T-CP/36/8/15067

Instruction Level Timing Mechanisms for Accurate Real-Time Task Scheduling

R. A. Volz
T. N. Mudge

Reprinted from: IEEE TRANSACTIONS ON COMPUTERS, Vol. C-36, No. 8, AUGUST 1987.

Correspondence

Instruction Level Timing Mechanisms for Accurate Real-Time Task Scheduling

RICHARD A. VOLZ AND TREVOR N. MUDGE

Abstract—**The scheduling of timed tasks is generally based, at the hardware level, upon the use of time intervals. For example, most microprocessor families provide their only hardware support for timing control in the form of a programmable interval timer chip accessible as an I/O device over the system bus. In this paper we will argue that a more natural and elegant solution bases timing on a local (to a particular CPU) absolute timer. Furthermore, we will show that the desired timing functions can be provided by simple extensions to existing CPU architectures. The widespread use of the "time interval" view has also influenced, in a negative way, the design of many programming languages. An important example is Ada, a language designed with real-time multitasking explicitly in mind. We will describe the difficulty with the current timing methods used in Ada, and present a method for overcoming the timing weakness by using the proposed timing mechanisms, while still remaining within the definition of the Ada language.**

Index Terms—**Ada, instruction set architecture, operating systems, programming languages, real time, scheduling, timing.**

I. Introduction

Real-time, multitasking processing requires that the activities of the processors be scheduled in accordance with both timing and external event requirements. Programming tools used for writing such applications should contain effective mechanisms for managing system resources to meet these kinds of requirements. Current time management tools have evolved from a separate development of hardware timers which may be added to the bus of a computer system and scheduling algorithms which use the timers. The consequence of this uncoordinated approach has been the development of timing systems which are largely interval based, inefficient, limited in resolution practically obtainable, difficult to use, and not readily extended to dealing with distributed systems. In this paper we argue that a unified approach to the development of software scheduling mechanisms and supporting hardware yields much more efficient, natural, and easy to use tools for time control. In particular, we suggest that timing control should be expressed in terms of absolute time at the language level and show that there exist simple extensions to CPU architectures which make the implementation particularly straightforward and efficient. The virtues of absolute time have been also discussed elsewhere [15], [16], but these discussions have not considered an instruction level approach.

The real-time performance of a system is highly dependent upon the performance of the scheduler, which in turn, is highly dependent upon the timing mechanisms available. Accordingly, real-time computer systems nearly always contain an interval timer, with either a fixed or programmable interval, and frequently, but not always, a time of day clock. If present, the time of day clock is usually of relatively low resolution, and of little value in scheduling tasks with a high repetition rate. Scheduling at the user level is typically accomplished by using some type of delay or wait function which puts the user process to sleep until a specified interval of time has elapsed [1], [2], [5]. The use of a fixed interval clock limits the practically achievable timer resolution because of the software overhead associated with each timer interrupt. This is a limiting factor for many real-time applications. A programmable interval clock can be used to reduce unessential software overhead, but care must be taken in managing it to avoid loss of clock ticks [3].

Most modern languages intended for concurrent and/or real-time applications either have mechanisms for timing control rooted in the use of time intervals or have none at all. Ada provides a "**delay** *interval*" construct to allow a process to delay its execution by the value of the variable *interval* [4, sect. 9.6]. Concurrent C and Concurrent Pascal utilize similar mechanisms [6], [7]. Modula-2 [8] provides no intrinsic timing mechanism at all. Real-Time Euclid [17] uses techniques of Leinbaugh [15] and Stoyenko [16] to achieve deadline scheduling of tasks.[1] Although interval based at the user level, it too would benefit from use of an absolute timer at the implementation level. Occam [9] appears to have a construct which references a desired (local) absolute time. It maintains an internal variable which represents time and can apparently be used to delay until *after* a desired time. The syntax for using the AFTER feature does not appear very stable, however; several different versions have appeared in Occam documentation over the past three years. Moreover, Occam does not maintain a global sense of time, and has a time resolution too limited for many serious real-time applications. Instruction level support for Occam's AFTER construct is available in the Transputer [9], which appears to be the first computer to provide direct support for an absolute sense of time at the hardware level, but details on its implementation are not readily available. Since process scheduling is done by the hardware, it is very difficult to make use of this timer to implement other kinds of scheduling systems.

Aside from being nonoptimal for real-time scheduling activities, interval based timing causes additional complications for distributed systems which must maintain time synchronism [10]. The adoption of an absolute sense of time at each node in the system simplifies user level synchronization problems by reducing them to simply maintaining synchronism among the individual system clocks.

The next section describes the difficulties with current timing mechanisms. Section III then presents a set of underlying support primitives based upon absolute timing which allow efficient implementation and illustrates their use in a simple scheduling algorithm. Section IV describes the use of the new constructs to simplify and improve the efficiency of timing control in Ada.

II. Current Modes of Operation and Their Limitations

The problems of concern are 1) scheduling a set of tasks $T_1, \cdots, T_n$, so that they are made ready at the times $t_1, \cdots, t_n$, where the times $t_1, \cdots, t_n$ are presumed to be known, 2) scheduling a task to be made ready after some interval of time has elapsed. Algorithms for selecting the times $t_1, \cdots, t_n$ or the interval are covered elsewhere, e.g., [11], and are not of concern here. What is of concern are the interactions between the timer and the scheduler.

In most existing implementations, the timing hardware is designed without regard for the types of scheduling algorithms best suited for

Manuscript received December 11, 1986. This work was supported by General Dynamics under Contract DEY-601540.

The authors are with The Robotics Research Laboratory, Department of Electrical and Computer Science, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 8715067.

[1] Deadline scheduling of tasks means specifying the time (or interval) by which a task must complete.

real-time operation and the schedulers must be written to accommodate existing hardware, with a resulting limitation in performance. The most common present mode of operation is to use a fixed interval timer which periodically interrupts the processor and invokes the scheduler. The scheduler maintains a list of scheduled tasks, the times at which they are to be made ready, and a software clock. Each time an interrupt occurs, the software clock is updated and the list of scheduled tasks is checked to see if any task on it should be made ready. The overhead associated with updating the clock and checking the task list after each interrupt places a lower bound on the clock interval which may be used, as this overhead must be incurred during each interval, regardless of whether or not there is a task to be scheduled.

The use of a programmable count down timer makes an alternative scheduling discipline possible. Such a timer always counts down at some basic rate (10–100 kHz are typical). Whenever zero is reached, an interrupt to the processor is generated and the scheduler invoked. Thus, overhead is incurred only when a scheduling operation is actually required, and, therefore, it is not necessary to choose a minimum interrupt rate on the basis of the fraction of processor time taken up by time management. This type of scheduling is preferred for real-time operations. In the case of the count down timer, however, the count in the timer must be updated by the scheduler to the interval required before the next task is to be made ready, and, as described below, this scheme is prone to errors which can result in a "drift" of the time kept by the system.[2]

Many vendors offer programmable timers that operate as described above which may be added as a device on the bus of a computer configuration and used for scheduling operations (e.g., the Intel 8254 programmable interval timer [12]). A few processors, e.g., the Intel 8096, even offer on-chip timers. All of these, however, suffer one major deficiency: they allow a cumulative loss of time under some circumstances. If the timer receives a new value by a store operation, the time between the occurrence of the previous interrupt and the store operation is lost. This is of no consequence if no pulse from the underlying clock generator arrived at the timer during the store. However, if such a pulse did arrive during or before the store, its effect will be wiped out by the store of a new value in the timer, resulting in a slow drift of the time maintained on the system (see Fig. 1). The likelihood of such loss of time is accentuated by the presence of other external devices which might interrupt the processor or cycle steal from the processor during time management.

One solution to this problem is to *add* the new interval to the timer rather than store to it. That is, if *interval* is the value of the interval timer, and *new_interval* is the next delay interval, it is necessary to achieve

$$interval \leftarrow interval + new\_interval;$$

rather than *interval* ← *new_interval*, to avoid a drift in time. However, the add operation must also be performed without losing account of any clock pulses that may occur during the add. If it were implemented in software, there would be the possibility of the add blocking the pulses, particularly if interrupts or DMA occurred simultaneously. Unfortunately, existing commercial timers do not support the desired "add to timer" function in hardware, and one can only approximate the desired behavior by making the timer management function run at the highest priority possible. Then, however, clock pulses and nonmaskable interrupts can still intercede and lead to cumulative time loss. A programmable count down timer with an "add to timer" function has been built in a real-time computing systems laboratory at the University of Michigan and incorporated into the scheduler in a real-time operating system [3]. In that system, the timer was a device on the Q-bus of an LSI-11/1 system. Cyclic timing intervals of 1 ms were realized with the system, intervals which were low for that generation of hardware.

[2] Throughout this paper we shall use the word "drift" to mean the cumulative loss of time due to missed clock pulses. Elsewhere it is often used to describe the variations in the behavior of the oscillator producing the clock pulses. This latter phenomenon is a function of the physical environment and cannot be entirely avoided.

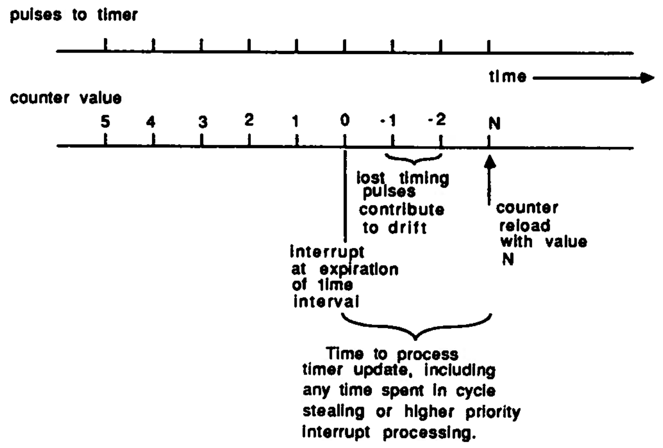
pulses to timer
time
counter value
5 4 3 2 1 0 -1 -2 N
lost timing pulses contribute to drift
counter reload with value N
Interrupt at expiration of time interval
Time to process timer update, including any time spent in cycle stealing or higher priority interrupt processing.

Fig. 1. Drift during a store to timer operation.

While the timing problems can be solved via an "add to timer" function or the use of a second register in conjunction with a programmable timer, as Digital Equipment Corporation and IBM do in the VAX and RT [2], [5], we believe the best solution to the problem of scheduling events in time is based upon a common sense of "absolute time." This is the approach that we shall present in this paper. Although it is not essential to the approach, the paper describes the placing of the necessary timing functions directly on the CPU chip. While perhaps infeasible a few years ago, most current microprocessor CPU chips can easily accommodate both the small amount of extra logic on-chip and the few additional instruction codes. These can be used in an upward compatible fashion to provide improved single chip real-time control processors.

III. Basic Timing Functions for Real-Time Task Scheduling

A. Programmable Absolute Timer

There are three basic questions to be answered in defining the new programmable absolute timer capability.

- How many bits should it have?
- Where should it be placed architecturally?
- How should it function?

The number of bits is a function of timer resolution and the maximum interval length to be measured. A 32-bit timer with a 10 μs resolution allows a maximum period of approximately a half a day. This is marginal for many applications. Since the number of bits is not critical to the remainder of the discussion, we assume a timer with 64-bits (this corresponds to a range of about 1/2 million years at a 1 μs resolution).

One of the key points to the proposed timer is that it be placed in the CPU. Proper timing operation requires several complex operations to be performed atomically, that is they must be performed *in toto* or not at all. This can be accomplished if the timer is part of the CPU, but is difficult to guarantee if the timer is only accessed through the bus. The need for atomic operation will become clearer as the necessary timer functions are described and illustrated below.

The proposed scheme requires the addition of two CPU registers, as shown below. The *todc* register holds the current time and is incremented at a fixed rate. The *cpr* register is

63 0

timer or time of day clock (*todc*)

63 0

compare register (*cpr*)

used to hold the time of the next event to be scheduled. The arrival of the event is checked for by continuously comparing the contents of *cpr* with that of *todc*. At every clock tick, the following operations are performed:

```
todc←todc + 1;
if todc ≥ cpr then
        cpr←1 ... 1;
        generate timer interrupt;
end if;
```

The comparison is atomic and is performed by hardware; clearly it should be performed within a clock tick. A convenient choice for the clock tick would be a multiple of the basic CPU clock. For example, in current 32-bit microprocessors a 2–4 phase clock would be appropriate and yield a tick period on the order of 1 μs. This degree of timer resolution is much finer than is currently typical. Just prior to generating a timer interrupt, *cpr* is loaded with its maximum value (all ones). This prevents subsequent meaningless interrupts from occurring before new event times are loaded into *cpr*. The timer interrupt is generated on every tick when *cpr* ≤ *todc*, even if *cpr* is, for whatever reason, loaded with a value less than the present time of day.

Given the above two registers and the continuous checking logic, four scheduling primitives can be defined that are sufficient for the implementation of a number of scheduling algorithms:

- set timer
- set compare register
- conditional set compare register
- read timer.

We next describe the operation of these instructions and then illustrate their use in a simple scheduling algorithm.

1) Set Timer: stodc(new_todc, A, B)

```
c(A, B)←todc;
todc←new_todc;
```

where *new_todc* is a 64-bit register pair or memory location, and *A* and *B* are a pair of 32-bit registers (we will assume for the purposes of this discussion that we are dealing with a 32-bit machine). In addition to providing a way to initially set the value of the clock, this instruction can be used to synchronize two or more loosely coupled CPU's. The value returned in the *A, B* register pair can be used for adjusting stored time values after *todc* has been changed to bring it into synchrony with an external clock. This operation should be atomic, i.e., no interrupts or DMA activity should intercede during the operation. Furthermore, *stodc* should be a privileged instruction.

2) Set Compare Register: scpr(new_compare_value, A, B)

```
c(A, B)←cpr;
cpr←new_compare_value;
```

where *A* and *B* are two 32-bit registers used to save the old value of *cpr* and *new_compare_value* is a 64-bit quantity from CPU registers or memory. It is not essential that *scpr(A, B)* be atomic for the purposes of this paper, although other considerations are likely to make it desirable. If interrupts, DMA or other events result in *new_compare_value* being less than *todc* when *scpr* is executed, then a timer interrupt will be generated on the next clock tick. However, since the timing is based on "absolute time" this local anomaly will not contribute to a cumulative bias as it can in similar situations when the underlying timing mechanisms are based on interval timing. If initialization is performed correctly, the accuracy rests solely with the ability to minimize the variations of the oscillator that increments *todc*. The contents of *todc* are never modified except when it is set initially or reset to effect synchronization with other processors or some absolute time base.

3) Conditional Set of Compare Register: cscpr(new_compare_value, flag, A, B)

```
flag←0;
if new_compare_value<cpr then
    c(A, B)←cpr;
    cpr←new_compare_value;
    flag←1;
end if;
```

This operation simply loads the compare register with a new event time if that time is earlier than the one presently in *cpr*. The 1-bit register *flag* is set to 1 if the exchange is made, and the old value is saved in a register pair. Typically, *flag* will be one of the CPU status bits. The instruction *cscpr* should be atomic. If the new value set into *cpr* is less than *todc*, that is, one has attempted to schedule something to occur prior to the present time, an interrupt will occur immediately after the next clock tick and the item to be scheduled can be handled at the nearest possible point in time to that originally desired. Again, because the scheme relies only on absolute time, there will be no permanent drift.

4) Read Timer: rtodc(save_timer_value)

```
save_timer_value←todc;
```

This operation saves a copy of the timer in *save_timer_value*, a 64-bit pair of CPU registers or memory locations.

B. Simple Scheduling Model

We can implement a task scheduler in several hierarchical levels. The bottom level makes use of the machine instructions and maintains the set of time scheduled tasks. The upper levels provide the interface to the user language, convert scheduling requests to the form required by the low level scheduler, and maintain the sets of ready and blocked (for I/O or synchronization) tasks. The low level scheduler is quite simple and can be used by a variety of high level scheduling algorithms. We consider first the low level scheduler.

Consider a set of tasks to be scheduled at various points in time. Order the tasks by the time at which they are to begin. For illustration purposes, we show the result as a linked list of task control blocks (TCB's) in Fig. 2. Each TCB contains, among other things, a pointer, *next*, to the TCB of the next task in the scheduling sequence, and a value, *scheduled_time*, specifying when the task is to begin. The variable *HEAD* points to the head of the list. Suppose that *cpr* has already been loaded with *HEAD.scheduled_time*, the time at which task T_1 is to begin. If *TCB_ptr* is a pointer to the TCB of a new task, that task can be added to the list of Fig. 2 as follows,

```
TCB_ptr.scheduled_time←calculate scheduled time;
TS(TCB_ptr);
```

where TS is the procedure

```
procedure TS(T:task_ptr) is
    interrupts off;
    cscpr(T.scheduled_time, flag, A, B);
    insert T in schedule list;
    interrupts on;
end TS;
```

The first two lines represent the actions of the high level scheduler. The procedure TS implements the low level scheduler. The action of the *cscpr* instruction permits the new task to be placed at the head of the queue of tasks to be processed if its *scheduled_time* is earlier than that of T_1. Of course, the *scheduled_time* for T_1 is not lost in such cases, and it will be reloaded into *cpr* at the appropriate time, as we will see. The insertion of *T* into the list can be done in the usual way by following the chain of TCB's until *T.scheduled_time* ≤ *next.scheduled_time*.

The actions performed by the scheduler upon the occurrence of a timer interrupt are equally simple, but do involve a simple link to the higher level scheduler. In particular, we suppose that the high level

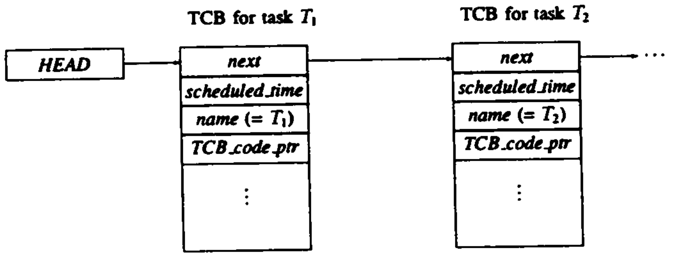

Fig. 2. List of TCB's.

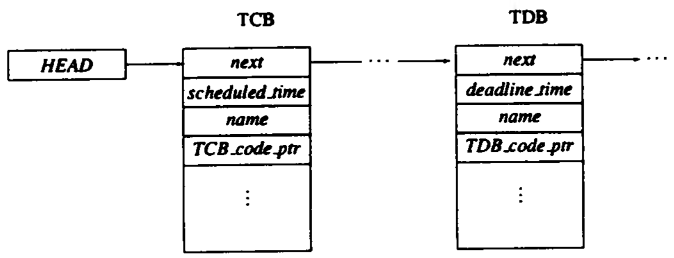

Fig. 3. Lists of TDB's and TCB's.

scheduler maintains a *ready* list of tasks from which it will select the task to be actually given the CPU resources. If the variable *NOW* is used to retain the *cpr* time which caused the timer interrupt, then the scheduler need only perform the following.

```
TIMER_INT_HND:
  interrupts off;
  save registers;
  NOW←HEAD.scheduled_time;
  repeat
    add task pointed to by HEAD to set of ready tasks;
    HEAD←HEAD.next;
  until NOW ≠ HEAD.scheduled_time;
  scpr(HEAD.scheduled_time, A, B);
  restore registers;
  interrupts on;
```

The **repeat**...**until** is executed at least once and moves all the tasks that were scheduled to run at the time given in *cpr* (which caused the interrupt) to the set of ready tasks. These tasks all have the same *scheduled_time*. The very next task in the list of TCB's is now at the head of the queue in Fig. 2 and has its *scheduled_time* placed in the compare register to await its start time. In this simple example, the registers which return the old value of *cpr* were not used because all of the needed information was held in the TCB's. However, in more complex scheduling methods, a need for them may arise.

To illustrate how this basic low level time scheduler can be embedded in a more sophisticated scheduler, consider one possible mechanization of a system to positively verify that tasks have completed within a specified deadline, and to raise an exception if they have not. In this case, each task that is to be scheduled must have two values of time associated with it (provided by a higher level scheduling algorithm or command—see the next section for an illustration): the time at which the task is to be made ready, and the time by which it is to be completed. To manage this, we introduce a task done block (TDB), which plays a symmetric role with respect to the TCB. The TDB is set by the higher level control algorithms to contain the deadline by which the task must be terminated, and contains a pointer to whatever exception handling code is to be used upon failure to complete in time. The use of the low level scheduler then proceeds as follows.

```
TCB_ptr.scheduled_time←calculate scheduled time;
TDB_ptr.deadline_time←calculate deadline;
TS(TCB_ptr);
TS(TDB_ptr);
```

This results in a scheduled list of the form shown in Fig. 3 with TCB's and TDB's interspersed (not necessarily alternating). The TDB's are queued in the same way as the task control blocks. If a task's deadline is reached, the TDB will invoke the exception code (which may be defined by either the system or the user) in the same manner as invoking another task. If a task is completed before its deadline, then it must invoke a higher level scheduling routine which will remove the TDB from the scheduling queue, in which case the TDB will never invoke the exception code. Obviously, in order to avoid the possibility of a false indication of deadline passage due to the system overhead in processing the completion call from the task, the exception code must begin by verifying that the task did, in fact, not finish.

One can actually overlay several scheduling concepts on the underlying time scheduling mechanism. In order to implement priority scheduling, one need only have the higher level scheduler maintain several priority queues, one for each priority level. The only change required in the low level scheduler is in *TIMER_INT_HND*. This routine must now look at the priority of the task (presumed to be in the TCB) and add the task to the correct queue.

Of course, the higher level scheduler may manipulate the ready queues without going through the low level timer scheduler. That is, during I/O or synchronization waits, it may remove a task from active status and place it on a *suspended* queue until the task is able to proceed, and then place it back on the ready queue. And tasks that do not require time scheduling may be placed in the ready queue by the high level scheduler without involving the low level time scheduler at all.

IV. Language Level Issues

The pervasiveness of the "time interval" view has influenced the design of programming languages in a way that has introduced unnatural complexity into the scheduling operations. An important example which is indicative of several other languages as well is Ada, a language designed with real-time multitasking explicitly in mind. Ada provides a predefined data type, DURATION; objects of this type represent time intervals. A language construct, **delay**, provides a delay for at least the length of time given by an argument of type DURATION. In addition, a predefined package, CALENDAR, is specified which provides a data type TIME, and a set of mathematical operations for dealing with TIMEs and DURATIONs. For example, it provides a function " – " which subtracts two TIMEs and yields a DURATION.

To illustrate the influence of interval timing, we present an Ada version of a commonly used timing loop for repetitive (at a fixed interval) operations and note that even though the language has interval-based timing mechanisms, the user must still maintain an absolute sense of time (at least locally). We consider a simple control loop, that, for instance, might be controlling the motion of a robot arm, which must be executed repetitively with period 0.01 s. Denote the control action by the subprogram *F*. Then using the Ada language syntax, the loop may be expressed as [13]

```
with CALENDAR;
declare
  use CALENDAR;
  INTERVAL : constant DURATION := 0.01;
  NEXT_TIME : TIME := FIRST_TIME;
begin
  loop
    delay NEXT_TIME – CLOCK;
    F;
    NEXT_TIME := NEXT_TIME + INTERVAL;
  end loop;
end;
```

where the package CALENDAR provides the data types DURATION and TIME, and the functions CLOCK (for returning the current value of TIME), " – " and " + " for operating on values of these types.

There are several observations to be made about this example. First, even though the example itself is intrinsically interval-based and the language provides interval-based timing, it is necessary for the user to implement an absolute (at least local to this problem) sense of time. This is necessary because of the unknown length of time required for the execution of F (indeed, it may not even be constant). Without the maintenance of an absolute sense of time, there could be a long-term drift in the timing of the loop which could be harmful to the operation being performed.

The syntax of the language reflects the bias toward an underlying interval timer. The example illustrates the inefficiency of this. One must first convert from a time interval specification to an absolute time specification and then back again. Furthermore, arithmetic involving data objects of type TIME is not necessarily efficient. On one compiler tested, the times required for " + " and " – " are on the order of 200 μs. Even worse, CLOCK function measurements on five Ada compilers showed times ranging from 94 to 3400 μs [14]. Furthermore, the underlying scheduling operations may suffer from one of the kinds of ills described in Section II.

The times required for timing operations are highly dependent upon the representation of the data types. The underlying use of an interval timer in conjunction with the requirement to provide functions returning the MONTH, DAY, and YEAR of the TIME make a record representation appealing. It is this underlying record implementation which leads to the large execution times.

The use of an underlying absolute timer would allow and encourage improvements in several ways. First, it would encourage an underlying representation of TIME objects as extended fixed point numbers, with conversion being performed for MONTH, DAY, and YEAR as necessary. Then the TIME arithmetic and CLOCK functions could be performed in microseconds rather than hundreds of microseconds. Time could easily be kept in Greenwich Mean Time allowing compatibility across time zones; the time zone would then be simply a setup parameter. Second, it would allow a more natural language expression of the timing function. Using the package facility, a new package could be defined, ABS_TIME, that supercedes the capabilities of CALENDAR and that is based on absolute time. Among other things it could provide 1) a new procedure DELAY_UNTIL(*T*:TIME), which would delay until the timer reaches the time passed in as a parameter; 2) a new type DURATION with 64-bit precision. These would typically be used in place of the current **delay** statement in Ada.

Then the above control loop may be expressed as

```
with ABS_TIME;
declare
  use ABS_TIME;
  INTERVAL: constant ABS_TIME.DURATION := 0.01;
  NEXT_TIME: TIME := FIRST_TIME;
begin
  loop
    DELAY_UNTIL(NEXT_TIME);
    F;
    NEXT_TIME := NEXT_TIME + INTERVAL;
  end loop;
end;
```

Several points are worth noting. First, no reading of the clock is necessary. Second, no subtraction of time is necessary. Both of these contribute to being able to execute much faster loops. Also, the expression of the timed loop is more natural and easier to understand.

There are, of course, a few other language constructs that have been proposed for specifying task timing information. CRASH [3] and Real-Time Euclid [17] both allow periodic execution of tasks to be specified. CRASH allows one task to schedule another with an "EVERY *interval* DO *task*" statement, while Euclid has a "PERIODIC FRAME *interval* FIRST ACTIVATION ATTIME *time*" statement as part of a process declaration. CRASH also has a "DO *task* AT *time*" statement. The "at time" parts of these map straightforwardly into the underlying implementation described above. The "interval" parts can be handled by having the scheduler perform actions similar to the timed loop described above. That is, if the interval is held in a variable *INT* in the TCB of each task, *TIMER_INT_HND* can be modified so that after adding the task pointed to by *HEAD* to the ready queue, it adds *INT* to *HEAD.scheduled_time* and reschedules the task. If deadline checks were being made, it would also add *INT* to *deadline_time* in the corresponding TDB block and reschedule it. Alternatively, at the completion of a task, the task will return control to the scheduler, and the scheduler could, as part of its actions, add *INT* to the previous timers and reschedule the task (and its TDB). In any event, the mapping of the more complex scheduling operations onto the proposed instruction level mechanism is straightforward and easier to understand than interval-based techniques because there is no need to worry about either doing certain critical sections of scheduler code within fixed time intervals or the myriad of things that can interfere with this.

V. Summary and Conclusions

The management of time is critical in real-time embedded systems. We have shown that basing time on a (local) absolute timer is more natural, leads to simpler implementations, and is easier to use. The timing registers and primitives introduced can be easily implemented on a CPU chip or on a timer board that can be attached to the system bus. We have illustrated both simple time scheduling using the proposed instructions and new high level language functions to allow more efficient and natural expression of timed loops.

The use of (local) absolute timers also simplifies the maintenance of synchronism across a set of machines and isolates the real problem, that of providing correct synchronized values of time to each of the processors in the system. Although beyond the scope of this paper, there are a number of mechanisms possible for solving this latter problem, e.g., radio and satellite broadcasts of digitally encoded time and geographical position to allow for transmission time compensation [10].

References

[1] Intel Corp., *iRMX-86 Reference Manuals*, Santa Clara, CA, 1980.

[2] Digital Equipment Corp., *VAX 11/780 Hardware Handbook*, Maynard, MA, 1978.

[3] R. A. Volz, "The CRASH—Compiler for real-time applications shop—Manual," Dep. Elec. Comput. Eng., Univ. Michigan, Ann Arbor, 1978.

[4] *Ada Programming Language (ANSI/MIL-STD-1815A)*, Washington, DC, 20301: Ada Joint Program Office, Dep. Defense, OUSD(R&D), Jan. 1983.

[5] G. Erwin, *IBM RT PC AIX Operating System Technical Reference*, Preliminary Edition, Document SV21-8009-1, Austin, TX, June 1986.

[6] N. H. Gehani and W. D. Roome, *Concurrent C*, AT&T Bell Labs. Rep., Murray Hill, NJ, 1985.

[7] P. B. Hansen, *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1977.

[8] N. Wirth, *Programming in Modula-2*. Berlin, Germany: Springer-Verlag, 1982.

[9] D. Pountain, *A Tutorial Introduction to Occam Programming*, INMOS Corp., P.O. Box 16000, Colorado Springs, CO 80935, 1985.

[10] R. A. Volz and T. N. Mudge, "Timing issues in the distributed execution of Ada programs," *IEEE Trans. Comput.*, vol. C-36, no. 4, pp. 449–459, April 1987.

[11] K. Ramamritham and J. A. Stankovic, "Dynamic task scheduling in distributed hard real-time systems," *IEEE Software*, vol. 1, July 1984.

[12] Intel Corp., *Microsystems Components Handbook*, Santa Clara, CA, 1984.

[13] J. G. P. Barnes, *Programming in Ada*, 2nd ed. London, England: Addison-Wesley, 1984.

[14] R. M. Clapp, L. J. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultze, "Toward real-time performance benchmarks for Ada," *Commun. Ass. Comput. Mach.*, vol. 29, pp. 760–778, Aug. 1986.

[15] D. W. Leinbaugh, "Guaranteed response times in hard-real-time environment," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 85–91, Jan. 1980.

[16] A. D. Stoyenko, "Real-time systems: Scheduling and structure," Dep. Comput. Sci., Univ. Toronto, Toronto, Canada, Dec. 1984.
[17] E. Kligerman and A. Stoyenko, "Real-time Eulcid: A language for reliable real-time systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 941–949, Sept. 1986.