XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa.
Buenos Aires, Setiembre 1988.

Procesamiento de eventos para sistemas de tiempo real

Wolfgang A. Halang

Bayer AG
Ingenieurbereich Prozessleittechnik
D-5090 Leverkusen, Bayerwerk, Alemania Federal

y

University of Illinois at Urbana-Champaign
Computer Systems Group, Coordinated Science Laboratory
1101 West Springfield Avenue, Urbana, IL 61801, EE.UU.

Resumen

En este articulo se describe una arquitectura extendida de computadores adecuada para mejorar el desempeño de sistemas concatenados que se encuentran en medios de tiempo real. Esta aumenta la confiabilidad y eficiencia explotando las posibilidades para procesamiento en paralelo inherentes a los sistemas de tiempo real. Se muestra que las transmisiónes internas de datos debidas a las operaciónes de conmutación contextual pueden ser eliminadas y la unidad central de procesos liberada de una cantidad considerable de trabajo rutinario proveyendo un modulo separado para funciónes mayores de sistemas operativos de tiempo real; esto es para reconocimiento de señales e interrupciónes, para administración del tiempo y las tareas, la transferencia de tareas entre diferentes estados y la inicialización del sistema. Considerando el modelo "en capas" de los sistemas operativos modernos de tiempo real estas funciónes constituyen el nucleo y la primera capa de un sistema operativo. Se desarolla una teoria matematica de la distribución de las tareas y sus condiciónes, bajo las cuales estas deben ser realizadas. Se describen las funciónes del procesador de eventos detallando un numero de algoritmos ejecutandose como reacciónes a los eventos ocurridos. Sus complejidades son esencialmente proporcionales al numero de tareas en observación. Esta implementación "hardware" de soporte tipico de tiempo real provee una separación fisica clara de las funciónes intrinsecamente independientes de reconocimiento de eventos y procesamiento de las tareas del procesamiento general. A la vez de reducir la carga y minimizar los tiempos de respuesta y reacción en general, la arquitectura hace posible gratizar una cota superior, predefinida, para el tiempo de reacción a eventos externos e internos.

1. Introduction

The purpose of this paper is to describe an extended computer architecture suitable of improving the performance of embedded systems encountered in hard real-time environments. With the latter term industrial, scientific, and military areas of application are meant, which are characterised by strict time conditions, that must not be violated under any circumstances. In contrast to this, commercial systems, e.g. for automatic banking and airline reservations, only need to fulfill soft real-time requirements, i.e. although they are designed with the objective of fast response time, the user may expect the completion of his transactions.

Real-time data processing systems are expected to recognise and to react to occurring events as soon as possible, that means instantaneously in the ideal case. Therefore, the response time is the most important measure for the performance of real-time systems. With presently available hardware a reaction can only be accomplished by interrupting the running task, determining the source of the event, and switching to a response program. Note that the running task is pre-empted although it is most likely independent on the just arriving interrupt. Furthermore, another task will not necessarily be executed before the current one, after the interrupt has been identified and acknowledged. Owing to this inherent independence, the possibility to apply parallel processing is given here. In order to preserve data integrity, in the conventional architecture the operating system and user tasks may prohibit to be interrupted during the execution of critical regions. Hence, there is a considerable delay between the occurrence of an event and its recognition, and an upper bound for it cannot be guaranteed [7]. This situation is further impaired when several events occur at (almost) the same time resulting in the mutual interruption of their service tasks and postponement of the low priority reactions.

In a conventional real-time computer, every interrupt causes a considerable overhead. This is very unproductive, since the context contained in the register sets needs to be saved and later reloaded. The majority of the interrupts is generated by the interval timer, typically 1000 times per second in order to realise system clocks with a one millisecond resolution. The clock interrupt handling routine updates its time and date variables and checks - mostly unsuccessfully - whether any time-scheduled activities have become due. It is clear that thus a considerable amount of a computer's available processing time is wasted. Furthermore, this kind of timing is not accurate, because the hardware timers have a low resolution, and due to the unpredictable operating system overhead that may supersede the timer routines.

In the course of the progressive increase of semiconductor devices' integration density, and the development of microprocessors, the cost relations have overturned; now, data transmission components within a computer system are more expensive than the processing and storage elements.

The advances in semiconductor technology, however, left the classical von Neumann architecture essentially unaffected. Hence, it appears to be necessary to consider structural changes, in order to minimise, or at least to reduce, the internal transmission expenses that have become the determinative factor of the hardware prices by exploiting possibilities for parallel processing inherent to real-time systems. Screening the pertinent literature, it can be concluded, that no attempt has been made yet to utilise hardware and firmware facilities for the implementation of typical hard real-time support features that cannot otherwise be realised. In this paper, therefore, it is shown that internal data transmissions due to context-switching operations can be eliminated and the CPU relieved from a considerable amount of routine work by providing a separate module for the task of event recognition and processing. By the advances in VLSI technology, this concept has now become realisable. Moreover, such a migration of functions is feasible, as was shown in [8-11], since it leaves invariant the execution sequence of a task set imposed by a given synchronisation scheme, and has further advantageous effects with regard to deadline driven task scheduling and virtual memory management [4].

2. Outline of the Concept for an Event Processor

A computer working in batch or time-sharing mode receives jobs for processing which it does not know in advance. They are to be executed as soon as possible. In contrast to this, the tasks of a process control computer are available in and known to the system. It has to continuously check the conditions for the activation of the various tasks, which are dependent on the actual time, external and internal events, and internal states. These task scheduling functions are assigned to a separate, independently working module, because they impose a heavy load on the CPU and because they are logically independent from general task execution. The resulting asymmetrical double processor configuration is depicted in Figure 1.

The execution of tasking operations is generally linked to conditional expressions, called schedules, which are dependent on the time and on binary quantities. The latter we call events. The module allows the usage of very general scheduling conditions, the most complex ones of which are temporal schedules whose initial times depend upon interrupts. In the next section, we shall present a mathematical theory of these schedules and of the conditions, under which they are fulfilled. The schedules are realised as Boolean procedures. It is the purpose of the event processor to observe the events and to evaluate the schedules. Time schedules are implemented as function procedures yielding, for every invocation, the next critical point in time, for which a tasking operation is planned. The time values obtained from all schedules in the system can be ascendingly ordered and the respective minimum is loaded into an "alarm clock".

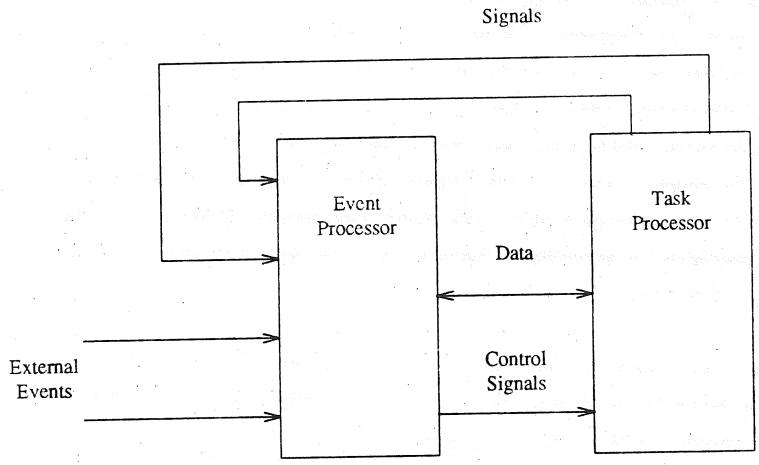

Figure 1

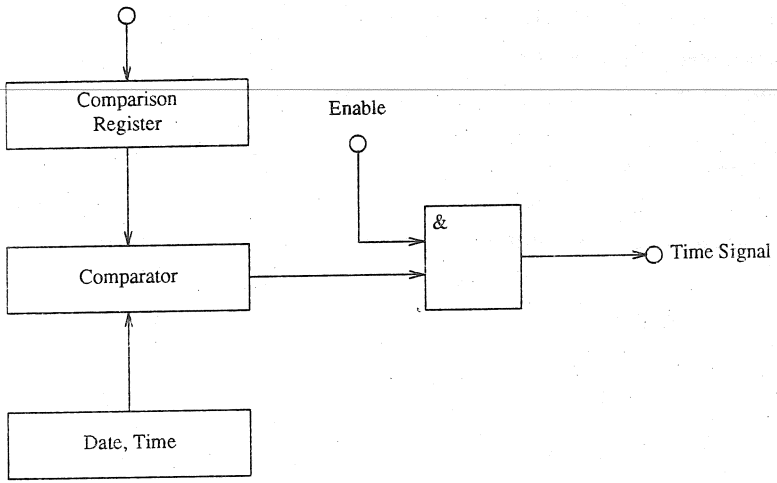

Figure 2

The event processor's time management is supported by an elaborate hardware module, which provides accurate time readings and eliminates all superfluous servicings of the clock. Such a timer was first described in [3] and later again in [12]; for full details the reader is referred to [5]. A schematic diagram with its main components is given in Figure 2. The clock is realised in form of of a divided, individually addressable counter yielding day, hour, minute, second, and an appropriate subdivision thereof. Opposite to the counter stands a correspondingly structured register. The outputs of both units are connected to a comparator generating a time signal upon equality. To keep the number of time events to be processed as low as possible, a signal is only raised when a moment is reached for which a certain action is scheduled.

The servicing of the various events is performed in the following way. In order to guarantee a predefined, short upper bound for the reaction time, the recognition of the events is carried through by continuous cyclic interrogation of the corresponding signal lines. This method is similar to the one employed in Programmable Logic Control (PLC) devices and was already used in the method of synchronous programming for the timely recognition of external events. The cyclic interrogation process has to be carried through with a high frequency, if the recognition time is to be kept short. In the course of the polling process, the event arrival times are saved for future reference. The occurrence of a time signal requires more service than that of other events. So, time schedules need to be handled by checking if they have been exhausted or by calculating the next critical moments. After determining the minimum of all these critical points in time, it is loaded into the clock's comparison register. Upon completion of each interrogation cycle, information is passed over to the operating system residing in the general processor. These data specify the set of schedules actually being fulfilled and the activities associated with them, which are now to be carried through.

Only fast event recognition and closely connected operations are carried out. Thus, the complexity requirements for the event processor are quite moderate. Since it also does not need to be freely programmable, it is realised as a rather simple, fully microprogrammed device, guaranteeing a high speed of operation. In the next but one section, the functions of the event processor will be described by detailing a number of algorithms running as reactions to occurred events. Their complexities are mainly linearly proportional to the number of observed schedules.

There are many possibilities to realise the data exchange between the system components. In order to prevent excessive overhead connected with writer/reader synchronisations, it appears most feasible to send operation parameters via first-in-first-out memories from the general processor to the event processor. Generally, the leading idea for selecting a certain means of data exchange with the event processor must be to achieve high speed combined with simplicity in order not to produce new bottlenecks.

3. Formal Description of the Tasks of the Event Processor

It has been shown in [2] that four states are sufficient to characterise the actual situation of the processing of a task. At any time every task is in exactly one of the states, which are defined as follows:

known: the task is known to the system, its scheduling condition, however, is presently not fulfilled;

ready: the task waits for the allocation of the resources required for its execution;

running: the task is being executed; and

suspended: the execution of the task was interrupted for synchronisation or user specified purposes.

The event processor manages the known tasks according to this model and carries out the transition to the state *ready*, when the corresponding condition are fulfilled. The other states and state transfers are organised by a different unit, viz. the dispatcher, whose description is outside of the scope of this paper. Furthermore, the event processor supervises the schedules of all other tasking operations.

In order to describe the event processor's duties, we consider the time instant t_0, at which the $m(t_0)\geq 0$ tasking operations T_i, i=1,...,m, may be known. The latter are combined into the set T. The elements E_j of the set of events E:={ E_j| j=1,...,n}, whose cardinality $n\geq 0$ may not depend on the time, are conjunctions of the actual events E'_j and possible maskings $M_k^{(j)}$:

$$E_j=E'_j\cdot\prod_{k=1}^{l_j}M_k^{(j)},\ l_j\geq 0,\ j=1,...,n.$$

With F:T→S the set T of tasking operations is surjectively mapped onto the set S:={S_k(t,E)| k=1,...,l(t_0)}, $l(t_0)\geq 0$, of scheduling conditions. Between the events and these run conditions for the tasks there exists the relation

R_1:={(E_j,S_k)| j=1,...,n, k=1,...,l, S_k depends explicitly on E_j}$\subset E\times S$.

Each of the explicitly time dependent schedules in S implies a sequence of instants, at which the corresponding tasking operations may have to be executed. We join all these instants into a countable set Z:={$t_1<t_2<\cdots$}$\subset[t_0,\infty)$ having a strictly monotonous increasing order. There exists a further relation between this set Z and the set of scheduling conditions

R_2:={(t_i,S_k)| i=1,...,|Z|, k=1,...,l, S_k depends explicitly on t and implies t_i}$\subset Z\times S$.

In order to reduce the event processor's processing expense, we assume without loss of generality,

that if it holds $(E_j,S_k)\in R_1$, j∈{1,...,n} and k∈{1,...,l}, than only E_j, however not $\bar{E}_j$, is present in the expression for S_k. This does not represent a restriction, because E_j can be replaced by the event $\bar{E}_j$ or both E_j and $\bar{E}_j$ can be elements of E, respectively. Hence, the Boolean function S_k, k∈{1,...,l}, can change its value from 0 to L at the instant $t\in[t_0,\infty)$ if and only if

1. $(t,S_k)\in R_2$ holds, or
2. at least one E_j, j∈{1,...,n}, with $(E_j,S_k)\in R_1$ has changed its state from 0 to L.

Therefore, it is the responsibility of the event processor to evaluate all scheduling conditions $S_k\in S$, $1\le k\le l$, for which holds

1. $(t,S_k)\in R_2$, if there is a $t_i\in Z$ with $t=t_i$, i∈{1,...,|Z|}, or

2. $(E_j,S_k)\in R_1$, if the event E_j, $1\le j\le n$, has assumed the state L at instant t.

Finally, let

$S^{(t)}:=\{S_k\in S \mid S_k(t-0)=0 \text{ and } S_k(t+0)=L,\ k=1,...,l\}$

and $t\in[t_0,\infty)$, then the tasking operations contained in $\dot{F}^{-1}(S^{(t)})$ must be executed. In order to perform the mentioned processing steps, the task scheduler needs to have the scheduling conditions available in the form of Boolean procedures. If the schedule S_k is explicitly dependent upon time, then additionally a function procedure s_k is required with the property

$$s_k(t)=\begin{cases}\min\{Z_k\cap(t,\infty)\}, & \text{for } Z_k\cap(t,\infty)\neq\emptyset\\ \text{"exhausted indicator"}, & \text{otherwise}\end{cases}$$

where Z_k designates the subset of Z which is implied by S_k. Based on this construction, it is not necessary to have all elements of Z and the entire relation R_2 available at any time. Instead, it is sufficient to evaluate at the instant t_0 all procedures s_k, to join all results into the set Z', to arrange them in increasing order, and to store the following subset of R_2:

$R'_2:=\{(t_i,S_k)\in R_2 \mid t_i\in Z',\ i=1,...,|Z|,\ k=1,...,l\}$

The smallest element of Z' is loaded into the comparison register of the "alarm clock" (cp. Figure 2), which generates a signal when this instant is reached. In contrast to the usage of a relative timer, no program run-times need to be taken into account when this method is applied. Besides the activities

already mentioned above, the event processor has to carry out the following operations, when the time signal arrives at the instant $t_i \in Z$, $i \in \{1,...,|Z|\}$:

1. Remove the smallest element from the set Z': Z':=Z'-{t_i};

2. Evaluate all procedures s_k with $(t_i, S_k) \in R_2$, $1 \leq k \leq l$;

3. Join the instants obtained in step 2 with Z' and sort the resulting set Z';

4. Check if the schedules S_k associated with the exhausted time schedules s_k cannot assume the value L any more; if this is the case, remove all corresponding elements from R_1;

5. Form the up-to-date subset R'_2 of R_2;

6. Load the comparison register of the alarm clock with the minimum of Z'.

4. Task Scheduling Algorithms

After having compiled in the preceding section the tasks of the event processor and having described its essential data structures, we want to detail now the module's function by specifying a number of algorithms. All these routines are tasks themselves, which are executed by the unit and activated by external signals.

Besides the already defined sets, in the following the set of time procedures s:={$s_k(t)$| k=1,...,l} will appear as operand. The mapping zp:S→s is given by the association of a procedure s_k with each schedule $S_k \in S$. The alarm clock's comparison register is designated by *vr*. A set is being sorted in ascending order by submitting it as actual argument to the procedure *sort*. The minimum of a set is provided by the function min. The signal *eof* indicates, whether there is presently no more control information to the read by the event processor. The parameters of tasking operations to be carried through are transmitted to the dispatcher, which has the address *dp*. All other items appearing in the following programs are auxiliary variables; in particular, t stands for the actual time.

A reset signal initialises the event processor and its data structures:

on reset **do**
T:=F:=S:=s:=zp:=Z':=R_1:=R_2:=∅ , enable:=**false** , vr:=∞
od

The event processor commences its activities with loading the alarm clock after occurrence of a start signal:

```
on start do
 if Z' ≠∅ then vr:=min(Z') ; enable:=true
od
```

At any time after a reset signal, the parameters of tasking operations and scheduling conditions can be transmitted to the event processor, which is requested to accept the corresponding data by a load signal. The following program module inserts the parameters read in into the unit's data structures. If new scheduling conditions for already known tasking operations are specified, then the old schedules will be overwritten. The procedure *delete* which erases these conditions will be defined further below.

```
on load do
 while not eof do
  get(TN,SN,sn,EN) ;
  if TN∈T then Σ:=F(TN) ; F:=F-{(TN,Σ)} ;
   if F⁻¹(Σ)=∅ then delete(Σ) fi fi ;
  T:=T∪{TN} , S:=S∪{SN} , s:=s∪{sn} , F:=F∪{(TN,SN)} ,
  zp:=zp∪{(SN,sn)} , R1:=R1∪{(e,SN)| e∈EN} ;
  if sn≠ nil then
   tn:=sn(t) ;
   if tn="exhausted" then
    if not [tz>t=>SN(tz,E)] then delete(SN) fi
   else Z':=Z'∪{tn} , R'2:=R'2∪{(tn,SN)} ; sort(Z') ;
    if tn=min(Z') then vr:=tn fi
   fi
  fi
 od
od
```

After a prevent signal the event processor removes scheduled tasking operations from his lists; hence, it performs the operation inverse to load. If there is no further tasking operation associated with the schedule of the one to be prevented, then also the schedule will be erased by calling the procedure *delete* .

```
on prevent do
 while not eof do
  get(TN) ;
  T:=T-{TN} , SN:=F(TN) ; F:=F-{(TN,SN)} ;
  if F⁻¹(SN)=∅ then delete(SN) fi ;
 od
od
```

The following procedure *delete* removes a scheduling condition and all information related to it from the event processor's data sets and, if need be, loads a new value into the comparison register of the clock.

```
proc(proc bool) delete=(proc bool Σ):
[σ:=zp(Σ) ; S:=S-{Σ} , s:=s-{σ} , zp:=zp-{(Σ,σ)} ,
 for all e∈E with (e,Σ)∈R1 do R1:=R1-{(e,Σ)} od ;
 M:=F⁻¹(Σ) ; T:=T-M ,
 for all m∈M do F:=F-{(m,Σ)} od ;
 if σ≠nil then
  for all tz∈Z' with (tz,Σ)∈R'2 do
   R'2:=R'2-{(tz,Σ)} ;
   if R'2∩{(tz,SN)| SN∈S}=∅ then Z':=Z'-{tz} fi
  od
  vr:=if Z'≠∅ then min(Z') else ∞ fi
 fi]
```

Now, we turn to the actual tasks of the event processor, viz. the evaluation of scheduling conditions when events occur. Upon every 0-to-L-transition of an $e \in E$ the following program is executed:

```
on e=0→L-transition do
 for all SN∈S with (e,SN)∈R1 do check (SN) od
od
```

whereby all schedules are checked which are in relation to the event e. If a scheduling condition assumes the logical value L, the associated tasking operations are transmitted to the dispatcher for execution:

```
proc(proc(real,[1:n]bool)bool) check=
      (proc(real t,[1:n]bool E)bool Σ):
⌈if Σ(t,E) then
 for all TN∈T with (TN,Σ)∈F do put(dp,TN) od
 fi⌋
```

When the time signal of the alarm clock occurs, all schedules are checked as outlined above, which are related by R'_2 to the actual minimum of Z'. The smallest element from Z' and the corresponding tupels of R'_2 are removed. The time schedules associated with the checked task scheduling conditions are evaluated and the set Z' and R'_2 are updated with the resulting instants. Schedules corresponding to exhausted time schedules are deleted from the event processor's lists, if they cannot assume the value L any more. Finally, the clock's comparison register is loaded with the new minimum of Z'.

```
on time_signal do
 tmin:=min(Z') ; Z':=Z'-{tmin} ;
 for all SN∈S with (tmin,SN)∈R'2 do
  check(SN) , tn:=zp(SN)(t) , R'2:=R'2-{(tmin,SN)} ;
  if tn="exhausted" then
   if not [tz>t=>SN(tz,E)] then delete(SN) fi
  else Z':=Z'∪{tn} , R'2:=R'2∪{(tn,SN)}
  fi
 od ;
 sort(Z') ; vr:=if Z'≠∅ then min(Z') else ∞ fi
od
```

On the premises that the number of tasking operations associated with a single schedule - which is normally 1 - is small, the complexity of the program handling the occurrence of an event in E is proportional to the cardinality of S. Let N(|Z'|) operations be necessary to sort Z'. Then the program responding to a time signal requires the execution of O(|s|+N(|Z'|)) instructions. The prevent routine's complexity is linearly dependent on the number of input data, the one of the load routine is proportional to the product of N(|Z'|) and the number of the received data sets.

The signals can be realised as leading edge triggered flip-flops. As already mentioned, the sets S and s consist of procedures, that have to be made available to the event processor. The elements of T contain an identification, parameters for the dispatcher and for the later program execution. The set Z' and the mappings F and zp can be represented by one-dimensional arrays, and the three relations R_1, R'_2 and F^{-1} as inverted files or as two-dimensional bit arrays. In the latter, case a number of the above mentioned statements can be realised by bit chain operations.

5. Evaluation

For the evaluation of the described computer architecture some qualitative considerations are sufficient, since a quantitative evaluation based on analytic modeling was already carried out by Tempelmeier [8,10,11].

First of all, it has to be stressed that the evaluation criteria for real-time embedded systems are quite different from those ones used with respect to batch or time-sharing computer systems. For real-time systems the throughput and the CPU utilisation are less important. What the user expects, is the reliable and predictable fulfillment of his requirements. Also, the cost of a process control computer has to be seen in a larger context. Naturally, the cost of a two-processor system is higher than that of a conventional von Neumann computer. Since the latter cannot guarantee reaction times, it may be unable to cope appropriately with exceptional and emergency situations of the external technical process. In comparison to the costs of a damage of such a process, which is caused by a computer's malfunction, the price for an event processor will be almost negligible.

The problem of prolonged reaction times caused by long phases, during which the operating system disables the recognition of interrupts, was pointed out by Schrott [7]. This measure is usually applied to avoid interrupt cascades while executing elementary operating system functions and to synchronise the access to basic operating system lists. Our architecture solves this problem by distributing the intrinsically independent functions of event recognition and task administration and execution to separate units.

In the early days of real-time data processing the fundamental requirements of timeliness and simultaneousness [6] were realised by the user himself. He employed the method of synchronous programming to schedule, within his application software, the execution of the various tasks. To this end, he usually wrote his own organisation program, a "cyclic executive". Thus, a predictable software behaviour could be realised and the observation of the time conditions could be guaranteed. Later, this method was replaced by the more flexible approach of asynchronous programming, which is based on the concept of the task. Tasks can be activated and run at any time, asynchronously to a basic cycle. The flexibility and conceptual elegance of the method was gained by renouncing predictability and guaranteed time conditions. The event processor has been designed to solve these problems of asynchronous programming.

The most important measure for the performance of real-time systems is the response time. It depends not only on the overall computing speed of a system, but also on the software organisation and here especially of the operating system, since its routines are executed together with the user tasks in an interleaved manner. Thus, the overhead becomes part of the task response times. Also the

interrupt reaction times depend on a system's hardware and software characteristics. As mentioned above, owing to the internal organisation of an operating system and the necessary functions to be performed, e.g. context-switching, there may be a considerable delay of unpredictable length before a conventional system can acknowledge a received interrupt. Both stated problems have been attacked with the here described approach. They could be solved by the observation of the inherent independence between running user tasks or operating system functions on one hand, and operating system routines or external requests on the other. Based on their independence, the mentioned activities were assigned to different devices and can be carried out in parallel, which reduces both task response and interrupt reaction times.

Conclusion

The clear physical separation of the intrinsically independent functions event recognition and processing from general task processing yields a number of improvements as against conventional structures, especially with respect to enhancement of performance. Based on their independence, the mentioned activities can be carried out in parallel, which reduces both task response and interrupt reaction times. By providing a special device for the handling of all events, unnecessary context-switchings are avoided and the normal program flow is only interrupted when required by the scheduling algorithm. Nevertheless, the event servicing tasks will be processed under observation of their due dates, but in a way disturbing the currently active tasks as little as possible. Besides being reduced, the operating system overhead becomes predictable and an upper bound independent of the actual workload for the time required to react upon events can be guaranteed. In general, the transfer of the event processing functions to specialised hardware contributes to enhancing reliability and efficiency essential in real-time applications. Finally, the event processor provides and relies on a more accurate timing facility than conventional process control computers.

To summarise, we have presented here a special module for major functions of real-time operating systems, viz. for interrupt and signal recognition, for the time management, the administration of task schedules, the transfer of tasks between different states, and the initial start-up of the whole system. Considering the layer model of contemporary real-time operating systems [1], these functions constitute the kernel and the first layer of an operating system.

References

[1] Baumann, R. et al., 'Funktionelle Beschreibung von Prozessrechner-Betriebssystemen'. Richtlinie VDI/VDE 3554. Berlin-Cologne: Beuth-Verlag 1982

[2] Eichenauer, B.F., 'Dynamische Prioritaetsvergabe an Tasks in Prozessrechensystemen", PhD Thesis, Universitaet Stuttgart, 1975

[3] Halang, W.A., 'Ansaetze zu funktionsorientierten Prozessrechnerstrukturen', PhD Thesis, Universitaet Dortmund, 1980

[4] Halang, W.A., 'Implications on Suitable Multiprocessor Structures and Virtual Storage Management when Applying a Feasible Scheduling Algorithm in Hard Real-Time Environments', SOFTWARE - Practice and Experience 16 (8), 761 - 769, 1986

[5] Halang, W.A., 'Architectural Support for High-Level Real-Time Languages', Proceedings International Conference on Software Engineering for Real-Time Systems, Cirencester, 28 - 30 September 1987

[6] Lauber, R.J., 'Prozessautomatisierung I'. Berlin-Heidelberg-New York: Springer-Verlag 1976

[7] Schrott, G., 'Ein Zuteilungsmodell fuer Multiprozessor-Echtzeitsysteme', PhD Thesis, Technical University Munich, 1986

[8] Tempelmeier, T., 'Antwortzeitverhalten eines Echtzeit-Rechensystems bei Auslagerung des Betriebssystemkerns auf einen eigenen Prozessor', PhD Thesis, Technical University Munich, 1980

[9] Tempelmeier, T., 'Auslagerung eines Echtzeitbetriebssystems auf einen eigenen Prozessor', Proceedings Fachtagung 'Prozessrechner 1981', pp. 196-205, 1981

[10] Tempelmeier, T., 'Antwortzeitverhalten eines Echtzeit-Rechensystems bei Auslagerung des Betriebssystemkerns auf einen eigenen Prozessor', Teil 2: Messergebnisse, Report TUM-I8201, Technical University Munich, 1982

[11] Tempelmeier, T., 'Operating System Processors in Real-Time Systems - Performance Analysis and Measurement', Computer Performance, Vol. 5, no. 2, pp. 121-127, 1984

[12] Volz, R.A., and Mudge, T.N., 'Instruction Level Mechanisms for Accurate Real-Time Task Scheduling', Proceedings of the 1986 IEEE Real-Time Systems Symposium, New Orleans, 2 - 4 December 1986, pp. 209-215, 1986