

# Multiplexing Real-time Timed Events

Mike Holenderski, Wim Cools, Reinder J. Bril and Johan J. Lukkien

*Technische Universiteit Eindhoven (TU/e)*

*Den Dolech 2, 5600 AZ Eindhoven, The Netherlands*

*m.holenderski@tue.nl, w.cools@student.tue.nl, r.j.bril@tue.nl, j.j.lukkien@tue.nl*

**Abstract**—This paper presents the design and implementation of RELTEQ, a timed event management algorithm based on relative event times, supporting long event interarrival time, long lifetime of the event queue, no drift and low overhead.

RELTEQ has been conceived to replace and improve the existing timed event management approach in  $\mu$ C/OS II, a real-time operating system used by one of our industrial partners. Experimental results confirm a lower overhead of the proposed method in terms of both processor and memory requirements compared to the existing approach.

## I. INTRODUCTION

Real-time systems need to schedule many different timed events (e.g. programmed delays, arrival of periodic tasks, budget replenishment, time keeping). On contemporary computer platforms, however, the number of hardware timers is usually limited, meaning that events need to be multiplexed on the available timers. Typical requirements of real-time systems on timed event management are support for (i) high resolution event times, (ii) long event interarrival time, (iii) long system lifetime, (iv) low overhead, and (v) no drift.

This work is performed in the context of the CANTATA [CANTATA, 2006] project in cooperation with an industrial partner in the surveillance domain. Their current system is implemented on top of the  $\mu$ C/OS II operating system [Labrosse, 1998], which supports only delay events. Every delay event is represented with a separate counter, which is decremented upon every tick, hence the tick handler complexity is linear in the number of events.

In order to use a combination of periodic tasks and reservations to exploit fluctuating network bandwidth [Holenderski et al., 2008] and implement it in  $\mu$ C/OS II we need a general mechanism for different kinds of timed events.

Event queues are a common method for multiplexing many events on a single timer. In this paper we present a design and implementation of an event queue satisfying the above real-time requirements while reducing the processor and memory overhead compared to existing solutions.

The arrival time of event  $e_i$  is denoted as  $t_i$ . An event queue stores the events in a queue sorted by their arrival time. The event time can be expressed in absolute or relative terms.

Most computer platforms have a *clock*, which monitors the oscillations of a crystal and periodically increments a counter [Liu, 2000]. The clock can be programmed to generate an interrupt when the counter reaches a certain expiration count,

This work has been supported in part by the Information Technology for European Advancement (ITEA2), via the CANTATA project.

called the *expiration time*. The combination of a clock and an expiration time is called a *timer*. A timer can be either one-shot or periodic. A *one-shot timer* is set to expire once. A *periodic timer* will expire periodically with the expiration time defining the interval between two consecutive expirations. The interrupt generated by a periodic timer is called a *tick*.

The clock granularity (i.e. the interval between two consecutive increments of the counter) determines the resolution of timers (i.e. the smallest interval between two different time values measured by the timer). The resolution of event times is derived from the timer driving the event queue. Usually a one-shot timer will offer a higher event resolution than a periodic timer.

Managing the timed events in the event queue can be considered independently from the driving timer.

The main contribution of this paper is the design and implementation of RELTEQ, an event queue based on relative event times, which in contrast to existing approaches satisfies the above real-time requirements without tradeoffs. We implemented RELTEQ in  $\mu$ C/OS II on top of a periodic timer.

The remainder of this paper discusses related work in Section II, causes and measures for avoiding drift in Section III, and the RELTEQ event queue in Section IV with experimental results in Section VI.

## II. RELATED WORK

### A. Time models

*Linear time model:* Typical operating systems for medium size machines use a linear time model, where time is represented using an  $n$ -bit variable. An example of a linear time model is illustrated in Figure 1.

The disadvantage of a linear time model is that it imposes a finite lifetime: the system cannot represent times past  $2^n$ . In case of 16-bit time representation and 1 millisecond clock granularity the system has a lifetime of about one minute. Increasing the lifetime requires either using a larger number of bits or setting a lower time resolution. Unfortunately, both solutions can be inappropriate for an embedded system with stringent memory requirements and real-time constraints.

*Circular time model:* The circular time model handles the overflow condition occurring when the  $n$ -bit variable used to represent the system time passes from  $2^n - 1$  to 0.

Figure 1 compares the existing time models with  $n$ -bit time representation. Linear time models are limited to a fixed time interval; they do not allow to schedule events later than  $2^n$ . Circular time models can be visualized as a sliding window,

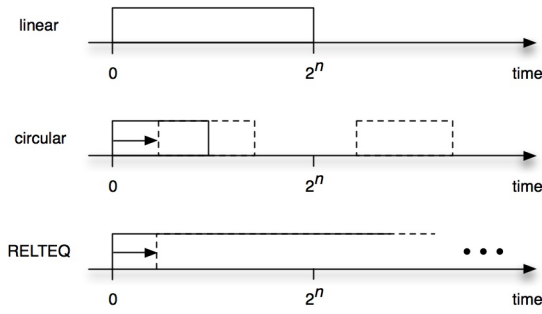


Fig. 1. Comparison of the linear, circular and RELTEQ time models with  $n$ -bit time representation. Solid boxes represent the initial event time range. Dashed boxes represent event time range as time progresses. Box size represents the maximum event interarrival time.

where the window size represents the maximum event interarrival time. As time progresses and events are executed, the time interval is shifted, allowing to schedule events beyond  $2^n$ . Using the circular time model the maximum event interarrival time is  $2^{n-1}$ , since at least one bit is necessary to discriminate between different cycles.

RELTEQ is based on a circular time model while reducing the overhead for handling overflows.

### B. Absolute times

Time is always expressed relative to some reference. In case of absolute times the reference time is usually implicit. For example, the standard time measure in many modern computers is expressed in terms of seconds since 0:00 on January 1st, 1970. However, since the reference time does not change, the number of bits required to represent the absolute time monotonically increases with later events, which in the case of the linear time model leads to limited system lifetime. The circular time model gets rid of the life time limitation at a cost of reduced maximum interval between events, as shown in Figure 1.

Examples of real-time systems with the circular time model and absolute event times are the extension of VxWorks with hierarchical scheduling in [Behnam et al., 2008] and RTLinux [RTLinux, 2007]. The SHaRK real-time operating system [SHaRK, 2006], [Gai et al., 2001] uses absolute arrival times and allows to choose between an implementation on top of a periodic or a one-shot timer. The arrival times are represented by a pair of long integers (*seconds*, *nanoseconds*). The event queue is implemented as a single list.

[Carlini and Buttazzo, 2003] present the Implicit Circular Timers Overflow Handler (ICTOH). It requires managing the overflow at every time comparison and is limited to timing constraints which do not exceed  $2^{n-1}$ . [Buttazzo and Gai, 2006] present an implementation of ICTOH minimizing the time handler overhead.

### C. Relative times

In case of relative times the reference time is explicit. In the  $\mu$ C/OS II [Labrosse, 1998] and Resource Kernel [Rajkumar et al., 1998] real-time operating systems, the arrival time of

every event is relative to the current time. It is implemented by representing each arrival time with a counter which is decremented upon every periodic timer tick. Since the reference time (i.e. the current time) is constantly increasing, relatively few bits are required to represent the arrival times. However, this approach incurs large performance overhead, since at every tick the arrival times of *all* events in the event queue have to be decremented.

RELTEQ, which is based on relative times as well, provides long event interarrival times *and* long system lifetime, while minimizing the memory requirement compared to absolute event times, and processor requirements compared to the current  $\mu$ C/OS II implementation based on relative times.

Figure 2 compares existing time models, where  $I$  is the maximum interval between any two events in the queue,  $L$  is the system lifetime, and  $P$  is the largest time which can be represented. In case of  $n$ -bits  $P = 2^n$ .

Model	$I$	$L$	event time
Linear	$P$	$P$	absolute
[Fonseca, 2001]	$P/4$	$\infty$	absolute
[Carlini and Buttazzo, 2003]	$P/2$	$\infty$	absolute
[Park et al., 2001]	$P$	$\infty$	absolute
$\mu$ C/OS II	$P$	$\infty$	relative
RELTEQ	$\infty$	$\infty$	relative

Fig. 2. Time model comparison.  $I$  is the maximum interval between any two events in the queue,  $L$  is the system lifetime, and  $P$  is the largest time which can be represented.

Note that according to [Carlini and Buttazzo, 2003] their approach has a lower memory and processor overhead than [Park et al., 2001].

## III. DRIFT AND ITS CAUSES

*Absolute jitter* defines the bound on the error between computer time and wall clock time [Klein et al., 1993]. A timed event management system is said to have *drift* if the absolute jitter is unbounded. We can identify the following causes for drift:

- 1) Hardware is inaccurate.
- 2) Granularity of event times is not a multiple of the granularity of the clock.
- 3) One-shot timer is programmed by setting the next expiration time relative to the current time.
- 4) Granularity of event times is not a multiple of the granularity of the tick.<sup>1</sup>

The first two causes derive from the limitations of the hardware and pose limitations on the application and are outside the scope of this paper. The other two causes can be addressed by proper programming measures.

The third cause describes a scenario where upon timer expiration the one-shot timer is programmed by setting the next expiration time to the next event, followed by handling the current event. Depending on whether the event times are absolute or relative, setting the next timer expiration time may

<sup>1</sup>Note that the clock granularity determines the resolution of timers, while the timer resolution determines the resolution of events.

require computing the difference between the next and current event, or simply taking the next event time, respectively. In both cases there is the overhead of setting the timer and possible interference from higher priority interrupts or disabling of timers, which will accumulate over time and thus lead to drift.

One possible solution for the drift problem of the one-shot timer is using a periodic timer instead. A fine granularity of the periodic timer will allow a high resolution of the events, but will also come at a high overhead, which gives a tradeoff between resolution and overhead.

Another solution is to use a High Precision Event Timer (HPET) [Intel, 2004], which is present on many modern computers with Intel processors. It avoids the drift problem of a one-shot timer by expressing the expiration time in terms of crystal oscillations. The HPET is programmed by setting the next expiration time *relative to the last expiration*.

The fourth drift cause is due to events arriving in between ticks. Such events will be handled upon the next tick, which in case of relative event times may lead to accumulation of drift. It can be remedied by expressing the next event time in absolute terms, bounding the jitter by at most one tick.

In the remainder of this paper we assume that the hardware is accurate, that the granularity of the clock is sufficient to express the event times, and that the event queue is driven by a periodic timer (since this is also the case in  $\mu\text{C}/\text{OS II}$ ).

#### IV. RELTEQ

The main ideas behind RELTEQ are to use relative event times and to insert “dummy” events to extend the maximum interarrival time between proper events.

Rather than storing the event times relative to the current time like it is done in  $\mu\text{C}/\text{OS II}$ , RELTEQ stores the arrival times of events relative to each other, by expressing the arrival time of an event relative to the arrival time of the previous event, with the arrival time of the head event being relative to the current time, as shown in Figure 3.

Two operations can be performed on an event queue: new events can be inserted and the head event can be popped. When inserting a new event  $e_i$ , the event queue has to be traversed, accumulating the relative times of the events until a later event  $e_j$  is found, with  $t_j < t_i$ , where  $t_i$  and  $t_j$  are both absolute times.

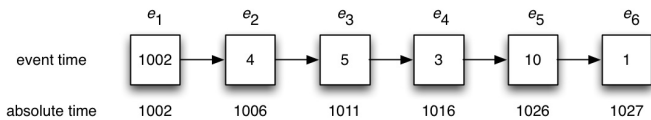


Fig. 3. Example of the RELTEQ event queue.

##### Unbounded interarrival time between events

For an  $n$ -bit time representation, the maximum interval between two consecutive events in the queue is  $2^n$ . This is already an improvement on the circular time models presented in Section II-A, where the maximum interval between *any* two

events is  $2^{n-1}$ . RELTEQ allows for *arbitrary* interval between any two events by inserting “dummy” events, as shown in Figure 4.

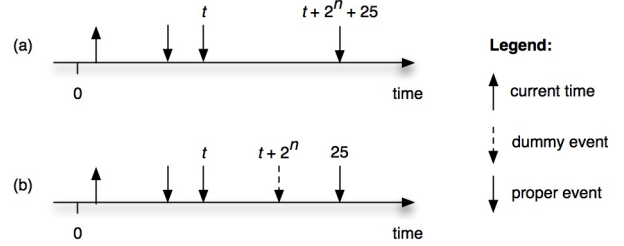


Fig. 4. Example of (a) an overflowing relative event time (b) RELTEQ inserting a dummy event to handle the overflow.

If  $t$  represents the event time of the last event in the queue, then an event  $e_i$  with a time larger than  $2^n$  relative to  $t$  can be inserted by first inserting dummy events with time  $2^n$  at the end of the queue until the remaining relative time of  $e_i$  is smaller or equal to  $2^n$ .

##### The first event

In order to avoid drift due to events arriving between two ticks (as described in Section III), the arrival time of the first event is expressed in *absolute* time.

Since the event time of the first event is stored as an absolute time it may overflow. We address this issue by inserting dummy events at times when the absolute time would overflow, as shown in Figure 5.

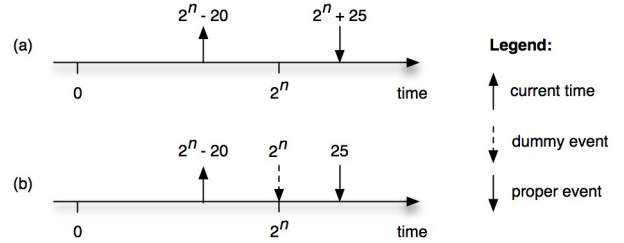


Fig. 5. Example of (a) overflowing absolute time of the first event (b) RELTEQ inserting a dummy event to handle the overflow.

The time will overflow once in  $2^n$  ticks (assuming an  $n$ -bit time representation), requiring to insert one dummy event every  $2^n$  ticks. Since the number of proper events within that time interval is likely to be high, the overhead of using dummy events to handle absolute time overflows is small.

##### Implementation

We have implemented RELTEQ on top of the  $\mu\text{C}/\text{OS II}$  rel-time operating system. Similar to the standard  $\mu\text{C}/\text{OS II}$  implementation of delay events, the RELTEQ events (i.e. periodic task arrival and delays) are stored in the Task Control Blocks (TCB), which is a structure managed by  $\mu\text{C}/\text{OS II}$  for keeping track of task parameters, such as priority, blocking state, etc. As currently one queue is used per event type, the

TCB structure is extended to contain information for  $n$  events (where  $n$  is the number of different kinds of events).

Each event contains the following information

- Link to TCB containing the next event of this type
- Link to TCB containing the previous event of this type<sup>2</sup>
- Timestamp associated with this event (relative to previous event)

For every type of event, a Queue structure is defined, which is basically a pointer to the TCB containing the first event.

The Queue structures together with the event information in the TCBs form a double linked list for each queue. By storing the event information inside of the  $\mu C/OS$  II TCBs, we won't need to store any pointers to the task associated to the event. The event kind is stored only once, in the Queue structure.

The field that was originally used for storing the delay (in the TCB) in  $\mu C/OS$  II is now no longer used.

Figure 6 illustrates the differences between the data structures in RELTEQ and standard  $\mu C/OS$  II.

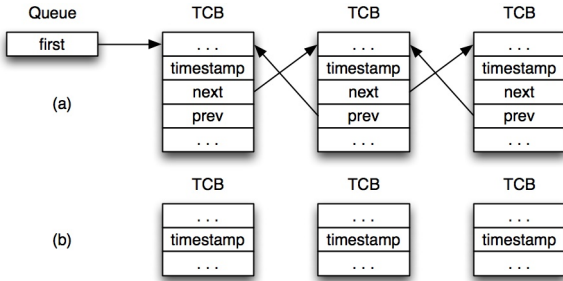


Fig. 6. Data structures for timed event management in (a) current implementation of RELTEQ in  $\mu C/OS$  II (b) standard  $\mu C/OS$  II.

As the TCBs are used to store the events, one event of every type can be stored per task (i.e. each task can have 1 Period Event and 1 Delay Event at the same time, just like  $\mu C/OS$  II can have only delay 'event' / task). Keeping the number of RELTEQ events limited also imposes an upper bound on memory usage and queue processing time.

The set delay function of  $\mu C/OS$  II has also been changed: instead of setting the delay value in the TCB, it inserts a Delay Event in the RELTEQ Queue. Delays larger than  $2^n$  are handled in the same way as in standard  $\mu C/OS$  II: looping over as many delays of  $2^n$  as necessary (the current implementation does not support dummy events).

The tick interrupt service routine of  $\mu C/OS$  II has been replaced with a routine to handle all the due events and update the queue.

The RELTEQ source code can be downloaded from [RELTEQ, 2009].

## V. DISTINCTION BETWEEN EVENT QUEUE AND TIMER

We can identify two layers in the RELTEQ architecture: the timed event management, as discussed in this paper, is happening on the event queue layer, while the timer expiration

<sup>2</sup>The previous pointer is currently only used for removing events (a delay event can be removed by other means than simply being due and handling, e.g. when the time-out for a semaphore is removed).

is managed by the timer layer. Ideally it should be possible to implement the RELTEQ algorithm on top of different timers with minimal effort.

The choice of data structures of the event queue will have an impact on the memory requirements of the whole timed event management system and the performance of inserting new events. The time representation of events in the queue will have consequences for the fourth drift cause in Section III, while the first three drift causes can be countered by appropriate choice of the timer. In the RELTEQ approach the real-time requirements of long event interarrival time and long system lifetime are addressed on the event queue level.

The resolution of events is determined by the timer driving the event queue, in particular by the granularity of the periodic timer or the resolution of the one-shot timer. The particular timer choice will also have impact on the overhead of programming the timer and the overhead of interference due to timer expirations.

## VI. EXPERIMENTAL RESULTS

We verified that the RELTEQ implementation in  $\mu C/OS$  II, offering a more general approach to timed event management, does not suffer from larger overhead than the current  $\mu C/OS$  II implementation.

The experimental setup consisted of one low priority task  $\tau_l$  with a fixed computation time (a loop with a fixed iteration count) and 59 higher priority instances of a task  $\tau_d$  which would loop over a delay command. We ran the same task set under  $\mu C/OS$  II with RELTEQ and the standard  $\mu C/OS$  II, and compared the response time of  $\tau_l$ . Since the system was fully utilized during the response time of  $\tau_l$ , the difference in its response time represents the overhead of the tick handler handling the delay events of the high priority instances of  $\tau_d$ .

Our initial results, based on a tick period of 1ms and delays of 200 ticks times the priority of the task (priorities ranged from 1 to 60, with 1 being the highest priority), show a 3% shorter response time under RELTEQ.

Regarding the memory requirements, RELTEQ requires two additional pointers per event kind per task, and one additional pointer for the head of each queue.

## VII. CONCLUSION

RELTEQ is an elegant timed event management method based on relative event times, supporting long event interarrival time, long lifetime of the event queue, no drift and low overhead. It was implemented in the  $\mu C/OS$  II real-time operating system to replace and improve its current timed event management system. Experimental results confirm a lower overhead of the proposed method in terms of processor requirements at a small cost of memory requirements, compared to the existing approach.

## VIII. FUTURE WORK

*Avoid drift without HPET support*

The RELTEQ approach is not limited to the current  $\mu C/OS$  II implementation. It is well suited for implementation on top

of a one-shot timer for exploiting its high resolution and low overhead compared to a periodic timer with fine granularity. However, an event queue driven by a one-shot timer may suffer from drift (as discussed in Section III).

One solution is to use a High Precision Event Timer, as discussed in Section III. It is to be investigated how to avoid drift on platforms *without HPET support*.

#### *Generalize implementation to arbitrary events*

The current RELTEQ implementation embeds the event queue inside the TCBS. This is sufficient for events pertaining to tasks only (such as periodic arrival or deadlines), however, implementing reservations will require an additional data structure for handling events such as budget replenishment or budget expiration.

#### REFERENCES

- [Behnam et al., 2008] M. Behnam, T. Nolte, I. Shin, M. Åsberg, R. J. Bril. *Towards hierarchical scheduling on top of VxWorks*. In *Proc. of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 63–72. 2008.
- [Buttazzo and Gai, 2006] G. Buttazzo, P. Gai. *Efficient EDF implementation for small embedded systems*. In *Proc. of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. 2006.
- [CANTATA, 2006] CANTATA. *Content Aware Networked systems Towards Advanced and Tailored Assistance*. 2006. URL <http://www.hitech-projects.com/euprojects/cantata>.
- [Carlini and Buttazzo, 2003] A. Carlini, G. C. Buttazzo. *An efficient time representation for real-time embedded systems*. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pp. 705–712. ACM, New York, NY, USA, 2003.
- [Fonseca, 2001] P. Fonseca. *Approximating linear time with finite count clocks*. Tech. rep., Dep. de Electronica, Universidade de Aveiro, 2001.
- [Gai et al., 2001] P. Gai, L. Abeni, M. Giorgi, G. Buttazzo. *A new kernel approach for modular real-time systems development*. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*. 2001.
- [Holenderski et al., 2008] M. Holenderski, R. J. Bril, J. J. Lukkien. *Using fixed priority scheduling with deferred preemption to exploit fluctuating network bandwidth*. In *Proc. of the Work in Progress session of the Euromicro Conference on Real-Time Systems*. 2008.
- [Intel, 2004] Intel. *IA-PC HPET (High Precision Event Timers) Specification*, 2004.
- [Klein et al., 1993] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, M. G. Harbour. *A practitioner's handbook for real-time analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.
- [Labrosse, 1998] J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. CMP Books, 1998.
- [Liu, 2000] J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.
- [Park et al., 2001] M. Park, L. Sha, Y. Cho. *A practical approach to earliest deadline scheduling*. Tech. rep., School of Electrical Engineering and Computer Science, Seoul National University, 2001.
- [Rajkumar et al., 1998] R. Rajkumar, K. Juvva, A. Molano, S. Oikawa. *Resource kernels: A resource-centric approach to real-time and multimedia systems*. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*. 1998.
- [RELTEQ, 2009] RELTEQ. 2009. URL <http://www.student.tue.nl/P/w.a.cools/>.
- [RTLinux, 2007] RTLinux. *Rtlinux version 3.2*. 2007. URL <http://www.rtlinuxfree.com>.
- [SHaRK, 2006] SHaRK. *Soft hard real-time kernel*. 2006. URL <http://shark.sssup.it>.