RTS-102
Temporal Event Management in EPOS

ABSTRACT

This paper presents the design, implementation, and evaluation of two time management strategies for the EPOS operating system: one based on a classic periodic time events, and one based on single-shot timers. Our results show that a properly configured periodic timer may yield high precision and low overhead. Single-shot timers, on the other hand, are usually able to match the precision of periodic timers, and introduce less jitter into the system.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

General Terms

Design, Performance

Keywords

Embedded Operating Systems, Time Management

1. INTRODUCTION

The notion of time is essential to any real-time embedded system. The system needs to keep track of how much time it is running for scheduling purposes, such as preemption of the running task, and must provide time services to the application, such as delays and alarms.

For many years, time management was done almost in the same way in different operating systems: using hardware timers configured to periodically trigger an interrupt, and using the system software to manage these *ticks*. Although this approach is simple and allows multiplexing several timing requisitions with different intervals with a single hardware timer, it incurrs in overhead to the system, since the interrupt handler runs periodically even if no time event is scheduled for a given time. During its execution, the timer interrupt handler must execute several operations, such as checking if are there tasks to be scheduled, alarms to be triggered, and if the running task must be preempted because its quantum is over.

The use of periodic time management can result in lack of precision in time services. For example, in a Linux sytem configured with a 10ms *tick* (period of the hardware timer), a 15ms time delay request of a task after a system tick may result in a total waiting time of 30ms. The interval requested will only start to count in the next tick, and will schedule this request for the next tick from that one (20ms from the count time). This lack of precision may be unacceptable for some applications.

Some strategies were created to improve time management efficiency in general purpose operating systems [5]. As an example, Windows Vista time manager can adjust the period of the hardware timer according to the system load. Linux offers to the application two timers interfaces, one that is a standard interface widely used and a high resolution timer interface. However, the excessive use of the standard timer interface inside the Linux Kernel results in unwanted overhead to the system [7].

Embedded Real-time systems have requirements that are not essential in general purpose systems, such as low-power consumption and high precision in periodic tasks activations. These requirements can only be obtained with low overhead time management implementations [6]. New approaches were created to satisfy these requirements, such as one-shot timers, soft-timers [1], smart-timers [8], and firm-timers [3].

Soft-Timer [1] triggers the time manager in the return of every system call, and not only in the hardware timer events. This results in a reduction in the number of context switches, and increase in the time service precision, since system calls tend to be much more frequent than the timer interrupts. Although system calls are widely executed in every system, its rate is not predictable. Therefore this approach cannot guarantee continuous precision, and periodic timer interrupts are used to satisfy minimum operating system requirements. *Smart-Timers* [8] have three main properties: accurate timing services with configurable maximum latency; reduced management overhead by triggering groups of nearby events together; and reduced overhead by avoiding unnecessary periodic events. Finally, *Firm-Timers* [3] combine three different approaches to time management: one-shot Timer, soft-timer, and periodic timer to reduce the number of timer interrupts, decreasing the overall overhead of periodic timers in the system.

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

2. TIME MANAGEMENT IN EPOS

EPOS (Embedded Parallel Operating System) [2, 4] is a multi-platform, component-based operating system. In EPOS, traditional operating system services are implemented through adaptable, platform-independent *System Abstractions*. Platform-specific support is implemented through *Hardware Mediators*. Mediators are functionally equivalent to device drivers in UNIX platforms, but do not compose a traditional hardware abstraction layer. Instead, they make use of platform-independent interfaces to sustain the interface contract between abstractions and the hardware. Through the use of static metaprogramming and function inlining, mediator code is *dissolved* into abstractions in compile-time.

Time is managed in EPOS by the families of components shown in figure 1. The **Clock** abstraction is responsible for keeping track of the current time, and is only available on systems that feature a real-time clock device, which is in turn abstracted by a member of the **RTC** family of mediators. The **Chronometer** abstraction is used to measure time intervals, through the use of timestamp counter (**TSC**) mediator. If a given platform does not feature a hardware timestamp counter, its functionality may be emulated by an ordinary periodic timer.

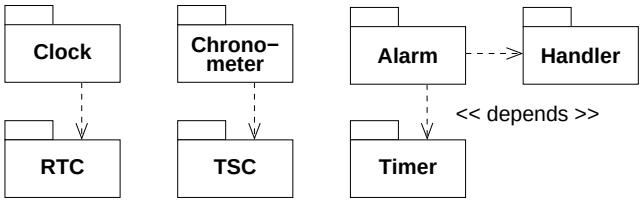


Figure 1: Families of Time Components

The **Alarm** abstraction can be used to put a thread to *sleep* for a certain time. It can also be used to generate timed events. For this purpose, an application instantiates a handler and registers it with Alarm specifying a time interval and the number of times the handler object is to be invoked. EPOS allows application processes to handle events at user-level through the **Handler** family of abstractions depicted in figure 2. The **Handler_Function** member assigns an ordinary function supplied by the application to handle an event. The **Handler_Thread** member assigns a thread to handle an interrupt. Such a thread must have been previously created by the application in the suspended state. It is then resumed at every occurrence of the corresponding event. Finally, the **Handler_Semaphore** assigns a semaphore, previously created by the application and initialized with zero, to an event. The operating system invokes operation **v** on this semaphore at every event occurence, while the handling thread invokes operation **p** to wait for an event.

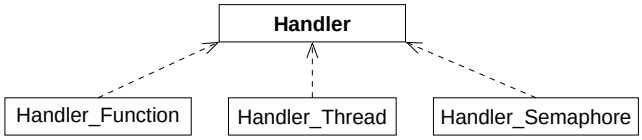


Figure 2: Event Handlers

Figure 3 illustrates how an application might use time

```
static int  iterations  = 100;
static Alarm::Microsecond time = 100000;

// ...

int main()
{
    OStream cout;
    Clock clock;
    Chronometer chron;

    // Read current system time
    cout << "Current␣Time:␣" << clock.now() << endl;

    // Create a handler function, and associate it
    // to a periodic time event.
    Handler_Function handler(&func);
    Alarm alarm(time, &handler,  iterations );

    // Start a chronometer, and put this thread to sleep
    // Afterwards, stop and read the chronometer
    chron.start ();
    Alarm::delay(time * ( iterations  + 1));
    chron.stop ();
    cout << "Elapsed␣time:␣" << chron.read() << endl;

    return 0;
}
```

Figure 3: Using Time Services in EPOS

services in the EPOS system. This application reads current system time through a **Clock** object, measures time intervals with a **Chronometer**, and registers a periodic **Alarm** with an associated **Handler_Function**.

The **Timer** class abstracts timing hardware. In a periodic event model, the platform's timer is set with a constant (configurable) frequency. When a new alarm event is registered, its interval is converted to timer *ticks*, with $T = \frac{I}{F}$, where T is the number of ticks, I is the desired interval, and F is the timer frequency. The event is then inserted into an ordered requests queue. Due to rounding errors, the numbers of ticks may not correspont to the exact desired interval. When a timer interrupt is triggered, an interrupt handler, registered by the **Alarm** abstraction, increments the ticks counter, and promotes every alarm in the event queue by a tick. If the event at the head of the queue has no more ticks to count, its handler is released.

In this work, we designed and implemented a new event model, based on *single-shot* timers. In a single-shot model, the platform's timer is programmed according to the interval of the next event. New events are enqueued according to relative interval. The **Alarm** interrupt handler promotes every alarm in the event queue with the time elapsed since the last interrupt, and programes the timer with the next event's period. Since there is no conversion to ticks, there is no precision loss when programming the interval.

In order to generate interrupts at a given frequency or period, hardware timers are usually configured with a frequency prescaler (relative to processor clock), and a comparison register. The timer hardware counts a tick each clock period, and triggers an interrupt when the ticks counter reaches the value on the comparison register. Valid hardware periods are given by

$$\frac{D}{C} \leq P \leq \frac{D \times (2^r - 1)}{C} \quad (1)$$

where D is the clock prescaler, C is the system's clock frequency, P is the timer's period, and r is the timer's resolution. Thus, when the desired event interval is larger than the timer's hardware resolution, the **Timer** mediator needs to count ticks in software.

In order to allow waiting for a period larger than the timer's hardware resolution, without incurring in overhead when this is not necessary, we introduced a **MAX_PERIOD** *configuration trait* to the timer implementation. This trat allows compile-time specialization for either hardware or software based counting, as necessary. When software counting is activated, the **Timer** mediator handles its own interrupt, in which it increments a ticks counter. When this counter is equal to the desired period, a new interrupt is triggered by the software, to be handled by the **Alarm** abstraction. As was the case with periodic timers, there may be rounding errors introduced by the conversion from interval to ticks. Since the interrupt to be handled by the **Alarm** changes according to configuration, the timer informs its IRQ through a class method.

3. CASE STUDIES

Our first case study was designed to measure the periodicity of events with different periods, in different timer clock configurations. Figure 4 presents the application implemented for this test. The main thread creates an alarm event with several iterations, and waits for the completion of all handlers. For each test, the period of each event was configured with a different value, from 100 μs to 10 s. Each event handler stores the timestamp of its execution instant. The difference between two consecutive timestamps results in an interval that, ideally, should be equal to the requested event period.

```
static Chronometer chron;
static Microsecond time_stamps[11];
volatile static int n = 0;

void handler(void) {
    time_stamps[n++] = chron.read();
}

int main() {
    // Register the events
    Handler_Function handler(&handler);
    Alarm alarm(PERIOD, &handler, 11);

    // Wait for all handlers to finish
    while(n<11) { }

    // Print intervals
    for(unsigned int i = 0; i < 10; i++)
        cout << time_stamps[i+1] - time_stamps[i] << endl;
}
```

Figure 4: Periodic Alarm Application

For every period and clock frequency, the periodic timer's frequency, and the single-shot timer's maximum interval were configured with *ideal* values. Thus, for example, if the event's period is 1 s, the periodic timer's frequency was set to 1 Hz, and single-shot timer's maximum interval was set to 1 s. From this follows that, whenever possible, the periodic timer's behaviour was equivalent to that of a single-shot timer: the timer's interrupt is only triggered when there is an event to run. Likewise, the single-shot timer's implementation only falls back to software tick counting when strictly necessary (i.e. when the requested event period is larger than the hardware's resolution). Thus, our tests yeild the best possible results for each timing strategy.

Periodic Timer			
Period (μs)	Timer Clock		
	7200 Hz	28800 Hz	115200 Hz
100	277	173	147
1000	1944	1214	1032
10000	10138	10034	10008
100000	100138	100034	100008
1000000	1000138	1000034	1000016
10000000	10001388	10000346	10000173

Single-Shot Timer			
Period (μs)	Timer Clock		
	7200 Hz	28800 Hz	115200 Hz
100	268	164	136
1000	1240	1031	1031
10000	10128	10059	10033
100000	100128	100059	100036
1000000	1000128	1000094	1152970
10000000	10138457	10282728	10921704

Table 1: Total Period

We executed this application in an 8-bit AVR microcontroller, with a 16-bit timer/counter. We configured this timer with three different clock frequencies: 7200 Hz, 28800 Hz, and 115200 Hz. Table 1 presents the average total period of each event, configured with different clock frequencies, in each timing strategy. It may be noted that, while the single-shot timing strategy usually presents better results than its periodic equivalents, both strategies present considerable errors for short periods when a "slow" (e.g. 7200 Hz) timer clock is used. Furthermore, when the requested period exceeds the maximum hardware period, and the single-shot timer falls back to software tick counting, errors for this strategy increase considerably. Likewise, when using a very "fast" (e.g. 115200 Hz) timer clock, the overhead of reprogramming the single-shot timer may exceed the overhead of counting ticks in periodic timers. Finally, it should be noted that these values represent a best-case scenario for periodic timers, since the timer's period as close to the event period as possible. On the other hand, these values represent a typical case for single-shot timers whenever the maximum event period is smaller or equal to the maximum timer hardware period.

Figure 5 presents error curves (actual period relative to ideal period) for each tested period, in each clock configuration, for each timing strategy. In every case, the error rates decrease as the requested period increases, and as the timer's clock speed increases. With larger periods and faster timer clocks, the overhead and rounding errors of the timing system are minimized. The upwards slope in the single-shot tests represents the moment where the timer's

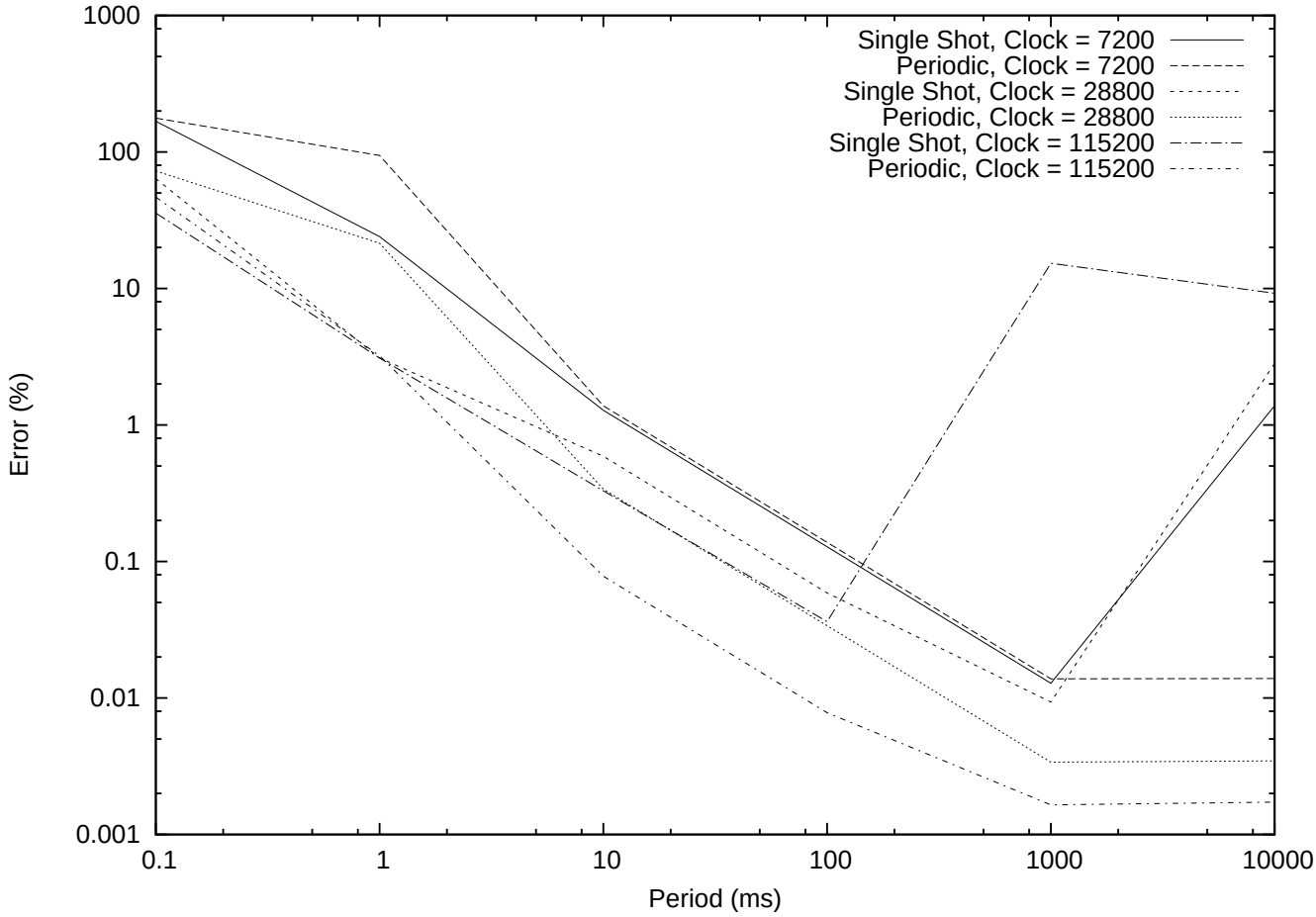


Figure 5: Error rates for different periods

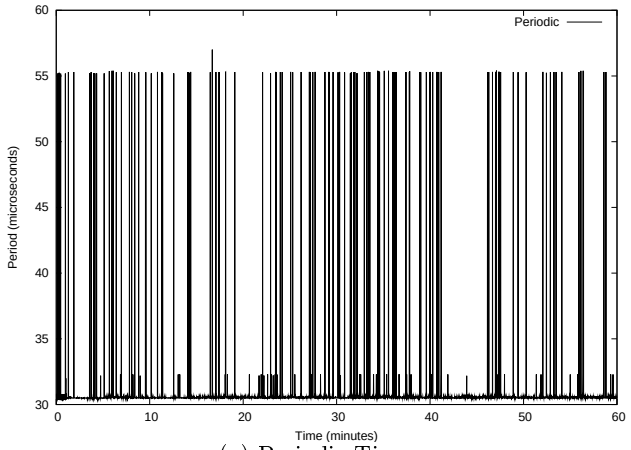
hardware maximum period is exceeded, and the timer falls back on software tick counting.

While a periodic timer may have equivalent or even superior performance than a single-shot timer, its implementation may interfere with other parts of the system. While a single-shot timer only generates interrupts when there is an event to be handled, a periodic timer generates interrupts at a constant rate, regardless of whether there is an event to handle or not. Thus, a periodic timer service may introduce jitter to other threads in the system. In order to measure this phenomenon, we devised a simple actuation system, in the form of a *glowing leds* application. In this application, a thread "glows" the leds at different intensities along time (figure 6. An alarm event periodically changes the intensity of each led. In order to create the illusion of glowing, the main thread must be executed at a regular period, with no interruptions other than the one that changes the intensity of each led.

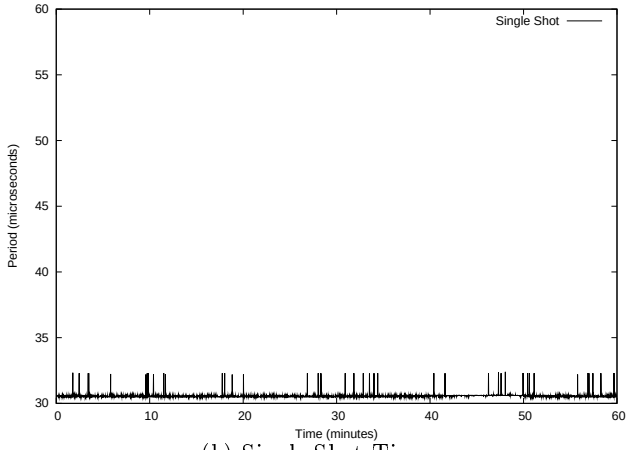
We executed this application on the same AVR platform as the last test. Timer's clock was set to 125000 Hz, the periodic timer's frequency was set to 1000 Hz. The event period for the alarm that changes leds intensity was set to 20 ms. Figure 7 presents the period of the thread that handles the leds. In order to measure this period, the output of one of the leds was connected to a digital osciloscope. When the periodic timer was used, the average period of this thread

```
int print_leds(void) {
    while(1) {
        unsigned char leds = 0;
        for(unsigned char i = 0; i < NUM_LEDS; i++) {
            // leds |= ... ;
        }
        CPU::out8(Machine::IO::PORTA, ~leds);
    }
}

void change_led_intensity(void){
    for(unsigned char i = 0; i < NUM_LEDS; i++) {
        // led_intensity[i] == ... ;
    }
}

int main() {
    Handler_Function handler(&change_led_intensity);
    Alarm alarm_a(20000, &handler, Alarm::INFINITE);
    Thread * a = new Thread(&print_leds);
    int status_a = a->join();
}
```

Figure 6: Glowing leds application



(a) Periodic Timer



(b) Single-Shot Timer

Figure 7: Thread period along time

was 31.34 μs with a standard deviation of 4.36 μs (13.91%). When the single-shot timer was used, the thread's period was 30.56 μs, with a standard deviation of 19.1 ns (0.62%). Since the single-shot timer only generates interrupts when there are events to handle, it does not interfere with the main thread in the system, yeilding a more regular period.

4. CONCLUSION

Common sense dictates that a single-shot timer implementation must always be superior to an equivalent period timer. However, our test have shown that a properly configured period timer may have, in the best case, superior performance to a single-shot timer. Furthermore, a single-shot timer is limited to hardware resolution, and must fall back to software tick counting when this resolution is exceeded.

Since, however, periodic timers always generate interrupts regardless of events to handle, they may introduce jitter into the system and. Furthermore, keeping a timer "always on", regardless of need may be considered harmful in power-aware systems. As is usually the case with embedded systems, each application must find its most suitable timing strategy. By providing alternative implementations through uniform, configurable interfaces, EPOS allows applications to find their best compromise.

5. REFERENCES

[1] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Trans. Comput. Syst.*, 18(3):197–228, 2000.

[2] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[3] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. Supporting time-sensitive applications on a commodity os. In *OSDI*, 2002.

[4] H. Marcondes, A.S. Hoeller, L.F. Wanner, and A.A.M. Frohlich. Operating systems portability: 8 bits and beyond. *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, pages 124–130, 20-22 Sept. 2006.

[5] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 30 seconds is not enough!: a study of operating system timer usage. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 205–218, New York, NY, USA, 2008. ACM.

[6] John Regehr and Usit Duongsaa. Preventing interrupt overload. *SIGPLAN Not.*, 40(7):50–58, 2005.

[7] Suresh Siddha, Venkatesh Pallipadi, and Arjan van de Ven. Getting maximum mileage out of tickless. In *OLS '07: Proceedings of the Ottawa Linux Symposium*, 2007.

[8] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. General purpose timing: the failure of periodic timers. Technical Report 2005-6, School of Computer Science and Engineering, the Hebrew University, Jerusalem, Israel, February 2005.