# Periodic Timers Revisited: the Real-time Embedded System Perspective

Antônio Augusto Fröhlich*, Giovani Gracioli, João Felipe Santos

*Software/Hardware Integration Lab, Federal University of Santa Catarina, 88040-900, Florianópolis, Brazil*

## Abstract

Common sense dictates that single-shot timer mechanisms are more suitable for real-time applications than periodic ones, specially in what concerns precision and jitter. Nevertheless, *real-time embedded systems* are inherently periodic, with tasks whose periods are almost always known at design-time. Therefore a carefully designed periodic timer should be able to incorporate much of the advantages of single-shot timers and yet avoid hardware timers reprogramming, an expensive operation for the limited-resource platforms of typical embedded systems.

In this paper, we describe and evaluate two timing mechanisms for embedded systems, one periodic and another single-shot, aiming at comparing them and identifying their strengths and weaknesses. Our experiments have shown that a properly designed periodic timer can usually match, and in some cases even outperform, the single-shot counterpart in terms of precision and interference, thus reestablishing periodic timers as a dependable alternative for real-time embedded systems.

*Corresponding author.
*Email address:* `guto@lisha.ufsc.br` (Antônio Augusto Fröhlich)

*Keywords:*
Real-Time Operating System, Time Management, Embedded System

## 1. Introduction

The notion of time is essential to any real-time embedded system. The system needs to keep track of time flow to schedule tasks and also to provide time services, such as delays and alarms, to applications. Historically, OSs have been implementing time management almost in the same way: a hardware timer is configured to periodically trigger interrupts, thus giving rise to a system time unit called *tick*. Ticks define the minimum perception of time flow within the system and rule every sort of time-driven events. The mechanism is usually implemented with a single hardware timer, whose interrupt handler is overloaded with operations around task scheduling and timed event propagation.

Though well-accepted in the realm of general-purpose systems, time management strategies based on periodic timers face strong criticism from the real-time system community [1]. For instance, in a system with a periodic timer configured to generate 10ms ticks, a 15ms delay request may result, in the worst case, in a waiting time of 30ms[1]. In addition to the lack of precision in time services, the periodic timer handler is constantly activated, even if no action needs to be taken, causing overhead and interference for the running tasks. These limitations fostered the mechanism of *single-shot* timer, with dedicated timers being programmed to fire exactly when an action has to be

[1]If the request is posted just after a tick is generated, counting will start on the next tick. Moreover, 15 might be rounded up to 20, yielding a total wait of 30ms.

performed [1, 2, 3].

Nonetheless, despite these unquestionable issues about periodic timers, while performing experiments in the context of a previous paper [4], we realized that single-shot supremacy might be indeed more closely connected to implementation issues than to the concept itself. Some of the aspects that called our attention were:

- Real-time embedded systems are intrinsically periodic. Even simple software architectures, such as cyclic executives, have a period derived from the main loop length. Real-time scheduling in more complex embedded systems is essentially periodic (e.g., Earliest Deadline First, Rate Monotonic, etc).

- Single-shot timers are usually multiplexed on a few physical timers, demanding constant hardware reprogramming, what, in some systems is much slower than software reprogramming.

- Single-shot timers, in practice, do overflow, demanding some sort of periodic fall-back mechanism.

In this way, a carefully designed periodic timer mechanism could, in theory, incorporate much of the advantages of single-shot timers. Indeed, even commercial OSs now feature improved periodic timers: Windows Vista can adjust the period of the hardware timer according to the system load [5]; Linux delivers a secondary high resolution timing interface to applications [6]. Therefore, this paper investigated this hypothesis by comparing the behavior of both time management strategies (i.e., periodic and single-shot) in the same scenario (i.e., hardware platform, OS, and applications). Our results confirmed that a properly configured and implemented 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 can cause less interference in the system when a periodic timer is not well-configured.

The remainder of this paper is organized as follows: section 2 presents related work; section 3 presents the design and implementation of single-shot and periodic timing mechanisms; section 4 describes the experimental evaluation of proposed mechanisms, along with a comparative analysis; section 5 closes the paper with our final considerations.

## 2. Related Work

Tsarfrir et. Al. discuss some issues of periodic time management, as lack of precision, and power consumption [1]. This work proposes a solution based on *Smart Timers*, which have three basic properties: 1) Accurate timing with configurable maximum latency; 2) Reduced management overhead by triggering combined nearby events, and (3) Reduced overhead by avoiding unnecessary periodic events.

Kohout presents a strategy to efficiently support RTOS, using core components implemented in hardware [7]. The main objective is to reduce the impact caused by the RTOS in the application. This impact is measured in terms of response time and CPU usage. This work introduces the Real-Time Task Manager (RTM), a memory-mapped on-chip task manager designed to deal with task scheduling, time management and event management. The RTM support for time management functions causes a 10% reduction of CPU time (with 24 tasks).

Aron and Druschel introduced the concept of *Soft-Timer*, which triggers the time manager in the return of every system call, and not only in the hardware timer events. These results have shown a reduction in the number of context switches, and an increase in the time service precision, since system calls tend to be much more frequent than 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 OS requirements.

Goel et Al. combined three different time management approaches (single-shot timer, soft-timer, and periodic timer) to build an efficient, high-resolution, low-overhead mechanism they called *Firm Timer*. The combination allows a reduction in the number of timer interrupts, thus reducing the overall overhead of periodic timers in the system [3].

In summary, all these works focus on eliminating the side-effects associated to the maintenance of a global system's tick counter. In particular, activation of the timer interrupt handler solely to increment the tick counter is strongly avoided. For this purpose, single-shot mechanisms are proposed. The performance evaluation of such mechanisms, however, is mostly done in the context of general-purpose systems, disregarding hardware reconfiguration times and the periodic nature of real-time embedded systems.

## 3. Design and Implementation

The design and implementation of the single-shot timer and periodic timer were carried out in the Embedded Parallel Operating System (EPOS) [8]. EPOS is a multi-platform, component-based, embedded system frame-

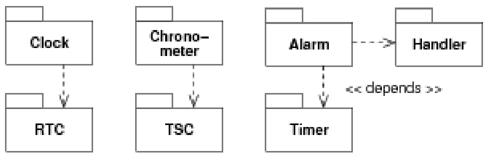


Figure 1: EPOS timing components.

work, in which traditional OS services are implemented through adaptable, platform-independent *System Abstractions*. Platform-specific support is implemented through *Hardware Mediators* [9], which are functionally equivalent to device drivers in UNIX, but do not build a traditional HAL. Instead, they sustain the interface contract between abstractions and hardware components by means of static metaprogramming techniques, thus “dissolving” mediator code into abstractions at 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 a timestamp counter (**TSC**) mediator. If a given platform does not feature a hardware TSC, its functionality may be emulated by an ordinary periodic timer.

The **Alarm** abstraction can be used to generate timed events, and also to put a thread to *sleep* for a certain time. For this purpose, an application instantiates a handler and registers it with an Alarm specifying a time period and the number of times the handler object is to be invoked. EPOS allows application processes to handle events at user-level through the **Han-**

**dler** 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 OS invokes operation **v** on this semaphore at every event occurrence, while the handling thread invokes operation **p** to wait for an event. Although not in the scope of this work, it is worth to mention that the same mechanisms are used in EPOS for real-time thread scheduling [10].

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 and relative request queue. Thus, manipulation of this queue affects only its head, because it keeps all values relative to the first element. Due to rounding errors, the number of ticks may not correspond to the exact desired interval. When a timer interrupt is triggered, an interrupt handler, registered by the **Alarm** abstraction, increments the tick counter, thus promoting every alarm in the

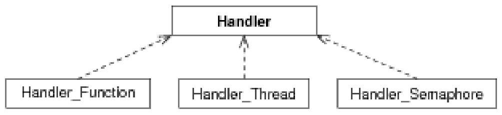
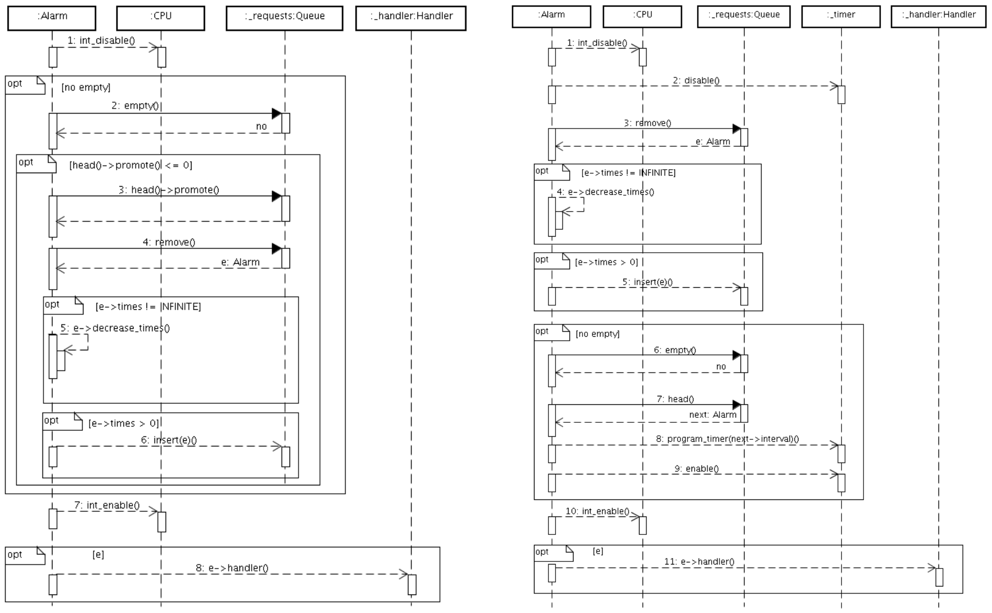


Figure 2: EPOS event handlers.



(a) (b)

Figure 3: (a) Periodic alarm handler sequence diagram (b) Single-shot alarm handler sequence diagram.

event queue by a tick. If the event at the head of the queue has no more ticks to count, its handler is released. The corresponding interrupt handler is depicted in the UML sequence diagram of Figure 3(a).

Figure 4 presents the class diagram of **Alarm** and **Chronometer** abstractions for the AVR-8 architecture. The **Alarm** abstraction uses two of the hardware timers available in the AVR microcontroller. One of these timers, **ATMega128_Timer_3**, is used to control the **Alarm** request queue. With the system clock configured at 7.2 MHz and a prescale of 1024, this 16-bit timer allows the scheduling of events spanning approximately 9 seconds. The other timer, **ATMega128_Timer_1**, generates periodic interrupts that trigger the system scheduler. These interrupts only occur when the scheduler

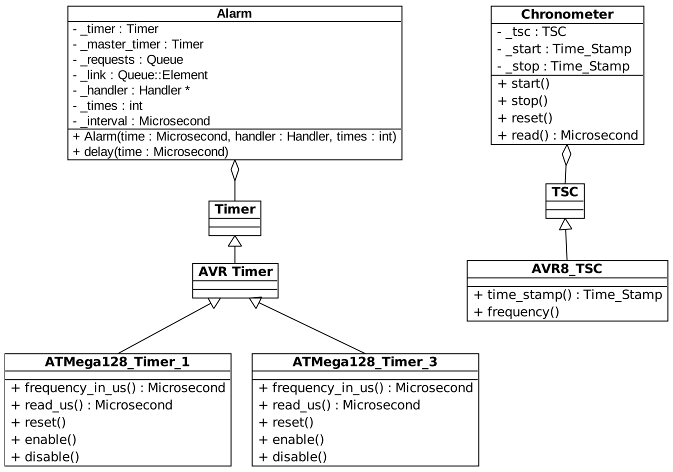


Figure 4: Class diagram of EPOS abstractions and mediators for the AVR-8 architecture.

is active, with a period that corresponds to the system's configured *quantum*.

Figure 5 illustrates how an application uses time 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**.

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 based on the interval to the next event. New events are enqueued according to their relative interval order. The **Alarm** interrupt handler promotes every alarm in the event queue with the time elapsed since the last interrupt, and reprograms the timer with the next event's interval. Since in this scheme there are no conversions between intervals and ticks, precision loss is restricted to physical timer resolution.

In order to generate interrupts at a given frequency, hardware timers are usually configured with a frequency prescaler (relative to processor clock),

```
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 5: Example of time service utilization in Epos.

and a comparison register. The hardware timer counts a tick at each clock period, and triggers an interrupt when the ticks counter reaches the value in the comparison register. Valid hardware periods are given by the following equation:

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

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 hardware timer resolution, without incurring in overhead when this is not necessary, we introduced a **MAX_PERIOD** *configuration trait* to the timer implementation. This trait 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 tick 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 period to ticks. Since the interrupt to be handled by the **Alarm** changes according to configuration, the timer informs its IRQ through a class method. The corresponding interrupt handler is depicted in the UML sequence diagram of Figure 3(b).

```
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 6: Periodic alarm application.

## 4. Experimental Evaluation

In order to evaluate the proposed time management strategies, we devised some experiments. Our first experiment was targeted at measuring the actual period of events with distinct periods, in different timer clock configurations. Figure 6 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 $\mu$s 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.

For every period and clock frequency, the periodic timer's frequency, and the single-shot timer's maximum period were configured with *ideal* values.

**Periodic Timer Period (us)**

| Target | Measured | | |
|---|---|---|---|
| **Period** | **7200 Hz** | **28800 Hz** | **115200 Hz** |
| **100** | 277 | 173 | 147 |
| **1.000** | 1.944 | 1.214 | 1.032 |
| **10.000** | 10.138 | 10.034 | 10.008 |
| **100.000** | 100.138 | 100.034 | 100.008 |
| **1.000.000** | 1.000.138 | 1.000.034 | 1.000.016 |
| **10.000.000** | 10.001.388 | 10.000.346 | 10.000.173 |

Table 1: Difference between expected and measured periods using periodic timer.

**Single-Shot Timer Period (us)**

| Target | Measured | | |
|---|---|---|---|
| **Period** | **7200 Hz** | **28800 Hz** | **115200 Hz** |
| **100** | 268 | 164 | 136 |
| **1.000** | 1.240 | 1.031 | 1.031 |
| **10.000** | 10.128 | 10.059 | 10.033 |
| **100.000** | 100.128 | 100.059 | 100.036 |
| **1.000.000** | 1.000.128 | 1.000.094 | 1.152.970 |
| **10.000.000** | 10.138.457 | 10.282.728 | 10.921.704 |

Table 2: Difference between expected and measured periods using single-shot timer.

Thus, for example, if the event's period was 1 s, the periodic timer's frequency was set to 1 Hz, and single-shot timer's maximum period was set to 1 s. From this follows that, whenever possible, the periodic timer's behavior 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 yield the best possible results for each timing strategy.

We executed this experiment in an 8-bit AVR microcontroller (ATMega128), with a 16-bit timer/counter. We configured this timer with three different clock frequencies: 7200 Hz, 28800 Hz, and 115200 Hz. Tables 1 and 2 present the average total period of each event, configured with different clock frequencies, in each timing strategy. It should 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 was configured as close to the event's period as possible. Nonetheless, these values represent a typical case for single-shot timers when-

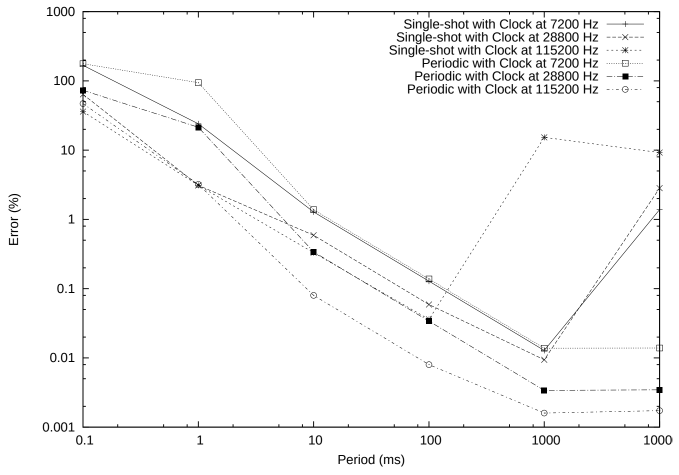


Figure 7: Error rates for different periods.

ever the maximum event period is smaller or equal to the maximum timer hardware period.

Figure 7 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 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. A single-shot timer only generates interrupts when there is an event to be handled, while a periodic timer generates interrupts at a constant rate, regardless of whether there is an event to handle or not. Thus,

| **Interrupt / Approach** | **Periodic** | **Single-Shot** |
|---|---|---|
| Counting ticks interrupt | 14 $\mu$s | - |
| Alarm release interrupt | 42 $\mu$s | 212 $\mu$s |

Table 3: Time spent in handling the alarm interrupt using periodic and single-shot timers.

a periodic timer service may interfere with other threads in the system. In order to evaluate this phenomenon, we measured the time spent in handling an alarm event in both approaches using the same AVR platform. At the beginning of the alarm handler, we turned a LED on and before leaving the handler we turned it off. We connected a digital oscilloscope to the output of the LED to measure the elapsed time. For this test, an alarm triggered at every 5 ms and the timer was configured with a frequency of 1000 Hz and a clock of 125000 Hz. A periodic timer generates four interrupts before the alarm is released, that is, during 4 interrupts it will only count ticks. The single-shot timer only triggers when it is necessary, but it must reprogram the timer after each event. The goal of the experiment is to measure the interference of both approaches in the system as a whole. Obtained values are shown in Table 3. For an interrupt that counts ticks, the periodic timer took 14 $\mu$s and for an interrupt where is necessary to release the alarm it took 42 $\mu$s. The single-shot timer, however, took 212 $\mu$s for releasing the alarm event. This high overhead is due to the time needed to reprogram the hardware timer.

In order to evaluate the behavior of a poorly configured periodic timer, we devised a simple actuation system, in the form of a *Pulse-Width Modulator* (PWM) software application. In this application, a thread "glows" LEDs at different intensities along the time (Figure 8). An alarm event periodically

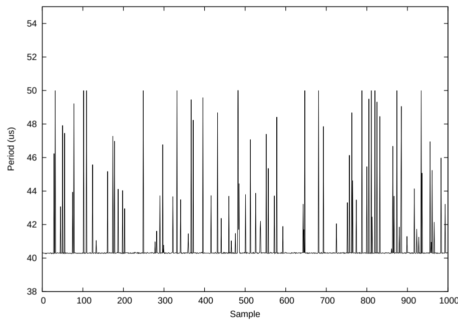
```
int print_leds(void) {
    while(1) {
        unsigned char leds = 0;
        for(unsigned char i = 0; i < NUM_LEDS; i++) {
            // leds |= ... ;
        }
        CPU::out8(Machine::IO::PORTA, ~leds);
    }
}
void change_led_intensity(void){
    for(unsigned char i = 0; i < NUM_LEDS; i++) {
        // 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 8: PWM led glowing application.

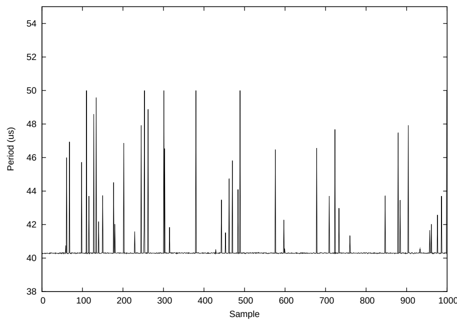
changes the intensity of each LED (duty cycle). In order to create the illusion of glowing, the main thread must be executed at a regular period, with no interruptions other than the one that changes the intensity of each LED.

We executed this application on the same AVR platform. The event period for the alarm that changes LEDs intensity was set to 20 $ms$. The timer's clock was set to 125000 Hz, and the periodic timer's frequency was set to 1000 Hz (1 interrupt at every 1 $ms$), 500 Hz (1 interrupt at every 2 $ms$) and 50 Hz (1 interrupt at every 20 $ms$, the best case for the periodic timer). Figure 9 shows the interference caused by the timer mechanisms on 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 oscilloscope. Note that the small period variance for single-shot is not related to interrupt handling interference, but to the very own nature of the PWM algorithm. We have also measured the the output delay in changing a pin (LED on and off) in the same AVR platform through the digital oscilloscope to corroborate our results. We ran this experiment for approximately 8 hours getting 10000 values for the rise and fall times of a pin. The average rise time was 7.25 $ns$ with a standard deviation of 1.42 $ns$ and the average fall time was 6.41 $ns$ with a standard deviation of 0.13 $ns$. These values represent less than 0.05% of the measured value, considering a period of 20 $ms$. This proves that our tests methodology is correct.

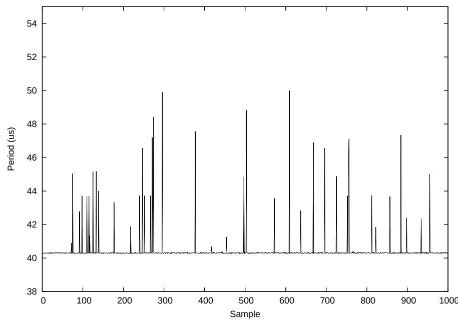
Figure 10 summarizes the interference caused by timers mechanisms on the period of the PWM thread in terms of standard deviation. When the single-shot timer was used, the thread's average period was 40.52 $\mu$s, with a standard deviation of 1.12 $\mu$s (2.77% of the thread's average period). When the 1000 Hz periodic timer was used, the average period of this thread was 40.75 $\mu$s with a standard deviation of 1.72 $\mu$s (4.23% of the thread's average period). Since the periodic timer is not well configured, it generates interrupts that interfere with application. On the other hand, the periodic timers configured with 500 Hz and 50 Hz presented average periods of 40.51 $\mu$s and 40.44 $\mu$s respectively, but the 500 Hz periodic timer obtained a higher standard deviation (1.19 $\mu$s of the thread's average period). Since the 50 Hz periodic timer will only generate interrupts when necessary, it presented the best average period and standard deviation (40.47 $\mu$s and 0.97 $\mu$s). This example shows how a bad periodic timer configuration can affect the system



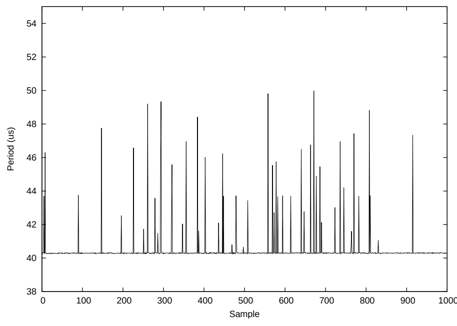
(a) Periodic timer at 1000Hz



(b) Periodic timer at 500Hz



(c) Periodic timer at 50Hz



(d) Single-shot

Figure 9: Timer interference on the PWM thread's period along the time: (a) using periodic timer configured with 1000Hz; (b) periodic timer with 500Hz; (c) periodic timer with 50Hz; and (d) using single-shot timer.

performance.

Our last experiment was designed to evaluate the performance of both approaches in a scenario with concurrent time events. The objective is to measure the execution time of the periodic interrupt handler in a multi-threaded environment. Figure 11 shows the application used in this test. The main function creates threads which execute functions (*func_ a*, *func_ b*, etc). When the alarm triggers, the thread is resumed (as described in section 3)

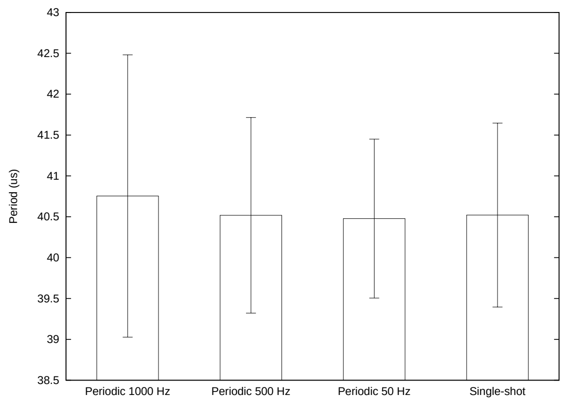


Figure 10: Mean period and standard deviation for each timer strategy and configuration with relation to Figure 9.

and repeats the loop. We ran this experiment in the same AVR platform. Due to memory restriction, we only vary the number of threads from 2 to 6. All threads have the same priority and their periods were 10 $ms$, 30 $ms$, 60 $ms$, 90 $ms$, 120 $ms$, and 150 $ms$ respectively. The hardware timer was configured with a frequency of 100 Hz (period equal to 10$ms$) and clock frequency of 28800 Hz.

For this test, we have changed the periodic timer interrupt handler in order to release all time events which have their ticks less or equal to zero in the same interrupt handler. Figure 12 exemplifies the new interrupt handler scenario. It is important to highlight that the timer job ends when it releases all threads. The execution order of threads is guaranteed by the thread's priority in the scheduler. In this case, the execution time of an interrupt handler is not constant, it varies depending on the number of threads that reached their period in that interrupt.

Table 4 shows the periodic interrupt handler execution time for this appli-

```
Thread *a,*b,*c,*d,*e,*f;

int func_a() {
    while(1) {
        //suspends itself
        a->suspend();
    }
}
int func_b() {
    while(1) {
        b->suspend();
    }
}
 ....
int main() {
    //creates all Threads and their Handlers
    a = new Thread(&func_a);
    Handler_Thread handler_a(a);
    b = new Thread(&func_b);
    Handler_Thread handler_b(b);
     ....
    //creates all Alarms
    Alarm alarm_a(Period_A, &handler_a, Alarm::INFINITE);
    Alarm alarm_b(Period_B, &handler_b, Alarm::INFINITE);
     ....
    int status_a = a->join();
    int status_b = b->join();
     ....
}
```

Figure 11: Threads application test.

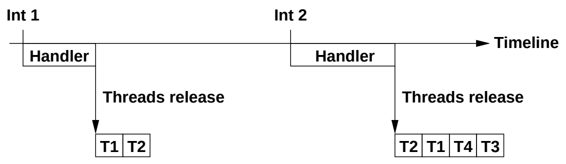


Figure 12: Periodic timer interrupt handler in a multi-threaded scenario. Note that the interrupt handler has different execution times.

| **Number of** | **Interrupt Handling Time (us)** | | | |
|---|---|---|---|---|
| **Threads** | **Average** | **Max** | **Min** | **Std** |
| **2** | 81 | 111 | 63 | 23 |
| **3** | 84 | 117 | 63 | 25 |
| **4** | 91 | 178 | 63 | 36 |
| **5** | 97 | 190 | 63 | 41 |
| **6** | 107 | 263 | 63 | 55 |

Table 4: Interrupt handling time of the Threads application varying the number of threads.

cation varying the number of threads. When running the application with 2 threads with periods of 10 $ms$ and 30 $ms$, for instance, the average execution time was 81 $\mu$s, the worst-case execution time was 111 $\mu$s, the best execution time was 63 $\mu$s, and the standard deviation was 23 $\mu$s. In comparison with the single-shot interrupt handler execution time in Table 3, the periodic interrupt handler presented a better performance up to 5 threads. With 6 threads, the worst-case execution time of the periodic handler (263 $\mu$s) was worse than single-shot (202 $\mu$s).

Two final remarks about the experiments carried out:

- Deeply embedded system

  Our work is focused on deeply embedded systems. Such systems present serious resources limitations, such as power consumption, processing power, and memory. Moreover, these systems are designed to run a specific set of applications, whose requirements are known at design-time. Therefore, configuring the periodic timer to fit system needs is a fully valid approach.

  Furthermore, the time spent by reprogramming the hardware timer in the single-shot implementation in a deeply embedded system is high, since this task involves calculations, like divisions and multiplications, in order to adjust the next time requested by the application to the hardware timer period.

- Epos dependency

  The experiments described here used the Epos system, but the basic idea can be applied to virtually any embedded operating system. The problem itself is related to how the periodic timer is implemented and not to the embedded operating system. A smart periodic time management implementation can supply and adjust to the needs of embedded or real-time applications.

## 5. Conclusion

Common sense dictates that single-shot timer mechanisms are better than periodic ones. Nonetheless, our experiments have shown that, for real-time embedded systems, a properly configured periodic timer can usually match the single-shot approach in terms of performance and interference. Indeed, they shown that a periodic timer can, in some specific cases, outperform an equivalent single-shot mechanism. This apparently unaccountable outcome arises basically from the intrinsically periodic nature of embedded systems and from the way timers are implemented in such systems.

A periodic timer mechanism does not require the hardware timer to be reprogrammed for each event. The hardware timer is programmed during system initialization to trigger interrupts with a frequency that best matches the periods of events that will be handled by that system. This, in combination with a properly designed event queue, can render a simple, fast, and regular timer interrupt handler. Furthermore, a single-shot timer is limited by hardware resolution, and must fall back to software tick counting when its resolution is exceeded.

Although this scenario of tailored periodic timer mechanisms does not fit the all-purpose essence of ordinary OSs, which must work in a best-effort to accommodate a myriad of application demands, it does fit well in the realm of real-time embedded systems. It is not our intention, however, to promote periodic timers as a generally better alternative for such systems than single-shot. There are many cases in the literature for which single-shot approaches have proved superior, in particular, concerning power efficiency and jitter. Our main intention is to reestablish periodic timer mechanisms as a concrete alternative for real-time embedded systems.

**Acknowledgment**

We would like to thank and acknowledge LISHA members Lucas Wanner, Roberto de Matos, and Danillo Moura Santos for the work on the initial implementation of single-shot timers for EPOS and for the assistance with the experiments reported in this paper. We would also like to thank Prof. Rômulo de Oliveira for the prolific discussions about timing in real-time

systems.

## References

[1] D. Tsafrir, Y. Etsion, D. 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, 2005.

[2] M. Aron, P. Druschel, Soft timers: efficient microsecond software timer support for network processing, ACM Trans. Comput. Syst. 18 (2000) 197–228.

[3] A. Goel, L. Abeni, C. Krasic, J. Snow, J. Walpole, Supporting time-sensitive applications on a commodity os, in: OSDI.

[4] G. Gracioli, D. Santos, R. de Matos, L. Wanner, A. A. Fröhlich, One-shot time management analysis in epos, in: Proc. of the XXVII SCCC, IEEE, Punta Arenas, Chile, 2008.

[5] S. Peter, A. Baumann, T. Roscoe, P. Barham, R. 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, ACM, New York, NY, USA, 2008, pp. 205–218.

[6] S. Siddha, V. Pallipadi, A. van de Ven, Getting maximum mileage out of tickless, in: OLS '07: Proceedings of the Ottawa Linux Symposium.

[7] P. Kohout, B. Ganesh, B. Jacob, Hardware support for real-time operating systems, in: CODES+ISSS '03: Proceedings of the 1st

IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, ACM, New York, NY, USA, 2003, pp. 45–51.

[8] A. A. Fröhlich, Application-Oriented Operating Systems, number 17 in GMD Research Series, GMD - Forschungszentrum Informationstechnik, Sankt Augustin, 2001.

[9] F. V. Polpeta, A. A. Fröhlich, Hardware mediators: A portability artifact for component-based systems, in: EUC, pp. 271–280.

[10] H. Marcondes, R. Cancian, M. Stemmer, A. A. Frohlich, On the design of flexible real-time schedulers for embedded systems, Computational Science and Engineering, IEEE International Conference on 2 (2009) 382–387.