Timers

A one-shot timer is a timer that measures a single interval. After the interval expires, the timer terminates [Austin Group 2004]. The logical application of such a timer is an operation that has to be executed once after a fixed period of time [Sridhar 2003].

1.4.2 Periodic Timers

A periodic timer measures a constant interval repeatedly and fires an event after each interval expires [Austin Group 2004]. After the end of such an interval, the timer immediately starts to measure the same interval duration again. This process it continuous for as long as the timer is required to run [Sridhar 2003].

1.4.3 Waitable Timers

A "waitable" object is a synchronization object that is placed in a suspended state until a specified interval has elapsed at which point its state is set to signalled. The waitable object could also be configured to wait on an I/O device connected to for example the serial port. A process can therefore "wait" until the timer object's state becomes signalled using it as a queue to for example initiate a routine [Henderson 2003].

There are two types of waitable timers that can be created: manual-reset and synchronization. Unlike a normal timer, multiple processes can "wait" for a single waitable object's state to transition to "signalled". This is summarized in Table 1-1.

Table 1-1: Timer Classification

Object	Description
Manual-reset object	A waitable object whose state remains signalled until a waiting process acknowledges the signal and resets it.
Synchronization object	A waitable object whose state remains signalled until a waiting process acknowledges the signal. The object will reset automatically.

An object of either type can be used in the implementation of a periodic timer. If a waitable object is incorporated into an algorithm that repeatedly sets it to a suspended state, the resulting behaviour of such an application is comparable to that of a periodic timer.

Another process is able to wait for the object to become signalled, execute the operations for which it requires periodic scheduling and wait for the object to become signalled again. In the case of the manual-reset object, the process resets the state of the object before waiting for it to become signalled again.

1.4.4 Duration Measurement Timers

A duration timer is used to determine the length of an operation. Such a timer could typically be used to quantify the amount of time required to execute certain operations within a software application. A timestamp may be taken at the beginning and another at the end of the operation. The difference between the two time measurements is the duration of the operation.

In Figure 1-4, a duration measurement timer is requested to provide a timestamp, namely T0. After the operation, another timestamp is taken – T1. Therefore the duration of the operation equates to T1-T0.

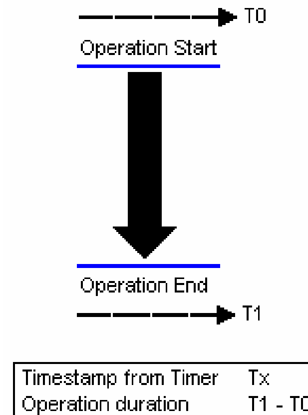

Figure 1-4: Duration Measurement Timer

1.5 Analysis Methods

Since chapters 2 and 3 through 5 present results attesting to the performance of the timers under scrutiny within the particular chapters, it is necessary to provide an explanation of the methods followed to perform the analysis. As will be seen, the systems under scrutiny are:

- Loop Timer
- System Timer
- Multimedia Timer
- POSIX Timer
- External Timers
- WIN32 Sleep Timer (Non Multimedia)
- WIN32 Sleep Timer (Multimedia Timer)
- Thread Induced Waitable Timer

An application is developed for each timer, in terms of which the following happens whenever the timer fires: a) the application reads a timestamp value; and b) the timestamp value recorded in dynamic memory for storage in an output file after the conclusion of the timer's run. This is done repeatedly, so that measurements are taken over a specified period of time. These timestamp values are subsequently utilised to calculate, in each case, the timer's actual interval size for each of its firings (activations). These values are plotted against the timestamps, converted into a time value.

Also, the maximum and minimum interval durations are recorded – These are used to calculate the timer resolution (see section 1.2).

For each timer, these statistics are recording for the following frequencies:

? 1 kHz (1 ms intervals)

? 50 Hz (20 ms intervals)

? 1 Hz (1000 ms intervals)

The reason why these frequencies were selected was because an external real-time timing source was available (see section 2.5) that could be used to benchmark the real-time accuracy of the various timers (as is done in section 4.5). This external real-time timer provides hardware signals at these frequencies.

The resource consumption of each timer is quantified. The minimum, maximum and average is recorded and the resource usage is presented in graphical form.

1.6 Dissertation Layout

The remainder of the dissertation is outlined in this section. Chapter 2 focuses on the background of existing timers, from the hardware timers to software timers currently available under open source and commercial operating systems. The performance and accuracy of these timers are also discussed, and existing timing mechanisms will be shown to be deficient.

Accordingly, among the goals of this research is to determine whether a work around could be devised to compensate in the areas where the current timers are lacking; thus leading into to culmination of the chapter – the problem statement (section 2.6).

To ensure the reader understands the process that culminated in the implementation of an alternative timer, chapter 3 focuses on design-, implementation decisions made, as well as analysis methods used. The main reasoning is to enhance the readers grasp of the solution in chapter 4. Chapter 4 will first focus on rejected solutions and show how they eventually suggested an alternative solution. The performance and accuracy thereof is also under scrutiny.

Further testing of the proposed solution is presented in chapter 5, aimed at exploring whether the solution is viable and practical for application in a real-time environment.

Finally, the conclusion is provided in chapter 6.

Chapter 2

Currently Available Timing Mechanisms

This chapter looks at timing mechanisms currently available under the common open source and commercial operating systems – Linux, Unix and Windows (WIN32 platform). Note that the performance results of the timers presented in this chapter, have been derived in conformance with the analysis methods explained in section 1.5. A question that could be raised is whether timers on the computer's hardware exist that may be used effectively as a real-time timer. This is addressed in section 2.1. Special timing mechanisms do exist on current computer hardware. This is the subject of section 2.2.The remaining timers that require investigation are the software timers provided by common operating systems, specifically those provided by the WIN32 and POSIX standards. Sections 2.3 and 2.4 are devoted to these timers. Finally, to ensure real-time timing, such implementations on consumer level hardware usually incorporate an external reliable hardware source of known and extremely high accuracy. This will be discussed in section 2.5. However, the main goal of this chapter is to provide a statement of the problem that this research sought to solve. This is provided in section 2.6.

2.1 Current Hardware Based Timers

The timers currently in use on current x86 computer platforms and the reason why they were not considered for a timing source in a real-time environment are discussed in this subparagraph. These timers are commonly used as the basis for the software timers under the operating system implementation. There are three such timers, each of which is now discussed in turn.

2.1.1 8254 Programmable Interrupt Timer (PIT)

In 1981, IBM introduced the 8254 Programmable Interrupt Timer, with a resolution of one millisecond [Peng 2002]. It is essentially a software programmable counter/timer device designed to address control problems in microcomputer system design [Intel 1993]. Provided by the PIT is a 16-bit counter that is capable of handling clock inputs at 10MHz. The timer provides accurate time delays under software control, eliminating the need for such loop timers as the one in section 2.2.2. The programmer has the ability to program the 8254 for the desired delay. After the desired time has elapsed, the 8254 will interrupt the CPU. The software that wishes to use the timer is simply tasked with waiting for such interrupt.

On consumer level Intel microcomputers, the 8254 interfaces to the system is an array of I/O ports: three counters and a fourth is an interface to a control register that is used for mode management. The counters are independent and may operate in different modes simultaneously. Possible modes are:

? *Interrupt on Terminal Count (Mode 0)*

  An initial count is loaded into the counter register. The counter is decremented on each clock input, and an interrupt is fired when the counter equals 0. At this point a new counter may be loaded.

  It is typically used as an event counter.

? *Hardware Re-triggerable One-Shot (Mode 1)*

Like Mode 0, an initial count is loaded into the counter register, with an interrupt when this counter is decremented to 0. However, this countdown may be repeated without a new counter being loaded into the register

? *Rate Generator (Mode 2)*

An initial count is loaded into the counter register, decremented and an interrupt fired as with Mode 0 and Mode 1. However, when the interrupt is fired, the count is loaded again automatically and the process is repeated. Therefore Mode 2 is periodic.

Mode 2 is typically used to generate a Real Time clock interrupt.

? *Square Wave Mode (Mode 3),*
Software Triggered Strobe (Mode 4) and
Hardware Triggered Strobe (Re-triggerable) (Mode 5)

These 3 modes are similar to mode 2 and produce a periodic timer in each case, but with different implementations on the hardware level. Details are beyond the scope of this text, but are available in the 8254's datasheet [Intel 1993].

Mode 3 is typically used for baud rate calculation.

Since I/O operations through the IO ports the timer uses to interface to the system are expensive, the PIT is not viable for generating small intervals, and the problem is aggravated by the fact that the clock inputs are handled at 10 MHz, i.e. every 100 ms. This implies that the smallest interval that the timer could measure reliably is in the order of 100 ms.

2.1.2 Real Time Clock (RTC)

The real-time clock (RTC) was shipped for the first time in 1984, in addition to the 8254 [Peng 2002]. The RTC performs two main functions, namely:

? Keeping track of time

? Storing system data, even when powered down [Intel 1995].

The RTC is sourced from a 32.768 kHz crystal and runs off a 3V battery. The RTC may also be used to generate wake up calls for the system, up to 30 days in advance.

The RTC provides 3 interrupt services:

? A time of day alarm (range 1 second to 30 days)

? Periodic interval timing (interval range 120 ?s to 500 ms)

? End of update cycle notification

Using the 32.768 kHz crystal, the RTC's time is updated every second, thus keeping track of seconds, hours, days, weeks, months and years with daylight savings capabilities.

As is the case with 8254 timers in section 2.1.1, the RTC has to be accessed via an I/O port that is generally mapped to IRQ 8 on the computer's motherboard. The high cost of I/O communication renders the RTC incapable of being used for a reliable 1 ms interval on a software level [Peng 2002].

2.1.3 Advanced Programmable Interrupt Controller (APIC)

The Advanced Programmable Interrupt Controller (APIC) was designed by Intel specifically for use in multi-processor environments to solve inter-processor interrupt routing issues. According to [Wikipedia 2006], it consists of two parts:

- The local APIC (LAPIC) – this is integrated into the CPU system. Should it be a multi processor environment, an LAPIC exists for every CPU. Not only does the LAPIC manage all external interrupts for the processor its part of, it is also able to generate and accept interrupts. These interrupts form the basis for the LAPIC interval timer.
- The Input/Output APIC (IOPIC)

  This is used throughout the system's peripheral buses. It routes interrupts it receives from these buses to the LAPIC via the use of a redirection table.

Spurious interrupts also occur within the APIC that may be mistakenly construed to be a genuine interrupt [Intel 1997]. It suffers from severe jitter in its interrupt latency, in other words the time that elapses from the moment an interrupt is generated to the moment that it is serviced varies and is not predictable. The consequence of this is that the jitter is exhibited in the APIC interval timer as well [Wikipedia 2006].

2.2 High Resolution Hardware Counters

This section is concerned with hardware counters located on current x86 central processing units (CPUs). First the counters themselves will be explained in section 2.2.1. This is followed by the description of a software implementation that makes use of these counters to calculate timestamps in sections 2.2.2 through 2.2.2.3. Section 2.2.2 introduces the loop timer and states the need for the timer. Section 2.2.2.1 presents the algorithm used to the implement such a timer. The performance results of this timer are presented in section 2.2.2.2. The conclusion is given in 2.2.2.3 as well as the motivation for not relying on the loop timer as a final implementation of a real-time software timer.

2.2.1 Hardware Timestamp Counters

The following are counters available on modern hardware:

- Power Management Timer (PM Timer)

  Modern personal computer hardware provides a counter on the Advanced Configuration and Power Interface (ACPI) [Intel 1999]. The clock is also called the Power Management Timer (or the PM clock). The ACPI requires a mechanism to measure the ACPI system idle time. This PM Timer manages a counter incremented at a fixed frequency of 3.579545 MHz. The current value of the counter is stored in a register that may be accessed programmatically. According to the ACPI specification [Intel 1999], this register is referred to as the PM_TMR_BLK register.
- Timestamp Counter (TSC)

  The Timestamp Counter is a 64-bit counter on the CPU, supported since the Pentium family of processors [Dongarra et al. 2001]. The counter is set to 0 on every hardware reset of the computer and incremented every processor clock cycle, making it independent of processor speed. It is thus very fine-grained with the accuracy limited to the CPU frequency. This counter is guaranteed to monotonically increase, except for the 64-bit wraparound, which is several thousand years in the Pentium family of processors.

These counters 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 timestamp value.

A timer was implemented that makes use of the counter generated by the ACPI clock or Timestamp Counter (TSC) in order to generate a timestamp at set intervals. This approach is described in section 2.2.2.

2.2.2 Loop Timer

With the availability of the high-resolution hardware counters described in section 2.2.1, it is possible to develop a timer with a maximum deviation within acceptable limits (refer to section 2.6). To illustrate this, a loop timer was designed and implemented as discussed below.

The loop timer queries a high-resolution counter until a suitable interval has expired. At this point the timer event is triggered, and the process restarts.

Since the value read from such a counter is an incremental value indicating the number of times a clock cycle has occurred, the actual timestamp (in time units such as milliseconds, not in clock cycles) has to be calculated. This calculation is discussed in section 4.1.

2.2.2.1 Algorithm

The following pseudo code is a representation of the loop timer algorithm:

```
?previous := current timestamp
?current  := ?previous
While the timer is running
      While (?current - ?previous < X ?s)
            ?current := timestamp from the
                        high precision counters
      End While
      ?previous := ?current
      Fire the timer event
End While
```

Algorithm 1 - Loop Timer Algorithm

The algorithm commences with the calculation of an initial timestamp, designated $?_{previous}$ and assigns it to the current timestamp variable, designated $?_{current}$. The timer enters a loop for the duration of its execution.

Another loop tests the time elapsed against the required interval designated X. This is accomplished by taking the difference between the current timestamp calculated from the high-resolution counter and the previous timestamp, $?_{previous}$. While this value is less than the required interval value X ?s, the next timestamp is read and stored in $?_{current}$.

When the difference between $?_{current}$ and $?_{previous}$ is greater equal to X ?s, the current timestamp $?_{current}$ is saved in $?_{previous}$. The timer event is fired, and the process restarts unless the timer should terminate.

2.2.2.2 Results

As is clear from Algorithm 1, at no point is a wait instruction used to prevent the timer from consuming all available CPU resources. Therefore the timer's CPU usage equates to 100% of the available resources leaving none available to do anything else. The accuracy, however, is extremely high.

The loop timer was developed in C++, based on reading Timestamp Counter (TSC) and the Power Management (PM) clock counter values (The WIN32 API determines which counter to use). It can be set to fire at various frequencies.

The accuracy of the results for different frequencies is presented in the following sections. Note that frequency in the current context refers to the rate at which a timer event is fired, i.e. to the inverse of the timer interval, measured in appropriate time units.

2.2.2.2.1 Loop Timer Performance (1 kHz)

The results for a frequency of 1 kHz are presented in Figure 2-1. The desired interval size should thus be as close to 1 ms (or 1000 ?s) as possible. As is clear from the figure, the timestamps measured is extremely accurate, only deviating once significantly from the desired interval duration over a period of approximately 32 seconds (32000 ms).

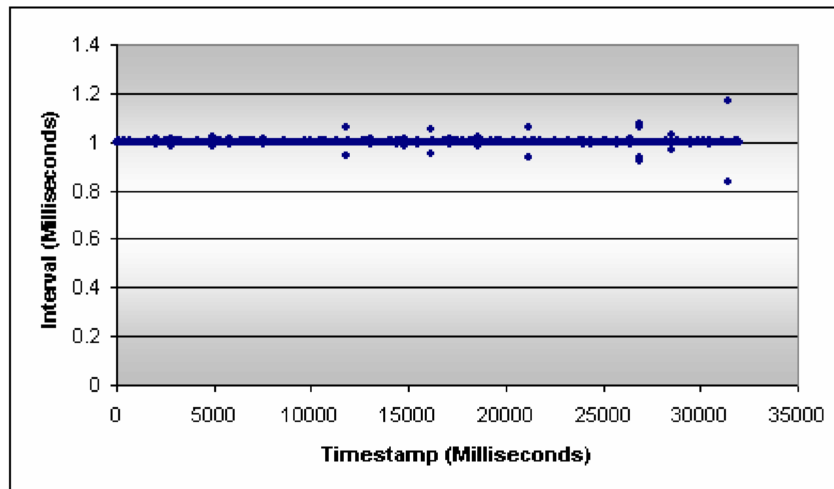

Figure 2-1: Loop Timer Performance (1 kHz)

For this timer, the maximum interval duration is 1.167 milliseconds (1167 ?s). The minimum interval duration is 1 ms (1000 ?s). Thus a resolution of 1 ms ? 167 ?s was achieved over the time span for which measurements were taken.

2.2.2.2.2 Loop Timer Performance (50 Hz)

The results for a frequency of 50 Hz are presented in Figure 2-2 and therefore the desired interval size should be as close the 20 ms (20000 ?s) as possible. Although the figure only shows the results for the first 10 seconds, the experiment was conducted over a period of 60 seconds. The reason for showing a subset of the results in the graph is to increase readability.

For the 50 Hz timer, the maximum interval duration is 20.002 ms (20002 ?s). The minimum interval duration is 19.998 ms (19998 ?s). Accordingly, the resolution equates to 20 ms ? 2 ?s. This is an extremely accurate measurement. However, this

achievement is overshadowed by the timer's consumption of all available CPU resources.

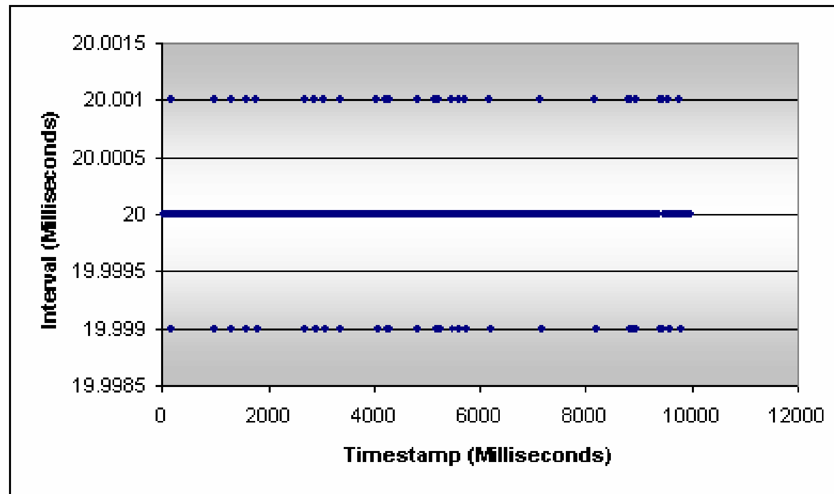

Figure 2-2: Loop Timer Performance (50 Hz)

2.2.2.2.3 Loop Timer Performance (1 Hz)

The results over a period of 60 seconds for a frequency of 1 Hz (desired interval size: as close to one second as possible) are presented in Figure 2-3.

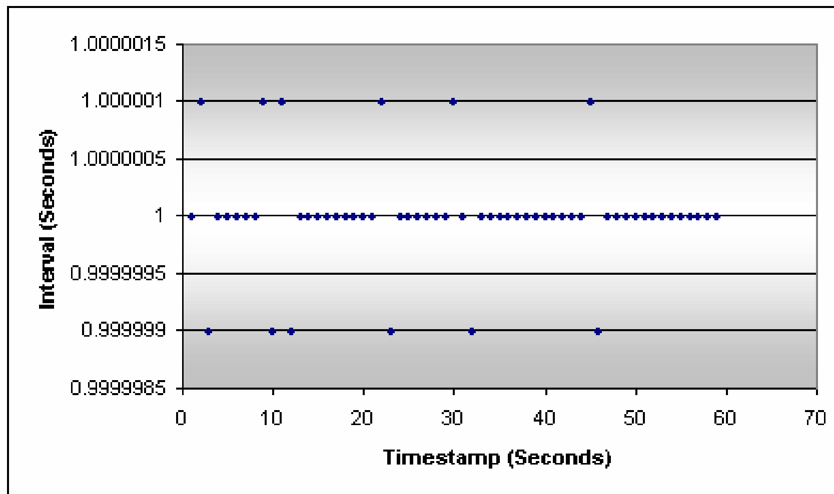

Figure 2-3: Loop Timer Performance (1 Hz)

With the maximum interval duration of 1.000001 seconds and minimum interval duration of 0.999999 seconds, the maximum deviation on this interval is 1 ?s, which is again extremely accurate. Accordingly, the resolution equates to 1 second ? 1 ?s.

2.2.2.2.4 CPU usage

The CPU usage was the same for all the cases discussed in sections 2.2.2.2.1 to 2.2.2.2.3 is depicted in Figure 2-4. The average CPU resources consumed by the timer were 83.59%. However, the total CPU usage was 100% throughout the operation of the loop timer. Since the timer was not the only process running on the

system on at the time, it is assumed that the other processes were consuming the CPUs resources.

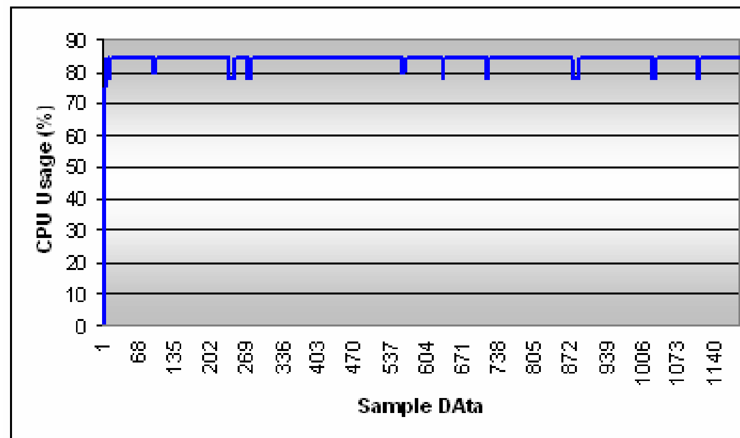

Figure 2-4: Loop Timer CPU usage

2.2.2.3 Conclusion

From the foregoing it is clear that, in terms of accuracy; the loop timer is extremely accurate. The reason for rejecting the solution is it consumes all available processing time on the CPU.

It is apparent from the implementation of the loop timer in section 2.2.2.2 that the polling of the counter consumes 100% of the processor resources available to it. This is unacceptable since no resources would be available for an application that encapsulates such a loop timer to execute other operations.

For more information, see 4.1.

2.3 WIN32 Timers

The timers discussed in sections 2.1 and 2.2 are all based on the hardware itself and can be used independently of the operating system running on the computer. However, the WIN32 operating system does provide two timing mechanisms of its own, albeit based on the hardware discussed in the previous sections: a system clock and a multimedia timer. From the point of view of the developer, these timers may be used on a software level via API (refer to 3.2) calls provided by the operating system, instead of accessing the hardware directly.

This section discusses these mechanisms and reports on various experiments in which timers of various interval lengths were built based on these timing mechanisms. It is also highlighted why these timers were deemed inappropriate for real-time application.

2.3.1 System Timer

In the article *Guidelines For Providing Multimedia Timer Support* [Peng 2002] explains the inner workings of the WIN32 timing system. According to the article, the WIN32 platform uses a periodic clock interrupt to keep track of time, trigger timer objects and manage thread execution. At boot time, this clock interrupt interval varies between approximately 10 ms and 15 ms. The result is therefore that the clock interrupt is updated every 10 ms to 15 ms, depending on the system. For the purpose of this discussion, 10 ms will be assumed [Abeni et al. 2002].

On receipt of a clock interrupt, the WIN32 Operating System is mainly tasked with the following operations:

- Update the timer tick count. The primary purpose of the timer tick count is to give effect to the abstract notion of time that Windows uses to keep track of time of day and to keep track of a thread quantum's time. Therefore a timer tick constitutes 10 ms to 15 ms in elapsed time.
- Check the timer objects expiration. The operating system checks if the interval of any timer object has expired. Should this be the case, the system schedules a Timer Deferred Procedure Call (DPC). A DPC is specified by the application that requested the timer object. Such a DPC specifies an event that has to occur when a specified interval has elapsed. Such timer objects are used by the operating system to track deadlines and to signal applications when a deadline is reached.

When a WIN32 operating system boots up, the current value 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 clock can be used to determine the duration of intervals when constructing a system timer.

Such a system timer may be constructed using WIN32 API calls in languages such as C++ and Visual Basic [MSDN 2003]. A pointer to a function that contains the routines that have to be executed at each timer event is given to an API call that creates a timer object. An application was developed to create such a timer object via the WIN32 API and specified to fire timer events at intervals of 1ms, 20ms and 1000ms. The results are presented in the following sections.

2.3.1.1 1 kHz Interval System Timer

Figure 2-5 depicts the performance of the System timer with a desired frequency of 1 kHz. Therefore the interval should be as close to 1ms as possible. Although the experiment was conducted over a period of sixty seconds, the figure only shows the first 200 milliseconds to increase the readability of the graph.

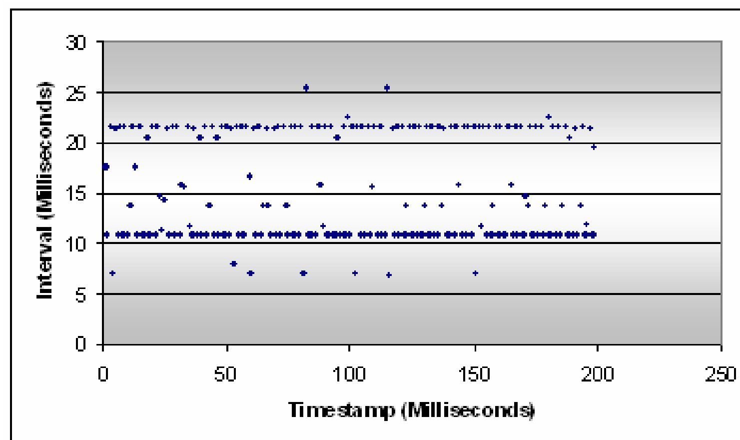

Figure 2-5: 1 kHz System Timer

As can be seen, the timer does not provide the required 1 ms second interval. Instead it provides a maximum interval of 25.431 ms and a minimum interval of 5.881 ms. The timer's maximum deviation is thus 25.431 ms – 1 ms = 24.431 ms with the resolution equating to 1 ms ? 24.431 ms.

2.3.1.2 50 Hz Interval System Timer

Figure 2-6 depicts the performance of the System timer with a desired frequency of 50 Hz. Again, for the sake of readability; the figure only shows the first 10 seconds of the 60 second period over which the experiment was conducted. The required interval should be as close to 20 ms as possible.

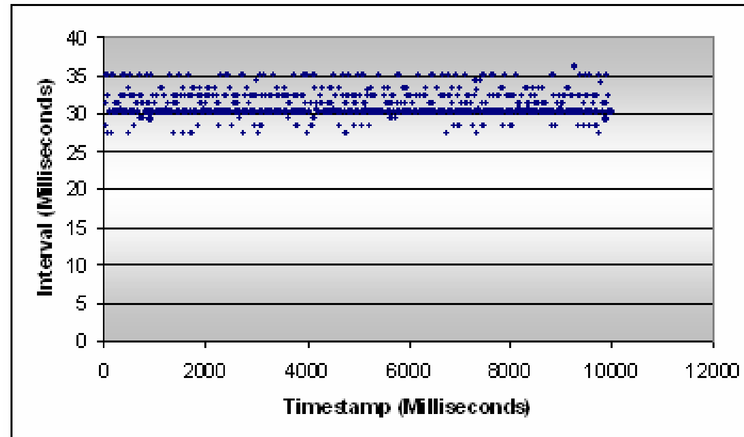

Figure 2-6: 50 Hz System Timer

During the first 10 seconds depicted in the graph, the timer did not manage to produce an interval smaller than 25 ms. Over the entire period the timer was running, it managed to generate a minimum interval of 27.291 ms. The recorded maximum interval was 36.151 ms. The timer's maximum deviation is thus 36.151 ms –20 ms = 16.151 ms and the resolution 20 ms ? 16.151 ms.

2.3.1.3 1 Hz Interval System Timer

The period of 60 seconds shown in Figure 2-7 depicts the performance of the System timer with a desired frequency of 1Hz. The required interval should therefore be in the close vicinity of 1s.

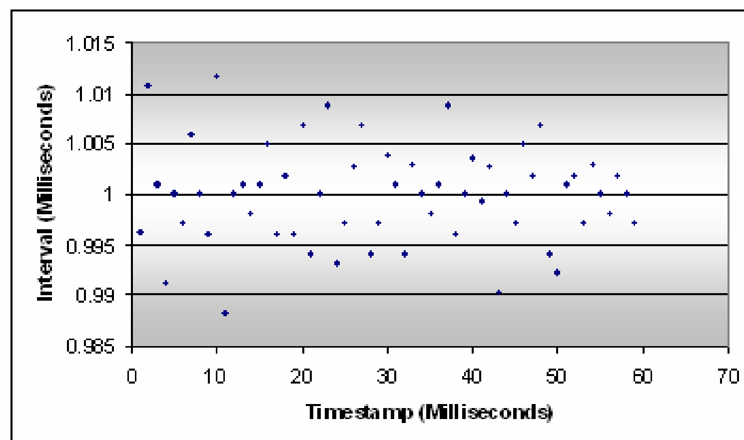

Figure 2-7: 1 Hz System Timer

As can be seen, the timer seldom provides 1-second interval and deviates from this required duration in general. The maximum interval recorded was 1.011667 s and a minimum interval 0.988237s. The timer's maximum deviation is thus 1s-0.988237s = 0.011763 s. The timer's resolution is therefore 1 s ? 11.763 ms.

2.3.1.4 Conclusion

The performance of a system timer based on the system clock as measured above was enough to disqualify it for use as a soft real-time timer. It did not achieve the granularity to enable it to even come close to measuring the required interval.

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 to 15ms [Peng 2002]. The System timer is dependant on the message queue of the application that spawned it. Every WIN32 application that opens a Windows frame requires such a queue to process messages from the operating system. When the operating system determines that the system timer's interval has elapsed, a message that instructs it to fire the timer event is place in this message queue.

This could explain both the maximum and minimum intervals recorded for 1 kHz timer in section 2.3.1. Due to the 10 ms to 15 ms interrupt the operating system uses to update the system time, the expected interval size should be between 10ms and 15ms. However, the range recorded was 5.881ms to 25.431ms. If the system timer starts an interval ? ms after the interrupt to the operating system, 10 ms – ? ms remain before the operating system inspects the timer again. This explains how the minimum interval could be less than 10ms.

The maximum interval can be explained if the periodic interrupt occurred to the operating system at an interval of 15 ms. For example, the timer interval commences at the start of the 15 ms gap between interrupts. When the next interrupt comes around, the timer would have been waiting for ± 14 ms for the message that enables it to fire the timer event. Since the message from the operating system that causes the interval to expire is placed in the message queue of the application that encapsulates the timer, it is possible for the message to be stuck in this queue for an undetermined amount of time. If the message gets stuck in this queue for the next 10 ms, the timer will experience and interval of ± 25 ms.

It is clear that the system timer cannot compete with the loop timer (in section 2.2.2) since it does not accurately measure out a 1 ms interval. Furthermore, when the frequency was decreased, the accuracy failed to improve to a point where the result was satisfactory. The system timer is inaccurate to such an extent when compared to the loop timer that the amount of CPU resources that it consumes is irrelevant. Although it uses considerably less CPU resources than the loop timer, the system timer is so inaccurate that it could never be used to measure an interval of one millisecond reliably. Therefore the CPU usage of the system timer is not presented in this dissertation.

2.3.2 Multimedia Timer

Multimedia timer services allow applications to schedule timer events with the greatest resolution (or accuracy) possible for the x86 personal computer (PC) platform. These multimedia timer services allow one to schedule timer events at a higher resolution than through other conventional timer services, such as the system timer (refer to 2.3.1).

The drawback of the system timer is obvious as the minimum interval duration that can accurately be measured is comparable to the interval of the clock interrupt that the operating system uses to update the system time. In applications where accurate timing is required to schedule events, these events could be late by 10 ms or more. Applications typically associated with smaller than 10 ms intervals are related to multimedia implementations.

For example a multimedia application may require an event such as audio playback to occur every 23 ms. The measurement of this interval is depicted in Figure 2-8.

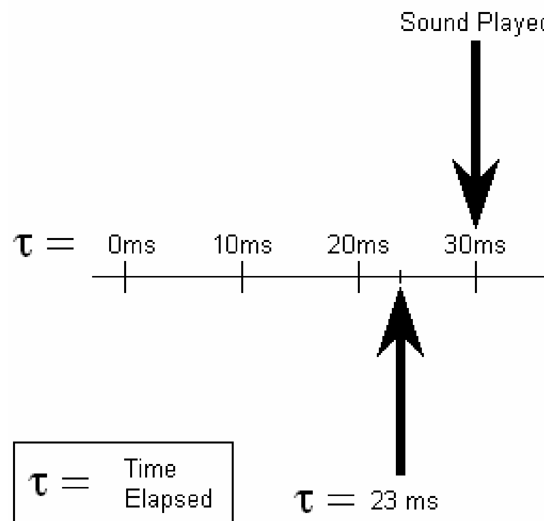

Figure 2-8: System Time Depiction

The multimedia application will configure a timer object to indicate when an interval of 23 ms has elapsed. Figure 2-8 shows the accumulation of elapsed time, as it is measured by the operating system. It is clear from the figure that although timer object interval should elapse after 23 ms, this is only determined after the third timer tick, or 30 ms. Therefore, the sound is played 7 ms late.

This is a clear drawback of the timing architecture of the WIN32 operating system. A solution to this problem was to lower the interval of the clock interrupt that the operating system uses to update the system time. This resulted in the multimedia timer.

To overcome the deficiency of the of the system timer, the clock interrupt should ideally be 1ms and not 10ms, as depicted in Figure 2-9. When the 23rd timer tick is received, 23 ms will have passed and the sound is played at exactly the right time.

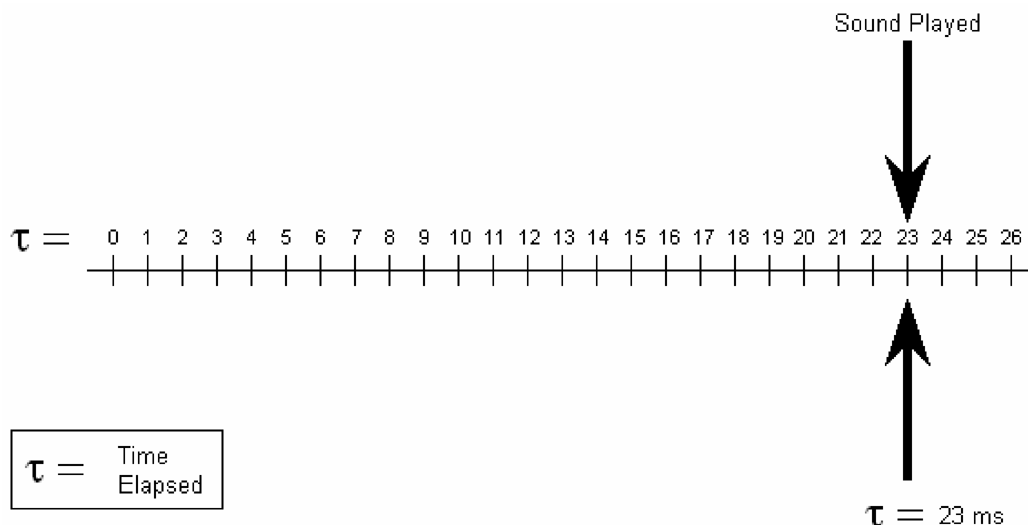

Figure 2-9: Ideal System Timer Depiction

However, according to [Peng 2002], Microsoft has found that the impact of lowering the clock interrupt to 1ms degrades system performance significantly, to the extent of not being worth the cost, specifically in terms of cache consistency and power management. Lowering it to 2ms however, has negligible effect. Therefore, in the multimedia application, the sound will be played within 1ms of the time that it is supposed to be heard, as depicted in Figure 2-10.

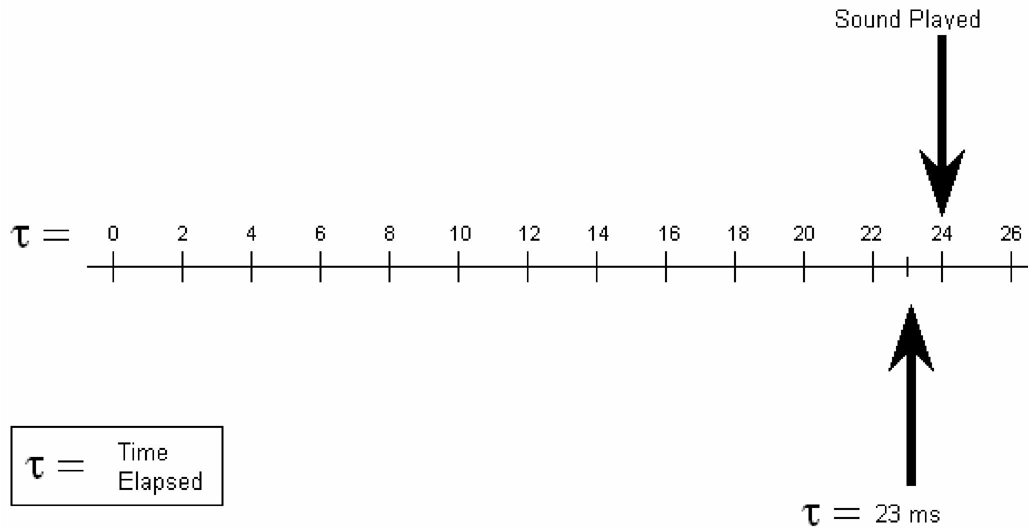

Figure 2-10: Actual Multi-Media Timer Depiction

Using a periodic clock interrupt is a common concept in contemporary operating system design and results in weighing increased accuracy against degrading system performance [Yoav et al. 2003]. The result still remains that the WIN32 platform provides interval accuracy with a variance of ?1ms, as will be seen below.

2.3.2.1 1 kHz Interval Multimedia Timer

The multimedia timer was executed over a period of 60 seconds and the minimum and maximum interval durations were recorded.

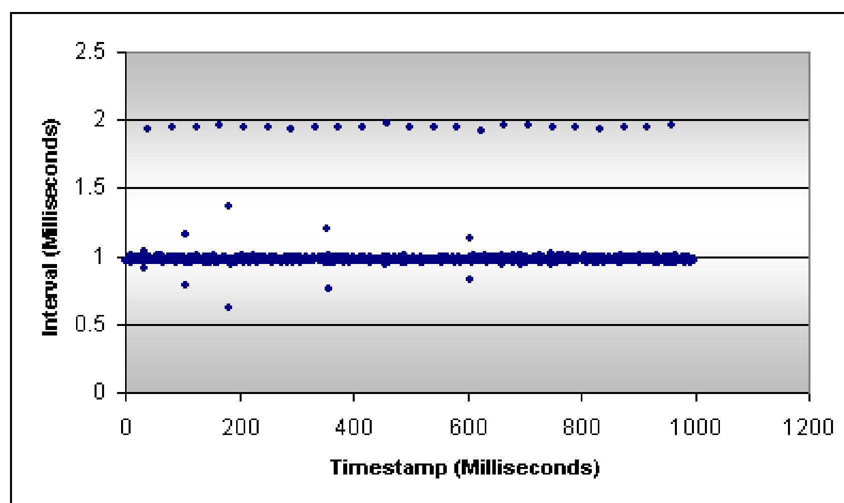

Figure 2-11: Multimedia Timer 1 kHz Frequency

Janno Grobler
jannogrobler@gmail.com
M.Sc Computer Science
University of Pretoria

Figure 2-11 illustrates the interval duration of the multimedia timer the first second. The performance of the timer over the remaining 59 seconds was similar to its performance in the first second. The frequency was set to 1 kHz (i.e. 1ms intervals).

From the figure it is clear that the timer stays within the range 1 ms to 2 ms. The maximum interval recorded was 1.983 ms and the minimum interval 0.503 ms. The timer thus exhibits a maximum deviation of 983?s and the resolution is therefore 1ms ? 0.983 ms.

Note that out of the 60000 intervals recorded in this experiment (there are a thousand one millisecond intervals within a second), the interval was greater than 1.5 ms 1439 times. The interval duration of 1.5 ms is significant since it exceeds the required interval size by half the length of the desired interval. Therefore, 2.398% of the intervals deviated from the desired interval of one millisecond by half the desired inter or more (500 ?s in this case).

2.3.2.2 50 Hz Multimedia Timer

The multimedia timer was configured generate intervals of 20 ms (a frequency of 50 Hz) and executed over a period of 60 seconds. Only the first 5 seconds are shown in Figure 2-12.

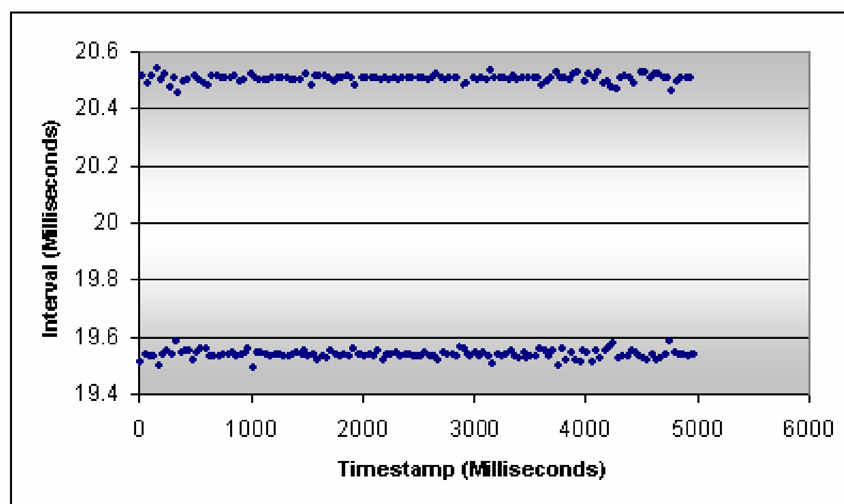

Figure 2-12: Multimedia Timer 50 hz Frequency

Figure 2-12 clearly illustrates the interval duration of the multimedia timer stays within the range 19.5 ms and 20.5 ms. This remains true for the 55 seconds not shown on the graph. The maximum interval was recorded at 20.613 ms and the minimum at 19.444 ms, which constitutes a resolution of 20 ms ? 613 ?s.

Of the 3000 intervals recorded (remember that there are fifty 20 ms intervals in a second), 1062 intervals deviated from 20 ms by 500 ?s and more. In other words, 35.4% of the time the multimedia timer deviated from 20 ms by 500 ?s and more.

2.3.2.3 1 Hz Interval Multimedia Timer

The final frequency tested was 1Hz, in other words, intervals of one second. Figure 2-13 illustrates the interval duration of the multimedia timer over a period of approximately 60 seconds. Over this period, the maximum interval recorded was 1.000139 s and the minimum interval 1.000069 s. The timer thus exhibits a maximum deviation of 139 ?s, which is extremely accurate. The resolution of 1 s ? 139 ?s

confirms the multimedia timer is capable of generating intervals of one second reliably.

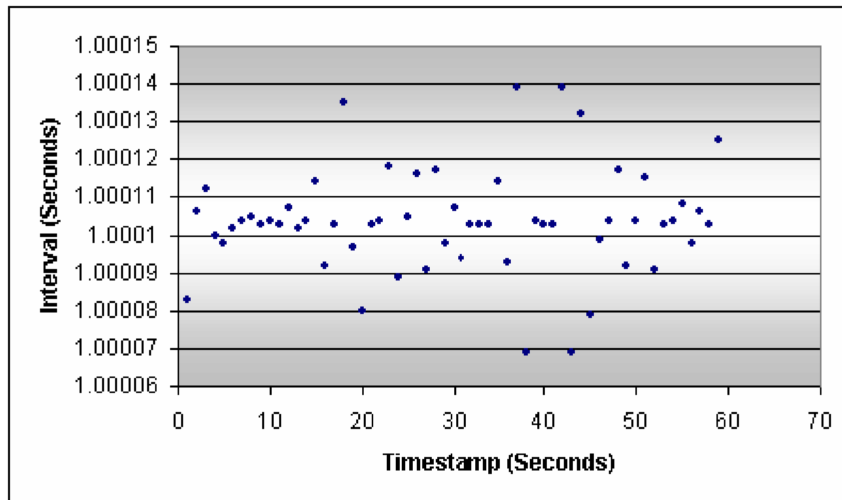

Figure 2-13: Multimedia Timer 1 Hz Frequency

2.3.2.4 CPU Usage

The multimedia timer consumes the very little in terms of CPU resources. In all the cases presented in sections 2.3.2.1 to 2.3.2.3, the multimedia timer consumed the same amount of resources. The CPU usage is shown in Figure 2-14.

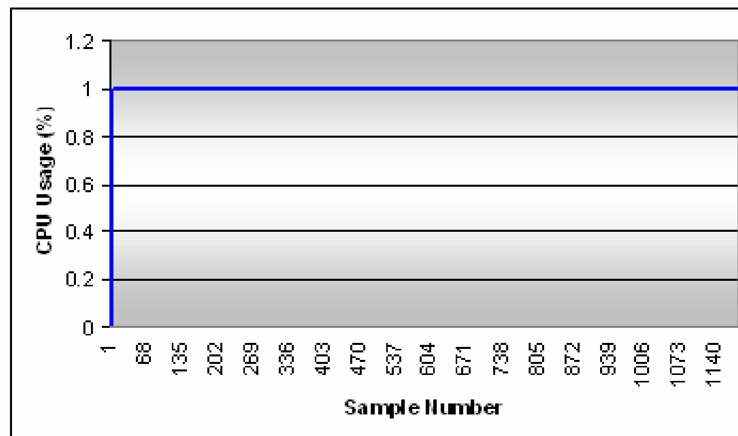

Figure 2-14: Multimedia Timer CPU Usage

The average amount of CPU resources consumed by the multimedia timer was 0.9957%. Therefore, the multimedia timer consumes a very small portion of the available processing power, especially compared to the 83.59% of the loop timer (refer to section 2.2.2). Therefore, the application that encapsulates such a multimedia timer has ± 99.0043% of the CPU resources to its disposal, depending on other the processes running on the system.

2.3.2.5 Conclusion

As was stated in section 2.3.2, Microsoft tests have determined that lowering the interval at which the system time is updated to 2 ms (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 1 ms is used [Peng 2002]. It was for this reason that the clock update interval for the multimedia timer was kept at 2 ms.

Although it was determined that the multimedia timer is capable of generating a 1 ms interval (refer to section 2.3.2.1), 2.398% of those intervals deviated from the required interval by more than 500 ?s. In the case of the 50 Hz multimedia timer (refer to section 2.3.2.2), this percentage was increased to 35.4 %. It does however deliver a reliable performance for an interval of one second (refer to section 2.3.2.3). However, the multimedia timer uses no more that 1% of the available CPU resources, as long as the clock update interval for the multimedia timer is kept at 2 ms [Peng 2002].

Although the results clearly show that the multimedia timer outperforms the loop timer in terms of resource consumption and the system timer in terms of accuracy (refer to section 2.3.1), it is less accurate than the loop timer. The multimedia timer is able to measure an interval of one millisecond with a maximum deviation of ± 1 ms whereas the loop timer was able to do it reliably to 200 ?s.

2.4 POSIX Timer

Section 2.3 focussed on software timers provided under the WIN32 platform. The POSIX standard for UNIX based operating systems such as Linux, UNIX, Solaris and AIX [Beal et al. 2003] provide timing mechanisms of their own. :

? BSD Timers:

  The BSD timers are interval timers with a granularity of around 10 ms. This seems similar to the system timer (refer to section 2.3.1) and it is expected that the BSD timers is similar to its WIN32 counterpart. However, the result exhibited by such a timer is presented nonetheless.

? IEEE 1003.1 Real-Time Timers – These timers are supposed to have the ability to measure intervals with a maximum deviation in the order of microseconds. However, IEEE 1003.1 is a real-time extension to UNIX, and therefore not included in the standard Linux kernel distributions. The IEEE 1003.1 standard is an extension to a UNIX base operating system, also known as the Real-Time Extension, or Real-Time Linux. However, as is the case with the real-time extensions to the WIN32 platform (refer to section 1.3 and also section 3.3) the Real-Time Linux timers are beyond the scope of this dissertation.

The results of these timers performance are presented below.

2.4.1 BSD Timers

These timers are included with the normal Linux kernel distributions. Three types of timers are available:

? A timer that is able to measure intervals. When the interval expires, a signal is sent to the application that encapsulates the timer as a notification.

? A timer that quantifies the processor time used by the timer.

? A profiling timer that measures the processor time used by the timer in addition to the time the processor spends on system calls related to the timer.

Therefore, the BSD timer capable of measuring intervals is under investigation in this section. An application was developed that initialises and executes such a timer. The results are presented below.

2.4.1.1 BSD Timers (1 kHz Frequency)

The results for a frequency of 1 kHz are presented in Figure 2-15. The figure shows the performance of the timer over a period of 32 seconds. It is clear that the performance of the BSD timer differs from that of the WIN32 system timer discussed in section 2.3.1.1.

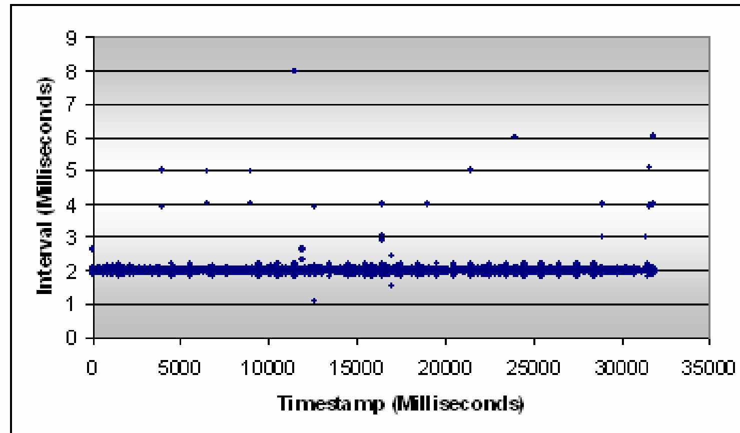

Figure 2-15: BSD Timer (1 kHz)

The minimum interval duration recorded was 1.070 ms and the maximum 7.976 ms. Compared to the system timer, this is a vast improvement, however, its performance is surpassed by that of the multimedia timer. The figure shows that the BSD timer was able to measure an interval close the one millisecond. Over a period of 60 seconds, the timer was able to do this once.

The number of intervals that were greater than 1.5 ms equates to 99.99633% of the intervals measured. Accordingly the timer exhibits a resolution of 1 ms ± 7.976 ms.

2.4.1.2 BSD Timers (50 Hz Frequency)

Figure 2-16 shows the performance of the timer over a period of sixty seconds. The frequency of the timer set to 50 Hz and was supposed the measure intervals of 20 ms. The minimum interval duration recorded was 20.885 ms and the maximum 28.014 ms.

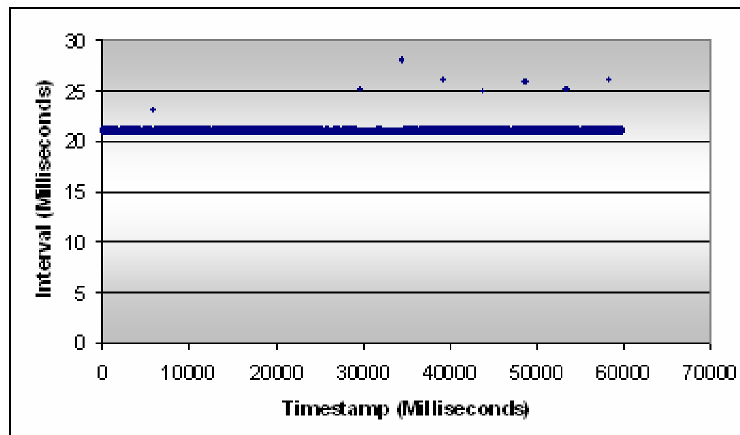

Figure 2-16: BSD Timer (50 Hz)

Janno Grobler
jannogrobler@gmail.com
M.Sc Computer Science
University of Pretoria

Therefore the resolution of the timer in this case is 20 ms ± 8.014 ms. Again the BSD timer outperforms the system timer in that it is able to generate an interval of 20 ms within a smaller margin of error.

However, the multimedia timer, whose resolution was 20 ms ? 613 ?s (refer to 2.3.2.2), surpasses the BSD timer. As is clear from the figure, the timer was not able to generate an interval smaller than 20.885 ms. Accordingly, the required interval duration is exceeded by more that 500 ?s 100% of the time.

2.4.1.3 BSD Timers (1Hz Frequency)

With the frequency set 1 Hz (presented in Figure 2-17), the minimum interval duration recorded was 1.000830 seconds and the maximum 1.011517 seconds. The timer was run for a period of 60 seconds, as shown in the figure.

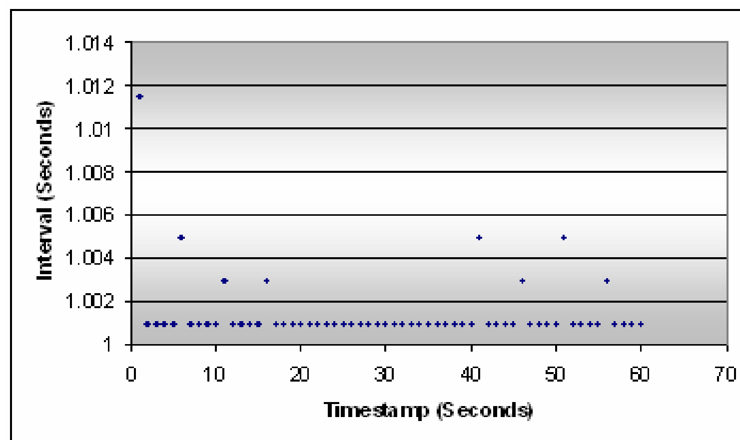

Figure 2-17: BSD Timers (1 Hz)

Therefore a consistent one second interval was not achieved. The desired interval size should be as close the one second as possible. The resolution of the timer is 1 s ± 11.517 ms.

On this occasion, the performance of the BSD timer is comparable to the 1 s ± 11.763 ms of the system timer (refer to section 2.3.1) at a frequency of 1 Hz. However, as was the case with the previous frequencies, the multimedia timer whose resolution was 1 s ? 139 ?s (refer to section 2.3.2.3), outperforms the BSD timer.

2.4.2 Conclusion

Section 2.4.1 shows that the BSD is capable of better performance than the system timer, but is surpassed by the multimedia timer. Since the loop timer outperforms the multimedia timer in terms of accuracy, it exhibits better performance than the BSD timer as well. Since a timer with better accuracy exists and the maximum deviation of the timer is in the order of milliseconds, the CPU usage of the BSD timer is not presented (as was the case with the system timer in section 2.3.1). Even if the BSD timer put the CPU under less strain than the multimedia timer (which is difficult considering the average CPU usage of the multimedia timer is 0.9957%), the multimedia timer is a better timing mechanism since it is more accurate.

2.5 External Timers

Hard real-time systems that rely heavily on precise timing constraints will typically be found in, for example, avionics systems [Newcomer 2000]. Such systems use high-resolution hardware timers as the timing source. As mentioned in section 1.3, such

hard real-time systems have to be thoroughly tested. These testing systems are often based on commercial operating systems such as Windows 2000 and XP.

To circumvent the inadequacies of WIN32 timers that were exposed in 2.3 above, external timers have been developed that deliver the required high-resolution timing mechanism to the computer. Since these devices are external to the PC, the communication is facilitated via the serial port.

Such an external timer device is designed to generate signals at a fixed and reliable frequency based on an oscillating quartz crystal. These signals are transmitted using the control line signals provided by the serial port technology. These include:

- RLSD – receive-line-signal-detect
- CTS – clear-to-send
- DSR – data-set-ready.

A common nominal signal frequency is 1 kHz, although these devices may be programmed to generate signals at other frequencies as well. Since more than one control signal may be transmitted via the control lines, such devices often transmit signals at different frequencies on different control lines simultaneously.

A typical configuration is a device that emits signals at 1 kHz (one millisecond intervals), 50 Hz (20 millisecond intervals) and 1 Hz (one second intervals).

An application was developed to catalogue the time at which the control signals are received. This application incorporates the waitable objects discussed in section 1.4.3. A waitable object is configured to repeatedly wait for the signals from the external timer. The timestamps at which these signals are received is recorded and accordingly, the results are presented in section 2.5.1 through 2.5.3.

2.5.1 1 kHz Synchronisation Signal

Figure 2-18 illustrates the elapsed time between the signals from the synchronisation box over a period of approximately 30 seconds where the interval frequency was set to 1 kHz (i.e. 1 ms intervals).

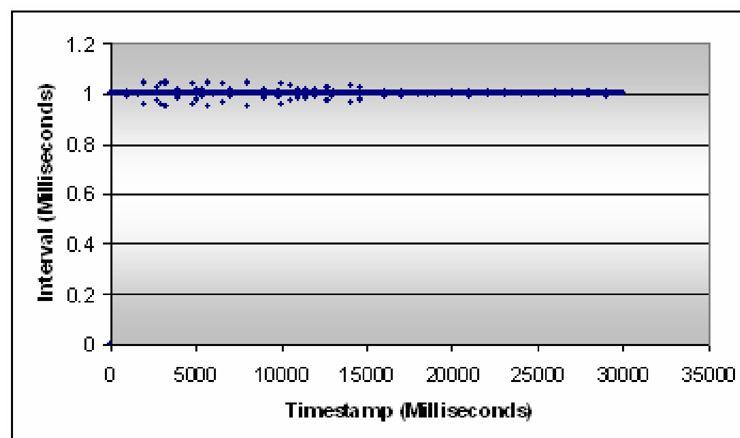

Figure 2-18: 1 kHz Synchronisation Box

Over the period of 30 seconds the minimum interval recorded was 0.953 ms and the maximum interval 1.047 ms. The timer thus exhibits a margin of error of 47 ?s. Since the crystal on which the signal is based oscillates at a constant interval, it is safe to assume that the deviation is due to delays induced by the operating system. However, a timer with a resolution of 1 ms ? 47 ?s is very accurate. It outperforms

the loop timer whose resolution was 1 ms ± 167 ?s (refer to section 2.2.2.2.1) for a frequency of 1 kHz.

2.5.2 50 Hz Synchronisation Signal

With the frequency set to 50 Hz (i.e. 20 ms intervals), the minimum interval recorded was 19.999 ms and the maximum 20.001 ms. This is illustrated in Figure 2-19 and depicts a period of approximately 60 seconds.

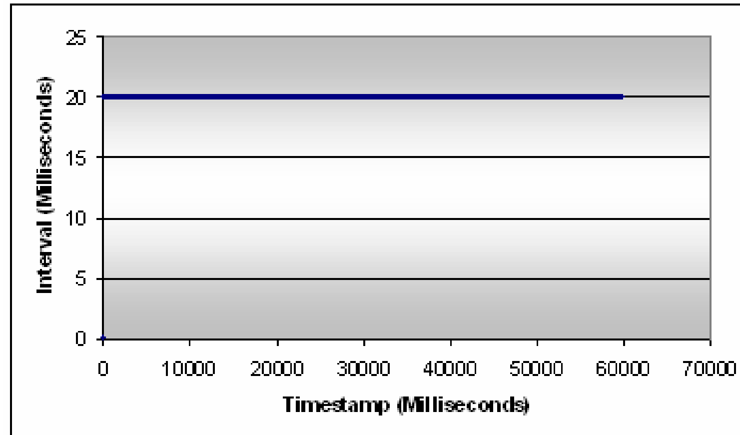

Figure 2-19: 50 Hz Synchronisation Box

In this instance, the performance of the loop timer is comparable to the signal from the synchronisation box. Therefore, the resolution exhibited was 20 ms ± 1 ?s. The loop timer's resolution was 20 ms ± 2 ?s for a frequency of 50 Hz. However, reading the signal from the serial port consumes virtually no CPU resources, whereas the loop timer consumes nearly 84% (refer to section 2.2.2.2.2).

2.5.3 1 Hz Synchronisation Signal

Figure 2-20 illustrates a period of seconds in which the 1 Hz (i.e. 1s intervals) from the synchronisation box. The minimum interval recorded was 0.999999 seconds and the maximum 1.0000000 seconds.

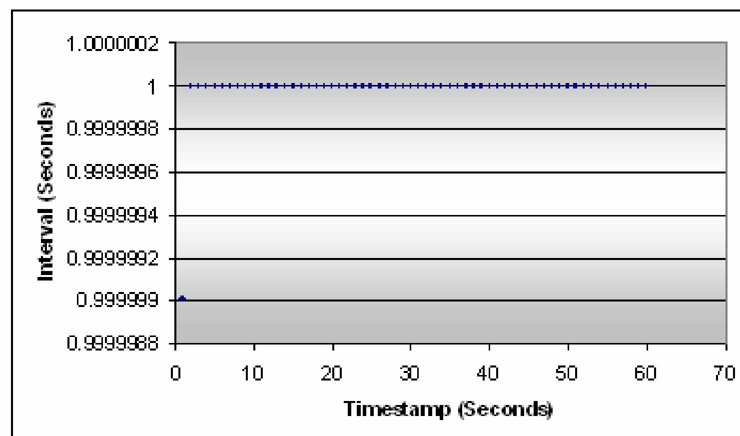

Figure 2-20: 1 Hz Synchronisation Box

The resolution in this case is 1 s ± 1 ?s which was the resolution calculated for the loop timer in section 2.2.2.2.3.

2.5.4 Conclusion

Since the external timer is an accurate real-time timer based on an oscillating quartz crystal, the results exhibited by the synchronisation serve to verify the accuracy of the "timestamp calculator" software that will be discussed in section 4.1. The timestamp calculator was used to measure the timestamp throughout chapter 2. Since the external timer, the synchronisation box, is a proven method of providing a real-time signal, the fact that the maximum deviations recorded in sections 2.5.1 through 2.5.3 were less than 50 ?s, it stands to reason that the timestamp calculator could be trusted to provide an elapsed time interval whose accuracy is in the order of 50?s.

The elapsed time measurements presented thus far, as well as in forthcoming chapters are thus assumed to be within about 50 ?s of the actual elapsed time, since all the measurements rely on the timestamp readings and calculations.

2.6 Problem Statement

As stated in 1.3, a soft real-time system allows the timer some latitude in missing deadlines. This is in contrast to hard real-time systems, where such a situation would constitute a system failure. Sections 2.1 through 2.5 describe the software timers provided by the WIN32 and Linux operating systems and some comparisons are made between them.

Reconsider the results presented in these sections. The following is apparent in regard to the smallest interval measured (1 kHz frequency = 1 ms intervals):

? The loop timer (section 2.2.2): Of all the timers whose performance was quantified in sections 2.2.2, 2.3.1, 2.3.2 and 2.4.1, the loop timer performed the best in terms of accuracy compared to the external timer in section 2.5. The resolution exhibited at 1 kHz was 1 ms ? 167 ms.

? The system timer (section 2.3.1): The minimum interval of 5.881 ms and a maximum of 25.431 ms were recorded. Since 1ms was required, the result is a timer with a resolution of 1ms ? 24.431 ms (25.431 ms – 1 ms).

? The multimedia timer (section 2.3.2): With a minimum interval of 0.503ms and a maximum of 1.983ms, when 1ms was required, a timer with a resolution of 1ms ? 0.983ms is yielded (1.983ms – 1ms). Therefore, the multimedia timer will miss the deadline by as much as 0.983ms.

? The POSIX timer (section 2.4): The resolution of this timer turned out to be 1 ms ? 6.976 ms. The maximum interval was 7.976 ms and the minimum 1.070 ms. Therefore, the timer could deviate from the required interval by almost 7 ms.

The comparison of the performance of the timers can be summarized as follows:

? The loop timer is the most accurate (the maximum deviation is less than 500?s).

? The system timer is the least accurate (the maximum deviation is more than one millisecond).

? The BSD timer is more accurate than the system timer, but less so than the multimedia timer (the maximum deviation is more than one millisecond as well).

? The multimedia timer is less accurate than the loop timer (the maximum deviation is in the order of one millisecond).

Since the multimedia timer and loop timer are the most accurate, their attributes are more conducive to the notion of a real-time timer than the system- and BSD timers.

Although the loop timer is the most accurate, it also puts the most load on the CPU. The multimedia timer consumes very little in terms of CPU resources, but its maximum deviation is ± 1 ms. Thus, from the evidence collected above and summarized in Table 2-1 it is clear that the highest precision software timer possible under the WIN32 operating system guarantees that the interval length of the timer will be within ? 1ms of what was desired duration, without consuming all available CPU resources.

The aim of this research is to determine if it is possible to design and implement a software timer under the WIN32 operating system that yields better performance than that of the multimedia timer. It is clear that to implement a timer that is efficient in terms of resource usage and accuracy, a compromise between the two is a probable side effect.

Nominally, it was decided aim at a margin of error that improves on the best margin of error exhibited by the timers studied above by at least 50% – thus a resolution of at least 1ms ? 500?s is sought.

Table 2-1: Timer Resolution Comparison

Timer	Resolution
WIN32 System Timer	1 ms ? 24.431 ms
WIN32 Multimedia Timer	1 ms ? 0.983 ms
Linux Timer	1 ms ? 6.976 ms

Referring to sections 2.2.2 and 2.2.2.2, it is clear that the design and implementation of the timer has to take processor usage into account, and therefore it is an additional requirement that the timer to be built should not consume all available resources.

It is also necessary for the timer to fire an event when the interval duration has elapsed as an indication of this event to the application using the timer.

Therefore, the problem statement is as follows:

A timer needs to be developed that presents a margin of error less than or equal to 500?s, that consumes minimal processor resources and that fires an event at the conclusion of each interval.

Chapter 3

Design and Implementation Decisions

This chapter describes the design and implementation decisions made to solve the problem as it is described in section 2.6. The aim of this chapter is to familiarize the reader with the process that was adopted, in other words to show how the problem was tackled. Firstly, the choice of programming language is discussed in section 3.1. Furthermore, since it is important that the notion of Application Programmer Interfaces (APIs) is understood, this is explained in section 3.2. Section 3.3 focuses on the choice of operating system followed by a discussion in section 3.4 on the important aspect of real-time systems, namely: Process Scheduling Priority.

3.1 Development Programming Language

The development language chosen was C++. The reasons are simple in that it provides easy access to both the WIN32 and POSIX APIs (section 3.2). In addition, it is widely used and therefore familiar to most researchers.

From the outset it was decided that the design of the timer would heavily rely on object orientation, the programming paradigm for which C++ was designed [Lischner 2003]. C++ also allows the programmer to implement and optimize software in an efficient manner [Bulka et al. 1999].

3.2 Application Programming Interfaces (APIs)

An API is a set of functions that is exposed by a piece of software and that may be used by another application to interact with that software [Palmer 2002]. An "Interface Definition" informs the API user how to use (or invoke) the software in other code [Webopedia 2003].

These functions may be a set of protocols, routines, and/or tools for building software applications. An efficient API will simplify software development in providing quick access to building blocks provided by another application [Wikipedia 2006].

A software module, containing the implementations of mathematical functions for example, may be made available to others through an API, thereby exposing these functions to another user's application. Let the name of such a module be the "Math Module".

This functions within the math module is made available via an API. This is depicted in section Figure 3-1.

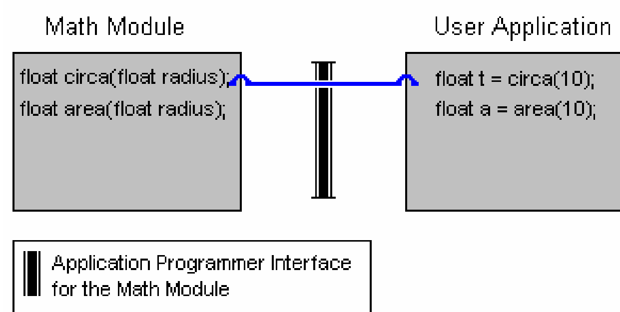

Figure 3-1: API Definition

As is shown Figure 3-1, the math module exposes two functions:

- *float circa(float radius);* - calculates and returns the circumference of a circle with a specific radius.
- *float area(float radius);* - calculates and returns the area of a circle with a specific radius.

The math module functionality will typically be compiled into a DLL (Dynamic linked library) under the WIN32 operating system or dynamically linked Linux library under Linux. The user's application will be either statically or dynamically linked to the DLL, and will be able to access the function via the DLLs API.

Both the WIN32 and POSIX environments provide APIs that give a developer the ability to access core functions within the operating system. These APIs were used to develop the solution. The ability to create threads, initialise platform timers and in the case of the WIN32 platform – to initialise the multimedia subsystem, is made available in the APIs.

3.3 Operating Systems

On the consumer end of computer hardware, specifically the x86-based platforms, the primary operating systems available to the consumer are the various WIN32 platforms (Windows 9x, Windows ME, Windows NT, Windows 2000 and Windows XP) and the UNIX platform that uses the POSIX standard (Linux – specifically the 2.6.x line of kernels).

As discussed in section 1.3, real-time extensions to these operating systems exist to address their real-time inadequacies (also according to [Klein et al. 1994] and [Hardeski 2001]). However, it was decided that a solution would be sought without the aid of such extensions.

3.4 Process Priority

The WIN32 operating system, like its POSIX counterpart, uses a process scheduling mechanism known as preemptive multitasking. This not only allows the operating system to ensure that each process being scheduled receives a fair amount of processing time – it also allows an external hardware event to interrupt the operating system (in other words the operating system may be preempted).

Processes demanding processing time may be divided into two groups:

- Processes waiting for input or a specified time to elapse (idle processes) for example a timer.
- Processes that are fully utilizing the processor.

The architecture of both the Linux and WIN32 operating systems consists of two levels: user mode and kernel mode. Applications in user mode are limited in terms of their access to system resources, whereas kernel processes have unrestricted access to system memory and external devices. However, in both systems, these processes are serviced at fixed at intervals with a typical size of 10 ms [Peterson et al. 1998]. Therefore a situation may arise where all processes are serviced in less than 10ms, however, the processes will not be serviced until the next 10ms interval. Moreover, certain kernel level processes are non-preemptable and may take an unknown amount of time to complete.

Both the WIN32 and Linux platform allow the user to set a process's priority. In the case of the various timer solutions presented in this dissertation, the timer should enjoy the highest possible priority to ensure that its timer measurement is as accurate as possible.

Under the WIN32 platform, the following are possible priority settings:

- IDLE_PRIORITY_CLASS – This class is for low priority threads that only need to run when the system is idle. Processes with this priority may be pre-empted by processes of higher priority.
- NORMAL_PRIORITY_CLASS – Processes with this priority are not in need of any special scheduling.
- HIGH_PRIORITY_CLASS – this class is used for a process that performs time-critical tasks that must be executed immediately. The threads of such a process may pre-empt any other thread with lower priority.
- REALTIME_PRIORITY_CLASS – This class is specified when a process requires the highest possible priority. Threads of such a process preempt the threads of all other processes, including operating system processes. This is also the priority class that the timer implementation will be awarded

Under the Linux platform, process priority allocation is somewhat simpler in that a process is given a priority number from 1 to 20, 20 being the highest priority. Any process of higher priority may pre-empt another process of lower priority. Therefore the timer implementations under the POSIX standard were given a priority of 20. Note that, even though a process may have a priority of 20, it cannot pre-empt kernel processes, at least not for current (at the time of this writing) normal Linux implementations. However, the Linux 2.6.x line of kernels are said to allow this and therefore this version of the research that relate to POSIX relies on this version of Linux [Santhanam 2003].

.

Chapter 4

Solution

In this chapter, a workaround is proposed for the shortcomings identified on the WIN32 platforms in regard to the software timers that it provides. However, before the timer implementation is presented, the timestamp calculator is discussed. This piece of software is significant as it is used in the solution to calculate timestamps. This is discussed in section 4.1. Initial solutions to this problem were not successful; however, those rejected solutions that had a direct hand in the final solution are presented in section 4.2. The actual solution, called a Thread Induced Waitable Timer or TIW timer, is presented in section 4.3. With a solution implemented, the validity and performance thereof has to be investigated, and this is done in section 4.4 where its performance is compared to the software timers discussed in chapter 2. As stated in section 1.5, the external hardware timer is the benchmark for the TIW timers. The TIW timer is compared to this timer in section 4.5.

4.1 Timestamp Calculator

A timestamp may be defined as the time at which a specified event occurs. This time is usually relative to another timestamp referred to as a base timestamp. In the case of the timestamp calculator, this is the time at which the timer is turned on. In other words, all the timestamps that the timestamp calculator generates represent an offset from the timestamp at which the calculator was initiated.

The timestamp calculator is the subject of the initial discussion in this chapter. This piece of software was used in the development of all the timers discussed in both this chapter and chapter 2, specifically for the calculation of interval duration. As was stated in section 2.5, these timestamps are within 50 ?s of the actual timestamp. Accordingly it is imperative to understand the timestamp calculator before the solution is discussed.

The timestamp calculator is based upon the high-resolution counters discussed in section 2.2. The timestamp calculator accesses one of these counters (depending on the processor) whenever a timestamp is required. The value of the counter as well as the knowledge of the frequency at which it is incremented may be combined into the calculation of a timestamp. The mechanisms used to accomplish this are discussed in section 4.1.1.

4.1.1 Query Functions

The WIN32 API provides the means to access the high-resolution counters. Two routines are available to developers that simplify the implementation. These routines are:

? QueryPerformanceFrequency

If the high-resolution performance counter is available, this function retrieves the frequency of the counter. If the hardware is a uni-processor system, the frequency is 3.579545 MHz whereas the frequency equates to the CPU's frequency in a multi-processor environment (as discussed in section 2.2). The units are in Hz (counter ticks per second). This routine is called once, since the frequency is constant and therefore there is no runtime overhead to access it.

? QueryPerformanceCounter

If the high-resolution performance counter is available, this function retrieves the current value of that counter. If the hardware is a uni-processor system, the counter is read from the Power Management timer whereas the CPUs timestamp counter (TSC) is utilised in a multi-processor environment (as discussed in section 2.2.1). As is apparent from section 2.2.2.2, the time to access this routine combined with the calculation of the timestamp is less than 50?s.

All the routines required to access these counters in the POSIX environment were implemented utilising assembler instructions as opposed to API calls. The equivalent implementations for the calls above were realized in the POSIX environment as follows:

? *QueryPerformanceCounter*

The TSC is accessed via assembler calls. Note that for readability, the assembler is presented in normal Turbo Assembler syntax, in other words the way it would be implemented under a WIN32 operating system. However, the Linux equivalent is provided directly thereafter.

```
//------------------------------------------------------------
// WIN32 Assembler presented for readability
//------------------------------------------------------------
CPUID;                        // Return the identification
                              // of the CPU for the rdtsc
                              // instruction
RDTSC;                        // Read the current value of
                              // the high-resolution counter
                              // into a 64 bit integer
mov var_low, EAX;             // Move the lower part of the
                              // 64 bit integer into EAX
mov var_high, EDX;            // Move the higher part of the
                              // 64 bit integer into EDX
```

Subsequently, var_low and var_high are combined to form a single 64bit value that represents the value read from the TSC on the CPU.

```
//------------------------------------------------------------
// LINUX (POSIX) Assembler – the actual implementation
//------------------------------------------------------------
asm("CPUID;");
asm("rdtsc;");
asm("mov %%eax, %0" : "=r"(var_low));
asm("mov %%edx, %0" : "=r"(var_high));
```

? *QueryPerformanceFrequency*

Unfortunately, the POSIX implementation under this routine is not as elegant. Again the implementation will first be presented in “WIN32” syntax in an endeavour to increase readability.

```
QueryPerformanceCounter (&start);   // Read an initial value
                                    // of the counter and
                                    // store it in a
                                    // variable “start”
sleep(1000);                        // Sleep for a second
```

```
QueryPerformanceCounter (&end);      // Read an end value of
                                     // the counter

frequency = (end - start);           // Therefore the
                                     // frequency is
                                     // calculated as the
                                     // difference between
                                     // start and end
```

4.1.2 Timestamp Calculation

The combination of the two routines discussed in section 4.1.1 may be used to calculate a timestamp. With the value of the current high-resolution counter known as well as its frequency, the time elapsed since the counter started incrementing may be calculated. Since the counter starts incrementing from zero the moment that electrical power is applied to the computer, the exact time since the PC was turned on can be determined. Likewise, the time that has elapsed since the previous occasion on which the counter was polled can be calculated.

Let ? be the current value of the high-precision counter and ? be its frequency. Remember that the frequency is specified in Hz and represents the number of times that the counter is incremented per second. Therefore the time elapsed since the computer was turned on ? (representing the current timestamp) is calculated as:

$$\tau = \frac{\varphi}{\phi}$$

In this case, the value ? is in seconds, but it may be processed to represent values in ms, ?s, ns etc. This is elaborated on in the following section.

4.1.2.1 Precision

The timestamp precision in the final version of the timestamp calculator is configurable. In other words, the timestamp calculator returns the timestamp in seconds, milliseconds, microseconds etc. This is accomplished through the division of the timestamp ? (as calculated above). This is required for the obvious reason that the timestamp calculator has to measure intervals considerably smaller than 1 second.

The timestamp calculator is configured via a floating-point parameter specifying the precision. The value specified and the precision obtained is tabulated in Table 4-1. Let ? denote the requested precision.

Table 4-1: Timer Calculator Precision

Precision Units	Requested Precision (?)
Seconds	1 (10 to the power 0)
Milliseconds	0.001 (10 to the power -3)
Microseconds	0.000001 (10 to the power -6)
Nanoseconds	0.000000001 (10 to the power -9)

4.1.2.2 Process of timestamp calculation

The following process of the timestamp calculator is designed in such a way that the timestamp calculated is relative to a base timestamp taken at initialisation of the timer. This process is as follows:

Janno Grobler
jannogrobler@gmail.com
M.Sc Computer Science
University of Pretoria

? When the timestamp calculator is initialised, an initial timestamp is calculated. This timestamp is saved as the base timestamp ?. The base timestamp ? is a representation of the time elapsed since the high-resolution counter started incrementing, in other words the number of counter ticks since the computer was turned on. The time elapsed ? is divided by the precision ? to obtain the base timestamp, therefore:

$$\beta = \left[\frac{\varphi}{\phi} \div \rho \right]$$

? Every time a subsequent timestamp is requested the current timestamp relative to ? is calculated and returned. Therefore, the current timestamp ? is calculated as:

$$\alpha = \left[\left[\frac{\varphi}{\phi} \div \rho \right] - \beta \right]$$

4.1.3 Class Description

As was stated in section 3, a fundamental design decision was to implement the solution using object orientation. Therefore timestamp calculator is designed as a class that may be instantiated by any application that requires such a component to calculate timestamps.

This class design is simple and consists of the following:

Timestamp Calculator Class

The C++ class definition of the timestamp calculator class is as follows:

```
class TimestampCalculator
{
  public:
    TimestampCalculator (double precision);
    ~ TimestampCalculator (void);

    bool enabled (void);
    void reset (void);
    unsigned long long getTimestamp (void);

  private:

    unsigned long long performanceFrequency ;
    unsigned long long performanceCounter ;

    unsigned long      baseTimestamp ;
    unsigned long      precision ;
};
```

The notable functions of the class are as follows:

Class Constructor (`TimestampCalculator (double precision); `)

Janno Grobler
jannogrobler@gmail.com
M.Sc Computer Science
University of Pretoria

Input Parameters: Single floating-point value – the precision ?.

The current timestamp is calculated and saved as the base time ?, with precision as specified by ?.

Reset Function (`void reset (void);`)

The reset function resets the base time to the timestamp at the moment the reset function is called.

Get Timestamp Function (`unsigned long long getTimestamp();`)

This function returns the current timestamp ?.

Typically, a single application will use only one instance of the timestamp calculator, which is the case in the final solution. However, it is of course possible to instantiate multiple instances of this class.

4.2 Rejected Solutions

During the course of this research, a number of unsatisfactory solutions to the problem of constructing a soft real-time timer were implemented. However, as is often the result of a trial and error process, elements of the final solution are frequently a part of a rejected implementation that preceded it. The rejected scenarios that contributed to the final solution are presented in the following sub-paragraphs. In fact, it is necessary to understand these rejected solutions in order to understand the eventual solution. The rejected solutions are all based on the WIN32 operating system platform.

4.2.1 WIN32 Sleep Timer (Non Multimedia)

The sleep timer is a waitable timer (see 1.4.3) based on the WIN32 system timers discussed in 2.3.1. As seen in 2.2.2 with the loop timer, it is essential that some mechanism be found to ensure that the timer does not take up all the available processing power of the CPU.

To accomplish this, we turn to a mechanism provided by the WIN32 API. The WIN32 API provides a mechanism to suspend a thread for a specified amount of milliseconds, called the "*Sleep*" function. The desired number of milliseconds for which the process is required to sleep is specified as a single parameter. The implementation of the sleep timer utilising this function is explained in the algorithm in section 4.2.1.1.

4.2.1.1 Algorithm

The algorithm consists of a single loop, wherein the execution thread is instructed to sleep for ? ms. The current timestamp $?_{current}$ is taken before the loop commences, and then after completion of every sleep interval. The value of $?_{current}$ is obtained from the timestamp calculator calculator's function, getTimestamp() discussed in section 4.1. At this point, a timer event may be fired.

This is depicted in algorithm 2.

```
?current := getTimestamp()
While the timer is running
      Sleep (for ? ms)
      ?current := getTimestamp()
      Fire Timer Event
End While;
```

Algorithm 2 - Normal Timer Algorithm

It should be assumed that every new assignment of a value to $?_{current}$ in the algorithm is accompanied by some statements necessary to store this value. Using this algorithm, the goal of the software timer is to generate intervals of duration ? ms.

4.2.1.2 Results

The performance of the timer is presented in this section. The analysis techniques are as described in section 1.5. The timestamp calculator was used calculate the timestamps.

4.2.1.2.1 1 ms interval (1 kHz)

The results for a frequency of 1 kHz are illustrated by Figure 4-1. The desired interval should be as close to one millisecond (1000 ?s) as possible. The maximum interval recorded was 16.336 ms (16336 ?s) and the minimum interval 14.528 ms (14528 ?s). The timer therefore provides a resolution of 1 ms ± 15.336 ms on 1 kHz frequency. Already it is clear that this timer is not effective with the realisation that although an interval of 1 ms was desired, an interval that deviates by ± 15 ms was obtained. Accordingly, the requirement of a maximum deviation of 500 ?s is not met.

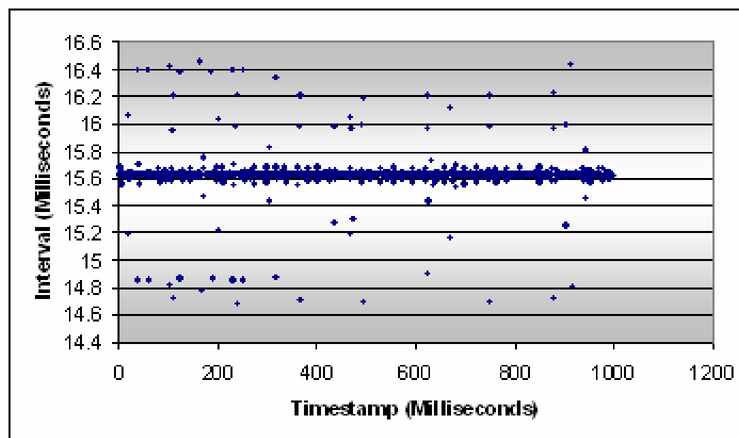

Figure 4-1: Sleep Timer (Non Multimedia) (1 kHz)

4.2.1.2.2 20 ms interval (50 Hz)

The performance for a frequency of 50 Hz is illustrated by Figure 4-2 and intervals close to 20 ms (20000 ?s) is expected. The maximum interval recorded was 31.879 ms (31879 ?s) and the minimum interval 30 ms (30000 ?s). The timer therefore provides for a resolution of 20 ms ± 11.879 ms on 20 ms interval. This is further proof

of the ineffectiveness of the timer. It seems that in actual fact, if an interval of 20 ms is desired, more accurate results could be obtained when required interval duration is set to one millisecond.

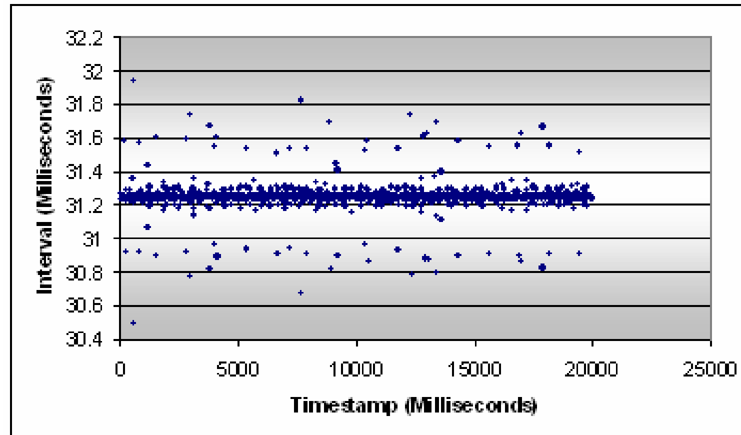

Figure 4-2: Sleep Timer (Non Multimedia) (50 Hz)

4.2.1.2.3 1 second interval (1Hz)

Finally, the timer was tested with a frequency of 1 Hz and its performance on this occasion is depicted in Figure 4-3. The target interval is as close to one second (1000000 ?s) as possible.

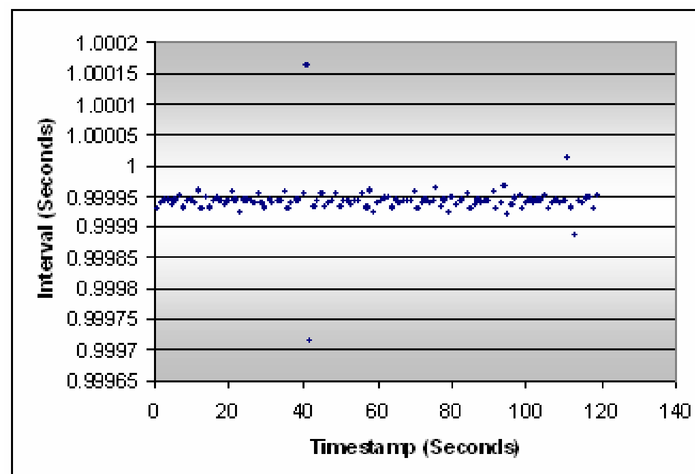

Figure 4-3: Sleep Timer (Non Multimedia) (1 Hz)

The maximum interval recorded was 1.000166 s (1000166 ?s) and the minimum interval 0.998872 ms (998872 ?s). The timer therefore provides for a resolution of 1 s ± 0.2 ms at a frequency of 1 Hz interval. This is actually an occasion where the where the timer exhibits acceptable results, although it fails on the smaller intervals.

4.2.1.3 Conclusion

It is clear from the results of the 1 ms and 20 ms interval tests that the timer is not suitable and the resolution is not high enough, even though timer is effective for a 1 Hz frequency. The requirement for the real-time software timer specified in section 2.6 requires the maximum deviation from the requested interval duration to be 500?s

or less and it is clear that the sleep timer does comply in these two cases. The excellent performance of the timer in the 1 Hz study is not adequate to allow the timer to be considered as a soft real-time solution. Neither is the fact that the timer consumes minimal system resources (due to the sleep instruction) nor the fact that the timer event is able to fire at the conclusion of every interval.

The reason for the poor performance of the sleep timer is that the sleep function only returns when the underlying operating system informs it that the desired sleep interval has elapsed. The reason for the tardiness of the operating system was covered in section 2.3.1. An attempt was made to solve this problem using the multimedia sleep timer in section 4.2.2.

4.2.2 WIN32 Sleep Timer (Multimedia Timer)

This timer is a derivative of the sleep timer discussed in 4.2.1. As discussed in section 2.3.2, it is possible to increase the frequency at which the system clock is updated when relying on the multimedia timer. This can be done via the multimedia sub-system initialisation API calls provided by the WIN32 operating system platform [MSDN 2005]. When the subsystem is enabled, the timer object, including the *Sleep* instruction is serviced at the highest rate provided by the WIN32 platform – 500 Hz or every 2 ms.

4.2.2.1 Algorithm

Algorithm 3 is essentially the same as Algorithm 2 discussed in section 4.2.1.1, barring the initialisation of the multimedia subsystem before the timer commences its loop, thus yielding Algorithm 3 below:

```
Initialise the Multimedia Subsystem
?current := getTimestamp()
While the timer is running
      Sleep (for ? ms)
      ?current := getTimestamp()
End While
```

Algorithm 3 - WIN32 Multimedia Sleep Timer Algorithm

4.2.2.2 Results

The performance of the WIN32 sleep timer is presented in this section. Again, the analysis techniques conform to the guidelines described in section 1.5 and the timestamp calculator was used to calculate the timestamps.

4.2.2.2.1 1 ms interval (1kHz)

To generate 1ms (1000?s), the frequency of the timer is set to 1 kHz and the results are presented in Figure 4-4. The figure shows the results over the first ±7 seconds to improve the readability of the graph. However, the experiment was conducted over a period of 60 seconds. The maximum interval recorded was 2.024 ms (2024 ?s) and the minimum interval 1.883 ms (1883 ?s).

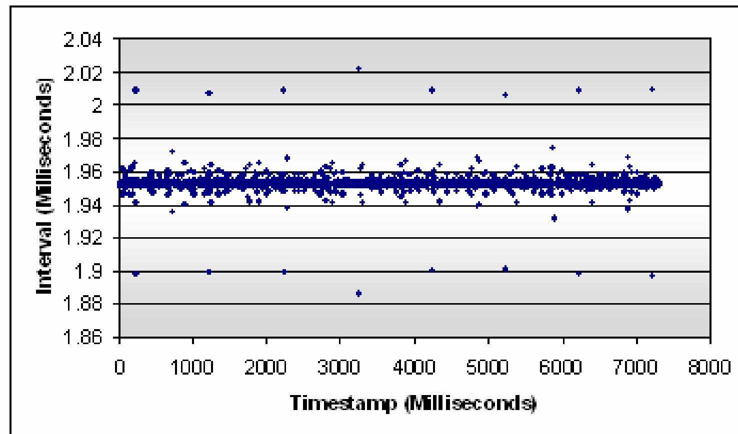

Figure 4-4: Sleep Timer (Multimedia) (1 kHz)

The timer therefore provides for a resolution of 1ms ± 1.024 ms on 1 ms interval. In fact, the average interval duration was 1.95 ms. This is an improvement from the result of the normal sleep timer in section 4.2.1.2.1. However, it only serves to illustrate the one millisecond margin error that is the best that the normal WIN32 platform provides (refer to section 2.6).

It should be noted that of the 60000 intervals recorded over the period of minute, the desired interval of 1 ms was exceeded by more than 1 ms (in other words, the number of intervals of 2 ms and more) 119 times. This equates to 0.198% of the intervals.

4.2.2.2.2 20 ms interval (50 Hz)

The results for a frequency of 50 Hz are illustrated by Figure 4-1. The desired interval should be as close to 20 ms (20000 ?s) as possible. The figure shows the performance over a period of approximately 15 seconds, although the experiment was conducted over the course of a minute.

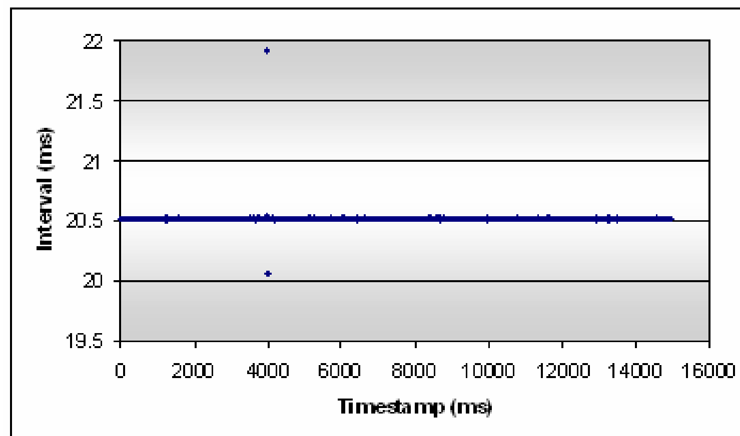

Figure 4-5: Sleep Timer (Multimedia) (50 Hz)

Over a period 60 seconds, the maximum interval recorded was 21.906 ms (21906 ?s) and the minimum interval 20.059 ms (20059 ?s). The timer therefore provides for a resolution of 20 ms ± 1.906 ms at a frequency of 50 Hz. As was the case with the 1 kHz sleep timer in section 4.2.2.2.1, the maximum deviation exceeds 500 ?s, which is outside the range of specified by the requirements in section 2.6.

4.2.2.2.3 1 second interval (1Hz)

Figure 4-1 illustrates the results for the multimedia sleep timer at a frequency of 1 Hz and accordingly the desired interval duration is one second (1000000 ?s). Results shown in the figure spans a period of 60 second.

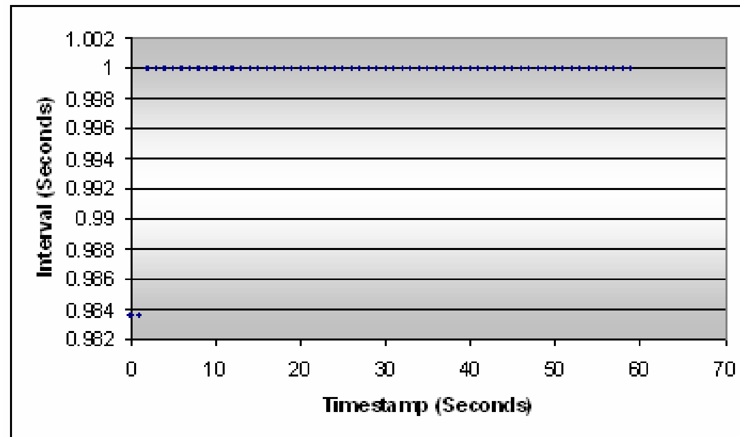

Figure 4-6: Sleep Timer (Multimedia) (1 Hz)

The maximum interval recorded over the period under scope was 0.999956s (999956?s) and the minimum interval 0.983568s (983568?s). The timer therefore provides for a resolution of 1 s ± 16.432 ms at a frequency of 1 Hz.

Strangely enough, where the 1 Hz interval yielded the best performance from the sleep timer in section 4.2.1, the worst performance is exhibited in the case of the multimedia sleep timer under discussion in this sub-paragraph.

4.2.2.3 CPU usage

The resource consumption recorded in sections 4.2.2.2.1 to 4.2.2.2.3 was basically the same and is represented by Figure 4-7. The maximum usage recorded at any given time was 1% and the minimum 0%. The average usage was 0.768%.

The multimedia sleep timer therefore consumes very little in terms of CPU resources and actually surpasses the performance of the multimedia timer recorded in section 2.3.2.4.

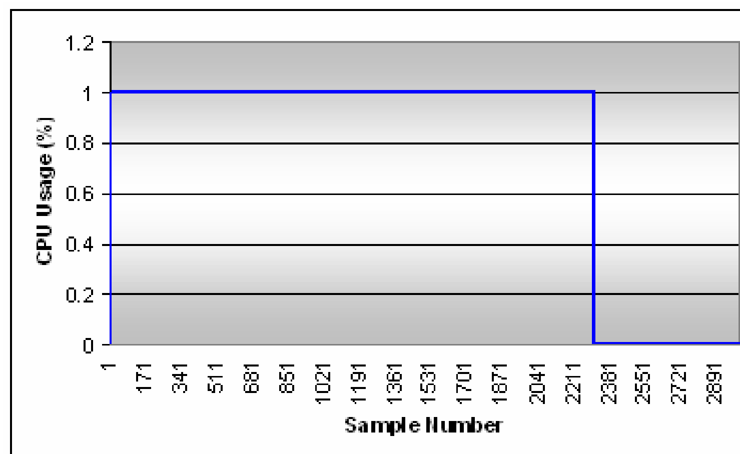

Figure 4-7: Sleep Timer (Multimedia) CPU usage

4.2.2.4 Conclusion

Although the multimedia sleep timer's performance is a considerable improvement over the sleep timer without the multimedia subsystem enabled, it still fails to meet the requirement of a margin of error less than 500 ?s. Again minimal resources are consumed (less than 1%) and an event is fired at the conclusion of each interval. However, as was the case with the normal sleep timer, this is insufficient to qualify it as suitable soft real-time timer as per the definition in section 2.6.

It is clear from the results that the smallest interval of time that a running process can be suspended is in the order of 1 – 2 ms. This is derived from the resolutions of the POSIX timer (section 2.4.1.1), the multimedia timer (section 2.3.2.1) and the multimedia sleep timer (section 4.2.2.2.1) at a frequency of 1 kHz.

Up to this point in the dissertation, the timer with the best resolution is the loop timer (section 2.2.2). The sleep timers were an attempt to introduce a waiting period into the loop timer to prevent it from consuming all available CPU resources. However, it is clear from the sleep timers' performance that to implement a timer with a maximum deviation of 500 ?s, it is necessary to find a mechanism to induce a waiting period of less than one millisecond to prevent it from consuming all available CPU resources.

4.2.3 Critical Section Timer

[Manko 2002] proposed a solution that uses critical sections that can be entered as soon as they become available. The development of this timer is an attempt to incorporate a waiting period in the timer to prevent it from consuming all available CPU resources, without inhibiting the generation of reliable intervals.

Two threads are created, each attempting to enter a critical section shared between them. When this is accomplished, the thread measures the required interval in the same way that the loop timer would (Refer to section 2.2.2). Subsequently the thread relinquishes the critical section to be entered by the second thread that in turn would follow the same process. And so it would continue. This is depicted in Figure 4-9.

The assumption was that forcing the threads to wait for the ownership of a critical section would induce sufficient idle time to allow the CPU to award its resources to other processes. This assumption proved to be incorrect and yet again no waiting period is induced and the timer consumes nearly all available processor resources. The average usage was 83.201% depicted in Figure 4-8.

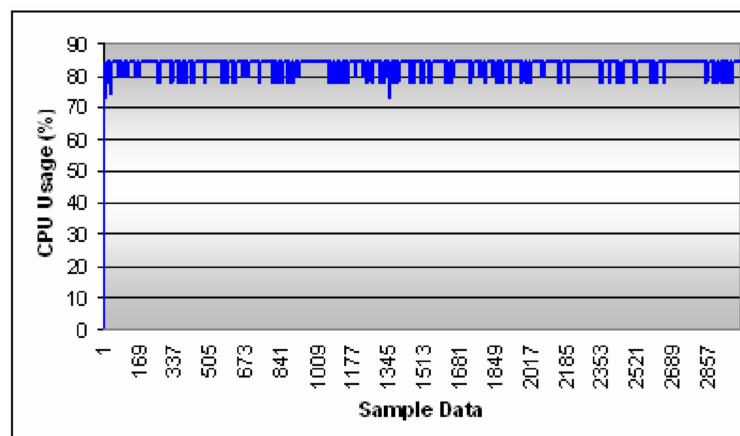

Figure 4-8: Critical Section Timer CPU usage

However, it exhibits the same accuracy as the loop timer, providing a margin of error less than 500 ?s (refer to section 2.2.2). The timer may be implemented to fire an

event at the conclusion of each interval. However, since it consumes most of the available CPU resources, it is not suitable for the soft real-time implementation defined in section 2.6.

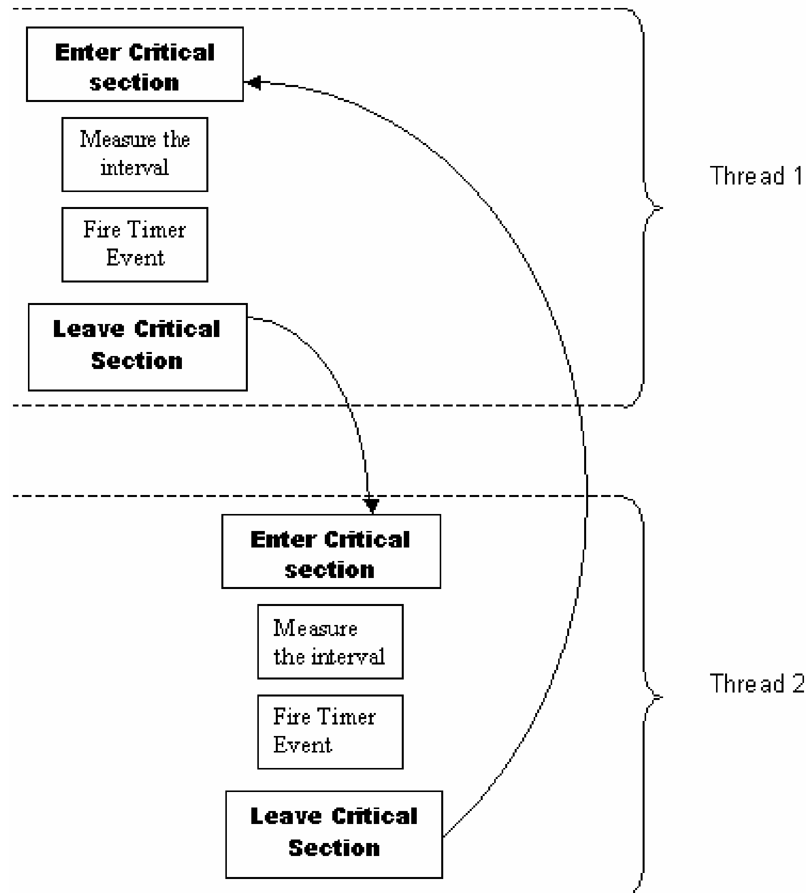

Figure 4-9: Critical Section Timer

4.3 Thread Induced Waitable Timer

As stated in section 2.2, high-resolution hardware counters are supported in modern hardware and provide a mechanism to compute accurate timestamps as discussed in section 4.1. The loop timer discussed in section 2.2.2 can be used to accurately measure a specified interval by reading one of these counters, but consumes all available CPU time in the process.

A mechanism needs to be found where these high-resolution hardware counters may be polled to determine timestamps without using all available processing time. The critical section timer (in section 4.2.3) attempts to solve this problem through the use of the concurrent process concept of critical sections, but as stated gives unsatisfactory results.

According to the empirical evidence of section 4.2.2.2.1, if the multimedia timer is initialised and the sleep function instructs a thread to sleep for an interval of approximately 1ms, then the thread sleeps for an average of 1.95ms instead.

Therefore the inadequacies of these solutions are known. To achieve a solution, these inadequacies have to be taken into account and remedied in the implementation.

Firstly a waiting period has to be enforced to prevent the timer from taking up all the available CPU processing time, in contrast to the loop timer and critical section timer. The waiting period has to be small enough for a 1 ms interval timer to be derived that has a maximum deviation of 500?s, unlike the sleep timers. A solution to this seemingly contradictory set of objectives presented itself, based on the use of multiple threads, inspired by the critical section timer. In fact, it turns out that two threads suffice to build the required timer.

This observation suggests a way of improving the margin of error of 1 ms, 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 use the timestamp calculator to determine how much time has elapsed. Only one of these threads are active at any given time, accomplished through the use of a common critical section, as is the case with the critical section timer. Both threads are started at the same time and both will try to enter the critical section. The timing between the two threads is shown in Figure 4-10. The thread that enters the critical section first is referred to as thread A in the rest of this section.

Initially, thread A will wait in a loop polling the high-resolution hardware counters until an interval of 1 ms has elapsed after which the timer event is issued followed immediately by the 2 ms sleep instruction. At this point thread B is made active and waits in a loop polling the high-resolution hardware counters for an interval of 1 ms, followed by the timer event and 2 ms sleep instruction as was the case with thread A. Since thread A would only have been sleeping for 1 ms by this time, 1 ms from the timer event issued by thread B has to elapse before thread A resumes. In this time, neither thread is executing. This process induces the required waiting period.

By the time that thread B has waited in the loop for 1 ms, thread A has approximately 1 ms of sleeping time left. Therefore when thread A completes its 2 ms sleeping time, approximately 0.95 ms (1.95 ms – 1 ms) has elapsed since the timer event issued by thread B. When thread A resumes, it waits in a loop polling the high-resolution hardware counters for on average approximately 50 ?s (950 ?s – 1000 ?s) until 1 ms has elapsed since thread B started its 2 ms sleep interval. This period is the ? ?period in Figure 4-10.

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

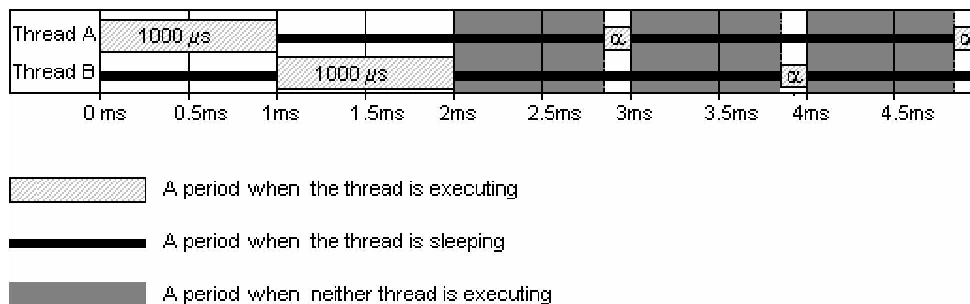

Figure 4-10: Thread Induced Waitable Timer Timing diagram

Our prior results already suggest that this wait in a loop where high-resolution hardware counters are polled, will endure on average for about 50?s, which does not seem too severe on the CPU. Furthermore, the data in Figure 4-4 suggests that there will occasionally be intervals that last for slightly longer than the required 2 ms. In the experiment in section 4.2.2.2.1, the interval duration was more than 2 ms 119 times over a period of 60 seconds. Therefore, over a period of 60 seconds, 0.198% of the

intervals were 2 ms and longer. However, they are well within the allowed tolerance of 500 ?s. Should this be the case, the active thread at this point will issue the timer event immediately and enter the 2 ms sleep interval. In order to compensate for the resulting drift, the following timer event will occur 1 ms from the time the previous one should have been issued (refer to 4.3.2).

The top-level algorithm of the timer is given in Algorithm 4.

```
Initialize thread A
Initialize thread B
?previous := getTimestamp()
Start thread A
Start thread B
While the timer is running
    Sleep for 1ms
Stop thread A
Stop thread B
```

Algorithm 4 - Normal Timer Algorithm

The first timestamp is taken from the high-resolution hardware counters and saved as $?_{previous}$. Thread A is started first and will start measuring the initial 1 ms interval immediately. Thread B will be started as well, but will wait for thread A to finish the initial 1 ms 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. The body of the loop consists of a single statement that suspends the main application thread for on average 1.95 ms (refer to section 4.2.2.2.1). This ensures that the main application thread does not consume all available processing resources and will continue until the timer is terminated. This loop is necessary to ensure that the main thread does not exit until the timer is terminated. When the timer is terminated, threads A and B have to be terminated as well.

However, such a configuration – i.e. where each thread operates entirely independently of the other – would be subject to more time drift in addition to the drift caused when the sleep instruction sleeps more than 2 ms. Instead of such independent functioning, the readings taken from the high-resolution hardware counters using the timestamp calculator should be stored in variables that are globally available to both threads. (Reads and writes to these variables should of course be in critical sections of code, protected by mutual exclusion mechanisms that prevent simultaneous access.) Call these variables $?_{previous}$ and $?_{current}$.

The timers implemented according to the discussed concepts, may be regarded as thread-induced waitable timers. For the remainder of this text, such a timer will be referred to as a TIW timer. The variables and their role in the algorithm of the TIW timer are presented in section 4.3.2.

4.3.1 Design

In accordance with the design decisions made in section 3 the design relies on object orientation. The design is simple and structured in such the way that the architecture is modular. However, the extent of object orientation is simple as no inheritance is used – for example.

Throughout the code snippets that will be presented in this section, the statements `#ifdef __WIN32__` and `#ifdef __POSIX__` often occur. This is to ensure the portability of the code, since the APIs for the different operating system vary. The statement instructs the compiler to include only the code that is relevant to a specific operating system.

The timer itself consisting of the following classes:

? **Thread Class**

This class encapsulates a mechanism to create a thread, providing it with a function to execute, initiate and terminate it.

Thread Class

The C++ definition of the thread class is as follows:

```
class ThreadClass
{
  public:
    ThreadClass ();
    ~ThreadClass ();

    void initialise(   void (*func)(),
                       CRITICAL_SECTION *criticalSection
                    );
    void start();
    void stop();
    void setInterval(unsigned short sleepInt);
  protected:
  private:
    void (*fire)();
    bool threadRunning;
    unsigned short sleepInt;
    CRITICAL_SECTION *criticalSection;
    THREAD_HANDLE threadHandle;
  friend
#ifdef __WIN32__
  unsigned long  CALLBACK
#else
  #ifdef __POSIX__
    void*
  #endif
#endif
  callback(void* argument);
};
```

Class Constructor and Destructor (*ThreadClass();~ThreadClass();*)

No significant operations are performed in these methods and they serve purely as class construction/destruction methods.

Initialisation function (*void initialise(…);*)

This function is used to initialise the class. The following are the parameters:

"func" – a pointer the function that contains the logic that the thread should execute

"critical_section" – a pointer to the critical section that will be shared among the threads

Start Function (*`void start();`*)

Starts the thread

Stop Function (*`void stop();`*)

Stops the thread

Set Interval (*`void setInterval(unsigned short sleepInt);`*)

Sets the interval for which the thread should sleep. Referring to the description of the TIW timer in the previous section, this interval would be set to 1 ms.

CALLBACK Function (*`callback(void* argument);`*)

Both the WIN32 and POSIX APIs require a callback of which the pointer has to be passed to the API function call that creates the threads. This function provides this pointer. The function pointed to by "fire" is called within this callback. The significance of this function is explained below.

Private Variables

"fire" – the pointer to the function that is specified to the thread class at initialisation and executed within the callback function.

"threadRunning" – a flag that keeps track of whether the thread is running

"sleepInt" – the interval specified by the setInterval operation

"criticalSection" – the critical section

"threadHandle" – a handle to the thread created using API funcion calls

? **Timestamp Calculator Class**

This class provides the capability to determine the current timestamp based on the high-precision counters discussed in section 2.2. The calculator was discussed in detail in section 4.1.

? **Multimedia Subsystem Class**

This class provides the capability of setting the operating system in the high-resolution clock interrupt mode that is provided by the multimedia subsystem, as discussed in section 2.3.2. As the following description will show, the class is simple, and serves only to activate/deactivate the multimedia subsystem.

Multimedia Subsystem Class

```
class MMSubsystem
{
  public:
    MMSubsystem (void);
    ~ MMSubsystem (void);
    bool initialise(unsigned int targetResolution);
    void close(void);
  protected:
  private:
    unsigned int timerResolution;
};
```

Janno Grobler
jannogrobler@gmail.com
M.Sc Computer Science
University of Pretoria

Class Constructor and Destructor (*MMSubsystem();~MMSubsystem();*)

No significant operations are performed in these methods and they serve purely as class construction/destruction methods.

Initialisation function (*void initialise(…);*)

This function is used to initialise the WIN32 Multimedia subsystem.

Close Function (*void close();*)

? **TIW timer Class**

This class provides the interface between the user application and the TIW timer classes. It is therefore the only class visible to the user application.

TIW timer Class

```
class TIWTimerClass
{
  public:
    TIWTimerClass ();
    ~TIWTimerClass ();
    void initialise(
                    void (*func)(void * args),
                    unsigned long interval,
                    unsigned long sleepInterval,
                    unsigned long noOfThreads,
                    unsigned long mod
                   );
    void start ();
    void stop ();
    bool isRunning ();
  protected:
  private:
    void (*timerCallback)(void* args);
    unsigned long sleepInterval;
    unsigned long interval;
    unsigned long noOfThreads;
    unsigned long mod;
    CRITICAL_SECTION criticalSection;
    vector <ThreadClass *> threads;
    unsigned long recordCounter;
    TimestampCalculator * timestampTimer;
    unsigned long long oldTime;
    HANDLE timerEvent;
    bool threadRunning;
#ifdef __WIN32__
  friend void
```

```
#else
  #ifdef __POSIX__
    void*
  #endif
#endif
                        tiwTimerCallback(void* argument);
};
```

Class Constructor and Destructor ***(TIWTimerClass(); ~TIWTimerClass();)***

No significant operations are performed in these methods and they serve purely as class construction/destruction methods.

Initialisation function (*void initialise(…);*)

This function is used to initialise the TIW timer.

"func" – This function encapsulates the functionality that has to be executed at each timer event. This pointer is saved in the private variable *timerCallback.*

"interval" – This parameter represents the desired interval duration. This is saved in the private variable *"interval"*

"sleepInterval" – The sleeping interval that is passed to the Sleep instruction in the case of the WIN32 operating system ("nanosleep" under POSIX). Therefore it represents the number of ms for which each thread will be suspended when the instruction is issued. This is saved in the private variable *"sleepInterval"*

"noOfThreads" – The number of threads that the timer should used. This is saved in the private variable *"noOfThreads"*

"mod " – The thread measures an interval specified by the "*interval*" parameter. However, the timer event need not be issued immediately after each interval. The number of intervals that has elapsed (see recordCounter below) is divided by mod. If the number of intervals is divisible by mod, the timer event is issued. This is saved in the private variable *"mod".*

Start Function (*void start();*)

Starts the TIW timer

Stop Function (*void stop();*)

Stops the TIW timer

Is Running Function (*void isRunning();*)

This function is an indication of whether the TIW timer is currently running. This is done via the Boolean private variable, threadRunning.

Private Parameters

"criticalSection" – This represents the critical section that is shared between the threads, as a single object.

"threads" – This is a link list that holds the pointers to all the threads currently in use.

"recordCounter" – The parameter keeps track of the number of intervals that have elapsed since the timer was started.

"timestampTimer" – This variable points to an instance of the timestamp calculator.

"oldTime" – this parameter is used to keep track of the previous timestamp (**?** $_{previous}$ in section 4.3.2)
"timerEvent" – This is a pointer to the timer event that issued at the appropriate times.

An user application can instantiate the TIW timer Class and provide it with the information necessary to initialise and run it. The information it has to provide to the timer is the following:

? The number of threads the TIW timer should use

? The required interval to measure

? The time period until the timer expires

? The sleep interval of the threads

? The modulo factor. This factor determines the number of intervals that has to pass before the timer event is fired.

Such an user application was developed, and was designed to "host" a TIW timer instance – with the following classes encapsulated:

? The User Application itself. This is the main application class for the user application, responsible for the initialisation of the remaining user application classes, discussed shortly, as well as TIW timer class. The user application basically acts as a host to the timer.

? The Command Line Class. This class enables the user application to be configured via the command line, with the same options as those taken by the TIW timer class. The command line options are primarily to configure the TIW timer. Should the command line options provided be incorrect, the following message will be displayed by the user application:

```
Usage : msc_timer <options>
Options:
--timestamps <no of timestamps>
--interval   <interval size>
--threads    <no of threads>
```

? The CPU Load class. For the sake of the analysis in this dissertation (refer to section 1.5), a mechanism is required to determine the CPU usage of the timer. The CPU Load class provides this capability.

Figure 4-11 presents the relationships between these classes in the form of a class diagram. The figure is divided into two sections. The top half represents the classes that are encapsulated in the TIW timer implementation, whereas the bottom half represents the user application.

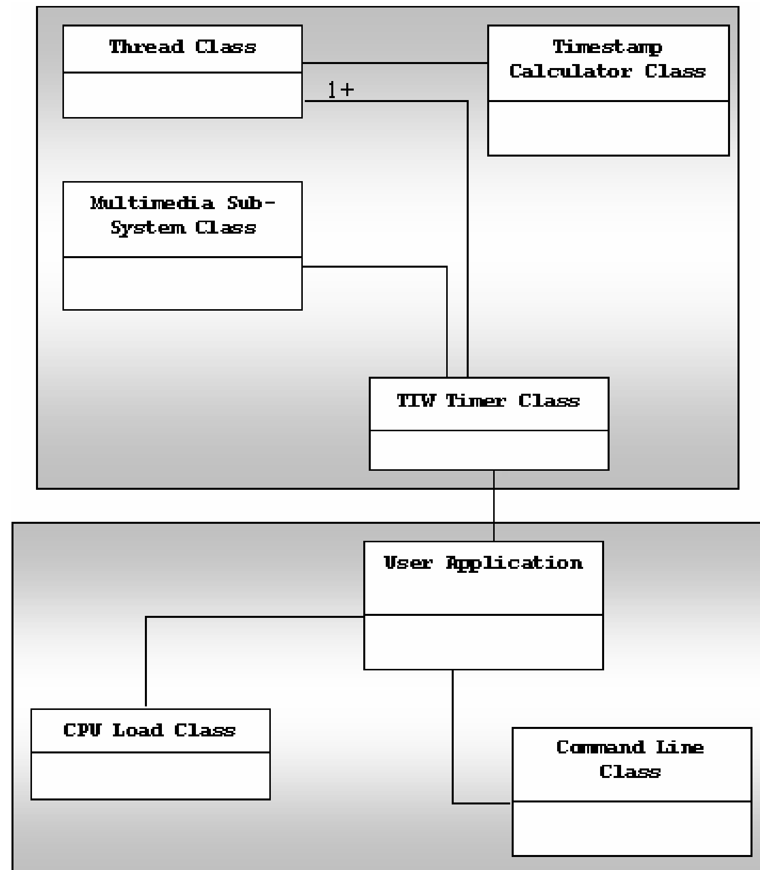

Figure 4-11: TIW timer Class Diagram

The sequence of operations of the system presented in Figure 4-11 is as follows:

- The user application reads the options from the command line using the command line class. There is therefore a one-to-one relationship between the user application and the command line class. The command line class will determine whether the command line options are valid or not.
- If the command line options specified are valid, the user application initialises the CPU Load class. Again, there exists a one-to-one relationship between the user application and the CPU load class.
- The user application initialises the TIW timer class, passing to it the options from the command line. The TIW timer uses this information to configure itself correctly.
- The TIW timer in turn initialises the operation parameters (from the command line). At this point, the TIW timer initialises the number of threads that were specified on the command line. These threads are subsequently started when the user application "*starts*" the TIW timer.
- At this point the user application enters a loop to keep it active, sleeping for intervals of 1 ms, taking the CPU load at the conclusion of each waiting period utilising the CPU load class. For the purposes of the TIW timer testing, this loop will terminate when the required number of timestamps was recorded. This number of timestamps is equivalent to the number of intervals that elapsed during the timer's execution.

? The "*function*" that contains the logical operations that the timer has to execute each interval, in this case is a routine that records the current timestamp. In other words, the current timestamp should differ from the previous one by the amount of time equivalent to the size of the interval duration.

? At the conclusion of the loop, the recorded timestamps are written to a comma-delimited file for analysis.

4.3.2 Algorithm

The first timestamp in Algorithm 5 is saved as $?_{previous}$. The interval at any point in time is given by $?_{current} - ?_{previous}$. A thread induced waitable timer for a 1 ms interval can thus be built by starting off a thread (thread A), waiting for 1 ms, recording the high-resolution hardware counters value as $?_{previous}$, and starting off a second thread (thread B). When each thread is started, both will execute the algorithm given in Algorithm 5.

```
While the timer is running
      Wait to enter the critical section
            // Initial value of ?previous read just before
            // thread 2 starts
            // Both threads execute the algorithm below
            For (duration of the test)
               ?current := getTimestamp();
               While (?current - ?previous <1000?s)
                          ?current := getTimestamp();
               ?previous := ?previous + 1000?s
               Fire the timer event
               Sleep for 2ms
            End for
      Leave the critical section
```

Algorithm 5 - Normal Timer Algorithm

It should be noted that $?_{previous}$ is incremented by the interval size just before the timer event is issued. Therefore $?_{current} - ?_{previous}$ represents time elapsed since the previous 1 ms interval should have ended. This compensates to an extent for the drift induced when the sleep instruction sleeps more than 2 ms. Therefore, when the sleep instruction suspends a thread longer than 2 ms, it results in a longer interval greater than 1 ms followed by a shorter interval less than 1 ms to compensate for the drift.

4.3.3 Results

This section is focused on the results recorded in terms of the performance of the TIW timer. In each case, a figure is presented depicting the recorded interval sizes generated by the TIW timer; over a fixed period of time. In addition, a figure is shown displaying the amount of CPU resources consumed during the period for which the timer was running.

4.3.3.1 TIW timer (1kHz Interval)

The results of the algorithm in section 4.3.2 for a frequency of 1kHz, is illustrated in Figure 4-12 and shows the performance over a period of 30 seconds. The command line parameters specified to the TIW timer are as follows:

```
Number of timestamps        – 60000
Interval Duration           – 1000 ?s
Requested Sleep Interval    – 1 ms
Number of threads to use    – 2
The "mod"                   – 1
```

Over a period of 60 seconds the maximum interval recorded was 1.036 ms and the minimum interval recorded 0.964 ms. Therefore an interval with a maximum deviation of approximately 50 ?s is achieved. Also as is clear from the figure, an interval with duration more than the desired 1010 ?s seconds is the exception and not the rule.

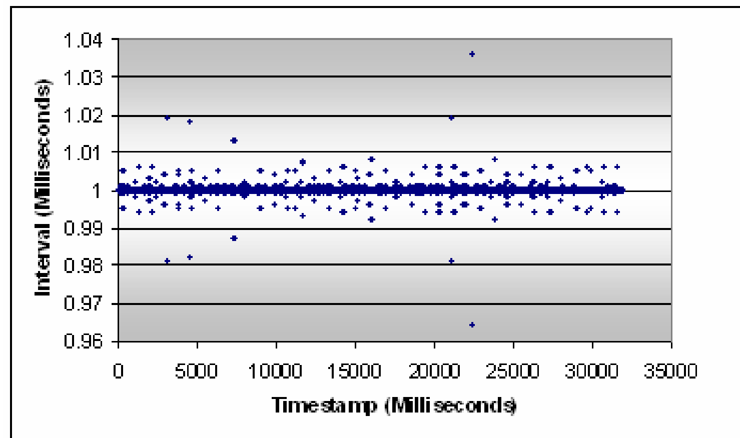

Figure 4-12: TIW timer (1 kHz Interval)

Accordingly the TIW timer exhibits excellent performance when configured to generate 1 kHz intervals. However, this would be in vain if the CPU was too intensively utilized. It turns out that the processor usage varied between 0% and 14.833%. This is illustrated in Figure 4-13, in which CPU utilization is plotted against sample number (oldest first). The reason for the difference in processing is due to the fact that the amount of work done by the threads in the ??period varies due to the performance of the *Sleep* instruction (refer to 4.2.2). Therefore the interval ?'s size varies depending on the duration of the *Sleep* interval.

However on average, processor utilization was 4.661%. The result is that the routines that use the TIW timer at a frequency of 1 kHz have on average ± 95.339% of the CPU to its disposal. It is clear from this result that that a waiting period was successfully induced and that an accurate 1 ms interval could still be generated.

Therefore the timer exhibits a resolution of 1ms ± 50?s with an average CPU load requirement of 4.661%. The CPU usage for the entire period the timer was running is depicted in Figure 4-13. However, this is a best-case scenario, as was determined after exhaustive testing. The deviation from the actual interval of 1000 ?s that is required never exceeded 500 ?s – never reaching this number in actual fact. This is illustrated in Figure 4-14. Section 5.4 discusses a further study in which the TIW timer was able to retain its accuracy over a period of one hour.

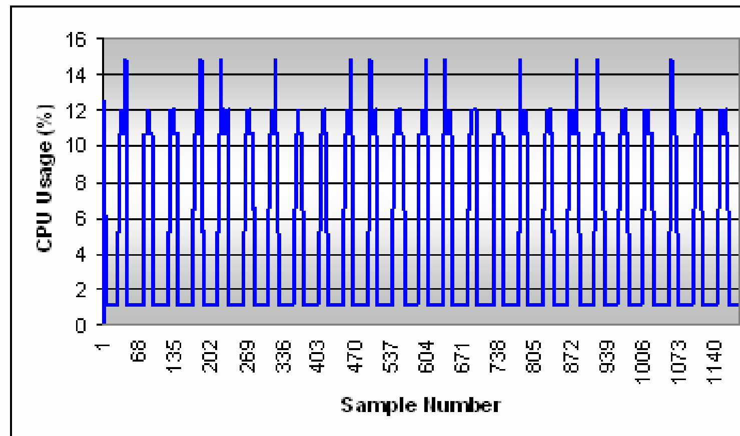

Figure 4-13: 1 kHz TIW timer CPU usage

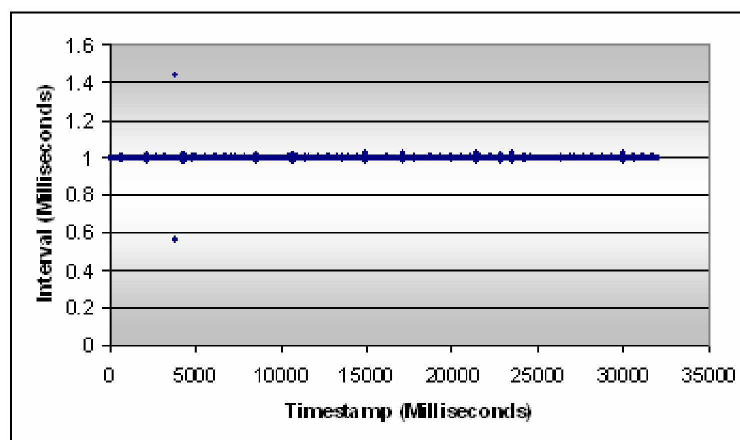

Figure 4-14: TIW timer 1 kHz Worst Case

Figure 4-14 illustrates the worst case recorded during the investigation of the TIW timer. The maximum interval in this case was 1.437ms (1437?s) with a minimum of 0.563ms (563?s). However, as is clear from the figure, an interval duration greater than 1.2 ms is an isolated event.

4.3.3.2 TIW timer (50 Hz Interval)

The results of this algorithm for a frequency of 50 Hz, is illustrated in Figure 4-15 and shows the performance over a period of 60 seconds. One would expect that if the TIW timer exhibits satisfactory results as in 4.3.3.1 – for larger intervals the performance should be better. The command line options are:

```
Number of timestamps          - 3000
Interval Duration             - 1000 ?s
Requested Sleep Interval      - 1 ms
Number of threads to use      - 2
The "mod"                     - 20
```

Note that with such a configuration, the TIW timer operates in exactly the same fashion as in the case of the TIW timer in the 1 kHz case in that each thread measures 1000 ?s intervals. However, the timer event is issued after every 20th

interval (in accordance with the "*mod*" command line option), instead of after each 1 ms interval.

Over a period of 60 seconds the maximum interval recorded was 20.018 ms and the minimum interval recorded 19.982 ms. Therefore an interval with a maximum deviation of approximately 18 ?s is achieved.

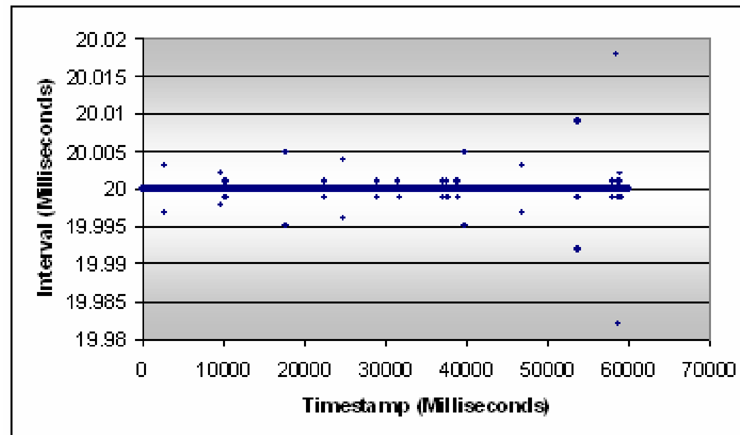

Figure 4-15: TIW timer (50 Hz Interval)

The processor usage varies between 0% and 14.833%, as was the case with the 1 kHz timer. The average processor utilization was 4.553%, again leaving the routines using the timer with around 95.447% of CPU resources on average. The CPU usage is plotted against sample number (oldest first) in Figure 4-16.

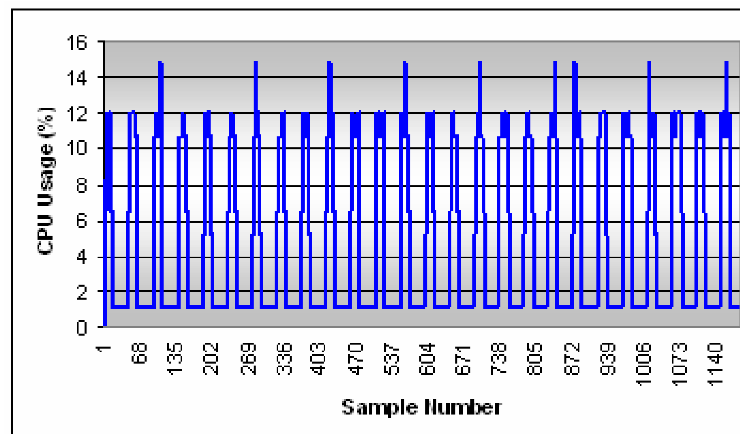

Figure 4-16: 50 Hz TIW timer CPU usage

Therefore the timer exhibits a resolution of 20 ms ± 18 ?s with an average CPU load requirement of 4.553%. This conforms to the requirement of a soft real-time timer in section 2.6.

4.3.3.3 TIW timer (1Hz Interval)

The results of this algorithm for a frequency of 1Hz, is illustrated in Figure 4-15 and shows the performance over a period of 65 seconds. Over this period, the maximum interval recorded was 1.000006 seconds and the minimum interval recorded 0.999994 seconds. Therefore an interval with a maximum deviation of approximately 6 ?s is achieved.

This was accomplished with the TIW timer configured as follows:

```
Number of timest amps          - 65
Interval Duration              - 1000 ?s
Requested Sleep Interval       - 1 ms
Number of threads to use       - 2
The "mod"                      - 1000
```

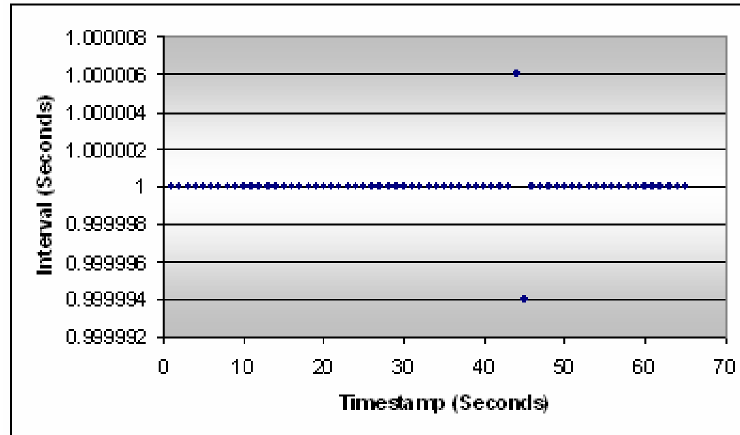

Figure 4-17: TIW timer (1 Hz Interval)

The processor usage varies between 0% and 14.833%. The average processor utilization was 4.902%. The CPU usage is therefore more or less consistent, regardless of interval that the timer is measuring. This is illustrated plotted against sample number (oldest first) in Figure 4-18.

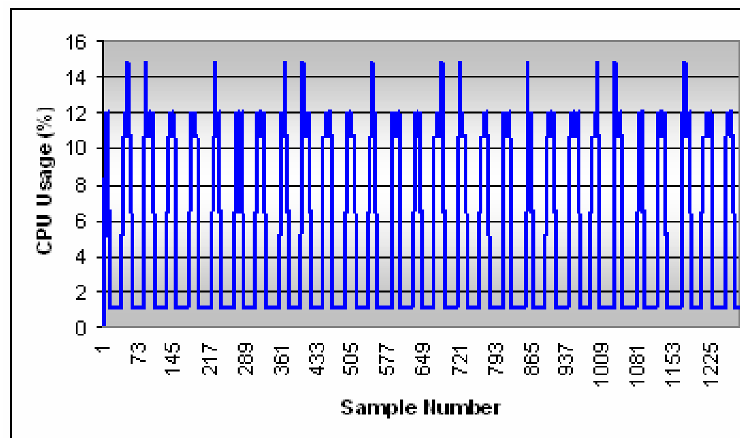

Figure 4-18: 1 Hz TIW timer CPU usage

4.3.4 An alternative TIW timer

The WIN32 Sleep Timer has the tendency to sleep for a little longer than required – 1 ms longer in the case discussed for a frequency of 1 kHz (see 4.2.2 where both the maximum and minimum intervals recorded were greater than 1 ms by ± 1 ms). The TIW timer was designed to use two overlapping threads that operate in tandem, thus compensating for the sleep timer's inaccuracy at high frequencies. The idea arose of basing a timer on one execution thread only, instead of two. The sleep interval is

chosen close to the desired interval and the impact on the performance of the TIW timer is investigated.

The design of the alternative method fits in with that of the TIW timer. In fact, the ability to configure the timer allows it to be set up in such a way that only one execution thread is used. This thread is requested to sleep for an interval of 1 ms less than the required interval. For example, in the case of the 20 ms interval (50 Hz frequency), the timer is configured to use one thread and its sleep interval is set to 19 ms.

This renders the operation of the TIW timer similar to that of the sleep timers in section 4.2. The results are to follow in the subsequent sections.

4.3.4.1 TIW timer Alternative (50 Hz Frequency)

As was stated before, the TIW timer did not need to be modified for this investigation. To achieve the required interval the TIW timer could be configured via its command line options detailed in section 4.3.1 to run with only one thread.

This configuration is as follows:

```
Number of timestamps        - 3000
Interval Duration           - 20000 ?s
Requested Sleep Interval    - 19 ms
Number of threads to use    - 1
The "mod"                   - 1
```

The results of the alternative algorithm for a frequency of 50 Hz, is illustrated in Figure 4-19 and shows the performance over a period of 60 seconds. Over this period, the maximum interval recorded was 20.005 ms and the minimum interval recorded – 19.995 ms. Therefore an interval with a maximum deviation of approximately 5 ?s is achieved. Immediately it is clear that in terms of accuracy, configuring the TIW timer this way results in the same performance as when multiple execution threads are used.

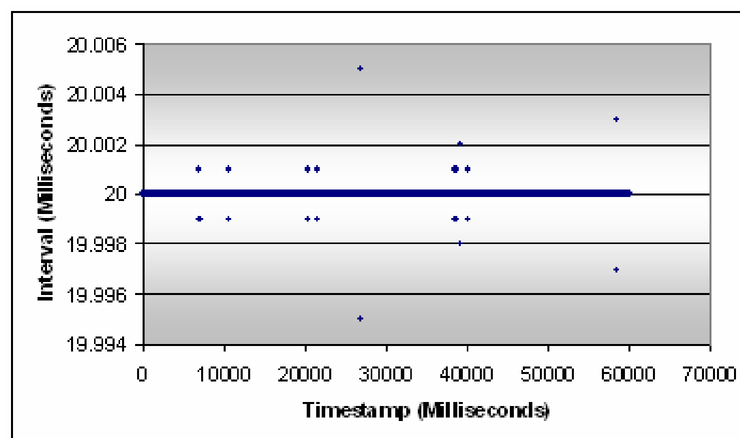

Figure 4-19: TIW timer Alternative (50 Hz Interval)

The processor usage varies between 0% and 16.5%. The average processor utilization was 2.990%. Therefore, in terms of processor usage, there is not too much of an improvement but an improvement none the less. This is illustrated plotted against sample number (oldest first) in Figure 4-20.

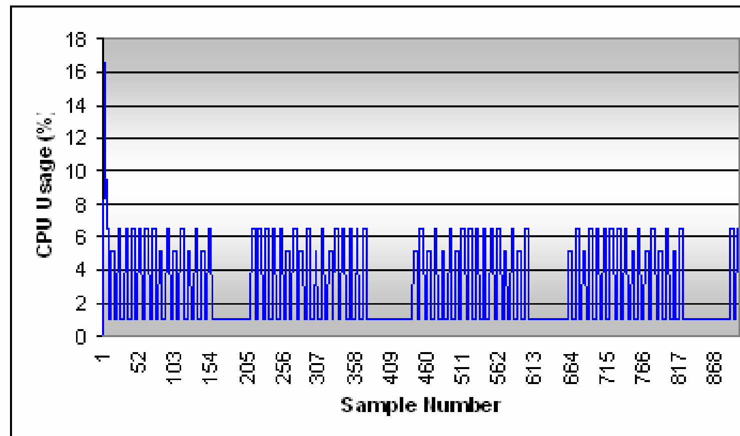

Figure 4-20: 50 Hz TIW timer Alternative CPU usage

Although its accuracy is the same and its average processor usage is slightly less than the "normal" TIW timer at 50 Hz (refer to 4.2.2.2.2), the overhead of continuously switching between multiple threads is eliminated in this alternative configuration. Where the normal TIW timer would be switching between two threads for the duration of the 20 ms interval, the TIW timer in this alternative configuration would be suspended for the better part of the interval.

4.3.4.2 TIW timer Alternative (1Hz Frequency)

The alternative solution was applied to the 1 Hz frequency timer as well.

The achieve the one second interval, the TIW timer was configured as follows:

```
Number of timestamps          - 60
Interval Duration             - 1000000 ?s
Requested Sleep Interval      - 999 ms
Number of threads to use      - 1
The "mod"                     - 1
```

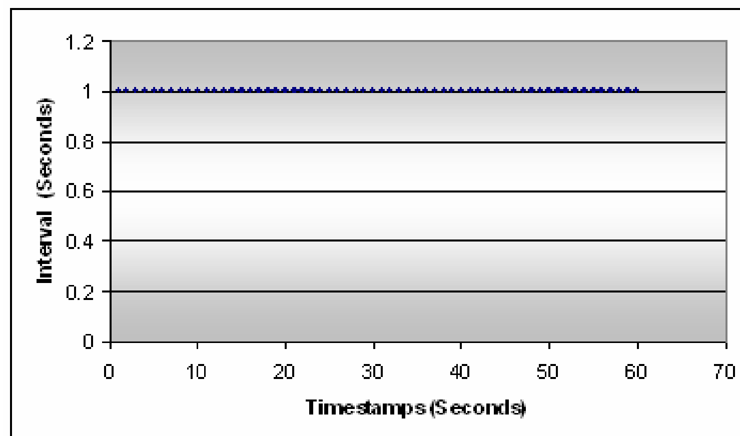

Figure 4-21: TIW timer Alternative (1 Hz Interval)

The results of the alternative algorithm at a frequency of 1 Hz, is illustrated in Figure 4-21 and shows the performance over a period of 60 seconds. Over this period, the maximum interval recorded was 1.000001 seconds and the minimum interval

recorded 0.999999 seconds. Therefore an interval with a maximum deviation of approximately 1 ?s is achieved.

The processor usage varies between 0% and 84.333%. The average processor utilization was 2.446%. This is illustrated plotted against sample number (oldest first) in Figure 4-22. As is clear from the figure is that the usage seems to spike initially to the maximum usage followed by a drop to below 2% for the remainder of the timer's operation.

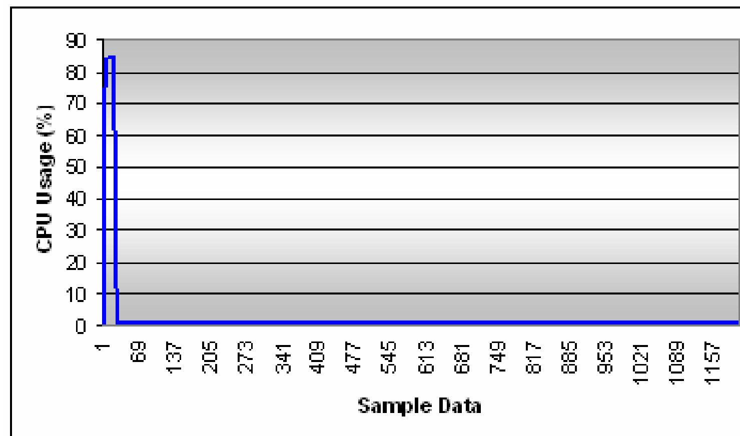

Figure 4-22: 1 Hz TIW timer Alternative CPU usage

The reason for this initial spike is due to the design of the TIW timer (refer to section 4.3.1. According to Algorithm 5, thread A will enter the loop and measure the required interval, which in this case is ± 999 ms. The first sleep instruction is only issued after thread A has measured the first interval, therefore, for the first 999 ms, the TIW timer consumes all available CPU resources. However, the TIW timer in this alternative configuration uses significantly fewer resources after this initial interval than its "normal" counterpart.

4.3.4.3 TIW timer vs TIW timer Alternative

In the previous sections it was shown that the TIW timer alternative solution based on one execution thread only, outperforms the two execution thread configuration for frequencies 50Hz and 1Hz in terms of CPU usage and thread switching overhead. Clearly, since it is impossible to generate a reliable 1 kHz frequency with a single thread (see section 4.2.2), the TIW alternative timer cannot be constructed to run at this frequency. However, it remains to be determined whether its better resolution would be retained across a range of frequencies lower that 1 kHz. This is the next matter to be explored, starting with a 500 Hz timer (2 ms intervals).

The first interval that is compared between the two versions of the timer has duration of 2 ms. To achieve the interval using the normal version of the timer, the configuration was as follows:

```
Number of timestamps        - 30000
Interval Duration           - 1000 ?s
Requested Sleep In terval   - 1 ms
Number of threads to use    - 2
The "mod"                   - 2
```

To achieve the same results using the alternative method, the configuration is altered slightly, with the resulting configuration as follows:

```
Number of timestamps         - 30000
Interval Duration            - 2000 ?s
Requested Sleep Interval     - 1 ms
Number of threads to use     - 1
The "mod"                    - 1
```

The performance of the normal TIW timer is shown in Figure 4-23. The performance is shown over a period of 60 seconds.

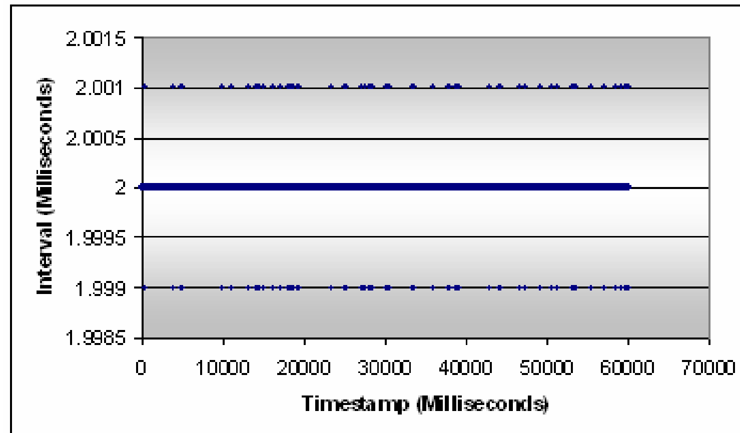

Figure 4-23: Normal TIW timer 500 Hz

A maximum of 2.001 ms (2128 ?s) and a minimum of 1.999 ms (1999 ?s) were recorded. Therefore, the resolution in this case is 2 ms ± 1 ?s is achieved. The average CPU usage was 4.873%, as illustrated in Figure 4-24.

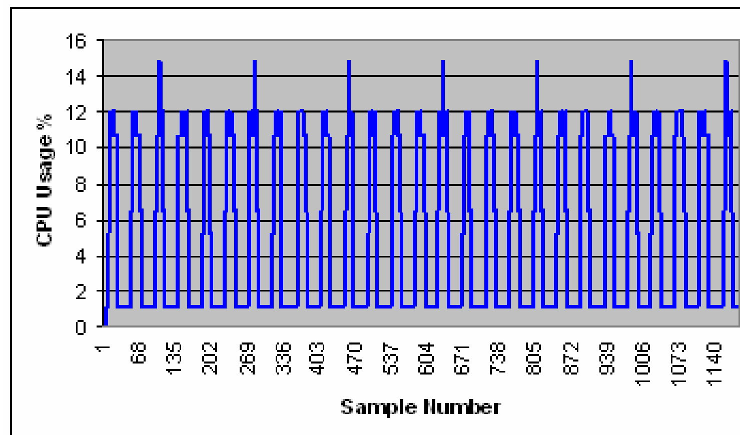

Figure 4-24: 500 Hz Normal TIW timer CPU Usage

Using the alternative version of the timer is presented in Figure 4-25. The period show in the figure is 60 seconds. A maximum of 2.015 ms (2015 ?s) and a minimum of 1.985ms (1985?s) were recorded. Therefore, the resolution in this case is 2ms ± 15?s is achieved. T

Therefore the normal TIW timer yielded a marginally better performance. However, as is clear from Figure 4-25, the 15 ?s spike in the alternative version's results occurred once. For the rest of the alternative timer's operation, the interval deviation

remained below 5 ?s. The conclusion therefore is that the performance of the two timers in this case is essentially the same in terms of accuracy.

However, the average CPU usage was 2.925% as apposed to the 4.873% of the normal solution. As is clear from Figure 4-24 and Figure 4-26, the alternative solution consumes a smaller percentage of the CPU.

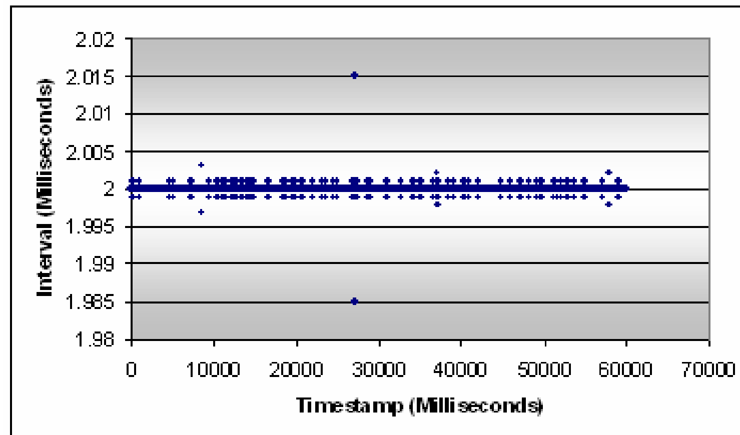

Figure 4-25: Alternative TIW timer (500 Hz)

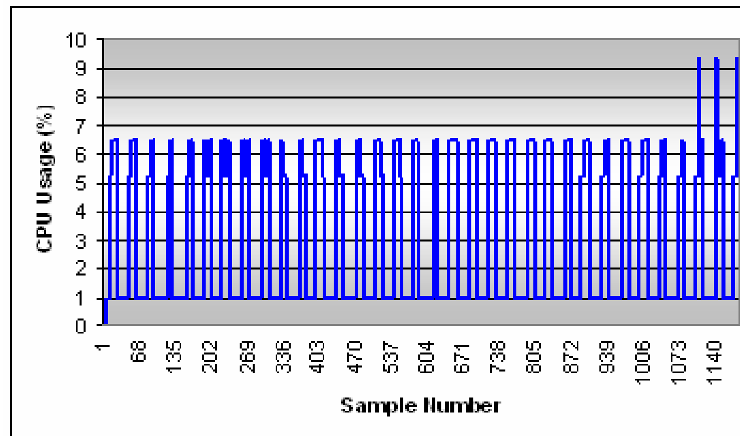

Figure 4-26: 500 Hz Alternative TIW timer CPU usage

For a interval of 3 ms, the configuration of the two timers were as follows:

? The normal TIW timer –

```
Number of timestamps          - 20000
Interval Duration             - 1000 ?s
Requested Sleep Interval      - 1 ms
Number of threads to use      - 2
The "mod"                     - 3
```

? The alternative TIW timer –

```
Number of timestamps          - 20000
Interval Duration             - 3000 ?s
Requested Sleep Interval      - 2 ms
Number of threads to use      - 1
```

The "mod" - 1

The normal TIW timer achieved a maximum of 3.232 ms and a minimum of 2.768 ms (refer to Figure 4-27). The average CPU usage was 4.826% illustrated in Figure 4-28. Again the experiment was conducted over a period of 60 seconds.

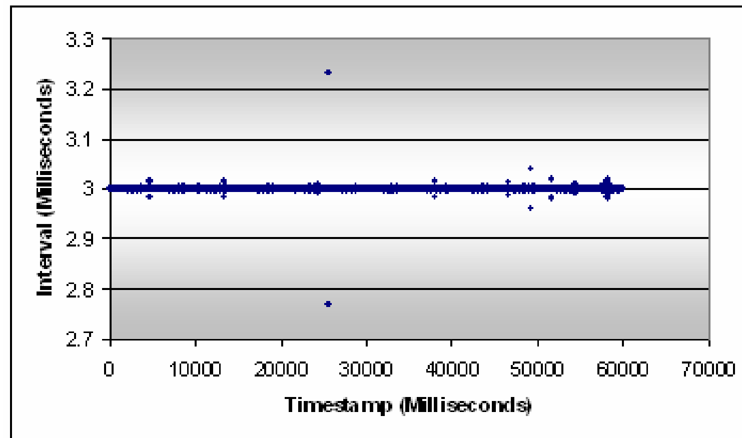

Figure 4-27: 333.33 Hz Normal TIW timer

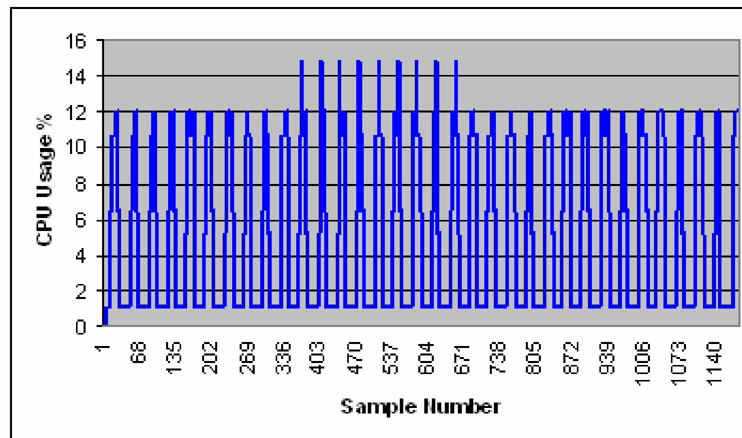

Figure 4-28: 333.33 Hz Normal TIW timer CPU usage

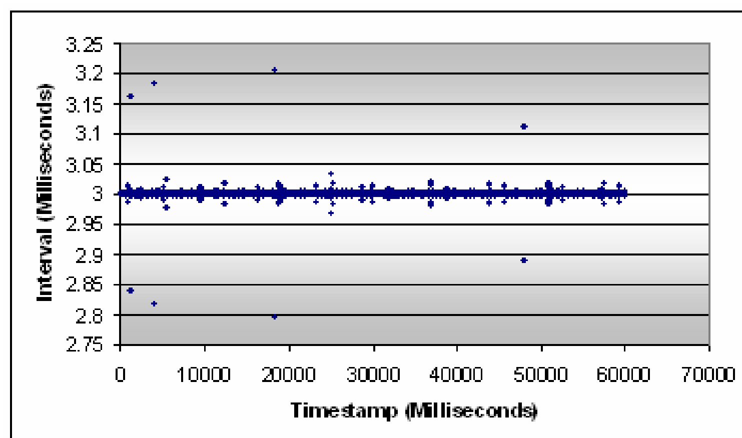

Figure 4-29: 333.33 Hz Alternative TIW timer

Janno Grobler
jannogrobler@gmail.com
M.Sc Computer Science
University of Pretoria

On the other hand, the alternative TIW timer achieved a maximum of 3.207 ms and a minimum of 2.793 ms (Figure 4-29). The average CPU usage was 2.915% as shown in Figure 4-30. These results yields resolutions of 3 ms ± 273 ?s and 3 ms ± 232 ?s – in other words, the performance is essentially identical in terms of accuracy.

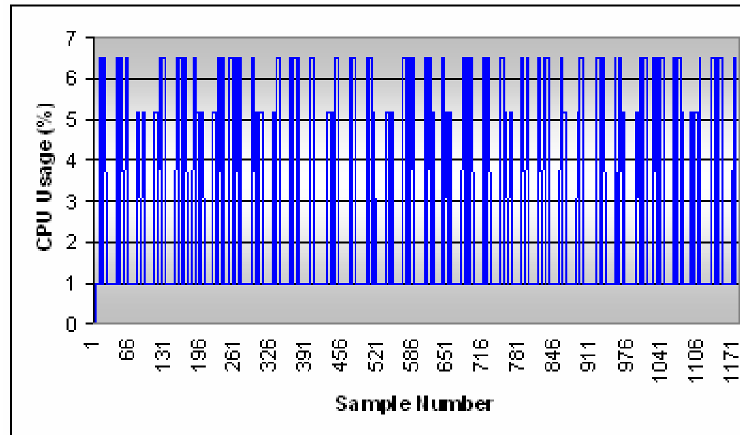

Figure 4-30: 333.33 Hz Alternative TIW timer CPU usage

The next interval under investigation was a 4 ms interval (250 Hz frequency). To achieve this interval, the timers were configured as follows:

? The normal TIW timer –

```
Number of timestamps          - 15000
Interval Duration             - 1000 ?s
Requested Sleep Interval      - 1 ms
Number of threads to use      - 2
The "mod"                     - 4
```

? The alternative TIW timer –

```
Number of timestamps          - 15000
Interval Duration             - 4000 ?s
Requested Sleep Interval      - 3 ms
Number of threads to use      - 1
The "mod"                     - 1
```

The normal TIW timer achieved a maximum of 4.360 ms and a minimum of 3.640 ms. The results for the normal TIW timer are illustrated in Figure 4-31. The average CPU usage was 4.841% (Figure 4-32).

The alternative TIW timer achieved a maximum of 4.234 ms and a minimum of 3.766 ms. Figure 4-33 illustrates the results of the alternative TIW timer. The CPU usage is shown in Figure 4-34. The average usage was 3.078%. These results yield resolutions of 4 ms ± 360 ?s and 4 ms ± 234 ?s in the two respective cases. Once more the results are essentially the same.

The conclusion therefore is that the normal TIW timer and its alternative configuration yield essentially the same performance in terms of accuracy. However, on the evidence at hand, the alternative configuration to the TIW timer yields better performance when it comes to CPU usage.

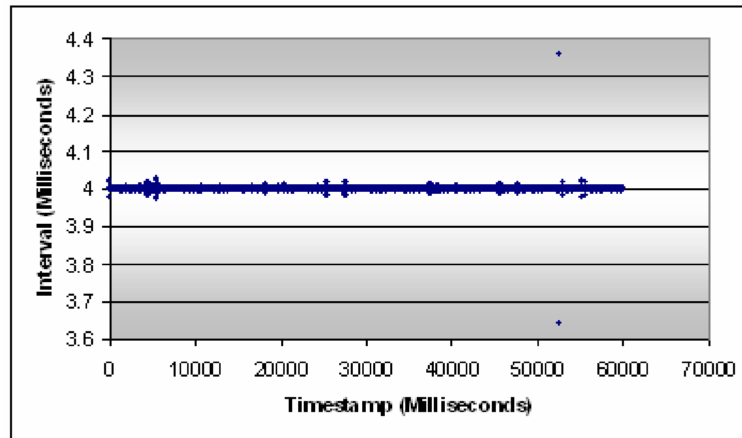

Figure 4-31: Normal TIW timer (250 Hz)

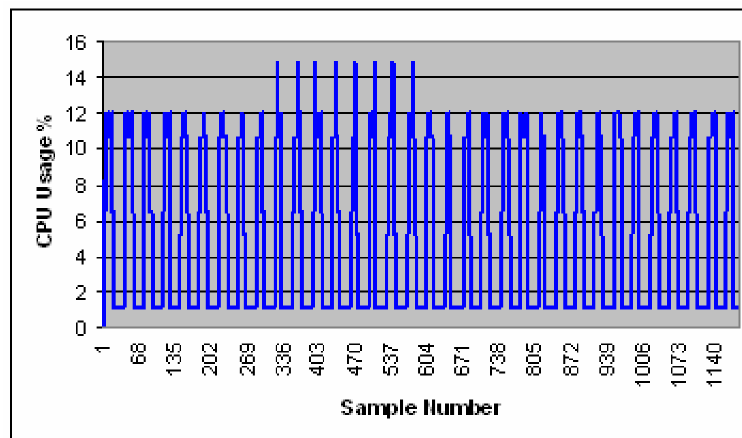

Figure 4-32: Normal TIW timer (250 Hz)

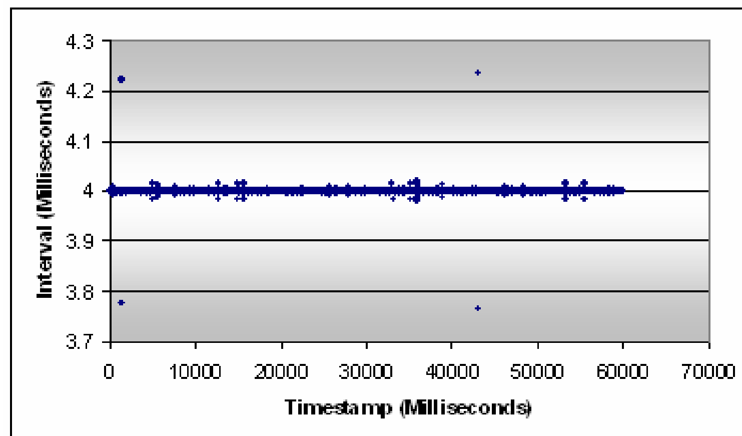

Figure 4-33: Alternative TIW timer (250 Hz)

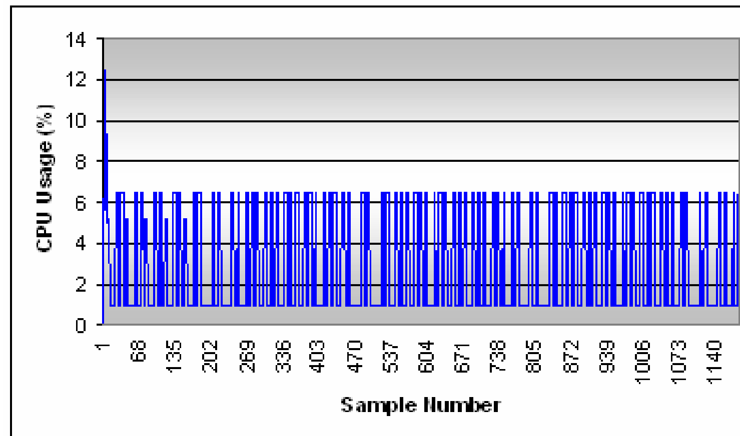

Figure 4-34: Alternative TIW timer (250 Hz) CPU usage

4.4 Comparison

This section compares the results from the TIW timer with current timers under the WIN32 and UNIX operating systems discussed in section 2.3 and 2.4. This serves to indicate whether the TIW timer's performance justifies it as an improvement over these timers. Since section 4.3.4 indicates that the performance of the alternative and normal configurations of the TIW timer are essentially the same in terms of accuracy, only the alternative solution is used for frequencies less than 1 kHz in the comparisons in section. This is due to the fact that it is less resource intensive.

Note that in each case, the maximum and minimum recorded intervals are presented in tabular form to ease the comparison. The values supplied are in the units noted in the column headings. The results for corresponding frequencies are tabulated in adjacent columns.

The results presented are also the "best" case scenario measured in section 4.3, as far as the TIW timer is concerned. Should the "worst" case scenario render performance below that of the existing timer under investigation, its results will be provided.

4.4.1 WIN32 Timers

The WIN32 API provides both the system timer (refer to section 2.3.1) and the multimedia timer (refer to 2.3.2). The performance of these timers is plotted against that of the TIW timer. These results are presented in subsections 4.4.1.1 and 4.4.1.2 of this section respectively.

4.4.1.1 System Timer

This section investigates the performance of the System timer versus the TIW timer. The result presented here is taken from the investigations in sections 2.3.1 and 4.3. The maximum and minimum intervals of each implementation are tabulated in Table 4-2.

The comparison in the table clearly indicates that the TIW timer is superior in every instance.

Table 4-2: System Timer vs. TIW timer

Interval	1kHz TIW timer (ms)	1kHz System Timer (ms)	50Hz TIW timer Alt. (ms)	50Hz System Timer (ms)	1Hz TIW timer Alt. (ms)	1Hz System Timer (s)
Maximum	1.036	25.431	20.018	36.151	1.000001	1.011667
Minimum	0.964	5.881	19.982	27.291	0.999999	0.988237

The following sections provide graphical representations of the results in Table 4-2.

4.4.1.1.1 1 kHz System Timer Comparison

Figure 4-35 Illustrates the difference between the minimum and maximum intervals produce by the System timer and TIW timer respectively.

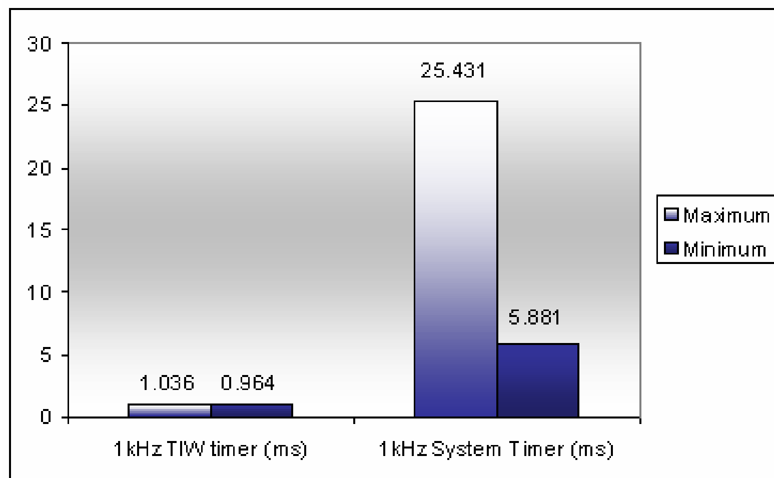

Figure 4-35: System Timer vs. TIW timer (1 kHz)

Since the desired interval size is 1 ms (1000 ?s), the TIW timer clearly outperforms the system timer with a maximum deviation in this case of 36 ?s as opposed to the ±25 ms deviation that the system timer shows. Not even the minimum interval recorded (5.881 ms) is close to the desired 1 ms interval.

Since the TIW timer was established to exhibit a maximum deviation of less than 500 ?s for the 1 kHz timer, the TIW timer clearly out performs the system timer.

4.4.1.1.2 50 Hz System Timer Comparison

Figure 4-35 illustrates the difference between the minimum and maximum intervals produced by the system timer and TIW timer respectively, for a frequency of 50 Hz. As was the case with the 1 kHz frequency (section 4.4.1.1.1), the TIW timer clearly outperforms the system timer, achieving the desired 20 ms (20000 ?s) within 500 ?s, as apposed to the 16.151 ms deviation in the case of the system timer. To add insult to injury, the TIW timer achieves this interval with a margin of error of 18 ?s in this case.

It is clear that the TIW achieves higher accuracy and is therefore shown to be superior yet again.

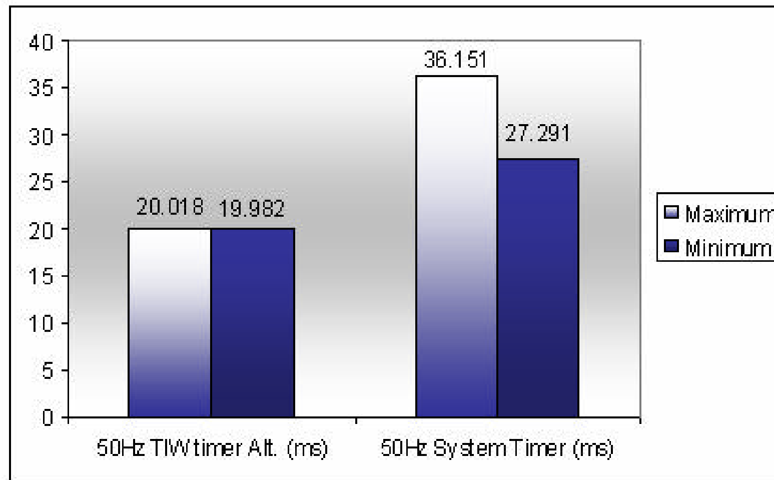

Figure 4-36: System Timer vs. TIW timer and TIW timer Alternative (50 Hz)

4.4.1.1.3 1 Hz System Timer Comparison

For a frequency of 1 Hz, the TIW timer achieves the required interval within 1 ?s, outperforming the system timer that suffers from a 11.667 ms maximum deviation. Therefore in the case of the 1 Hz frequency, the TIW timer outperforms the system timer.

This is illustrated in Figure 4-37.

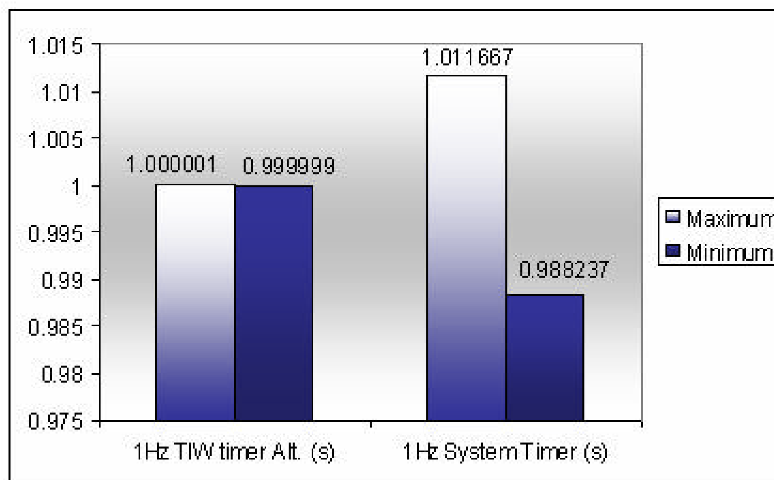

Figure 4-37: System Timer vs. TIW timer and TIW timer Alternative (1 Hz)

4.4.1.2 Multimedia Timer

This section investigates the performance of the multimedia versus the TIW timer. The maximum and minimum intervals of each implementation are tabulated in Table 4-3.

The result is a mixed bag since the multimedia timer provides good results for the 1 second (1 Hz frequency) and 20 ms (50 Hz frequency) intervals.

Table 4-3: Multimedia Timer vs. TIW timer

Interval	1KHz TIW timer (ms)	1KHz Multimedia Timer (ms)	50Hz TIW timer Alt. (ms)	50Hz Multimedia Timer (ms)	1Hz TIW timer Alt. (s)	1Hz Multimedia Timer (s)
Maximum	1.036	1.981	20.018	20.52	1.000001	0.999953
Minimum	0.964	0.545	19.982	19.516	0.999999	0.999941

4.4.1.2.1 1 kHz Multimedia Timer Comparison

Figure 4-38 illustrates the difference between the minimum and maximum intervals produced by the Multimedia timer and TIW timer respectively. As is indicated in the figure, the TIW timer provides better performance for a frequency of 1 kHz. The maximum deviation of the multimedia timer is clearly more than the 500 ?s that is a requirement of the TIW timer (refer to section 2.6) – clocking in at 981 ?s.

It is interesting to not that the multimedia timer generated an interval of less than 1 ms, and that it did so with a deviation of 455 ?s. It does not generate such an interval reliably though since the maximum interval was 1.981 ms

Therefore, the TIW timer is able to measure the interval with a smaller margin of error than the multimedia timer when a 1 kHz frequency is required.

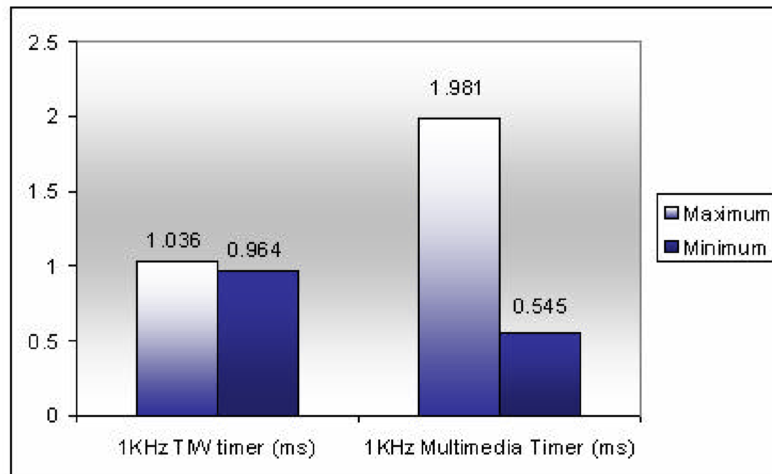

Figure 4-38: Multimedia Timer vs. TIW timer (1kHz)

4.4.1.2.2 50 Hz Multimedia Timer Comparison

The minimum and maximum intervals produced by the multimedia timer and TIW timer are depicted in Figure 4-39. As is clear from the figure, the maximums differ by a spread of ± 502 ?s and the minimum intervals by ± 466 ?s.

At first glance it seems as if the multimedia timer might conform the requirement that the maximum deviation should not be more than 500 ?s. However, if we take the results in sections 4.3.3.1 and 4.3.4.1 into account, the TIW timer outperforms the multimedia timer again with a maximum deviation of 18 ?s as opposed to the 520 ?s exhibited by the multimedia timer. Therefore, as was the case in with the 1 kHz timers in section 4.4.1.2.1, the TIW timer is justified as a replacement for the multimedia timer.

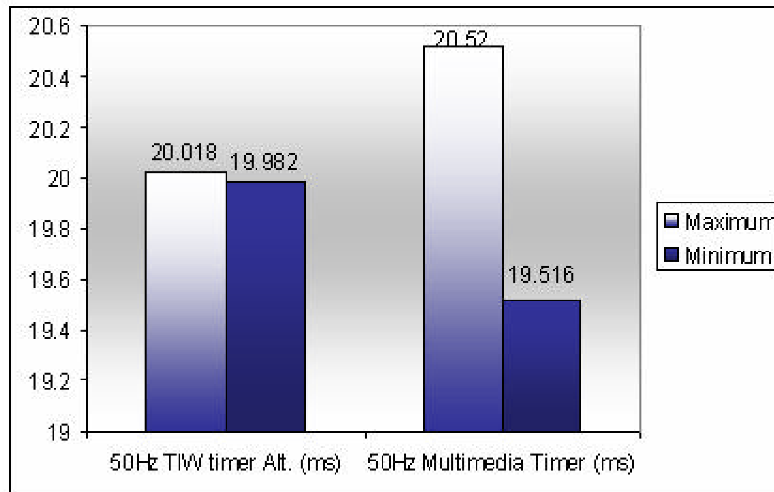

Figure 4-39: Multimedia Timer vs. TIW timer (50Hz)

4.4.1.2.3 1 Hz Multimedia Timer Comparison

As was the case in the previous sections, Figure 4-40 illustrates the difference between the minimum and maximum intervals recorded by the multimedia timer and TIW timer respectively. The results are even closer together than that of the 50 Hz TIW timer and 50 Hz multimedia timer.

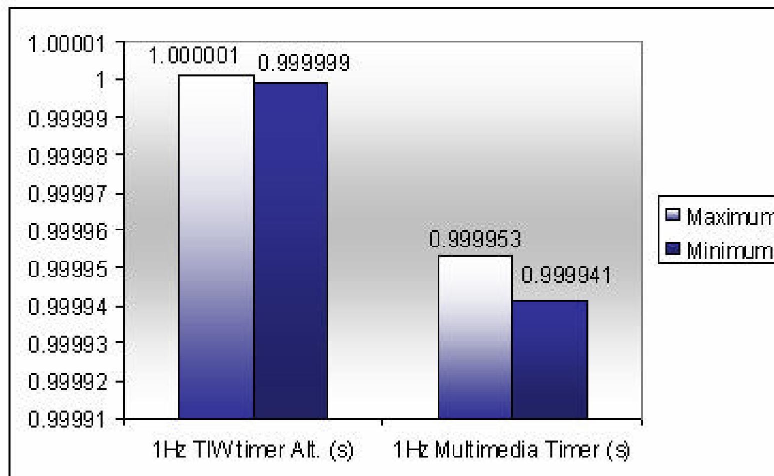

Figure 4-40: Multimedia Timer vs. TIW timer (1Hz)

Again the TIW timer outperforms the multimedia timer exhibiting a maximum deviation of 1 ?s. The multimedia timer on the other hand never recorded an interval duration closer than 47 ?s to the target of 1 second. The multimedia timer exhibits a maximum deviation of 59 ?s although it should be noted that in this specific case, the result is satisfactory for the soft real-time timer specified in section 3.

Therefore, either the multimedia timer or the TIW timer is suitable for this interval. However, since the TIW timer outperforms the multimedia timer on the other two intervals, its development is justified.

4.4.2 POSIX Timer

This section compares the performance of the TIW timer against that of the POSIX timer. It also compares the TIW timer under the UNIX and WIN32 environments. The first investigation, which is a comparison between the WIN32 TIW timer and the timer provided by the POSIX API, serves to illustrate why the soft real-time may not be realised under a normal distribution of the Linux platform. Remember that this text does not take real-time extensions like Real-Time Linux into account, as the goal of this dissertation was the development of a soft real-time timer without such assistance.

The second investigation aims to show whether the TIW timer implementation is viable on the Linux platform. The TIW timer has been implemented to be portable to the POSIX environment, as was illustrated by its design in section 4.3.1.

Firstly, a direct comparison is made between an implementation of the POSIX timer under the Linux environment using a 2.6.x kernel and the WIN32 TIW timer. This is tabulated in Table 4-4.

Table 4-4: POSIX Timer vs TIW timer

Interval	1KHz TIW timer (ms)	1KHz POSIX Timer (ms)	50Hz TIW timer Alt. (ms)	50Hz POSIX Timer (ms)	1Hz TIW timer Alt. (s)	1Hz POSIX Timer (s)
Maximum	1.036	7.976	20.018	28.014	1.000001	1.011517
Minimum	0.964	1.07	19.982	20.885	0.999999	1.000083

4.4.2.1 WIN32 TIW timer vs. POSIX Timer.

As is observed in Figure 4-41 the interval recorded with the POSIX timer deviates from the required interval of 1 ms by up to ± 7 ms. The WIN32 TIW timer is able to measure the 1 ms interval within a maximum deviation of 500 ?s. Figure 4-42 represents the performance of the TIW timer versus the POSIX timer at a frequency of 50 Hz. The latter exhibits an eight ms deviation against the maximum deviation of ±5 ?s exhibited by the TIW timer (refer to section 4.3.4.1).

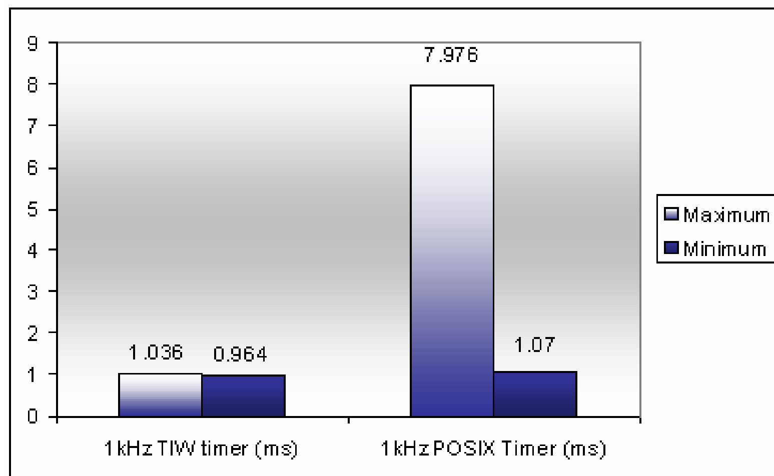

Figure 4-41: 1kHz TIW timer vs POSIX Timer

Figure 4-43 reinforces the fact that the POSIX timer is not reliable. It has a deviation of ± 12 ms for a required interval of 1 second (1 Hz frequency). Again the TIW timer

outperformed the POSIX timer with a maximum deviation of 1 ?s (refer to section 4.3.4.2).

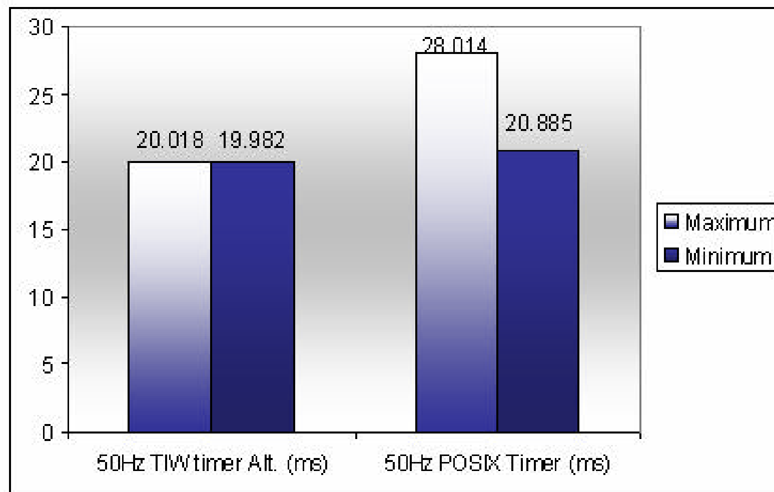

Figure 4-42: 50Hz TIW timer vs. External Timer

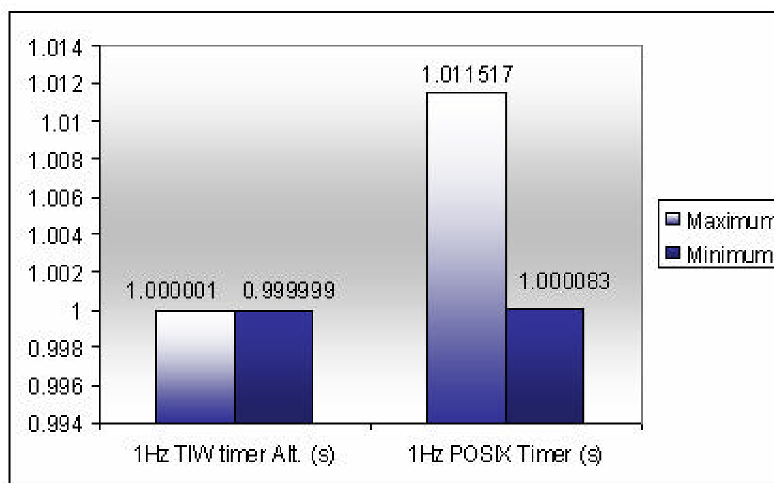

Figure 4-43: 1Hz TIW timer vs. External Timer

The reason for this problem under the Linux platform is similar to that of the WIN32 platform as the Linux timer objects are serviced at each system timer interrupt that occur at a rate of 100Hz [Yoav et al. 2003]. Another reason is the ability for a user process to pre-empt a kernel level process. In section 3.4 it was stated that certain kernel level processes are not preemptable.

Various extensions to the Linux operating system exist that address these issues. An example of this is the Kansas University RT Linux (or KURT) that claim to provide "*Microsecond timing resolution and event-driven real-time scheduling*" [Atlas et al. 1998] as well as the real-time Linux foundation that endeavours to provide a common real-time platform based on Linux [Wurmsdobler 2002].

However, as was stated in section 3.3, extensions such as these were not taken into account since the goal was to achieve the solution without them. The bottom line is that the WIN32 TIW timer outperforms the Linux POSIX timer.

4.4.2.2 WIN32 TIW timer vs. POSIX TIW timer.

The logical next step is to implement the TIW timer under Linux and measure its results. Obviously the architecture and performance of the operating systems differ and therefore the performance of the TIW timer may not be necessarily the same on both platforms.

With minor modifications, the TIW timer was successfully implemented under the Linux operating system. Figure 4-44 illustrates the timestamps generated by the TIW timer on the Linux platform for a 1 ms interval timer. The maximum interval recorded was 5.001 ms (5001 ?s) and a minimum of 1.001 ms (1001 ?s). This results in a resolution of 1 ms ± 4.001 ms. This is a far cry from the excellent performance of the TIW timer under the WIN32 operating system for this interval which meets the requirement of the resolution of 1 ms ? 500 ?s.

Therefore the WIN32 TIW timer outperforms the POSIX TIW timer when a frequency of 1 kHz is required.

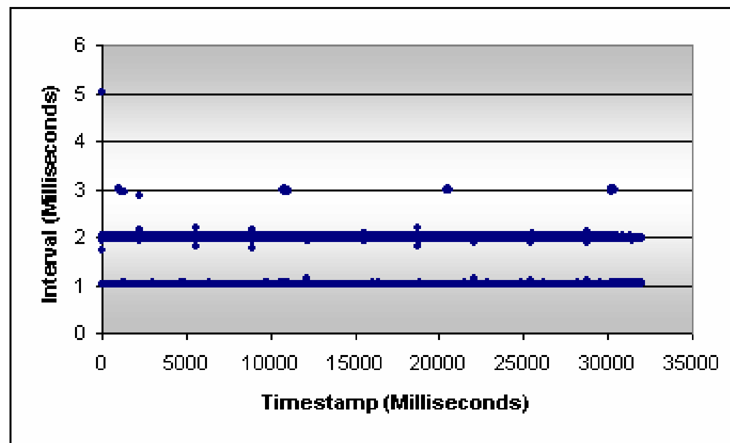

Figure 4-44: 1 kHz POSIX TIW timer

Figure 4-45 show the intervals generated by the POSIX TIW timer at 50 Hz. Once again the performance of the WIN32 TIW timer could not be matched with a maximum of 40.019 ms and a minimum of 29.871 ms. Therefore the resolution in this instance is 20 ms ± 20.019 ms, which is not comparable to the excellent resolution of the alternative WIN32 TIW timer – 20 ms ± 18 ?s.

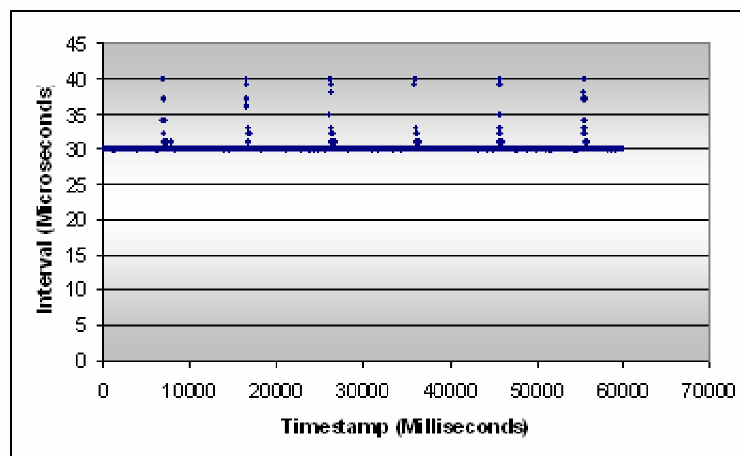

Figure 4-45: 50Hz POSIX TIW timer

The same is true of the 1 Hz POSIX TIW timer in Figure 4-46. The maximum recorded in this instance was 1.573653 seconds and a minimum of 1.499666 seconds. Therefore the resolution in this case is a staggering 1 second ± 574 ms. There is a vast difference between this resolution and the 1 second ± 1 ?s resolution of the 1Hz alternative WIN32 Timer.

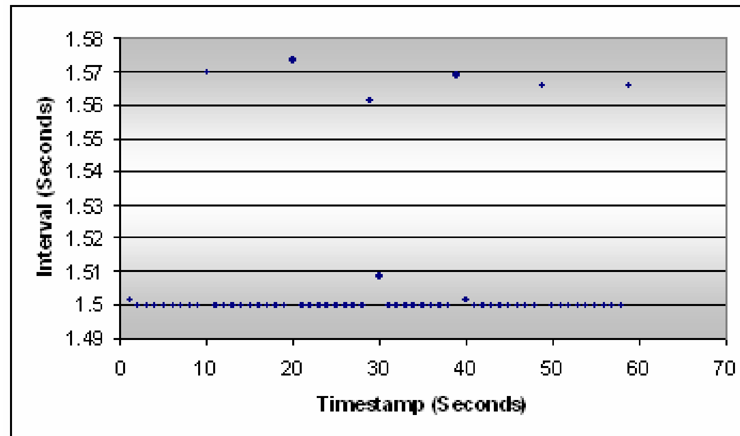

Figure 4-46: 1 Hz POSIX TIW timer

The TIW timer is heavily dependant on the amount of time that the "sleep" instruction suspends the execution threads. This interval of time was more or less consistent under the WIN32 platform (section 4.3). The main reason for the TIW timer ineffectiveness is the WIN32 "*Sleep*" instruction's replacement under the POSIX platform – "*nanosleep*". This instruction may also be instructed to suspend an execution thread for an elapsed time of 1 ms. However, the amount that the instruction sleeps is not consistent, rendering the design of the TIW timer ineffective in the POSIX environment [Locke 2005].

4.5 Benchmark (External Timer)

In this section, the TIW timer is compared to a true real-time timing source: the external timer discussed in section 2.5. These results are a testament to the true performance capabilities of the TIW timer, since these external timers are examples of hard real-time timing mechanisms.

The maximum and minimum interval of each implementation is tabulated in Table 4-5. Unlike the previous results, in this section, the timer with which the TIW timer is compared actually outperforms the TIW timer – hence the external timer's use as a benchmark.

Table 4-5: External Timer vs. TIW timer

Interval	1kHz TIW timer (ms)	1kHz External Timer (ms)	50Hz TIW timer Alt. (ms)	50Hz External Timer (ms)	1Hz TIW timer Alt. (s)	1Hz External Timer (s)
Maximum	1.036	1.047	20.018	20.094	1.000001	1.000017
Minimum	0.964	0.953	19.982	19.906	0.999999	0.999983

4.5.1 1 kHz TIW timer vs. External Timer.

Figure 4-47 depicts the difference between the minimum and maximum intervals produce by the external timer and TIW timer respectively. Unlike previous results, the external timer and TIW timer are almost identical when the best case performance of

the TIW timer is taken into account. The difference between the 1 ms intervals recorded for the timers are within 50 ?s with resolutions of 1 ms ± 36 ?s and 1 ms ± 47 ?s for the TIW and external timers respectively.

Likewise, the resolutions of the TIW and external timers for an interval 20 ms (50 Hz Frequency) are 20 ms ± 18 ?s and 20 m ± 94 ?s (Figure 4-48). The resolutions for the 1 Hz timer is 1s ± 1?s for the TIW timer and 1 second ± 17 ?s for the external timer (Figure 4-49).

The trend of these resolutions seems to suggest that the TIW timer outperforms the external timer. However, the TIW timer only stays within 500 ?s of the target interval, whereas the external timer stays within 100 ?s (refer to section 4.3.3.1).

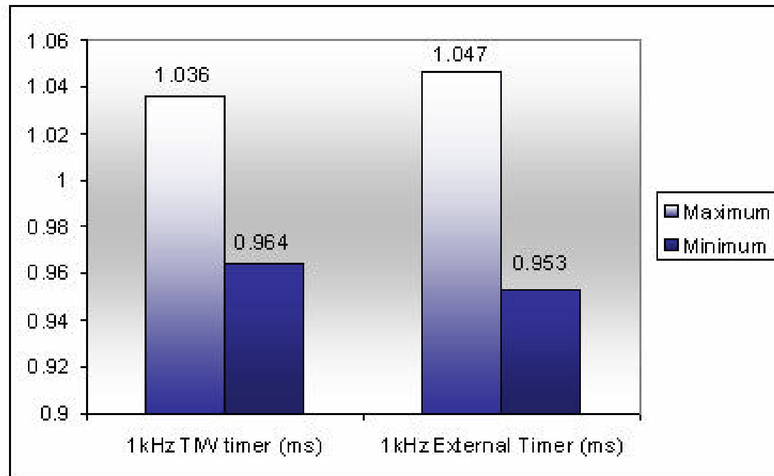

Figure 4-47: 1 kHz TIW timer vs. External Timer

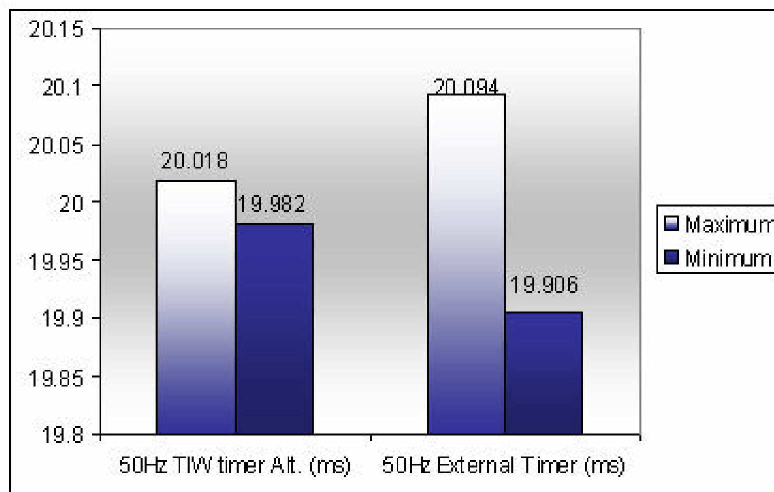

Figure 4-48: 50 Hz TIW timer vs. External Timer

As the external timer is based on an oscillating crystal, one would expect the external timer not to exhibit any deviation from the desired interval. However a discrepancy is detected (also mentioned in section 2.5.1). This discrepancy may be attributed to operating system, being induced by the time that elapses between the arrival of the signal at the serial port to the arrival of the signal at the application that is expecting it. Another contributing factor is the amount of time required for the Timestamp

calculator (see section 4.1) to read the high-precision counter and calculate the timestamp.

Since the performance accuracy of the external timer is closely tied with the performance of the operating system, the TIW timer may be concluded to merely circumvent operating system deficiency.

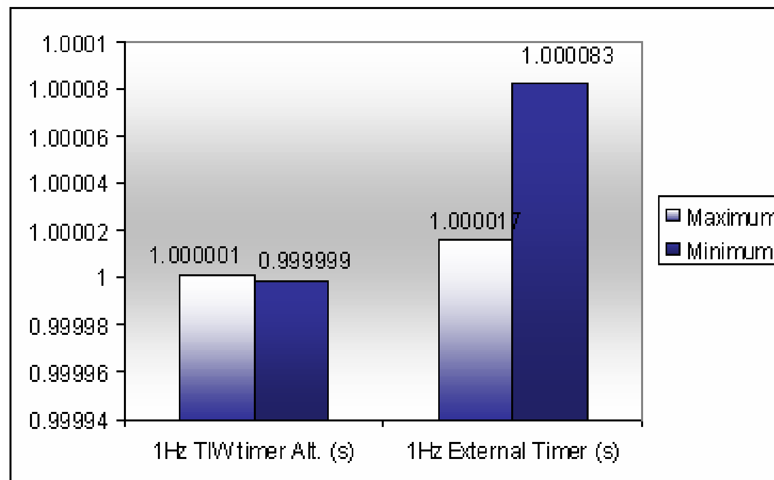

Figure 4-49: 1 Hz TIW timer vs. External Timer

The bottom line of this comparison is that the TIW timer is able to reliably generate intervals within 500 ?s of the interval duration that the hard real-time external timer is able to generate.

4.6 Timer Performance

This chapter presented both rejected and final solutions to the problem of developing a soft real-time timer that fits the description in section 2.6.

? The TIW timer was shown be exhibit a margin of error of 500 ?s or less (section 4.3.3)

? The TIW timer was shown to use minimal system resources. The average usage was quantified to remain below 5 % (section 4.3.3)

? The TIW timer was designed to fire a timer event at the conclusion of each interval (section 4.3.1)

Therefore, the TIW timer meets the requirement of a soft real-time timer discussed in section 2.6. The TIW timer was also shown to outperform the existing timers discussed in section 2 in terms of accuracy. Of the timers discussed, the loop timer was found to be the most accurate, measuring a one millisecond interval with a maximum deviation of 167 ?s (refer to section 2.2.2). The TIW timer is able to do so reliably with a margin of error less than 500 ?s. This margin of error is half that of the multimedia timer which can only produce 1 ms intervals with one millisecond of the desired interval (refer to section 2.3.2.1).

When the issue of CPU consumption is investigated, the TIW timer performs very well. The main drawback of an implementation such as the loop timer is that it put considerable strain on the processor. The consumption of the TIW timer remains on average below 5 %, as was shown in section 4.3.3. When the alternative configuration of the TIW timer is used for frequencies lower than 1 kHz, the average usage remains below 3 % (section 4.3.4). It is true that the multimedia timer is superior in this area, consuming an average of 0.9957% of the available resources

(section 2.3.2.4). However, as stated, the TIW timer cuts the margin of error exhibited by the multimedia timer by half. These may be summarised as the benefits of the TIW timer, as well as being a justification for its development.

The TIW timer does have one significant disadvantage however – the solution is ineffective under POSIX based platforms such as Linux.

This section was restricted to testing the timer's accuracy. However, further testing is required to attest to primarily the stability of the TIW timer. This is the subject of the next chapter.

Chapter 5

Further Testing

This chapter presents further testing of the TIW timer, since chapter 4 only covered the accuracy of the timer. It also aims to answer the question as to whether there is more to the TIW timer than just being able to generate a reliable 1ms interval. Section 5.2 looks into this question. But before that, section 5.1 investigates the effects of load on the TIW timer, whether it be load on a single timer, multiple timers on a single machine or the effects of running the timer along with other multimedia applications. Section 5.3 investigates the effect on the timer when the number of execution threads are varied, and finally section 5.4 investigates the performance of the TIW timer in a real-world application.

5.1 Load Testing

A timer with accurate time measurement is desirable, and the TIW timer has demonstrated this capability in section 4.3.3. However, a timer is of little value if it cannot be used in practical applications. Therefore, in addition to providing accuracy, it must demonstrate the same desirable performance when placed under strain. This section investigates the viability of the timer in terms of load – in other words if the timer can maintain its accuracy.

5.1.1 Load on a single timer

In this investigation, a single timer is placed under load. The timer is tested under both reasonable and unreasonable load (overload). Section 5.1.1.1 investigates the TIW timer's performance under reasonable load. The timer fires a timer event for a task that will consume all available CPU resources for a time period fractionally smaller than the interval measured by the timer.

Section 5.1.1.2 investigates the TIW timer's performance under overload, in other words the timer fires a timer event for a task that will consume 100% of the CPU resources for a time period equal or greater than the desired interval measured by the timer. In this case, the timer will be in a position where it has to "share" CPU resources with the application using it.

5.1.1.1 Reasonable Load

Figure 5-1 presents the results for the TIW timer under reasonable load. The timer is configured to measure an interval of 1 ms. The load consists of a function that measures an interval of 700 ?s using the loop timer algorithm discussed in section 2.2.2. Since the loop timer consumes all available CPU's resources, 100% of the processor is unavailable for a period of 700 ?s in each 1000 ?s interval measured by the TIW timer.

The maximum interval recorded was 1.402 ms and the minimum 0.598 ms. Therefore, under reasonable load a single timer demonstrates a resolution of 1 ms ? 402 ?s. The deviation is within 500 ?s. Therefore the TIW timer is still accurate and still maintains the required resolution of 1 ms ± 500 ?s.

The maximum percentage of CPU resources consumed at a given time was 80%. Note that this includes the resource usage introduced by the load as well. This is shown in Figure 5-2.

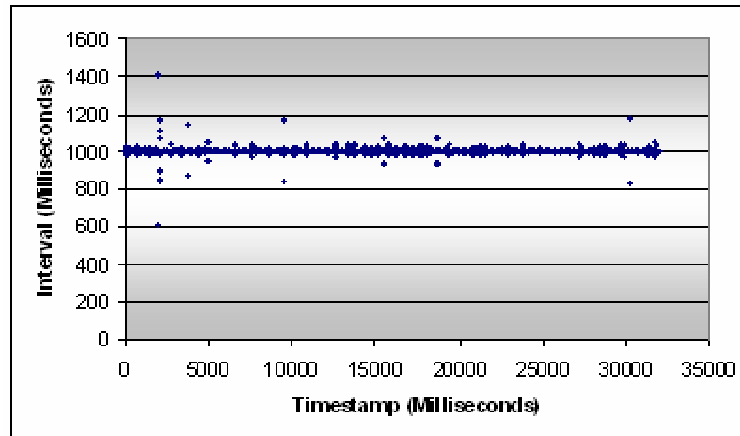

Figure 5-1: 1kHz Reasonable Load

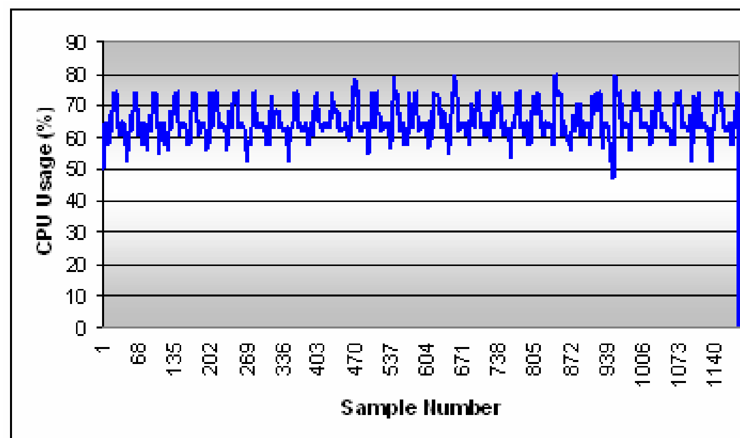

Figure 5-2: TIW timer Reasonable Load CPU usage

5.1.1.2 Overload

Figure 5-3 presents the results for the TIW timer under an overload. Again the TIW timer is configured to measure an interval of 1 ms, as was the case in the previous section. This time the load consists of a function that measures an interval of ?1000 ?s using the loop timer algorithm discussed in section 2.2.2. Therefore, the CPU will be kept occupied for the entire interval duration that it is required to measure. Referring back to the TIW timer investigation in section 4.3.2, it is expected that this will have an adverse affect on the TIW timer, since it leaves no time for the TIW timer to record the interval.

The maximum interval recorded was 1.951 ms and a minimum of 0.525ms. Therefore, under overload a single TIW timer exhibits a resolution of 1ms ? 951?s which is essentially the same as the 1 kHz multimedia timer (refer to section 2.3.2.1). This is a situation in which the task to be executed in during every interval takes the same amount of time or more time to execute as the interval duration measured by the timer.

Therefore, the TIW timer does not hold its own when required to measure time for unreasonable load. However, since the load is unreasonable, the TIW timer (or any other timer for that matter) could not be reasonably expected to measure such an interval.

Such a scenario constitutes poor design of the application using the timer. It is therefore safe to assume that the timer will be accurate under normal operational environments where a single timer is used. This is deduced from the results in section 5.1.1.1.

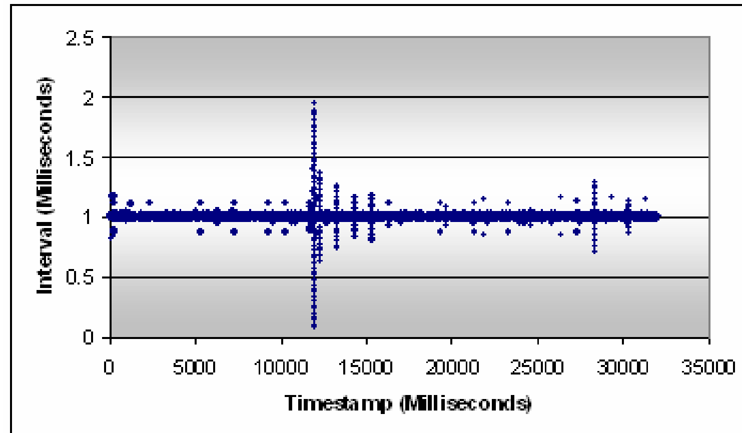

Figure 5-3: 1kHz Overload

The total processor usage was quantified. The maximum amount of CPU resources consumed at a given time is 100%, as expected. However, the average CPU resources consumed over the measured period are 90.916%. Note that these figures are an indication of the CPU usage of both the TIW timer and the load. This is shown in Figure 5-4.

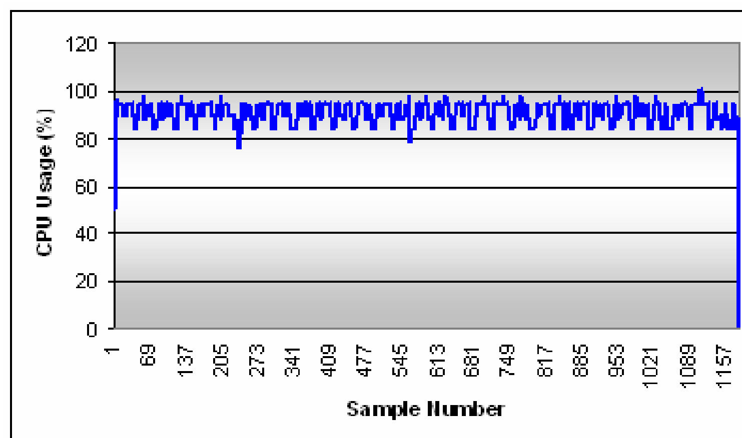

Figure 5-4: 1kHz Overload CPU Usage

5.1.1.3 Conclusion

This investigation illustrates that the TIW timer retains its accuracy with minimal adverse affect on its resource consumption when tasked to measure time for a routine that places reasonable load on the timer.

On the other hand, the timer exhibits a similar resolution to the WIN32 multimedia timer when the routine it is measures time for, is “unreasonable” in that it consumes the same amount of time or more time than the interval duration measured by the TIW timer.

5.1.2 Multiple Timers

In this experiment, the number of timers that can be executed on a single system is investigated. Real-time systems commonly need to employ more than one timer and therefore the TIW timer's performance in this regard deserves to be quantified. Since the performance of a single timer is known (section 4.3), this investigation firstly focuses on the performance of the TIW timer when of two instances are running simultaneously. This will be followed by the estimation of the maximum number of timers that may be running on a single system at the same time.

5.1.2.1 1 kHz, Two TIW timers running

For the purpose of this investigation, the timers will be referred to as Timer A and Timer B. These two timers were spawned within the same user application and tasked to provide timer events at a frequency of 1 kHz.

Figure 5-5 depicts the results from the Timer A. A maximum of 1.309 ms and minimum 0.691 ms was recorded. Therefore the timer exhibits a resolution of 1ms ± 309?s.

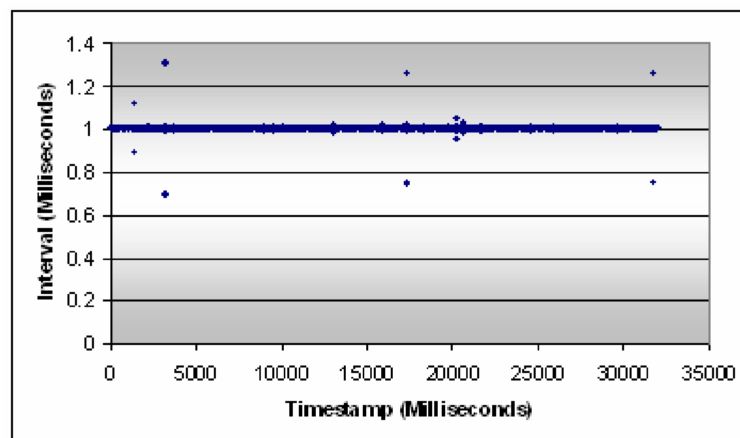

Figure 5-5: Timer A 1 kHz Test

As is clear from Figure 5-5 Timer A provides an interval accurate within 500 ?s. The figure also indicates that this deviation appeared irregularly. The rest of the time the interval remains below 1.100 ms. Therefore the TIW timer in the case of timer A exhibits the required resolution of 1ms ± 500?s or better.

The maximum amount of CPU resources consumed at any moment was 66% with an average of 5.077%. It is clear from Figure 5-6 that this is only an initial spike and that the rest of the time Timer A's performance is comparable to the single TIW timer in section 4.3.3. The CPU usage of Timer A is presented in Figure 5-6. The conclusion that applies to Timer A is that its performance is the same as if it was running on its own.

Figure 5-7 depicts the results from the Timer B. A maximum of 1.100 ms and minimum 0.9 ms was recorded. Therefore, timer B exhibits a resolution of 1ms ± 100 ?s. Timer B, like timer A, exhibits the required resolution of 1 ms ± 500 ?s or better.

Timer B consumed a maximum of 14.883% of the available CPU resources and a minimum of 0%. The average CPU usage was 4.885% as can be seen from Figure 5-8. Like timer A, timer B's performance is analoguous to the single TIW timer. Again it may be concluded that timer B performs the same, as it would have if it was running alone.

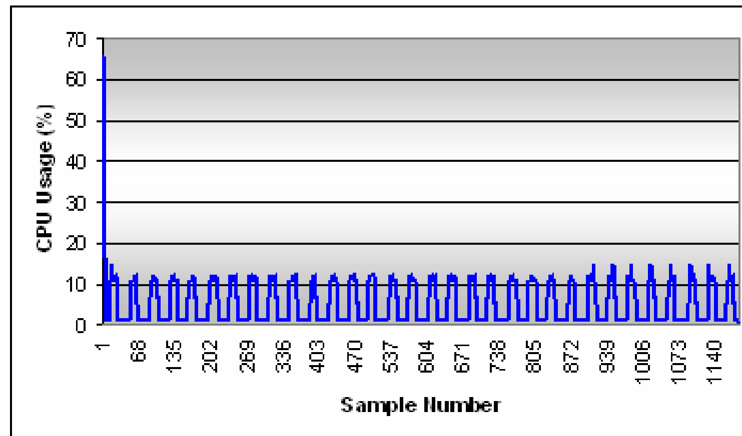

Figure 5-6: Timer A CPU Usage

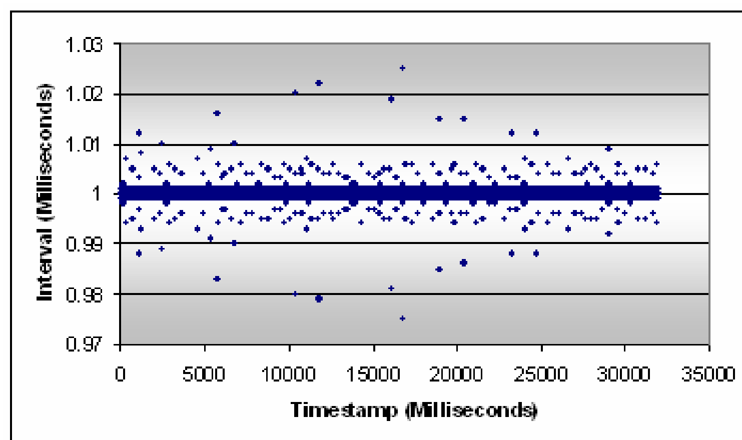

Figure 5-7: Timer B 1 kHz Test

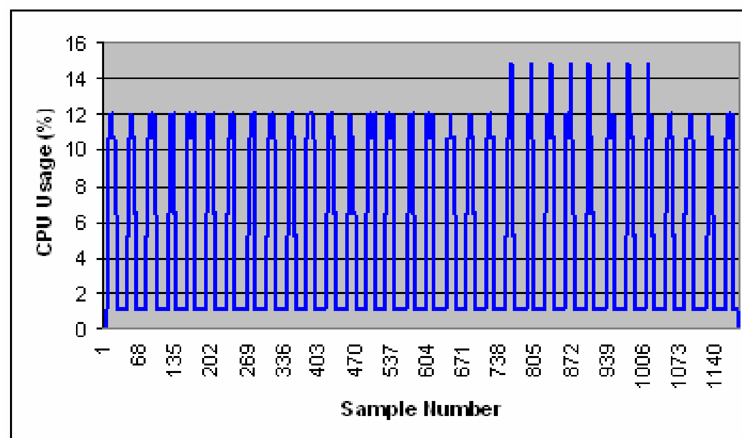

Figure 5-8: Timer B CPU usage

5.1.2.2 1 kHz six TIW timers running

From the previous section, it is known that two TIW timer's can be run together on a single machine. The following investigation endeavours to determine by how much

the number of TIW timers may be increased before the timer's performance starts to degrade significantly.

Six TIW timers were initialised and started – configured to fire timer events at a frequency of 1 kHz. Accordingly, the accuracy of all six timers was recorded. The timer that presented the best result had the following behaviour:

? The maximum interval recorded – 1.191ms

? The minimum interval recorded – 0.809ms

Therefore this particular timer's resolution is 1ms ± 191?s, thus conforming to the required resolution of 1 ms ± 500 ?s. This is shown in Figure 5-9.

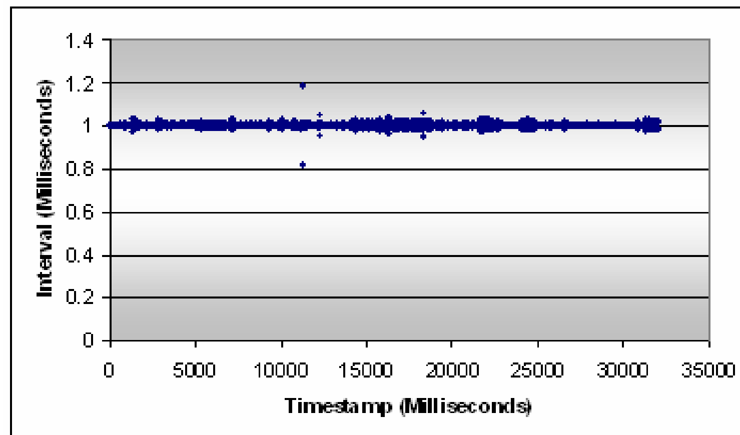

Figure 5-9: Best Timer with 6 Timers Running

The timer with the worst performance, however, exhibited the following behaviour:

? The maximum interval recorded – 1.993ms

? The minimum interval recorded – 0.007ms

With a resolution of 1 ms ± 0.993 ms (1.993ms – 1ms), the TIW timer's performance is considerably worse than that of a TIW timer that is running on its own (although still analogues to the multimedia timer). This is shown in Figure 5-9.

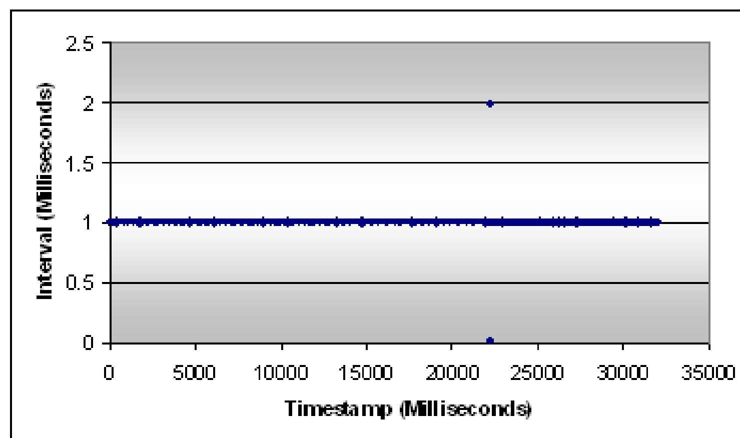

Figure 5-10: Worst Timer with 6 Timers Running

Although the "best" timer did indeed provide satisfactory performance results, it is not predictable which of the TIW timers among the six running concurrently will exhibit acceptable results, and which will not. Therefore, six TIW timers cannot be run

continuously on a single machine and be expected the generate 100% of their interval within a maximum deviation of 500 ?s.

This leads to the investigation described in the following section, where the test is repeated with five TIW timers at 1 kHz.

5.1.2.3 1 kHz Five TIW timers Running

Five TIW timers were initialised and started, with a configuration that will allow them to fire timer events at a frequency of 1 kHz, as was the case in the previous section. Again, the accuracy of the timers was calculated. Subsequently, the performance of the "best" and "worst" timer was used to quantify the results of the investigation.

The TIW timer with the best performance recorded a maximum interval of 1.034 ms and a minimum interval of 0.966 ms. Thus a resolution of 1 ms ± 34 ?s was achieved – well within the requirement of a soft real-time timer in section 2.6. This is shown in Figure 5-11.

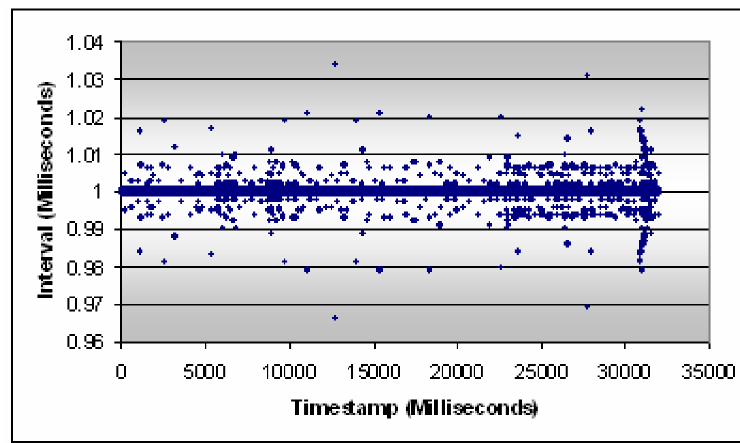

Figure 5-11: Best Timer with 5 Timers Running

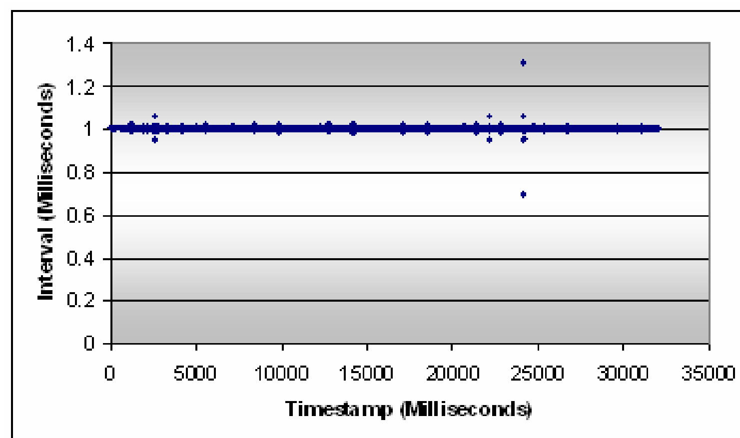

Figure 5-12: Worst Timer with 5 Timers Running

On the other hand, the timer with the worst performance exhibited the following behaviour:

- The maximum interval recorded – 1.375 ms
- The minimum interval recorded – 0.625 ms

The resolution is therefore 1 ms ± 375 ?s. As was the case with the "best" TIW timer, the resolution is within the required value of 1 ms ± 500 ?s.

This is shown in Figure 5-12. The figure indicates that the spike in interval occurred only once over a period of 30 seconds – the rest of the intervals were below 1.100ms. Therefore, 5 timers can be run concurrently on a single system and will still generate its intervals within a maximum deviation of 500 ?s.

5.1.2.4 Conclusion

When multiple TIW timers on a single machine are configured to generate timer events at a frequency of 1 kHz, the following is concluded:

- From section 5.1.2.1 it is obvious that two TIW timers can run concurrently on a single machine without notable degradation in performance.
- It is apparent from the performance of the worst timer in section 5.1.2.2 that six TIW timers may not be run reliably in a concurrent fashion at a frequency of 1 kHz. (Although the performance is analogues to that of the multimedia timer (refer to section 2.3.2.1))
- Section 5.1.2.3 shows that it is possible to run 5 TIW timers concurrently at a frequency of 1 kHz and thus the maximum number of TIW timers that may be running concurrently with reliability equates to 5 when a frequency of 1 kHz is used.

The test was repeated for the 50 Hz timer. However, the performance of TIW timer did not degrade significantly, even when a significant number of timers (20 in total) were executed. It was not possible to determine a maximum number of timers for this interval. However, since the 1 kHz frequency may be seen as the most "difficult" to achieve for the TIW timer (without considering section 5.2) and it is possible to run 5 of them concurrently at 1 kHz, the TIW timer's performance in this regard will be construed to be at most 5 timers at 1 kHz. This implies however that more timers may be run concurrently at lower frequencies.

5.1.3 Other Multimedia Applications running with the TIW timer

This section investigates the performance of the TIW timer whilst another multimedia application is running. When the multimedia subsystem is activated, the entire operating system is put into the high performance state [MSDN 2005]. Should another application be using the default WIN32 timers and another application initialises the multimedia subsystem, the first application will benefit from it.

As the name implies, the multimedia subsystem is used by multimedia applications such as the Windows Media Player. The purpose of this section is therefore to determine the TIW timer's performance while another multimedia application, such as the Windows Media Player, is running.

5.1.3.1 1 kHz TIW timer with other Multimedia Application

Figure 5-13 illustrates the performance of the timer with a frequency of 1 kHz required. At the time the TIW timer was started, an instance of the Windows Media Player is already running and is in the midst of video playback. Accordingly the multimedia subsystem is already activated by the TIW timer starts.

The maximum interval recorded was 1.866 ms and the minimum 0.133 ms over a period of 60 seconds (although only the first 30 seconds are depicted in the figure). The resulting resolution of the TIW timer in this case is 1 ms ? 866 ?s and therefore outside of the required resolution for a soft real-time timer defined in section 2.6. (It is

interesting to note again that this performance is on par with the multimedia timer (refer to section 2.3.2.1).

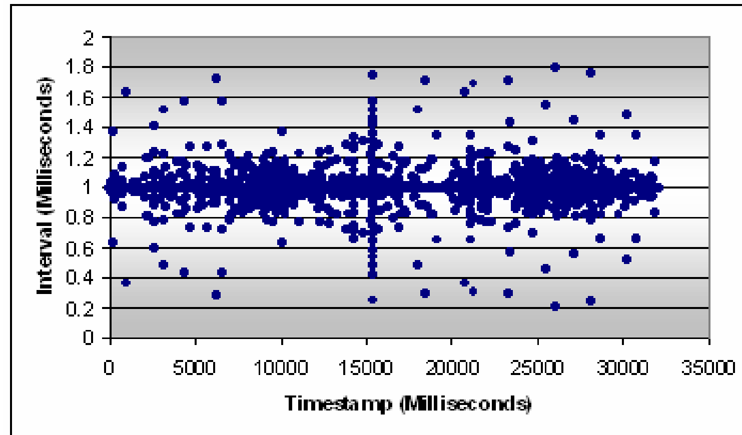

Figure 5-13: 1 kHz other Multimedia Application

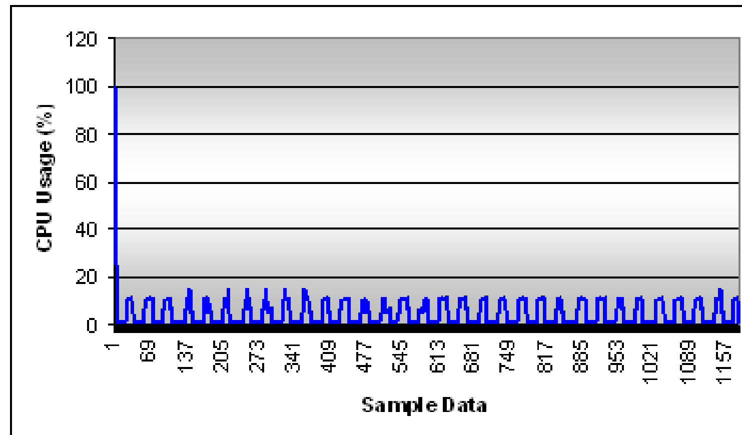

Figure 5-14: 1 kHz TIW timer with other Multimedia application CPU Usage

Furthermore, the CPU consumption is not completely consistent with instances where the TIW timer is running without interference from another application that uses the multimedia subsystem. The maximum amount of CPU resources consumed at a given time was 100%, an initial spike. However the average was 4.835%.

5.1.3.2 50 Hz TIW timer with other Multimedia Application

Figure 5-15 illustrates the performance of the timer with a frequency of 50 Hz required, again with the Windows Media Player presented the contents of a video recording in the background. Again the player was started up first with the TIW timer in tow.

The maximum interval recorded on this occasion was 20.636 ms and the minimum 19.364 ms. Therefore the timer exhibits a resolution of 20 ms ? 636 ?s, very similar to that of the multimedia timer with the same frequency (refer to section 2.3.2.2).

The maximum amount of CPU resources consumed at a given time was 16.5%. The average was 3.001%, a result consistent with that of a single TIW timer running without the added load of another multimedia application on the same machine.

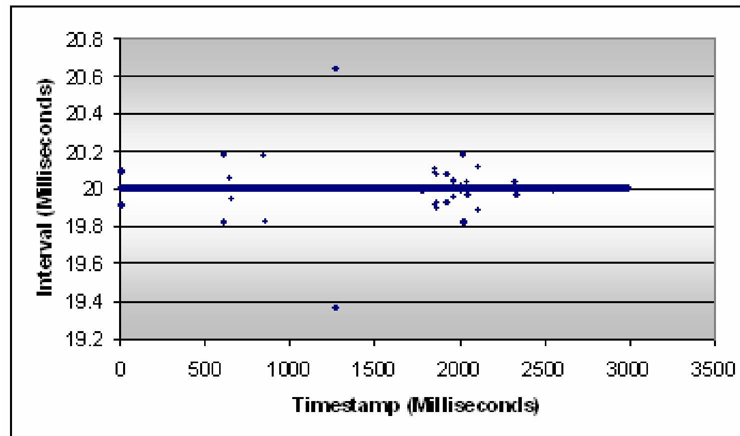

Figure 5-15: 50 Hz other Multimedia Application

5.1.3.3 1 Hz TIW timer with other Multimedia Application

Figure 5-16 illustrates the performance of the TIW timer configured to generate timer events every second – a rate of 1 Hz.

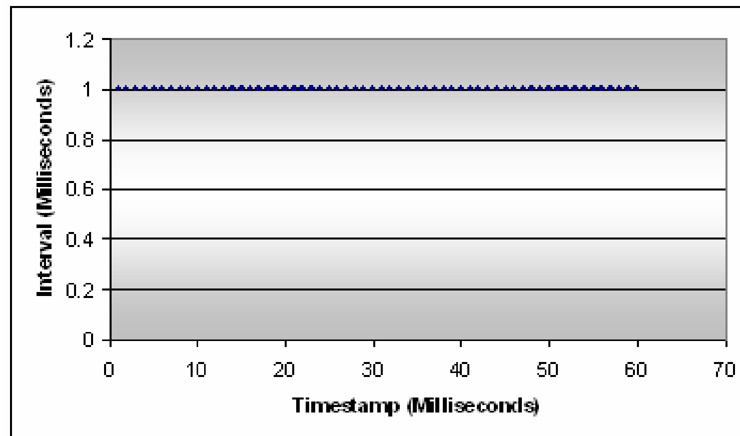

Figure 5-16: 1 Hz other Multimedia Application

As before, the Windows Media Player application is started before the TIW timer. The maximum and the minimum intervals recorded were both 1.0 second. Therefore the timer exhibits a resolution of 1 second ? 0 ?s. Comparing this resolution to that recorded for the single TIW timer with an alternative configuration in section 4.3.4.2, it is clear that the accuracy of the TIW timer is retained in this instance. The resolution on that occasion was 1 second ? 1 ?s.

The maximum amount of CPU resources consumed at a given time was 84.333%, an initial spike like that of the TIW timer with an alternative configuration in section 4.3.4.2. The average was 2.452% and when this is compared to the results in section 4.3.4.2, it can be surmised that the performance the TIW timer is not adversely affected by another multimedia application in this case.

5.1.3.4 Conclusion

Running a multimedia application concurrently with the TIW timer has notable effects on the performance of the timer – especially at higher frequencies. The TIW timer

displays the same performance as the multimedia timer at certain frequencies (in this case 50 Hz and 1 kHz, refer to sections 5.1.3.1 and 5.1.3.2).

However, at lower frequencies such 1 Hz (in section 5.1.3.3), the TIW timer retains its performance. Therefore, it is possible to run the TIW timer reliably alongside another multimedia application at lower frequencies – which may not be adequate for all soft real-time implementations although that TIW timer performs on par with the multimedia timer in terms of accuaracy.

5.2 Period Adjustment

In section 4.3, it was shown that the TIW timer is able to generate an interval of 1000 ?s (1 ms) utilising 2 threads. This section investigates whether it is possible to measure smaller intervals than 1 ms by varying the number of execution threads that the TIW timer is using.

The investigation shows that it is indeed possible to do so, given that the choice of thread numbers is made correctly. This section will only present the results of the TIW timer at higher frequencies than 1 kHz, showing that it is possible with the number of threads chosen wisely. Section 5.3, however, serves as an explanation of how the correct number of threads is chosen.

5.2.1 4 Threads, 500 ?s Interval (2 kHz)

In this first experiment, the objective was to double the highest frequency of the TIW timer; namely 1 kHz. Therefore the TIW timer is required to generate timer events at 2 kHz. To achieve this, the number of threads was increased to four, and the required interval set to 500 ?s. The complete configuration follows:

```
Number of timestamps          – 120000
Interval Duration             – 500 ?s
Requested Sleep Interval      – 1 ms
Number of threads to use      – 4
The "mod"                     – 1
```

The results are shown in Figure 5-17.

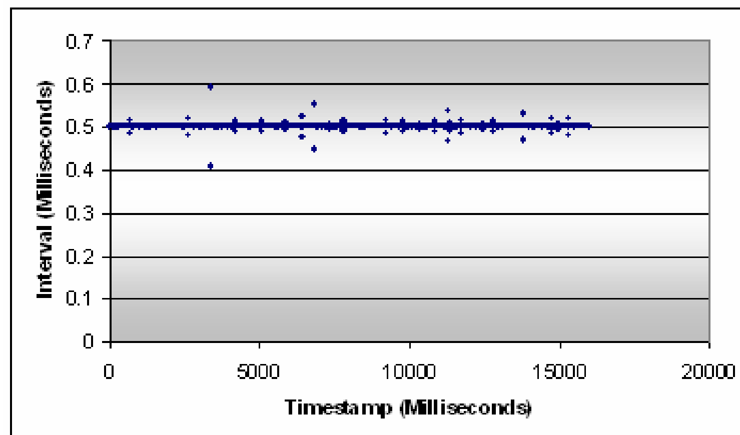

Figure 5-17: 2 kHz TIW timer

A maximum interval of 591 ?s was recorded and a minimum of 409 ?s over a period of 60 seconds with the first 16 seconds shown in the figure. Therefore, the timer exhibits a resolution of 500 ?s ? 91 ?s in this case. The worst resolution recorded for

the timer with this frequency is 500 ?s ? 250 ?s – 200 ?s being within the 500 ?s maximum deviation specified in section 2.6.

However, 500 ?s doesn't seem like an acceptable deviation for a timer with an interval size of 500 ?s. Since a resolution of 1 ms ± 500 ?s is acceptable for a 1 kHz with the maximum deviation fixed at half the interval size, it is reasonable to expect that the TIW timer at 2 kHz exhibits a maximum deviation of half its interval size, in this case 250 ?s. Figure 5-18 depicts the worst case recorded for the 2 kHz timer and shows that the maximum deviation is less than or equal to 250 ?s. The maximum interval recorded was 682 ?s and as the figure shows, it was an isolated incident.

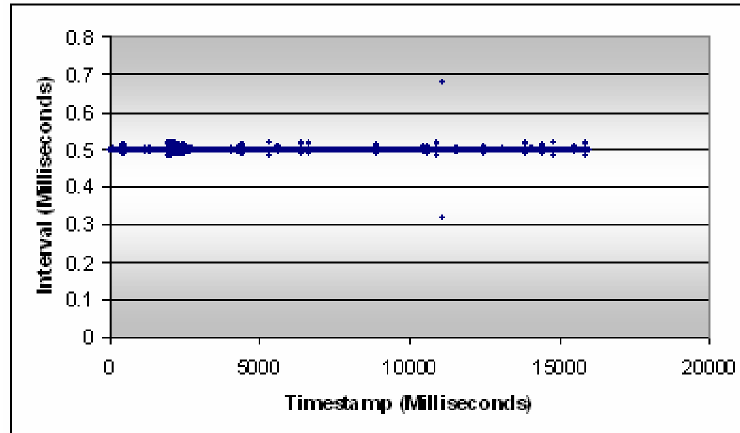

Figure 5-18: 2 kHz TIW timer – Worst Case

Since the 2 kHz TIW timer utilises more threads than the 1 kHz TIW timer, it is to be expected that there will be an increase in resource consumption. The maximum slice of CPU resources consumed by the timer shows an initial spike of 66% where after the usage remains below 30%. However, the average usage was 8.885% leaving whatever application requiring 500 ?s interval measurements with on average ? 91.115% of the CPU to its disposal.

This is depicted in Figure 5-19.

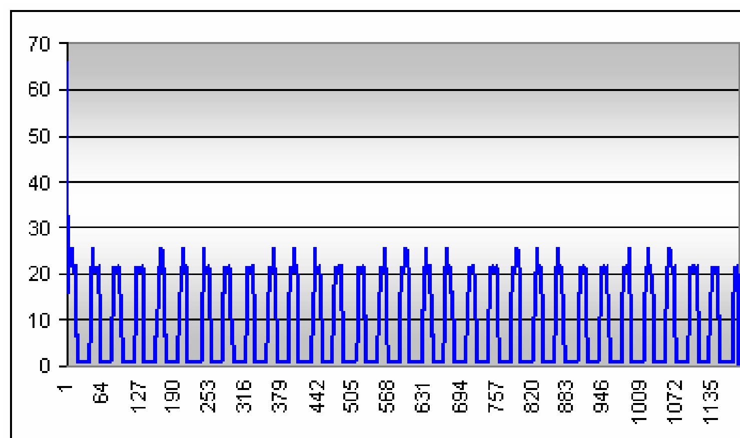

Figure 5-19: TIW timer 2 kHz CPU usage

5.2.2 8 Threads, 250 ?s Interval (4 kHz)

Since the TIW timer successfully doubled the 1 kHz frequency to 2 kHz, the next obvious step is to determine whether the frequency may be doubled again to 4 kHz. Therefore the number of threads was increased to 8, and the required interval set to 250 ?s. Figure 5-20 illustrates the results recorded with the TIW timer configured as follows:

```
Number of timestamps         - 240000
Interval Duration            - 250 ?s
Requested Sleep Interval     - 1 ms
Number of threads to use     - 8
The "mod"                    - 1
```

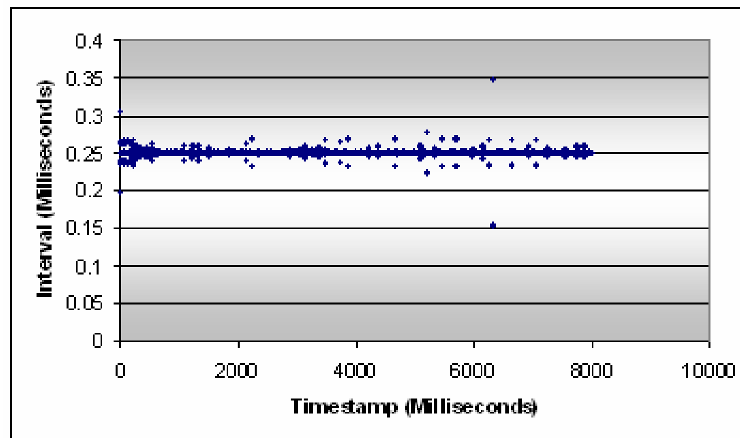

Figure 5-20: 4 kHz TIW timer

A maximum interval of 347 ?s was recorded, however, over a period of 60 seconds this only occurred once; in fact, Figure 5-20 illustrates the period of 8 seconds in which the maximum interval was recorded. The interval stays below 350 ?s save for the solitary maximum and spike. The minimum recorded interval was 153 ?s. The resolution is therefore 250?s ? 100?s. Again the maximum deviation is less than half the length of the interval and obviously within 500 ?s of the target interval duration.

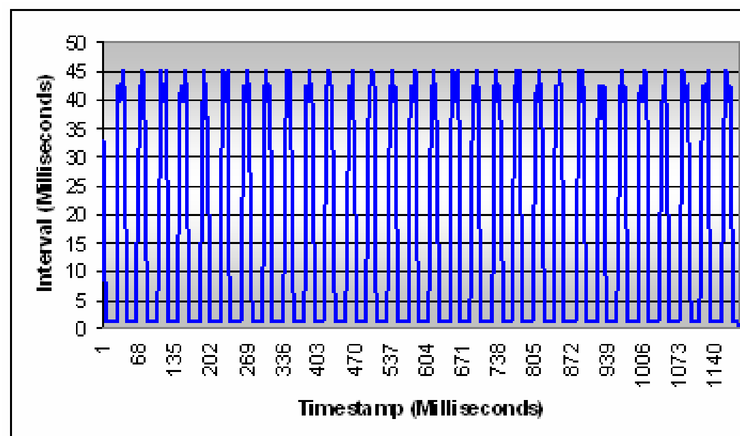

Figure 5-21: TIW timer 4 kHz CPU usage

Janno Grobler
jannogrobler@gmail.com
M.Sc Computer Science
University of Pretoria

Again the number of threads were doubled from the previous experiment in section 5.2.1 and accordingly and increase in resource consumption is to be expected. As Figure 5-21 illustrates, the CPU usage varies between 0% and 45.167% with an average of 16.594%.

5.2.3 16 Threads, 125 ?s Interval (8 kHz)

In this experiment, the goal was yet again to attempt to double the previous frequency (in this case from 4 kHz to 8 kHz). To achieve this, the number of threads was increased to 16, and the required interval set to 125 ?s. The full configuration of the timer is as follows:

```
Number of timestamps        - 480000
Interval Duration           - 125 ?s
Requested Sleep Interval    - 1 ms
Number of threads to use    - 16
The "mod"                   - 1
```

The results are shown in Figure 5-22 illustrates the results from this experiment over a period of 4 seconds. A maximum interval of 347 ?s was recorded, however, as shown in Figure 5-22, deviations greater than 75 ?s occur sporadically. The minimum recorded was 4?s and therefore the resolution in this case is 125?s ? 222?s. Therefore, the TIW timer does not measure the interval reliably for all the intervals generated and the maximum deviation is greater than the interval size. However, Figure 5-22 shows that the interval measured is always within 500 ?s of the desired interval duration. Since 75 ?s is half the length of the required interval of 125 ?s, it is concluded that the TIW timer's acceptable performance at 1 kHz and 2 kHz cannot be extrapolated to a frequency of 8 kHz.

With the increase of the number of threads to 16, the CPU usage is once again expected to increase and that is exactly what the timer demonstrated. The maximum % CPU resources consumed were 84.333% and the average CPU usage was 32.063% (refer to Figure 5-23).

Therefore, should an application require the TIW timer to measure 125 ?s intervals, the estimated interval will be accurate within 222 ?s and the application itself will have on average ? 77.937% of the CPU at its disposal.

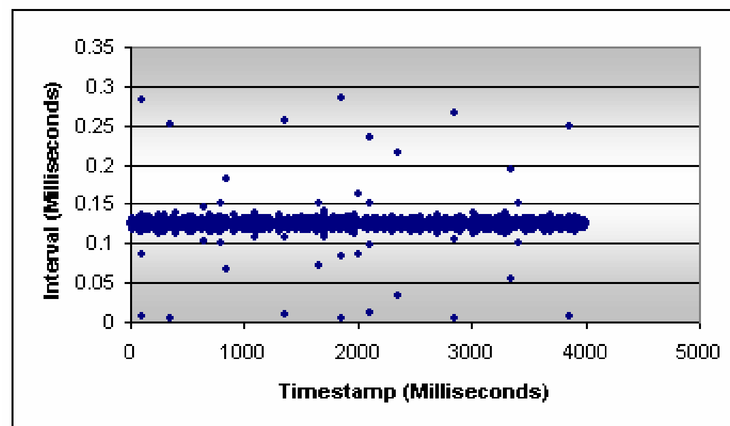

Figure 5-22: 8 kHz TIW timer

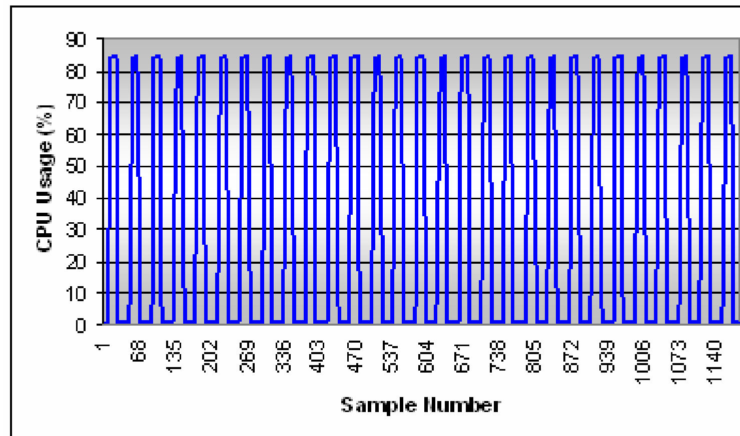

Figure 5-23: TIW timer 8 kHz CPU usage

5.2.4 5 Threads, 400?s Interval (2.5 kHz)

Up to this point, the intervals recorded were all divisors of 1000 ?s:

? 1000 ?s/2 = 500?s

? 500 ?s/2 = 250?s

? 250 ?s/ 2 = 125?s

Therefore in this experiment, the idea was to generate an interval less than 1000 ?s that is not a factor of 1000 ?s. The interval chosen was 400 ?s.

The results for this interval are presented in Figure 5-24. The maximum interval recorded was 600 ?s and a minimum of 200 ?s. Again the maximum deviation is half of the length of the required interval culminating in a resolution of 400 ?s ? 200 ?s. The CPU usage is presented in Figure 5-25. The average usage was 10.766%.

The TIW timer was tested at his frequency several times and the maximum deviation of up to 300 ?s was recorded. However, it was always an isolated incident with the rest of the timestamps recorded within 200 ?s of the required interval. However, as was the case with the 8 kHz TIW timer in section 5.2.3, the TIW timer is able to generate the required interval within 500 ?s, but fails produce intervals that do not exceed the required interval by more than half.

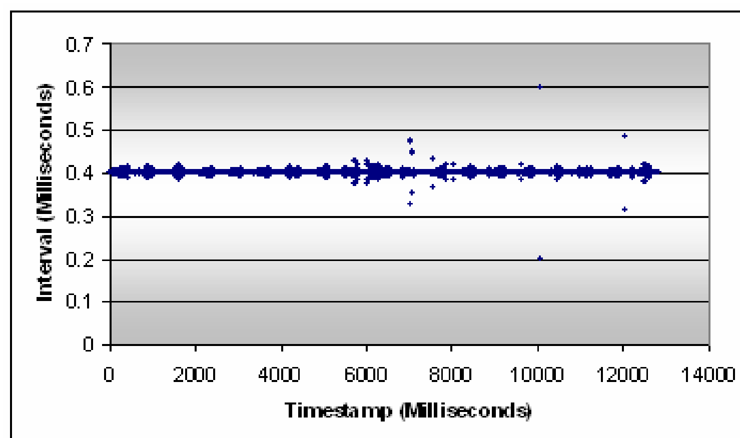

Figure 5-24: 2.5kHz TIW timer

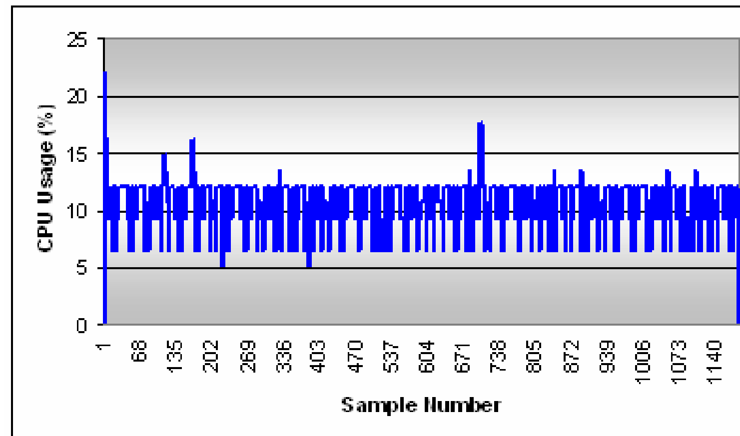

Figure 5-25: TIW timer 2.5kHz CPU usage

5.2.5 Conclusion

The definition of the soft real-time timer in section 2.6 specified a required resolution of 1 ms ? 500 ?s for the soft real-time timer. Therefore, the maximum deviation equates to half the duration of the required interval. The TIW timer achieved this for the 1 kHz frequency, as seen in section 4.3.3.1.

In section 5.2.1, the resolution exhibited was 500 ?s ? 200 ?s for a 2 kHz timer. Note that, not only is the maximum deviation less than 500?s; it is also less than half of the required interval, namely less than half of 500 ?s. In this sense, the 2 kHz TIW timer's performance is the same, relatively speaking, as the 1 kHz timer. This unfortunately does not apply to the 2.5 kHz timer in discussed in section 5.2.4. It was seen there that this timer had a resolution of 400 ?s ? 200 ?s in general, although an occasional interval may exceed 400 ?s by as much as 300 ?s.

However, it does apply to the 4 kHz timer discussed in section 5.2.2 where the resolution was 250 ?s ? 100 ?s. Although the maximum deviation is less than half of the interval size, it would compel the timer to generate intervals of ±150 ?s to compensate when the maximum deviation occurs. The resolution thus complies with the original timer specifications that were enunciated in section 2.6, but would probably be unacceptable for an application that really needed a 4 kHz timer.

The same is true of the 8 kHz timer in section 5.2.3. It has a resolution of 125 ?s ? 222 ?s – in other words the deviation is more than half the interval size. The maximum deviation is in fact greater than the interval size, which is probably not desirable for a real application that needed an 8 kHz timer, although these deviations are irregular and sporadic.

Taking these results into consideration it is clear that the TIW timer is capable of generating intervals smaller than 1000 ?s with a maximum deviation that was considered acceptable for the 1 kHz timer, namely 500 ?s. In fact, for timers with a frequency lower than 2.5 kHz (i.e. a required interval higher than 400 ?s), the maximum deviation was less than or equal to the duration of half the required interval, although the 2.5 kHz timer only exhibited occasional deviations that exceeded half the interval size.

Therefore, for timers with frequency of 2.5 kHz and higher, the maximum deviation will be less than 500 ?s, but it may also be greater than half the interval duration.

5.3 Number of Threads

The choice of number of threads is crucial to the TIW timer's operation, especially when generating intervals smaller that 1000 ?s (refer to section 5.2), as will be shown in this section. Refer back to the TIW timer timing diagram in Figure 4-10 for the 1 kHz timer using two threads. In the startup phase of the timer which precedes the first event that is fired, three intervals may be identified. The first two intervals occur successively, each enduring for 1000 ?s, and each being generated via the loop timer method. The third interval endures for another ? 1000 ?s, being the period during which both threads are "sleeping".

Now consider Figure 5-26 that represents the timing of the TIW Timer with a required interval of 500 ?s. Four threads (A, B, C and D) are initialized, each successively measuring out 500 ?s intervals, using the loop timer mechanism. After thread D has measured its interval, a fifth interval ensues during which all threads are sleeping. At the start of this fifth interval, thread A will have roughly 500 ?s of sleeping time left before it fires the first timer event.

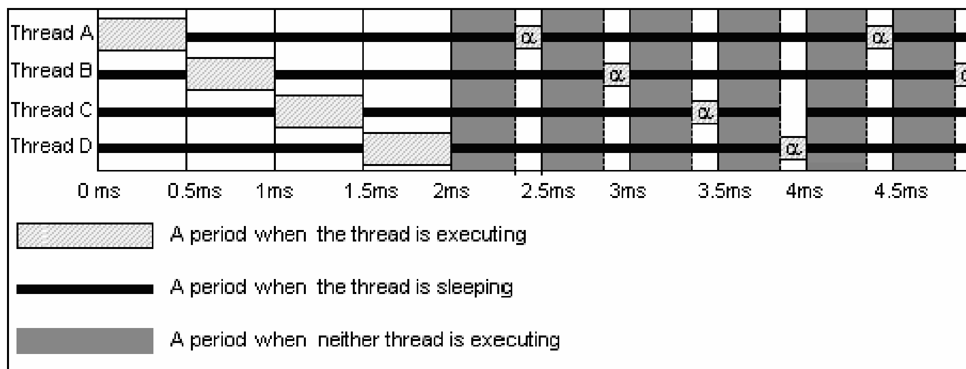

Figure 5-26: 2 kHz TIW timer Timing Diagram

Figure 5-27 represents the timing diagram for the 2.5 kHz timer and shows that the five threads (A through E) will each measure an initial 400 ?s interval using the loop timer mechanism. When thread E is done with this initial interval measurement, a sixth interval ensues during which all threads are sleeping. At the start of this sixth interval, thread A has roughly 400 ?s of sleeping time left.

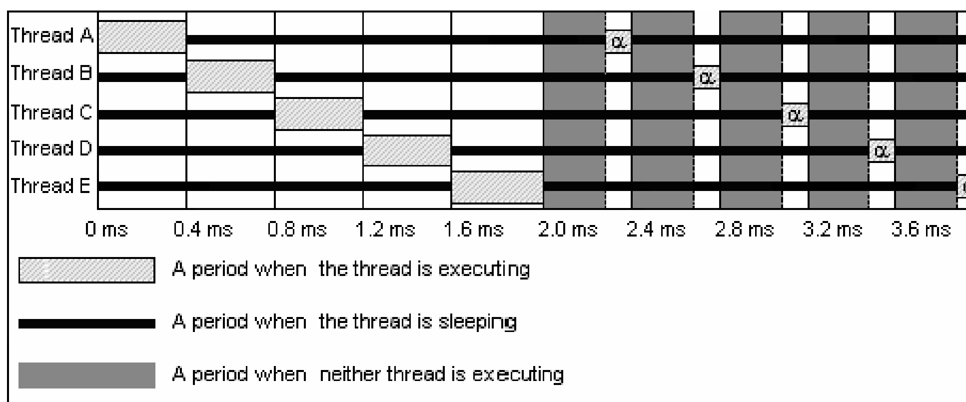

Figure 5-27: 2.5 kHz TIW timer Timing Diagram

The same is true of Figure 5-28, which represents the timing diagram for the 4 kHz timer. The eight threads (A through H) will start by measuring an interval of 250 ?s

each. When thread H has measured its interval, a ninth interval ensues during which all threads are sleeping. At the start of this ninth interval, thread A will have roughly 250 ?s left to sleep.

Therefore, in all cases, the first interval during which all threads are sleeping is approximately the same length as the timer's interval duration.

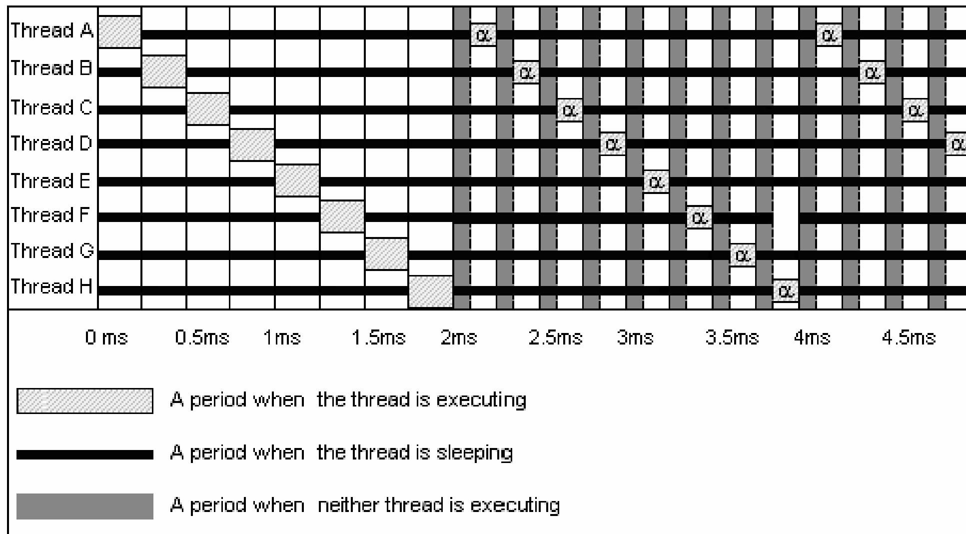

Figure 5-28: 4kHz TIW timer Timing Diagram

Now, consider all these timing diagrams (Figure 4-10, Figure 5-26, Figure 5-27 and Figure 5-28). In each of these diagrams, the time elapsed since the end of timer A's interval calculation and the firing of the first timer event (in other words the time elapsed since the end of timer A's interval calculation and the end of the first ? period), is 2 ms[2]. Within the 2 ms there is one period with a duration equal to that of the interval size during which all of the threads are asleep.

Let *N* denote the number of threads for a TIW timer that is to measure an interval of t milliseconds. In all the figures, *N* threads each measure one t-millisecond interval within this 2 ms period. Therefore, including the interval during which all threads are asleep, *N* intervals of t-milliseconds have been measured with an interval of t-milliseconds to come before thread A has completed its 2 ms suspension. In other words, the relationship: 2 = N X t holds in all theses cases, or N = 2/t.

Now consider Figure 5-29, which depicts the situation where the TIW timer is configured to use five threads to measure a 500 ?s interval (in other words, a number that is not equal to 2 ms divided by the interval size). The configuration of the TIW timer is accordingly as follows:

```
Number of timestamps        - 120000
Interval Duration           - 500 ?s
Requested Sleep Interval    - 1 ms
Number of threads to use    - 5
The "mod"                   - 1
```

It is clear that the interval during which all threads are sleeping is replaced by the fifth thread (thread E) running a loop timer during that time to measure an interval of 500

[2] For the purposes of this discussion, assume that there is no overshoot, i.e. that a = 0

?s. The resulting TIW timer will constantly consume 100% of the available CPU resources.

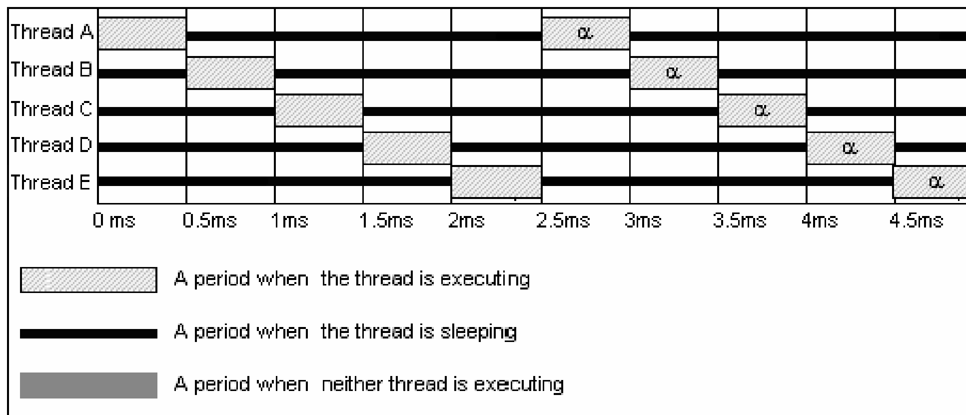

Figure 5-29: 2kHz TIW timer Timing Diagram – Wrong number of threads

Figure 5-30 shows the CPU usage for a TIW timer set up to measure a 500 ?s utilizing 5 threads. The average usage was 83.22%.

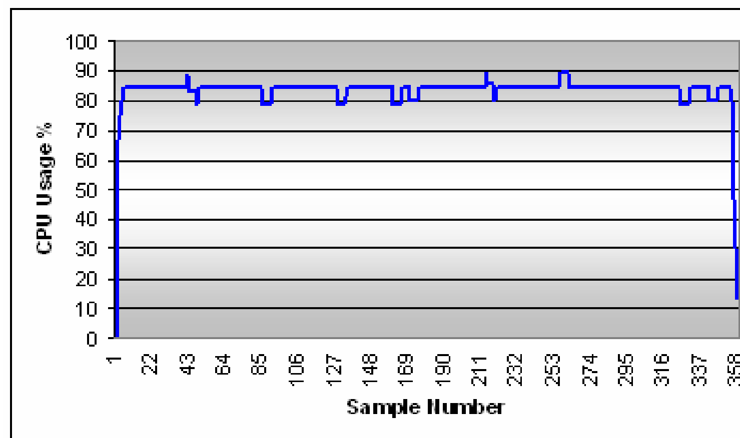

Figure 5-30: 2 kHz TIW timer, 5 threads CPU usage

However, the sleep instruction may suspend a thread for more than 2 ms, which does occur as shown in section 4.2.2.2.1. If for example, thread A sleeps for 2.100 ms after its initial interval in the figure, a period of 200 ?s exist after thread E has entered the sleep cycle for the first time where no thread will be running. However, the sleep instruction does not overshoot every time and the intervals where no thread is running is so small that it fails to induce a sufficient wait period that would prevent the TIW timer from using an unacceptable amount of CPU resources.

Figure 5-31 shows the TIW timer configured to generate timer intervals at 2 kHz using three threads (again a number that is not equal to 2 ms divided by the interval size). As is clear from the figure, when the third thread (thread C) finishes its interval calculation, a period of ± 1000 ?s ensues during which all threads sleep before thread A goes into a running state and measures out the first a interval. In fact, this same “hiccup” occurs so that every third timer event is cumulatively delayed by 0.5ms.

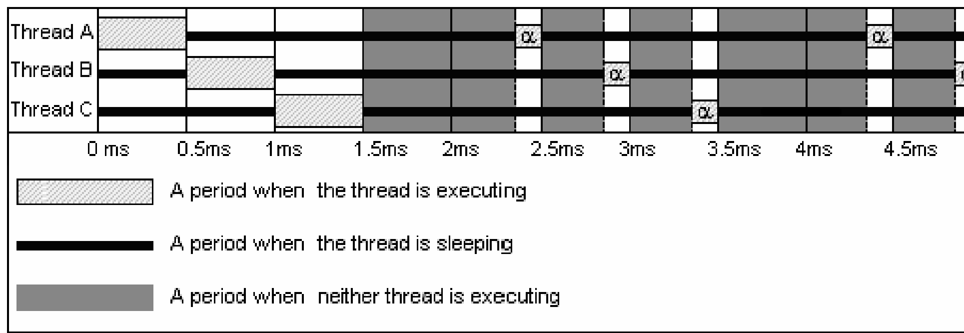

Figure 5-31: 2kHz TIW timer Timing Diagram – Wrong number of threads

Therefore, the number of threads is should be equal to the 2 ms divided by the interval size or either the situation depicted in Figure 5-29 and Figure 5-31 is the result. Since it is impossible to have a fraction of a thread, the intervals that can be generated are dependent on factors of 2000 ?s. For example, an interval of 300 ?s or 800 ?s can not be generated reliably since 2000 ?s is not divisible by these values. It is impossible to have 2.5 threads (2000/800 = 2.5). However, an interval of 333.33 ?s (3 kHz TIW timer) is possible since the result of the division of 2000 ?s by this number is 6 – a number of threads that the TIW timer is able use.

This is depicted in Figure 5-32. The maximum interval recorded was 365 ?s – a maximum deviation of 32 ?s in this case.

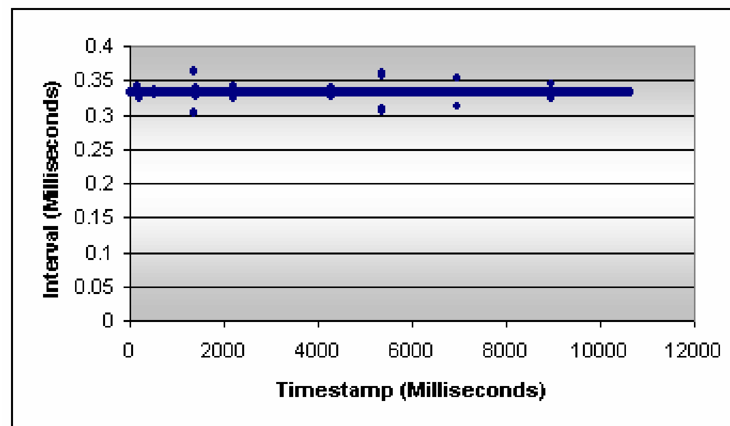

Figure 5-32: 3 kHz TIW timer

In the present study, the TIW timer with the largest number of threads had 16. To test larger factors is beyond the scope of the present study. The evidence gathered thus far suggests that intervals of, for example, 40 ?s could be generated using 50 threads, but that the resolution and CPU utilization would degenerate.

Note that this is only applicable to intervals of 1000 ?s or less. As was shown in section 4.3.4, the alternative TIW timer is able to generate reliable interval larger than 1000 ?s with the use of one thread.

5.4 A real world application

The inclusion of this sections stems from experienced gained by the author in the defense industry – specifically involvement in the development of a ground station for fighter aircraft. The basic idea of this system is that vast amounts of data are imported for analysis from data recorders on the aircraft. During this import process,

the data is converted into a format that eases the analysis process. These recorders record the data relative to time, and therefore every piece of data has a timestamp attached to it.

Therefore the data imported from the aircraft has timestamps connected to each parameter encapsulated in the data. However, the data is imported from two different sources on the aircraft. The first source is an embedded platform that uses a hard real-time timer to calculate the timestamps for the data. The second source is a commercial computer with the Windows XP operating system installed on it. The data is assigned timestamps on this second machine calculated using the system timer (see section 2.3.1) provided by the WIN32 API. When the timestamps were compared, the time measured from the WIN32 machine was greater than the corresponding timestamp from the hard real-time source (after ± one hour of data) by as much as 9 minutes.

This caused severe problem since the data from these two sources had to be matched according to timestamp. This led to the question of whether the TIW timer would still generate a reliable interval after an hour. Accordingly an application was developed that configured a TIW timer to fire timer events at a frequency of 1 kHz. An example of such a setup is given in section 4.3.3.1 with the exception that the number of timestamps was set to 3600000 (the number of one millisecond intervals in one hour). Therefore, the final timer event should fire after ± 3600000 ms have passed.

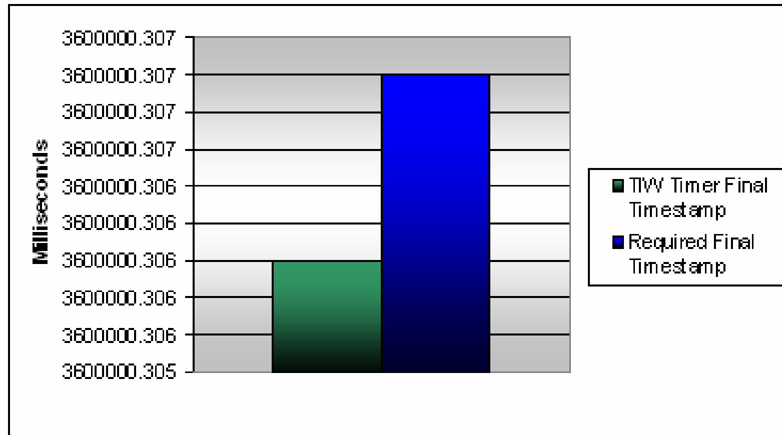

Figure 5-33: TIW Timer 1 hour: Final Timestamp

Figure 5-33 shows two bars representing timestamps in milliseconds relative to the moment that the timer was started. The first bar represents the time elapsed at the moment the TIW timer fires its final event after on 1 hour. The second bar represents the amount of time that should have elapsed. As is clear from the figure, the TIW fired its final event 1 ?s short of the time that it should have – in other words, within 500 ?s retaining the target deviation specified in section 2.6.

Another interesting note is that the maximum interval over this period was 1.111 ms. The 20 second period in which this maximum interval occurred is depicted in Figure 5-34. The yellow line represents what the interval size should be – 1 ms. It is clear that the interval durations of the TIW timer remain within close vicinity of this required interval length as the maximum deviation was 111 ?s.

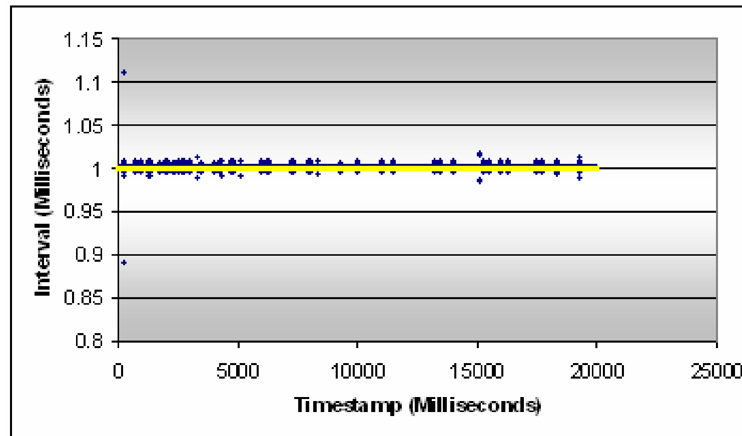

Figure 5-34: TIW Timer 1 kHz: One Hour

Since the TIW timer fired its final event 1 ?s before it was supposed to after running for an hour and this timer event was used to measure a duration of one hour, the measurement would be inaccurate by a margin of 1 ?s. As stated, this deviation is within 500 ?s and therefore very accurate. A further point of interest is that the TIW timer maintains accurate one millisecond intervals over the period of an hour.

Therefore this particular experiment shows that the TIW timer would be able to solve the problem in the real world situation described in the beginning of this section. If the system timer is replaced with the TIW timer to measure the timestamps, the discrepancy of 9 minutes could be reduced to 1 ?s.

Chapter 6

Conclusion

The goal of this chapter is to serve as conclusion to the research presented in this text. The chapter first focuses on how future trends in technology are likely to influence the TIW timer. It also presents a summary and a conclusion to this text.

Chapter 1 served as introduction and some general background to the subject matter discussed in this text. The definition of a timer was formalised along with the explanation of timer resolution and real-time. These concepts are important and serve to provide a better understanding of the problem statement that was given in section 2.6 in which certain requirements are placed on real-time timers, whether in a soft or hard real-time environment.

After providing some background on existing timers, chapter 2 shows that neither the Win32 nor standard POSIX timers are able to provide constant intervals that comply with a maximum deviation requirement of 500 ?s or less and that this failure may be attributed to the rate at which the operating systems service their timers. Another example from the defence industry illustrates this deficiency, specifically under the WIN32 operating system in section 5.4.

However, a workaround was found. Chapter 3 is a brief introduction to the design and implementation decisions that were used in implementing the workaround.

The workaround is in the form of the thread induced waitable timer (TIW timer) discussed in section 4.3. Within this section, the design and results for this timer were presented. The final solution went through a couple of rejected solutions before being formalised into the TIW timer. These solutions were presented in section 4.2.

The TIW timer is capable of providing a constant interval with a maximum deviation recorded on a 1 ms interval of 500 ?s while consuming on average less than 5% of the available processing power (refer to section 4.3). 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 2.6.

The TIW timer was shown to outperform the current WIN32 and standard POSIX timers. This was the subject of section 4.4. In that section, the TIW timer was also compared to an external hard real-time timer – the benchmark for the accuracy of the TIW timer. The comparison yielded satisfactory results for the TIW timer.

Further studies on the TIW timer were also presented, and these are summarised as follows:

? The performance of the TIW timer on the Win32 platform and under a UNIX platform was compared in section 4.4.2.2. Since the TIW timer's performance under the WIN32 operating system is superior to that of the standard POSIX timers, the logical next step was to implement the TIW timer under the Linux operating system. However, it was found that due to architectural and operational differences, the TIW timer is not a viable solution under the Linux environment.

? The effect of load on the TIW timer was investigated in section 5.1. The TIW timer was shown to retain its accuracy when measuring time for routines that placed a reasonable amount of load on the timer.

  The research done in section 5.1 specifically focused on the performance of the TIW timer when placed under load. 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 processor usage.

Untimely scheduling of the concurrent threads of the TIW timer may hamper the real-time accuracy of the solution [Chu 1997]. However, 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 such improvements in the form of hyper-threading technology on their Pentium 4 processor. This 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. 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 occupy 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 will have to continue supporting the execution of multiple concurrent 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 1 ms 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 high-resolution hardware counters, for greater accuracy.

- The number of TIW timer instances that can be run reliably together on a single computer was discussed in section 5.1.2. The maximum number of timers that may be run reliably together on a single machine at a frequency of 1 kHz was found to be five in total. However, it may be possible to execute more timers concurrently at lower frequencies.
- Determining the effects of other multimedia applications that are running on the same machine as the TIW timer was the focus of section 5.1.3. It was shown that the TIW timer is vulnerable to interference from other multimedia applications. It only retains is accuracy at lower frequencies such as 1 Hz. However, when the TIW timer is executed concurrently with a multimedia application, its performance is still comparable to that of the multimedia timer.
- Section 5.2 quantified the performance of the TIW timer at frequencies greater than 1 kHz. Since the TIW timer was successful in generating a relatively small interval of 1000 ?s, it was decided to investigate whether it would be possible to measure even smaller intervals. It was determined that the TIW timer is indeed able to measure intervals smaller than 1000 ?s, with maximum deviations within 500 ?s and with reasonable processor consumption (although the processor consumption increases as the frequency rises). This is accomplished through configuration of the TIW timer to use more than two threads.

  However, only for frequencies of 2 kHz and lower, does the maximum deviation of the timer not exceed half the duration of the required interval reliably. For frequency greater than 2 kHz, the TIW timer does succeed in generating intervals within deviations within half the interval size. However it is not guaranteed that the interval will not exceed that required duration by more than half.
- Ever since the initial design of the TIW timer was laid down (the final design was presented in section 4.3.1) it was clear that the number of threads that the TIW timer uses in a specific configuration impacts the accuracy of the timer significantly. In section 5.3, the goal was to quantify the effect of the number of threads. It was determined that for the TIW timer to generate a reliable interval two conditions have to be met: The number of threads should represent a whole number that is a factor of 2000.
- Finally the TIW timer's performance when required to solve a real world problem was tested in section 5.4. It was determined that the TIW timer is able to maintain its accuracy over a period of one hour. The TIW timer fired the timer event that indicated that an hour had passed only 1 ?s short of 60 minutes. Furthermore, the maximum deviation produced by the TIW timer over this period was 111 ?s.

The TIW timer is thus a flexible algorithm that conforms to the definition of a soft real-time timer and provides a timer event at the specified frequency that can be used by an application to measure time and schedule events. However, the TIW timer was found to be an effective solution under the WIN32 platform only and is unfortunately not as accurate under an operating system that uses the POSIX standard.

In addition to being an effective workaround for the inadequacies of timers currently available under the WIN32 operating system, the TIW timer is also positioned to take advantage of new processor technologies such as hyper-threading and the high performance event timer.

References

[Abeni et al. 2002] Abeni L, Goel A, Krasic C, Snow J, and Walpole J, 2002. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, CA, USA, September , 2002, IEEE Computer Society 2002, ISBN 0-7695-1739-0.

[Atlas et al. 1998] Atlas A, Bestavros A, 1998. Design and Implementation of SRMS in Kurt Linux. *Computer Science Department, Boston University*. September 1998.

[Austin Group 2004] Austin Group, 2004. The Single UNIX Specification. *The Open Group Base Specifications Issue 6*, IEEE Std 1003.1, 2004 Edition. http://www.unix.org/version3

[Barabanov 1997] Barabanov, M, 1997. A Linux-based Real-Time Operating System. *MS thesis,* New Mexico Institute of Mining and Technology, Socorro, New Mexico.

[Barr 1999] Barr M, 1999. Programming Embedded Systems with C and C++. *O'Reilly Media, Inc.* January 1999.

[Beal et al. 2003] Beal D, Ripoll I, Pisa P, Abeni L, Gai P, and Lanusse A, 2003. Linux as a Real-Time Operating System. Metrowerks Corporation – A Motorola Company, OCERA, May, 2003.

[Bulka et al. 1999] Bulka D, Mayhew D., 1999. Efficient C++: Performance Programming Techniques. *Addison Wesley Professional.* November 1999.

[Dongarra et al. 2001] Dongarra, J, London, K, Moore, S, Mucci, P, Terpstra, D, 2001. Using PAPI for Hardware Performance Monitoring on Linux Systems. In *Conference on Linux Clusters: The HPC Revolution*, Urbana, Illinois, June, 2001.

[Friesen 2001] Friesen J, 2001. Java 2 by Example, 2nd Edition. *Que Publishing*. December 2001.

[Gill et al. 2001] Gill D, Levine D, and Schmidt D, 1998. The Design and Performance of a Real-Time CORBA Scheduling Service. *Real-Time Systems Volume 20,* 117-154.

[Gopalan 2001] Gopalan K, 2001. Real-Time Support in General Purpose Operating Systems. *Research Proficiency Exam Report*, Dept. of Computer Science, State University of New York, Stony Brook, NY, January 2001.

[Hardeski 2001] Hordeski M, 2001. HVAC Control in the New Millennium. *Marcel Dekker.* January 2001.

[Henderson 2003] Henderson K, 2003. The Guru's Guide to SQL Server Architecture and Internals. *Addison Wesley Professional.* October 2003.

[Intel 1993] Intel, 1993. 8254 Programmable Interval Timer Datasheet. *Order Number* 231164-005.

[Intel 1995] Intel, 1995. Intel I/O Controller HUB 7 (ICH7) Family Datasheet. *Document Number* 307013-O01.

[Intel 1997] Intel, 1997. MultiProcessor Specification. Intel, May 1997.

[Intel 1999] Intel, 1999. Advanced Configuration and Power Interface Specification. Intel, Microsoft, Toshiba, February, 1999.

[Johnson et al. 2001] Johnson A.P, Macauley M.W.S, 2001. High Precision timing within Microsoft Windows: threads, scheduling and system interrupts. Elsevier Science B.V. July 2001.

[Klein et al. 1994] Klein M, Ralya T, Pollak B, Obenza R., Harbour M.G, 1994. A Practitioner's Handbook for Real-Time Analysis. *Springer. Part of Springer Science &Business Media.* ISBN: 0-7923-9361-9.

[Lamie 2003] Lamie E, 2005. Real-Time Embedded Multithreading: Using ThreadX and ARM. *CMP Books.* January 2003.

[Li et al. 2003] Li Q, Yao C, 2003. Real-Time Concepts for Embedded Systems. *CMP Books.* April 2003.

[Lischner 2003] Lischner R, 2003. C++ In a Nutshell: A Desktop Quick Reference. *O'Reilly Media, Inc.* May 2003.

[Locke 2005] Locke D. L, 2005. POSIX and Linux Application Compatibility Design Rules. http://www.douglocke.com

[Manko 2002] Manko E, 2002. Creating a High-Precision, High-Resolution, and Highly Reliable Timer, Utilising Minimal CPU Resources. *Codeguru.* http://www.codeguru.com

[MSDN 2003] MICROSOFT DEVELOPERS NETWORK (MSDN) April 2005. *What is a Timer?* http://msdn.microsoft.com

[Newcomer 2000] Newcomer Joseph M. 2000. THE CODE PROJECT. Time, the simplest thing. http://www.codeproject.com/system/simpletime.asp

[Palmer 2002] Palmer N, 2002. Getting a Handle on the Win32 API. Tufts University. March 2002.

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

[Perkins 2003] Perkins C, 2003. RTP: Audio and Video for the Internet. *Addison Wesley Professional.* June 2003.

[Peterson et al. 1998] Peterson P, Schotland T., 1998. Win32: A Suitable Standard for Real-Time Embedded Systems? *Real-Time Magazine 3Q98, p64 – 68,* 1998.

[Rieker 2004] Rieker R, 2004. Advanced Programmable Interrupt Controller.

[Santhanam 2003] Santhanam A, 2003. IBM developerWorks. Towards Linux 2.6. http://www-128.ibm.com/developerworks/linux/library/l-inside.html

[Sridhar 2003] Sridhar T, 2003. Designing Embedded Communications Software. *CMP Books.* June 2003.

[Timmerman et al. 2002] Timmerman M, Monfret J-C, 2002. DEDICATED SYSTEMS MAGAZINE 1997. Windows NT a Real Time OS? http://www.omimo.be/magazine/97q2/winntasrtos.htm

[Webopedia 2003] Webopedia, August 2003. *API Term definition.* http://www.webopedia.com

[WIKIPEDIA 2006 ref. 1] WIKIPEDIA: The Free Encyclopedia January 2006. *Intel APIC Architecture.* http://www.wikipedia.org

[WIKIPEDIA 2006 ref. 2] WIKIPEDIA 2006: The Free Encyclopedia January 2006. *Application programming interface.* http://www.wikipedia.org

[WIKIPEDIA 2005] WIKIPEDIA: The Free Encyclopedia January 2005. *Hyper-Threading.* http://www.wikipedia.org

[Wurmsdobler 2002] Wurmsdobler P, 2002. What is the Real Time Linux Foundation? *Real Time Linux Foundation, Inc*. September 2002.

[Yoav et al. 2003] Yoav E, Tsafrir D, and Feitelson D, 2003. Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, San Diego, California, USA, June, 2003, ACM Press, New York, NY, 172-183.

[Grobler et al. 2005] Grobler J.P, Kourie D.G, 2005. Design of a High Resolution Soft Real-Time Timer under a Win32 Operating System. In *Proceedings of the 2005 SAICSIT conference on Research for a changing world,*. White River, South Africa, September 2005, 226 – 235.