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, r.j.bril, j.j.lukkien}@tue.nl

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 μC/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. Our extension of μC/OS II with periodic tasks and reservations [Holenderski et al., 2008] requires 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 all the real-time requirements while reducing the processor and memory overhead compared to existing solutions.

An event and its arrival time (or simply the event time) is denoted as $e_i$. For example, we can say that $e_i$ represents the arrival of a periodic task and that it occurs at time $e_i$. 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.

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

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, 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, at an arbitrary time. 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 resolution of the one-shot timer or the granularity of the periodic timer, depending on which kind of timer is driving the event queue. Usually a one-shot timer will offer a higher event resolution than a periodic timer.

Existing timed event management approaches trade off the real-time requirements, such as (ii) vs. (iii), or (ii) vs. (iv). The main contribution of this paper is the design and implementation of RELTEQ, an event queue based on relative event times, supporting long event interarrival time, long lifetime of the event queue, no drift and low overhead in terms of both processor and memory. We implemented RELTEQ in μC/OS II on top of a periodic timer.

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 a 32 bit variable with 1 millisecond resolution [Racciu and Mantegazza, 2006]. In this case, the system lifetime has a value of about a few months. An example of linear time model is illustrated in Figure 1.

The disadvantage of a linear time model, however, is that it imposes a finite lifetime: the system cannot represent times past $2^n$. 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.
Drift is defined in terms of the absolute difference $\delta$ between the intended event time and the actual time when the event occurs. A timed event management system is said to suffer from drift if $\delta$ is potentially unbounded (get reference from Diana *). For example, if we assume that the time period between two ticks is 1ms, but in reality it is 1.1ms, then the timer will drift.

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. ¹

The first two causes pose limitations on the hardware and on the application and are outside the scope of this paper.

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

¹Note that the clock granularity determines the resolution of timers, while the granularity of a periodic timer determines the resolution of events.
absolute or relative, setting the next timer expiration time may 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. Upon every tick a counter is incremented and compared with the event time of the first event (which is expressed in terms of ticks). If they are equal the first event is popped form the event queue and executed. There is a trade off between the granularity of the periodic timer and the performance overhead of the clock handler. A fine granularity of the periodic timer will allow a high resolution of the events, but will also come at a high performance overhead. In any case the system may not loose timer interrupts (e.g. due to disabled interrupts or interference from higher priority interrupts). Modern real-time platforms have a timer overrun counter which indicates the number of timer interrupts which have arrived but were not handled [Liu, 2000].

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 absolute number 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 drift 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 platform uses a periodic timer (since this is also the case in µC/OS II).

IV. RELTEQ

Rather than storing the event times relative to the current time, we can store 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.

When inserting a new event, the event queue has to be traversed, accumulating the relative times of the events until a later event $e_j$ is found, with $e_i < e_j$, where $e_i$ and $e_j$ are both absolute times.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.png}
\caption{Example of the RELTEQ event queue.}
\end{figure}

Unbounded lifetime

Insert "dummy" events to extend the maximum interval between events in the event queue. Figure 4. (* elaborate *)

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4.png}
\caption{Example of (a) an overflowing relative event time (b) RELTEQ inserting a dummy event to handle the overflow.}
\end{figure}

The first event

The arrival time of the first event is stored in absolute time. This has several benefits. First, the event queue is resistant to drift, in case the granularity of event times is not a multiple of the granularity of the tick, as described in Section III.

Second, in case of hierarchical scheduling [Behnam et al., 2008], an absolute arrival time of the first event allows to limit the event queue processing only to the events which can occur within the currently active server. For example, by associating a separate event queue with every server, the arrival of periodic events in other servers can be ignored by simply ignoring the corresponding event queues.

Handling overflows of the first event time

Since the event time of the first event is stored as an absolute time it may overflow. We address this issue in a similar way to prolonging the life time of relative events in the remainder of the queue: by inserting dummy events at times when the absolute time would overflow, as shown in Figure 5.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig5.png}
\caption{Example of (a) overflowing absolute time of the first event (b) RELTEQ inserting a dummy event to handle the overflow.}
\end{figure}

\footnote{Is this paragraph necessary? Move this paragraph to future work or omit completely to avoid introducing a new topic of hierarchical scheduling.}
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 μC/OS II real-time operating system. Similar to the standard μC/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 μC/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).

(Where are the dummy events stored? *)

Each event contains the following information:

- Link to TCB containing the next event for this type
- Link to TCB containing the previous event for this type
- (relative relteq) timestamp associated to this event

For every type of event, a Queue structure is defined, which contains the following information:

- Type of Event (for now only: Period Event, Delay Event)
- Pointer to 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 μC/OS II TCBs, we don’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 μC/OS II is now no longer used.

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 μC/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 (just like the other μC/OS II structures).

The set delay function of μC/OS II has also been changed: instead of setting the delay value in the TCB, it inserts a Delay Event in the RELTEQ Queue. Furthermore, the tick ISR of μC/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 [?].

V. Simulation results

Show the improvement by comparing standard μC/OS II with μC/OS II + RELTEQ, based on direct measurements of the timer handler latency (μC/OS II Probe?), or indirect measurements (compare two runs of the same task set utilizing the system completely, under μC/OS II and μC/OS II + RELTEQ).

(Elaborate *)

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).

VI. 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 μC/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 both processor and memory requirements compared to the existing approach.

VII. Future work

Avoid drift without HPET support

The RELTEQ approach is not limited to the current μC/OS II implementation. It is well suited for implementation on top of a one-shot timer for exploiting its higher resolution compared to a periodic timer. 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


