An Efficient Time Representation for Real-Time Embedded Systems [*]

Alessio Carlini
University of Pavia, Italy
carlini.alessio@libero.it

Giorgio C. Buttazzo
University of Pavia, Italy
buttazzo@unipv.it

ABSTRACT

Real-time systems should provide a time representation mechanism which allows to specify timing constraints in a wide range and with sufficiently high resolution. Moreover, the system lifetime (that is, the longest absolute time that can be handled by the system) should be as long as possible, or possibly infinite. In powerful architectures, this goal is achieved by representing time through variables with a large number of bits. Unfortunately, in real-time embedded systems with small memory requirements, such a solution cannot be used, and a trade off needs to be found for memory, system resolution and the longest representable timing constraint. In such systems, the runtime overhead introduced by the time representation mechanism is also crucial and needs to be taken into account.

In this paper we present an efficient method for time representation suited for small embedded systems. The method allows to achieve an infinite lifetime, high resolution and deal with sufficiently large timing constraints. The proposed approach is compared with other methods proposed in the literature and it is proved to be more efficient in terms of both overhead and memory requirements. The method allows an efficient implementation of deadline-based scheduling algorithms (such as EDF) and time accounting mechanisms for capacity based servers and resource reservation policies. The proposed technique has also been implemented and tested in two real-time kernels for small embedded microcontrollers.

[*]This paper appears in Proceedings of the ACM Symposium on Applied Computing (SAC 2003), track on Embedded Systems, Melbourne, Florida, USA, March 9-12, 2003.

1. INTRODUCTION

Embedded real-time systems have often to satisfy a set of stringent requirements in order to be installed on top of an integrated device. Beside satisfying functional and non functional application requirements, an embedded system must typically run on small processors with little processing power and small memory space. Hence, it must have a small memory footprint. To save memory space, however, typical kernels for embedded systems do not include sophisticated mechanisms for handling real-time constraints and use simple scheduling and resource management schemes. As a consequence of such a design choice, these kernels are very reactive to events, but achieve poor resource utilization and exhibit low predictability during peak load conditions, due to priority inversion phenomena [14] and other side effects.

For example, the MCX11 kernel [16] is a kernel totally written in assembly code for the Motorola 68HC11 microcontroller. It can manage up to 126 tasks and has a memory footprint of 3 Kbytes. Task scheduling is performed based on a fixed priority scheme and task communication may occur through shared memory buffers or message queues. The kernel does not provide a notion of global time, but only allows posting events with a time granularity of 50 milliseconds. SSX5 is a commercial real-time system produced by Realogy [17] for M68HC12 microcontrollers. This kernel provides fixed priority preemptive scheduling and it is based on the single-shot execution model, in which tasks run to completion when activated. They are preempted as appropriate, but always complete before returning control to lower priority task that they have preempted. Single shot execution allows the use of a single stack, which leads to significant reduction in RAM requirement. However, tasks cannot be created dynamically. The ROM and RAM costs for some SSX5 objects commonly used in application is 723 bytes ROM for code and static data and 83 bytes on RAM. Other OS functions can be included when the application uses them. MaRTE OS [13] is another real-time kernel for embedded applications, which allows greater flexibility. Most of its code is written in Ada with some C and assembler parts and it follows the Minimal Real-Time POSIX.13 subset. ERIKA [9] is a modular kernel for real-time systems. It consists of two main layers: the Kernel Layer and the Hardware Abstraction Layer. The Kernel Layer contains a set of modules that implement task management functions, real-time scheduling policies (as cyclic executive, Rate Monotonic, Earliest Deadline First, etc.) and resource management protocols (like Priority Inheritance and Stack Resource Policy), interrupt processing and error treatment. All these modules are developed using portable C code. The Hardware Abstraction Layer (HAL) contains some very basic services that are architecture-specific, like context-switch, interrupt handling, low-level timer and memory management initialization. For architectures with very small RAM memory, ERIKA provides a mono-stack task management policy, where all tasks use the same stack, but it also allows applications to use a multi-stack model. MICOS [4] is a microkernel developed for the Motorola 68HC11 micocontroller, which is also available for Intel X86 architectures. To facilitate portability on different platforms, the kernel consists of two main layers: the kernel layer and the hardware layer. MICOS is one of the few kernels which implements dynamic EDF-based scheduling for better resource utilization. It handles hard real-time tasks, soft real-time tasks, and non critical activities. The time management method adopted in the kernel is the one described in this paper. It was used to implement a resource reservation mechanism for providing temporal isolation among tasks [1], as well as good aperiodic responsiveness. The reservation mechanism allows the user to reserve a fraction of processor to each soft task, to guarantee a minimum level of performance. The efficiency of the time management also allows monitoring task execution times for detecting execution overruns and collecting statistical information on timing parameters. The kernel with all real-time features requires 8 Kbytes of memory.

In a real-time system, the system's lifetime is the maximum time the system can operate without causing a real-time clock overrun. It can be extended either by increasing the number of bits of all time variables or decreasing the timer resolution.

For example, with a 10 ms resolution, a linear time clock represented on 16 bits has a lifetime of about 11 minutes, which can be extended up to one hour using a 50 millisecond resolution. However, this is not suitable for most real-time applications. On a 32-bit time representation the lifetime becomes about 16 months, which could be fine. However, handling 32-bit timing constraints on a 8-bit microcontroller would cause a large overhead, since each 32-bit integer operation must be split into several 8-bit instructions, also considering the reduced number of registers available (3 for the M68HC11).

Another solution to achieve a large lifetime with a small number of bit is to decrease the resolution. For example, a lifetime of 1 month with 16-bit time representation can be achieved using a resolution of 1.6 seconds, which is unsuitable for most practical cases.

As a consequence, when working with 8-bit or 16-bit microcontrollers, a long lifetime can be achieved either using a 32-bit linear timer (with large overhead) or with a 16-bit circular timer, by handling the overflow (in this case the lifetime becomes infinite). It is worth observing that, when using a circular timer, the overflow has to be managed at every time comparison, and hence it must be efficiently handled. The proposed method (ICTOH) is able to perform such an overflow management with a very small overhead compared with the existing techniques [12, 7, 6].

In general, a time management mechanism should have the following characteristics.

1. Time resolution should be as high as possible in order for the kernel to provide high responsiveness to asynchronous events. High timer resolution also allows to increase processor utilization in the case task periods or deadlines are not multiple of the system tick. In this situation, in fact, to avoid missing deadlines, a period which is not multiple of the tick, should be reduced to the closest multiple. Hence, the higher the resolution, the smaller the utilization increase due to the period reduction.

2. The maximum time interval P handled by the system should be as long as possible in order to manage tasks with large periods or long relative deadlines (with respect to the system tick).

3. In embedded systems with stringent memory requirements, the system time should be represented using the minimum number of bits. Such a requirement is in contrast with the previous ones and imposes a trade off in the kernel.

4. The time management mechanism should not introduce a large runtime overhead.

In general, possible compromises depend on several factors, including the processor speed, the available memory, the efficiency of the kernel, the time horizon required by the applcation, the maximum extension of the relative timing constraints, the task criticality, and the number of tasks in the system.

This paper provides the following contributions:

1. It presents a simple time management technique (ICTOH) that can be efficiently implemented in many embedded real-time systems with limited amount of resources. The method allows to achieve an infinite lifetime, high resolution and deal with sufficiently large timing constraints. The problem of time management is typically neglected in real-time kernels for small processors, since tasks are scheduled using a set of fixed priorities, thus no absolute deadline management is required.

2. It allows an efficient implementation of deadline-based scheduling algorithms (such as EDF) and time accounting mechanisms for using capacity based servers and resource reservation policies.

3. It compares the proposed approach with other methods proposed in the literature, showing that ICTOH is more efficient in terms of both overhead and memory requirements.

4. It presents some concrete implementation issues and provides real experimental data from an EDF-based kernel. In particular, the method has been implemented in the MICOS microkernel [4] built for a Motorola 68HC11 microcontroller and in the ERIKA kernel [9].

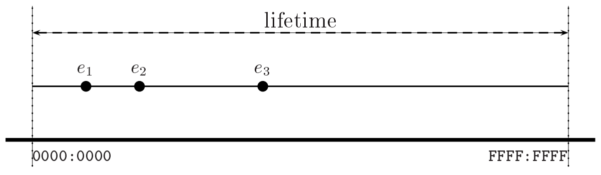

Figure 1: Linear time model with 32 bits.

The rest of the paper is organized as follows. Section 2 presents the problem by describing two intuitive approaches. Section 3 illustrates the proposed method and provides some implementation details. Section 4 shows the effectiveness of the proposed approach and compares it with other two methods described in the literature. Finally, Section 5 contains our conclusions and future work.

2. PROBLEM STATEMENT

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. In this case, the system lifetime (that is, the maximum absolute time that can be represented in the system) has a value of about a few months. An example of linear time model is illustrated in Figure 1. The main advantage of such a solution is that an event e_i precedes another event e_j if $e_i < e_j$. The disadvantage of a linear time model, however, is that it imposes a finite lifetime. Increasing the lifetime requires either using a larger number of bits or setting a lower time resolution. Both solutions can be inappropriate for an embedded system with stringent memory requirements and real-time constraints.

A reasonable compromise among the four requirements stated in the previous section is to use a circular time model. It differs from the linear one in that it handles the overflow condition occurring when the n-bit variable used to represent the system time passes from $2^n - 1$ to 0.

Figure 2 shows a circular time model, implemented using a 16-bit variable. In this model, each cycle has a length P = 10000H (exadecimal), hence two events with a time difference greater than or equal to P cannot be handled by the system without additional information. For example, Figure 2 shows two cycles in which four events are represented. Since events e_1 and e_2 have the same value ($e_1 = e_2$), they are considered simultaneous by the system, although they occur in two different cycles. Similarly, the time distance between e_1 and e_3 ($e_3 - e_1$) is considered the same as the one between e_1 and e_3' ($e_3' - e_1 = e_3 - e_1$).

3. THE ICTOH ALGORITHM

This section describes the Implicit Circular Timer's Overflow Handler (ICTOH) method which allows to efficiently represent times in a circular time model. We first introduce the following definitions.

Def. An event and its temporal reference on the circular timer is denoted as e_i. Thus, for example, we can say that e_i is a task activation and that $e_i = 04F3H$.

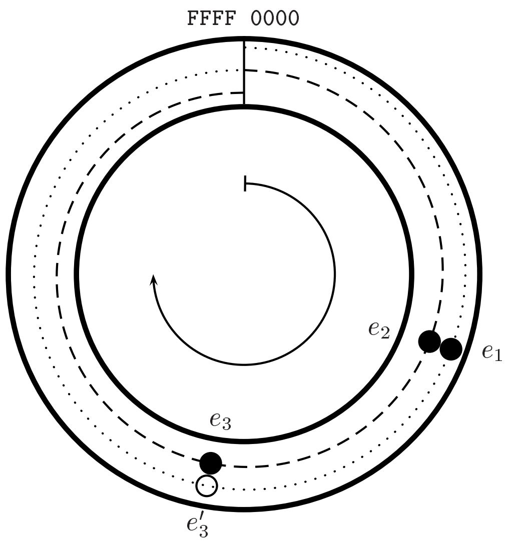

Figure 2: Circular time model with 16 bits.

Def. The set of temporal references stored in the system at time t is denoted as $E(t)$.

Def. The absolute time at which an event $e_i \in E(t)$ occurs is denoted as $t(e_i)$.

Def. The circular timer period is denoted as P. In other words, P is the minimum interval of time between two non simultaneous events characterized by the same representation in the system.

Def. We say that two events $e_i,\ e_j \in E(t)$ belong to the same cycle if the interval $[t(e_i),\ t(e_j)]$ represented on the circular timer does not include the values FFFFH and 0000H. For example, in Figure 2 e_1 and e_2 do not belong to the same cycle, whereas e_2 and e_3 do belong.

Then the ICTOH method can be defined as follows.

ICTOH: If events are represented by n-bit unsigned integers, such that

$$\forall t\ \forall e_i, e_j \in E(t)\ |t(e_i) - t(e_j)| < \frac{P}{2} \qquad (1)$$

then $\forall t\ \forall e_i, e_j \in E(t)$ we have:

1. $t(e_i) > t(e_j) \iff (e_i \ominus e_j) < \frac{P}{2}$, $(e_i \ominus e_j) \neq 0$
2. $t(e_i) < t(e_j) \iff (e_i \ominus e_j) > \frac{P}{2}$
3. $t(e_i) = t(e_j) \iff (e_i \ominus e_j) = 0$

where $\ominus$ denotes a subtraction (modulo 2^n) between n-bit integers, evaluated as an unsigned n-bit integer.

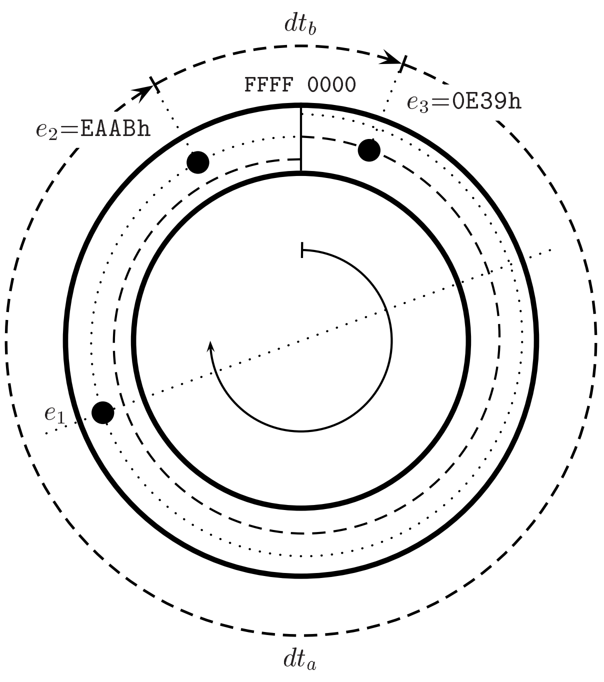

Figure 3: Example of events evaluated by ICTOH.

It is worth observing that for 8/16/32-bit integers such a subtraction operation does not require a special support since it is implemented in all CPUs. Figure 3 shows a set of events which satisfies condition (1).

Notice that condition (1) represents the price we have to pay for implementing a high resolution timer with an infinite lifetime. It means that the system can only handle tasks with timing constraints that cannot exceed the value of $P/2$ ticks.

So, for example, if the tick is set to 1 ms, and we use a 16 bit variable for storing the system time, then $P = 2^{16} = 65536$, meaning that the longest timing variable cannot be greater than 32.768 seconds. If the application includes a task with a greater period, we can increase the system tick, until a value equal to the least timing value in the system.

So, actually, $P/2$ represents the maximun ratio between the longest and the smallest timing parameter. If there is a task set where such a ratio is greater than $P/2$, then the ICTOH method cannot be used, and we are forced to pay a greater overhead for handling time variables with a higher number of bits.

The main property of the $\ominus$ operator is that

$$\forall a, b \in [0, 2^n - 1] \;\; unsigned(b \ominus a) = dist(a, b)$$

where

- $dist(x, y)$ is the distance from x to y evaluated on the time circle in the direction of increasing time values. Notice that $dist(x, y) = d$ means that if $t = x$ then, after a delay of d, we have $t = y$, independently from the fact that x and y belong to two different cycles.
- $unsigned(x)$ is the value of x, interpreted as an n-bit unsigned value. We recall that according to the 2's complement representation,

$$unsigned(x) = \begin{cases} x & \text{if } x \geq 0 \\ 2^n + x & \text{otherwise} \end{cases}$$

For example, when evaluating events e_2 and e_3 in Figure 3, we have that $dt_a = (e_2 - e_3) = DC72H > 8000H = P/2$. Hence, we conclude that e_2 must precede e_3 and that the actual time difference between the events is $dt_b = (e_3 - e_2) = 238EH < P/2$.

3.1 Extension

The constraint expressed by equation (1) can be relaxed if we consider disjoined sets of events.

Def. Two sets of events $E_i(t)$ and $E_j(t)$ are said to be disjoined if every element of the first set is never compared with an event of the second set.

Def. Let F be the set of all the disjoined sets in the system.

Then, we can formulate constraint (1) as follows:

$$\forall t \; \forall E_k(t) \in F(t) \; \forall e_i, e_j \in E_k(t) \; |t(e_i) - t(e_j)| < \frac{P}{2}. \quad (2)$$

In this way it is possible to manage temporal events which are spread in intervals greater than $P/2$, provided that events belonging to the same group are not separated by a time difference greater than $P/2$. Figure 4 illustrates two groups of events that satisfy constraint (2) and can be handled by the ICTOH method.

Task activation times and deadlines represent a typical example of two disjoint groups of events. In fact, although an absolute deadline is computed from task's activation time (by summing the corresponding relative deadline), the deadline event enters the system only after task activation, hence there is no need to compare the two events.

3.2 Implementation notes

Given a pair of events e_i end e_j represented through variables with 8, 16, or 32 bits, then by computing the difference $(e_i \ominus e_j)$ as a signed integer we can say that:

$$t(e_i) > t(e_j) \Longleftrightarrow (e_i - e_j) > 0$$
$$t(e_i) < t(e_j) \Longleftrightarrow (e_i - e_j) < 0$$
$$t(e_i) = t(e_j) \Longleftrightarrow (e_i - e_j) = 0$$

It is worth noting that such a result is valid only for variables represented on 8/16/32 bits, since only in this case all unsigned numbers with a value greater than or equal to

Language	**Linear timer implementation**	**ICTOH implementation**

C	**unsigned** e1,e2;
	
	if (e1 == e2)
	        / code for managing*
	        *simultaneous events */*
	else if (e1 > e2)
	        / code for managing*
	        *the case e1 > e2 */*
	else
	        / code for managing*
	        *the case e1 < e2 */*
		unsigned e1,e2;
		
		if (e1 == e2)
		        / code for managing*
		        *simultaneous events */*
		else if ((**signed**)(e1-e2) > 0)
		        / code for managing*
		        *the case e1 > e2 */*
		else
		        / code for managing*
		        *the case e1 < e2 */*

Assembler X86	_e1 **DW** _e2 **DW** **MOV** AX,_e1 **CMP** AX,_e2 **JZ** equal_e1_e2 **JG** greater_e1_e2 less_e1_e2: *;code for managing* *;the case e1 < e2* greater_e1_e2: *;code for managing* *;the case e1 > e2* equal_e1_e2: *;code for managing* *;the case e1 = e2*	_e1 **DW** _e2 **DW** **MOV** AX,_e1 **SUB** AX,_e2 **JZ** equal_e1_e2 **CMP** AX,0 **JG** greater_e1_e2 less_e1_e2: *;code for managing* *;the case e1 < e2* greater_e1_e2: *;code for managing* *;the case e1 > e2* equal_e1_e2: *;code for managing* *;the case e1 = e2*

Table 1: ICTOH vs. the simple linear timer: implementation issues.

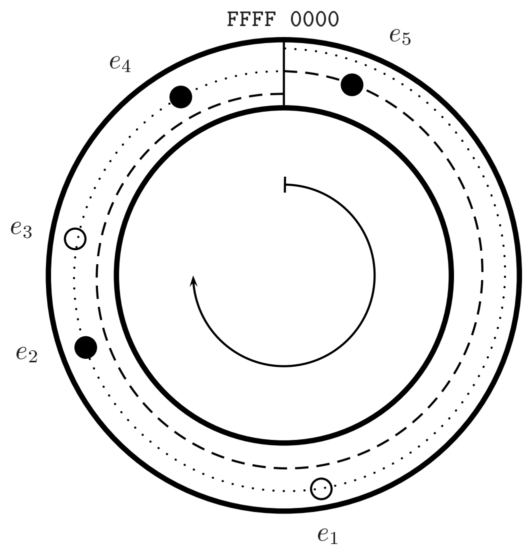

Figure 4: Two groups of disjoint events $\{e_1, e_3; e_2, e_4, e_5\}$ that can be handled by the ICTOH method.

$P/2$ (evaluated using the two's complement representation) are considered to be negative. For variables with a different number of bits, the test has to be performed as shown in the previous section, which however in several CPUs does not cause a larger overhead in terms of time and memory.

Table 1 compares the proposed ICTOH algorithm with the classical linear timer (without overflow management).

As clear from the assembler (X86) formulation, the difference between the two implementations is very small and the overhead introduced by ICTOH consists in just one assembly instruction. In fact, the execution time of a CMP instruction is similar to the one of a SUB instruction, and the additional instruction (CMP AX,0) required by ICTOH does not introduce a large overhead, being performed using a register and an immediate operand.

4. COMPARISON WITH OTHER METHODS

The implementation illustrated in Table 1 shows that the proposed method introduces a runtime overhead practically similar to the one introduced by the linear timer, which is the simplest possible implementation (which does not even handle the overflow). To handle the overflow, whenever an event exceeds the system lifetime, a routine must recompute all temporal events present in the system, by translating them of a fixed quantity. Such an approach is not efficient, especially in microcontrollers, since it introduces a large runtime overhead at every timer overflow.

In this section we compare the ICTOH approach with two other methods proposed in the literature. The first method was proposed by Park et al. in [12], while the second method was proposed by Fonseca in [7]. The same problem has also been addressed in [6], however the proposed solution is less efficient than those compared here.

4.1 Method 1

The method proposed by Park et al. in [12] has been devised to handle deadlines assuming they are never in the past ($d_i < t$) and that the longest relative deadline is less than the circular timer period ($D_i < P$). Hence, the method consists in handling those deadlines which exceed the current cycle (i.e., the one containing t).

The basic idea is to add the constraint that $\forall i\ D_i < P$. Moreover, an additional bit d_bit_i is associated to each temporal event to indicate whether it belongs or not to the current cycle. An additional bit t_{bit} is also used for the system time t as a base to identify the current cycle. Then, the method consists to invert the t_{bit} bit at each new timer cycle. Whenever a new deadline d_i falls in the current cycle the corresponding d_bit_i is inizialized at the value of t_{bit}, otherwise $d\_bit_i = \overline{t_{bit}}$. A comparison between two deadlines is performed on the additional bits first. If they are equal, the two values are compared as shown in Table 2.

```
unsigned    t,d1,d2;
char        flag,t_bit,d1_bit,d2_bit;

//t_bit management
if(t<0x8000 && flag==0){
        t_bit^ =1;
        flag=1;
}else if(t>=0x8000)
        flag=0;

//new deadline management
if(d1<t)
        d1_bit=t_bit^1;
else
        d1_bit=t_bit;

//Deadline comparison
if (d1==d2)
        // Code for managing the case d1 = d2
else if(d1_bit==d2_bit){
        if(d1>d2)
                // Code for managing the case d1 > d2
        else
                // Code for managing the case d1 < d2
}else if(d1_bit==t_bit){
        // Code for managing the case d1 < d2
}else
        // Code for managing the case d1 > d2
```

Table 2: Deadline comparison using Method 1.

It is worth noting that the additional bit used by this method introduces a large overhead due to its dynamic management. Moreover, storing this bit either increases the required memory or causes a larger runtime overhead. In fact, storing the bit in a dedicated byte is clearly a waste of memory. On the other hand, compressing those bits would increase the overhead for their manipulation.

4.2 Method 2

The method proposed by Fonseca in [7] also uses an additional bit, t_{bit}, but only for the t variable. Hence, such a bit can be stored in a dedicated byte without much problems. The major novelty of this method is to exploit the property of unsigned and signed integer numbers, which have two different discontinuity points: one from $(2^n - 1)$ to 0 and the other from $(2^{n-1} - 1)$ to (2^{n-1}).

By limiting the interval of the manageable events to $t \pm \frac{P}{4}$ and using the t_{bit} value, computed based on t, to specify which of the two representations has to be used, it is possible to manage time with a reduced overhead. The method is described in Table 3.

```
unsigned    e1,e2;
char        t_bit;

//Time management
t-= 0x4000;
if ((signed)t<0)
        t_bit=0;
else
        t_bit=1;

//Deadline comparison
if (e1==e2){
        // Code for managing the case e1 = e2
}else if(t_bit==0){
        if(e1>e2)
                // Code for managing the case e1 > e2
        else
                // Code for managing the case e1 < e2
}else {
        if((signed)e1>(signed)e2)
                // Code for managing the case e1 > e2
        else
                // Code for managing the case e1 < e2
}
```

Table 3: Event comparison with Method 2 [7].

This method is better than Method 1 because:

- It does not require an additional bit for each temporal event;
- Creating new events does not introduce overhead.

Nevertheless, this method is less efficient than ICTOH because it requires storing and managing the additional t_{bit}.

Method	Function	Code (Bytes)	Variables (Bytes)	T_{max} (μs)	T_{avg} (μs)
Method 1	Handling t_{bit}	28	2	20	11
	Handling e_i ($e_i = d_i$)	20	n	16	13
	Comparison	30	0	20	19
Method 2	Handling t_{bit}	18	1	13	12
	Handling e_i	0	0	0	0
	Comparison	28	0	16	15
ICTOH	Handling t_{bit}	0	0	0	0
	Handling e_i	0	0	0	0
	Comparison	15	0	11	9

Table 4: Comparison among the three methods on a MCU 68HC11 at 8Mz.

Moreover, when comparing two events, Method 2 is heavier than ICTOH, since it requires one more test (**if**(t_bit==0){...}).

Table 4, obtained by optimizing the code for the 68HC11 assembly language, summarizes the main features of the three methods presented in this paper in terms of code length, additional memory requirements, maximum and minimum execution time.

To better evaluate the difference among the three methods, we performed an experimental test on a kernel using an EDF scheduler on a Motorola 68HC11. The test has been run on 9 periodic tasks with periods equal to 15, 21, 43, 150, 186, 320, 521, 946, 1080 milliseconds respectively. The results of the test are reported in Table 5, which shows that ICTOH is more effective than the other methods in terms of both runtine overhead and memory requirements.

	Method 1	Method 2	ICTOH
Total execution time	18ms/s	12ms/s	5ms/s
Processor utilization	1.8%	1.2%	0.5%
Code size	270 Bytes	120 Bytes	30 Bytes

Table 5: Performance of the methods on a MCU 68HC11 system running 9 periodic tasks handled by EDF.

5. CONCLUSIONS

In this paper we presented an efficient method for representing time and compare timed events in small embedded processors with small memory requirements. The method allows to achieve an infinite lifetime and balance the system resolution with the longest relative timing constraint handled by the system. The method has been implemented on a kernel for a Motorola 68HC11 microntroller. For most control applications considered in our lab, the timer resolution has been set to 100 microseconds, giving a limit of about 3 seconds for the maximum relative deadline/period that can be specified on the tasks.

The performance of the proposed method has been evaluated and compared with two other approaches, resulting in a more efficient implementantion and a smaller overhead, comparable with the one of a linear timer model without overflow management.

As a future work, we plan to use such a method for implementing an execution time monitoring mechanism useful for developing capacity-based servers or performing statistical evaluations on timing events.

6. REFERENCES

[1] L. Abeni and G. Buttazzo, “Integrating Multimedia Applications in Hard Real-Time Systems”, *Proc. of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] L. Abeni and G. Buttazzo, “Support for Dynamic QoS in the HARTIK Kernel,” *Proceedings of the 7th IEEE Real-Time Computing Systems and Applications*, Cheju Island, South Korea, December 2000.

[3] G. Buttazzo, “HARTIK: A Real-Time Kernel for Robotics Applications”, *Proceedings of the 14th IEEE Real-Time Systems Symposium*, Raleigh-Durham, pp. 201–205, December 1993.

[4] Alessio Carlini, “A real-time kernel for embedded applications on a Motorola 68HC11 microcontroller”, Technical Report, Robotics Lab, University of Pavia, TR-2001-01, December 2001.

[5] M.L. Dertouzos, “Control Robotics: the Procedural Control of Physical Processes,” *Information Processing*, 74, North-Holland, Publishing Company, 1974.

[6] R. Elz and R. Bush, “Serial Number Arithmetic”, August 1996, Network Working Group, request for comments, 1982. URL: ftp://ftp.isi.edu/in-notes/rfc1982.txt.

[7] Pedro Fonseca, “Approximating linear time with finite count clocks”, Technical Report, Dep. de Electrónica, Universidade de Aveiro, Revista do DETUA vol.3,n.4,pp:359-361, ISSN-1645-0493, Sept.2001.

[8] P. Gai, L. Abeni, M. Giorgi and Giorgio Buttazzo, “A New Kernel Approach for Modular Real-Time systems Development”, *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, Delft, Netherlands, June 2001.

[9] Paolo Gai, “A Flexible and Configurable Real-Time Kernel for Time Predictability and Minimal Ram Requirements” Technical Report, Scuola Superiore S. Anna of Pisa, RETIS TR2001-02, March 2001. URL: http://gandalf.sssup.it/ pj/research.html.

[10] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli, “HARTIK 3.0: A Portable System for Developing Real-Time Applications,” *Proceedings of the IEEE Real-Time Computing Systems and Applications*, Taipei, Taiwan, October 1997.

[11] C.L. Liu and J.W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment,” *Journal of the ACM* 20(1), 1973, pp. 40–61.

[12] Moonju Park, Lui Sha, and Yookun Cho, “A Pratical Approach to Earliest Deadline Scheduling”, Technical Report, School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea, December 2001.

[13] M. A. Rivas and M. Gonzalez Harbour, “POSIX-Compatible Application-Defined Scheduling in MaRTE OS,” Proceedings of the Work-In-Progress session of the 13th Euromicro Conference on Real-Time Systems, Delft (NL), June 2001. URL: http://ctrpc17.ctr.unican.es/.

[14] L. Sha, L.R. Rajkumar, J.P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization,” *IEEE Transactions on Computers*, 39(9), 1990.

[15] M. Spuri and G.C. Buttazzo, “Scheduling Aperiodic Tasks in Dynamic Priority Systems,” *Real-Time Systems*, 10(2), 1996.

[16] The MCX11 kernel, URL: http://www.introl.com/introl-demo/demo/MCX11/contents.html.

[17] The SSX5 kernel, URL: http://www.realogy.com/.

Alessio Carlini’s biography

Alessio Carlini graduated in Computer Engineering from the University of Pavia in 2002. During his thesis, he designed and developed a real-time kernel for a Motorola 68HC11 microcontroller, aimed at running control applications for robotic systems. After his graduation, he worked as a research assistant in the Robotics Laboratory of the Computer Science Department of the University of Pavia, developing a vision-based obstacle avoidance system for an autonomous vehicle. Currently, he is working in Prisma Engineering, focusing on embedded systems and digital communication devices.

Giorgio Buttazzo’s biography

Giorgio C. Buttazzo graduated in Electronic Engineering from the University of Pisa in 1985, received a Master in Computer Science from the University of Pennsylvania in 1987, and a Ph.D. in Computer Engineering from the Scuola Superiore S. Anna of Pisa in 1991. He is an Associate Professor of Computer Engineering at the University of Pavia, Italy. His main research interests include real-time operating systems, dynamic scheduling algorithms, quality of service control, multimedia systems, advanced robotics applications, and neural networks.