June 1988

UILU-ENG-88-2228
CSG-87

COORDINATED SCIENCE LABORATORY
College of Engineering

NAG 1-613
IN-62-CR
149520
P-78

DEFINITION OF AN AUXILIARY PROCESSOR DEDICATED TO REAL-TIME OPERATING SYSTEM KERNELS

Wolfgang A. Halang

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) (CSG-87) UILU-ENG-88-2228	5. MONITORING ORGANIZATION REPORT NUMBER(S) None
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL *(If applicable)* N/A	7a. NAME OF MONITORING ORGANIZATION NASA
6c. ADDRESS *(City, State, and ZIP Code)* 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS *(City, State, and ZIP Code)* Washington, DC 20546
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NASA 8b. OFFICE SYMBOL *(If applicable)* N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA Grant Number NAG-1-613
8c. ADDRESS *(City, State, and ZIP Code)* Washington, DC 20546	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Definition of an Auxiliary Processor Dedicated to Real-Time Operating System Kernels

12. PERSONAL AUTHOR(S) Halang, Wolfgang A.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ____ TO ____	1988 June	74

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
real-time, multiprocessor architecture, auxiliary processor, operating systems

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

In order to increase the efficiency of process control data processing, it is necessary to enhance the productivity of real-time high-level languages and to automate the task administration, because presently 60% or more of the applications are still programmed in assembly languages. This may be achieved by migrating apt functions for the support of process control oriented languages into the hardware, i.e., by new architectures. Whereas numerous high-level language machines have already been defined or realised, there are no investigations yet on hardware assisted implementations of real-time features.

This research commences with summarising the requirements to be fulfilled by languages and operating systems in hard real-time environments. A comparison of the most prominent languages, viz. Ada, HAL/S, LTR, Pearl, as well as the real-time extensions of Fortran and PL/1, reveals how existing languages meet these demands and which features still need to be incorporated

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)* / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

REAL TIME OPERATION
KERNEL FUNCTIONS
PERIPHERAL EQUIP. (COMP.)
MULTIPROCESSING "
HIGH LEVEL LANGUAGES

ARCHITECTURE (COMP.)
COMPUTER SYSTEMS PERFORM
" " DESIGN

ORIGINAL PAGE IS OF POOR QUALITY

to enable the development of reliable software with predictable program behavior, thus making possible to carry out a technical safety approval.

With reservations, it can be stated that Pearl represents the closest match to the mentioned requirements. Therefore, extensions of this language are proposed to express the time behaviour, the surveillance of events and problem oriented synchronisation. Furthermore, statements are introduced to control the desirable operating system services and software verification features.

Taking the objectives of feasible processor scheduling, inherent deadlock prevention, minimisation of context-switching operations and guaranteed reaction times as a basis, then it is investigated which multiprocessor structures yield the best performance. Single processor systems that cooperate with devices specialised in carrying through operating system nuclei turn out to have advantages over classical von Neumann and symmetrical multiprocessor structures. A further result of these considerations is that nearly optimal look-ahead algorithms for the virtual storage administration, employing the code of entire tasks as paging element, are closely related to deadline driven scheduling.

An auxiliary processor dedicated to real-time operating system nuclei is defined. First, the design of the unit is outlined, and its concept is discussed in relation to comparable ones found in the literature. The services to be provided by the device are compiled and assigned to three different reaction levels. The basic level consists of special hardware features for timing and event recognition. These are driven by the primary reaction level, which handles the occurring events and manages the time schedules. It finally communicates the activities to be executed to the secondary reaction level, which controls the whole system and supports the application programs. The three reaction levels are constructively describec by detailing their functional units, internal data structures, and control algorithms.

Finally, the proposed architecture is qualitatively evaluated, especiall in comparison with the conventional one. Its feasibility is verified by showing, that the extended Pearl can be implemented on the considered architecture. In this context the special compiler activities and run-time features to be incorporated into the application programs are of particular interest.

Definition of an Auxiliary Processor Dedicated to Real-Time Operating System Kernels

Wolfgang A. Halang
Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, IL 61801, USA

Contents

This research was supported, in part, by NASA under NASA Grant Number NAG-1-613.

0. Abstract

In order to increase the efficiency of process control data processing, it is necessary to enhance the productivity of real-time high-level languages and to automate the task administration, because presently 60% or more of the applications are still programmed in assembly languages. This may be achieved by migrating apt functions for the support of process control oriented languages into the hardware, i.e. by new architectures. Whereas numerous high-level language machines have already been defined or realised, there are no investigations yet on hardware assisted implementations of real-time features.

This research commences with summarising the requirements to be fulfilled by languages and operating systems in hard real-time environments. A comparison of the most prominent languages, viz. Ada, HAL/S, LTR, Pearl, as well as the real-time extensions of Fortran and PL/1, reveals how existing languages meet these demands and which features still need to be incorporated to enable the development of reliable software with predictable program behaviour, thus making possible to carry out a technical safety approval.

With reservations, it can be stated that Pearl represents the closest match to the mentioned requirements. Therefore, extensions of this language are proposed to express the time behaviour, the surveillance of events and problem oriented synchronisation. Furthermore, statements are introduced to control the desirable operating system services and software verification features.

Taking the objectives of feasible processor scheduling, inherent deadlock prevention, minimisation of context-switching operations and guaranteed reaction times as a basis, then it is investigated which multiprocessor structures yield the best performance. Single processor systems that cooperate with devices specialised in carrying through operating system nuclei turn out to have advantages over classical von Neumann and symmetrical multiprocessor structures. A further result of these considerations is that nearly optimal look-ahead algorithms for the virtual storage administration, employing the code of entire tasks as paging element, are closely related to deadline driven scheduling.

An auxiliary processor dedicated to real-time operating system nuclei is defined. First, the design of the unit is outlined, and its concept is discussed in relation to comparable ones found in the literature. The services to be provided by the device are compiled and assigned to three different reaction levels. The basic level consists of special hardware features for timing and event recognition. These are driven by the primary reaction level, which handles the occurring events and manages the time schedules. It finally communicates the activities to be executed to the secondary reaction level, which controls the whole system and supports the application programs. The three reaction levels are constructively described by detailing their functional units, internal data structures, and control algorithms.

Finally, the proposed architecture is qualitatively evaluated, especially in comparison with the conventional one. Its feasibility is verified by showing, that the extended Pearl can be implemented on the considered architecture. In this context the special compiler activities and run-time features to be incorporated into the application programs are of particular interest.

1. Introduction

1.1. Motivation

The purpose of this paper is to define a computer architecture suitable of solving some reliability and performance problems 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.

When employing the conventional von Neumann architecture and contemporary real-time operating systems, one has the problem that it is not known in advance, when a program activity scheduled for a certain point in time will actually be carried out. Since the security of men and material is at stake, it is not acceptable any longer to trust only in the speed of the process control computer. Hence, it is the aim of this paper to introduce an architecture providing qualitative improvements by guaranteeing reliable program execution. Besides the strict observation of timing and security constraints, this also implies the a priori verifiability of software and the predictability of its execution.

In order to characterise the state-of-the-art, we now indicate some of its shortcomings. So, tasks are generally scheduled on the basis of fixed priorities, although it was shown [35,24,25], that the observation of strict deadlines typical for hard real-time environments cannot be guaranteed by this procedure. It seems that the implementation of a more appropriate scheduling algorithm, such as the deadline driven one, has not yet been realised, because of lacking hardware support and since the expense of updating the tasks' remaining run-times appeared to be too high. Not all problems can be solved if the application programmer has the possibility to change the priorities dynamically. Only his software becomes more complicated, because it has to perform functions that ought to be part of the operating system. Per definitionem [16], real-time operation requires the availability of results within a given time span after the arrival of the input data. In addition to the above mentioned reasons, this can generally not be secured, since the multitasking procedures seem to assume that a calling task requesting a resource can wait until that resource is available. The last statement was given in [37] with respect to Ada (TM), but it is certainly also true for other languages and associated real-time operating systems. The contemporary real-time systems do not provide accurate timing. This is due to the low resolution hardware timers applied and the unpredictable operating system overhead that may supersede the timer routines. Furthermore, as a consequence of the "semantic gap" between the requirements of hard real-time applications and the capabilities of available processors, only a part of the features requisite for efficient problem solving and for easy formulation of reliable software has already been realised in existing process control languages. Hence, extensive usage of assembly language programming is still prevailing when realising time-critical applications.

From the above-mentioned further goals of our development become evident. The architecture should support the real-time features of process control languages in such a way that error-prone and difficult to verify assembly language programming can be renounced. The implementation of necessary language and operating system features should not fail due to deficiencies of the hardware. Finally, a technical safety approval of the software packages to be utilised is to be enabled under special consideration of their time behaviour.

By describing an apt architecture, i.e. in a constructive way, it will be shown in the present paper, that these goals can be achieved with available technology. The main measure in this respect is to implement a feasible processor scheduling algorithm. Here the term "feasible" means, that under the condition, that this is actually possible, the algorithm orders at any point in time the tasks pending for execution in such a way, that they can be processed under observation of their deadlines. From the known algorithms, we select the deadline driven one, because it minimises the number of induced processor pre-emptions. Furthermore, as a by-product, it allows the early detection and hence the handling of a future processor overload situation. These features and the operating system's time management will be supported by an elaborate hardware module, which provides accurate time readings and eliminates all superfluous servicings of the clock. The mentioned device is also the basis of two further functions that will be implemented in order to achieve the goals we have set. Thus, the waiting times and the execution times of tasks and certain operations will be supervised and the exactly timed start of instructions will be made possible. The latter feature has never been realised [44], although it is needed for many applications, e.g. for direct digital control, for the supervision whether parameters vary within gliding limits, and for the fast acquisition of measurement data. To the end of verifying the software behaviour, a separate interrupt generation module will be provided, which allows an application oriented simulation, takes the operating system overhead into account, and yields exact results.

The computer structure we are proposing here is essentially an asymmetrical two processor architecture. It consists of a conventional general processor for executing the application tasks and the functions of the operating system shell, and of an auxiliary processor as carrier of the operating system kernel. This device comprises the above mentioned hardware elements and provides the features stated there in addition to those of customary real-time operating systems. Hence, we are considering an orthogonal extension of a classical von Neumann architecture by migrating "outboard" into appropriate hardware and firmware suitable operating system functions, which are able [21] to support process control oriented high-level languages. It will be shown that this architecture can guarantee predefined time frames for its reaction to external events and that it reduces the complexity of the operating system.

According to the pertinent literature [4,11,23,28,53], various approaches have already been made to support operating system concepts by architectural measures. As far as real-time features are concerned, it can be concluded that only some of those have been migrated, which were known from previous operating systems. No attempt, however, has been made to utilise hardware and firmware facilities for the implementation of typical hard real-time support features that cannot otherwise be realised. Although architectures were developed to meet some language constructs, especially of Ada, there was no integral effort yet to build a system according to the special requirements of a process control language suited for the application engineer and of a corresponding operating system [13]. That is the motivation for the attempt aiming into this direction, which is to be worked out in this paper.

1.2. Intuitive Concept of an Architecture

How should a process control computer system be organised internally? Supposedly, this question was not asked in the beginning of real-time data processing. It was already a big achievement to employ for the control of technical processes computers with the proven von Neumann architecture, that were only adapted to their application area by the provision of process peripherals and externally available interrupt lines. All other real-time requirements were met by software, viz. by operating systems as well as by usually very specific application programming. The problems mentioned above, however, could not be solved in a general way.

As common in engineering, there are always many possible designs for a system fulfilling a given set of demands - provided the problem is solvable with the available technology. That this can be done in our case is the objective of the present paper. In order to derive an appropriate processor architecture, we now consider some analogies from other fields where systems coping with real-time conditions were already developed earlier.

The first example is the system composed of a manager and his secretary. The duties of the secretary are the reception of mail and telephone calls, the elimination of unimportant things, and minimising the interruptions of the manager's work by visitors and callers. Furthermore, she schedules her chief's work by arranging the files in the sequence in which they are to be treated and by the administration of his meeting dates. Thus, the manager's work becomes less hectic - i.e. its "real-time conditions" are eased - and more productive, because he can perform his tasks with less frequent interruptions in a more sequential and organised manner.

Similar organisational structures have been developed in various areas in order to prepare and to schedule the work of either highly qualified and paid persons or of expensive resources. As examples the reception room of a doctor's surgery or the reception desk of an automobile repair workshop as well as the operations room of a batch-processing oriented computer centre are to be mentioned here. In these offices the tasks to be carried out by both single or multiple - usually specialised - resources or persons are organised, supported, and arranged in an appropriate order.

As final analogy we consider the human brain, which consists of cerebrum, midbrain, diencephalon, cerebellum, and extended spinal cord. The signals to and from various parts of the body are transmitted via the spinal marrow, which has some similarities with a computer bus. The nerves of the spinal marrow end at the side of the brain in the extended spinal cord, that is closely connected with the midbrain, the diencephalon, and the cerebellum. The four last mentioned organs have non-arbitrary and routine reflex functions. Thus, they control the metabolism, the bodies' position, heat, and water content, and regulate respiration and blood circulation. The organs are an important switching site between the nerves of the body and those of the brain. Furthermore, the immediate reflex centre is located here. In contrast to this, the other information processing functions of higher complexity, such as the evaluation of sensual impressions, the control of arbitrary actions, and all intellectual tasks are performed by the cerebrum.

By taking pattern from these models, we now define the structure of an apt real-time computer. The concept is displayed in Figure 1. The system, we are introducing here, basically consists of two dissimilar processors. One of them is a classical von Neumann processor. Its function is the execution of the users' tasks and of those operating system processes, which are not intrinsically different from the former. These are mainly services of the supervisor shell, such as data exchange with peripherals and file management, that are provided in the form of independent tasks or subroutines called by the users' tasks. From this outer layer of the software running on process control computers, the operating system kernel is clearly, i.e. physically, separated by migrating it to the second, the auxiliary processor. Generally speaking, it will be the carrier of the system functions event, time, and task management, communication, and synchronisation.

1.3. Overview of the Contents

In distinct contrast to the customary procedure of basing process control systems on hardware with (almost) minimum capabilities, we want to proceed from the special requirements of hard real-time environments and to make use of an integral view when developing apt computer systems. The objective of such a top-down approach is the definition of a hardware architecture as closely adapted to these applications and supporting specific operating system features as the state-of-the-art in hardware technology allows. As a prerequisite for carrying through this task, a synopsis is required of the elements which need to be implemented in order to meet our goals. Therefore, we shall commence our considerations in the next section by discussing the conditions imposed by industrial environments on real-time data processing systems. From this follow fundamental demands for process control systems and especially languages required to derive the specification of the single features. Then, we shall make an inventory of the real-time elements as provided by the main languages for this area of application, viz. Ada, Fortran, HAL/S, LTR, Pearl, and PL/1. Subsequently, it will be discussed how these languages fulfill our fundamental demands, giving rise to a synopsis of important but not yet or only seldom realised real-time features. With its emphasis on real-time features, the present investigation differs from the more general language comparison [45], where besides Ada and Pearl also languages were considered having no multiprogramming and process I/O facilities.

With reservations, it can be stated that Pearl represents the closest match to the mentioned requirements. But, to increase the efficiency of real-time data processing, it is necessary to enhance the productivity of the employed languages [34]. Therefore, extensions of this language are proposed to express the time behaviour, the surveillance of events and problem oriented synchronisation. Furthermore, statements are introduced to control desirable operating system services and software verification features.

In order to derive an appropriate architecture, we shall consider different computer structures and compare their suitability for, and performance in, process control applications. The starting-point of these considerations will be the use of a feasible processor scheduling method, guaranteeing the observation of task due dates. We shall see that, by employing the deadline algorithm, the occurrence of deadlocks is inherently prevented. The asymmetrical multiprocessor structure outlined in the sequel aims at executing the tasks with the minimum of outside influence. Thus, the handling of events of all kinds is transferred to separate specialised devices, relieving the universal processors of frequent context-switching operations. Our views will show that, if serious restrictions cannot be accepted, the requirements of hard real-time data processing may be met when the operating system nuclei are run on specific hardware.

The use of deadline driven task scheduling suggests as a by-product a method for virtual storage management, representing an implementation of the working set model typical for real-time applications. We shall state two algorithms essentially assigning storage according to the ordered list of ready tasks provided by the processor scheduling. The algorithms page storage on the basis of tasks, and are optimal as far as paging demands depend upon internally foreseeable events.

On the basis of the preceding considerations, the structure of the system and of the auxiliary processor is defined. This includes the assignment of functions to the various subsystems and the description of data exchange and control flow between them. The proposed architecture is then shortly discussed in comparison with other non-conventional ones found in the literature. After that the three levels of the auxiliary processor are detailed. Its basic one is composed of various hardware elements which support all time dependent features and the event recognition. These hardware modules are controlled on the next higher level by the primary event reaction processor. In order to guarantee predefined time frames for event recognition and servicing, its operation is similarly organised as that of a programmable control device. The algorithms for the time and event management performed on this level and the corresponding data structures are described. After that, the same is done for the secondary event reaction processor, which forms the highest of the three levels. Its duties are to carry out the specified routines which represent a real-time operating system kernel and to control the execution of the users' tasks in the general processor.

Finally, the proposed architecture is qualitatively evaluated, especially in comparison with the conventional one. Its feasibility is verified by showing, that the extended Pearl can be implemented on the considered architecture. In this context, the special compiler activities and run-time features to be incorporated into the application programs are of particular interest.

2. High-Level Language Real-Time Features

2.1. Review of Existing Languages

2.1.1. Selection of Reviewed Languages

Within the scope of this paper it is naturally impossible to consider all languages aiming at applications in real-time processing. The selection of the languages to be reviewed here is determined by their dissemination and their suitability in industrial environments. So most developments are either designed for special purposes or have been only implemented on a single computer model and did not find widespread usage or were even discontinued in an experimental stage. In [43] Algol 68 was considered in a comparison of real-time languages. Although it allows parallel processing, we shall omit it here since it does not provide real-time features. The older British developments Coral 66 and RTL/2 are designated as process control languages. But the actual languages only comprise algorithmic elements and, as far as real-time features, synchronisation, and I/O are concerned, they totally rely on operating system calls. Therefore, we shall not consider them here as well as Concurrent Pascal and Modula. The latter are operating system implementation languages and process control oriented elements are lacking. Particularly, Modula is not suited for industrial applications, since no I/O and timing facilities are incorporated and since it does not provide as language constructs the features necessary to model a set of tasks upon a technical process and to control this task set accordingly. Instead, Modula allows to write machine dependent peripheral driver routines and the users' own timing, synchronisation, and resource sharing mechanisms. Formulating real-time application programs in Modula would therefore require a considerable amount of non-problem-oriented work and would yield non-portable software. The above statements on real-time facilities hold even more for Modula-2 [60], especially with respect to tasking, since only the coroutine concept is implemented to express concurrency. There are many dialects of Basic available incorporating a minimum of real-time features aimed at scientific, not heavily used applications, where the control and evaluation of experiments is to be quickly programmed. Hence, these Basic derivatives are also outside of our scope.

2.1.2. Demands of Hard Real-Time Applications

The main objective for the employment of process control computers in industrial environments is to increase productivity. To this end the operation costs are to be lowered, especially by automating routine work, and participating resources are to be utilised most efficiently. The latter may not be confounded with computer internal or operating system resources - here, instead, men, machines, material, and energy are meant. In pursuing these goals, certain boundary conditions are to be observed. First, the security of the working people and of the capital invested into the process needs to be guaranteed. Then, the processing is subject to timing constraints with only small tolerances; and last but by no means least, reliable system operation characterised by high availability and low maintenance expense is expected.

In scientific applications slightly different requirements come to the fore. Here process control computers are mainly utilised for the acquisition and evaluation of measuring data originating in directly connected experimental set-ups. Thence, efficiency aspects become less important whereas tighter timing constraints are to be observed. The reasons for this are that the exact instances of monitoring operations are prescribed by the evaluation algorithms and that data may be received with very high frequencies.

Finally, for military use of real-time systems, emphasis is shifted towards security and reliability aspects, since personnel and extremely expensive equipment depend on their faultless functioning. The design of such systems has to provide enough capacity reserves and redundancy in order to be able to cope with extraordinary requisitions.

From the above mentioned we shall now derive fundamental demands for real-time languages and systems.

Whereas hardware prices have drastically dropped in the past, the costs for developing software have considerably risen. Therefore, the latter need to be lowered in order to enable an overall cost advantage when utilising automatic process control. As commonly known, one means of achieving this is to replace assembly by high-level language programming. Such languages must reflect the users' way of thinking, and the users generally are engineers and technicians - but not computer scientists. So the language features should be easily conceivable, safe to handle, and application oriented. Furthermore, wide use of dynamic language elements seems questionable and, also for security reasons, it must be possible to completely relinquish their employment. In the past concessions have been made when designing real-time languages to enable their implementation under already existing operating systems. This contradicts a feasible proceeding, because operating systems ought to support the language features in an inconspicuous manner and to bridge the gap between language requirements and hardware capabilities. Hence, the development of a process control system should commence with the definition of a suitable language. Then the hardware architecture is to be designed, enabling the implementation of the language as easily and efficiently as possible and thus keeping the operating system and its overhead relatively small. In the case where the mentioned implementation and operating efficiency of a real-time system conflicts with the efficiency and the safety of the process to be controlled, the latter has naturally to be estimated higher.

In many applications where the security of personnel has to be guaranteed, real-time computers are working in parallel with hard-wired logic controlling, safety measures for the industrial processes. The reason for this is that procedures for the technical safety approval have been developed for such devices and laid down as obligatory directions, which are widely used by the competent authorities. Unfortunately, corresponding methods for licensing computer systems are not available yet. The difficulties encountered when trying to define such proceedings mainly concern the software, since hardware set-ups could be handled quite similarly to other electronic equipment. The first measure to overcome these problems is replacing assembly by high-level language programming. To enhance the software verifiability, real-time languages should enforce simple program formulation in easy to survey modules with clear interfaces and as little side-effects as possible. But not only the software correctness in the sense of mathematical mappings as in batch processing environments has to be proved, also its intended behaviour in the time dimension and the interaction of concurrently active tasks need verification. To this end, language features for the specification of maximum task run-times, of updated residual run-times required by tasks before their completion, and of timing constraints for certain operations, as well as for enabling the determination of module run-times by the compiler are necessary. Furthermore, resource claim capabilities have to be provided that allow the deadlock preventing scheduling of tasks. The latter should be based on algorithms guaranteeing the observation of the tasks' due dates, provided that this is possible. If, finally, the time frame for the operating system overhead is known, then the program behaviour as independent from external events becomes foreseeable. When the above mentioned features are given, an off-line simulation of the software to be approved can be carried through, not only simplifying the verification process, but also allowing the inclusion of randomly occurring external events into the consideration. In order to facilitate real-time systems to cope with worst-case conditions these events ought to be generated accordingly for simulation purposes. This test procedure oriented at hard real-time environments would yield exact information, if time constraints can be observed and whether due dates can be granted, distinctively contrasting to the prevailing practice of assuming that "computers are so fast that they will easily master the given workload".

2.1.3. Synopsis and Discussion of Real-Time Features Realised in Available Languages

The availability of, and partly further specifications on, the real-time features of the six languages under consideration have been compiled in Table 1. The information used in the course of this was obtained from the references [1,2,5,9,29,30,36,40,43,48].

Table 1
Survey on Real-Time Language Elements in Ada, Industrial Real-Time Fortran, HAL/S, LTR, Pearl, and Real-Time PL/1

Category	Feature	ADA	FORTRAN	HAL/S	LTR	PEARL	PL/1
Conventional elements	Bit processing	(n) (1)	y	y	y	y	y
	Reentrant procedures	y	n	y	y	y	y
	File handling	y	y	y	y	y	y
	Process I/O	y	y	n	y	y	y
Tasking	*Declaration of tasks*	y	y	y	y	y	y
	Hierarchy of tasks	y	n	y	n	y	n
	Stati of tasks available	2	n	n	n	n	3
	Controllability of tasks	poor	complete	limited	limited	complete	poor
	Exception handling in tasks	y	n	y	n	y	y
	Task scheduling	y	y	y	y	y	y
	Implied scheduling strategy	Prio., FiFo	—	Prio.	Preemp. prio.	Prio.	Prio., OS-dep.
	Usage of priorities	y	n	y	y	y	y
	Changebility of priorities	y (2)	n	y	n	y	y
Synchro-nization	Synch. mechanisms available	y	y	y	y	y	y
	Semaphors	y (2)	y	n	y	y	n
	Further means of synch.	Rendezvous Signal (2) Shared varia.	Resourcemark	Compool Lock	Blockstruct.	Bolt Blockstruct. Implicitly	Shared obj. Lock
	Resource reservation	y	y	y	y	y	y
	Stati of resources available	y	y	n	y	n	y
	Resource allocation strategy	FiFo	OS-dep.	Prio.	Prio.	Prio., OS-dep.	Prio.
	Deadlock prevention supported	n	n	n	n	n	y
Events	Event mechanisms available	y	y	y	y	y	y
	Interrupt handling	y (1)	y	(y)	y	y	y
	Enable/disable interrupt	n (1)	y	(y)	(n)	y	y
Timing	Date/time available	y	y	y	(y)	y	y
	Cumulative run-time available	y (2)	n	(y)	n	n	n
	Forms of time scheduling	Delay	various	various	Delay Cycl.	various	Delay Fixed date
	Timing control of synch. op.s	y	n	n	n	n	y
Verification	Aids for testing	Raise except. Call interr. entries	Set eventmark	Simulation Run-time det. Trace	Mapping Trace Flag display	Induce event Trigger int.	n

(1) Only indirectly possible in the final Ada version [3].
(2) Not any longer available in the final Ada version [3].

The first category of features listed in Table 1 comprises elements required in process control applications but of a conventional character. This may cause contradiction as far as process input/output is concerned. Apart from their time scheduling, however, these operations are carried through by corresponding driver routines of the operating system in a manner similar to standard I/O.

In all languages considered, parallel processing is organised on the basis of tasks. These can be hierarchically ordered as in Ada, HAL/S, and Pearl. Each language has as a foundation a different model of task states. But it is not possible to interrogate as to which state a task is presently in. Only Ada and PL/1 allow a determination of whether a task is active or already terminated. To control the execution of tasks and their inter-status transfer, tasking operations are provided as language features. In this respect Pearl and Fortran offer a wide range of capabilities that may even be scheduled in Pearl. In Ada and PL/1 it is only possible to activate and abort tasks, whereas additionally a wait operation may be requested in HAL/S and LTR. Furthermore, the last two allow task activations to be scheduled for the occurrence of simple timing and other events. After the schedules for their initiation are met, the tasks compete for resources, especially for processors. All languages except Fortran imply that the appropriate dispatching decisions are then made on the basis of task priorities that may be dynamically changed, however not in LTR. According to [1], an operating system underlying Ada programs may also utilise the first-in-first-out strategy.

All languages considered provide means for the synchronisation of task executions and for the reservation of resources to be used exclusively. The most common synchronisation feature is the semaphore. In addition, various other concepts are realised in the different languages. The availability of resources can be checked in Ada, Fortran, LTR, and PL/1. The employed resource allocation schemes are either priority based or operating system dependent. Only Ada systems work according to the first-in-first-out strategy. Deadlock prevention as objective of the language has only been found in PL/1.

Tasks may communicate with each other in all six languages via single bit messages called events. The handling of interrupts is also a common feature. However, they cannot be enabled or disabled in an Ada program.

As we shall see in section 2.1.4, all languages are lacking in important timing features. The cumulative run-time of a task which must be known to perform deadline driven scheduling is available in Ada. For simulation purposes this information is given by HAL/S measured in machine cycles. Whereas the capabilities of Ada, PL/1, and LTR for the time scheduling of task executions and operations are very limited, the former two languages allow a supervision of whether synchronisation operations are completed within predefined time frames. The other languages offer a wide range of time scheduling features.

Only HAL/S provides run-time determination and simulation facilities to aid the verification of real-time software. For this purpose in three other languages one has only the possibility of generating events under program control.

When comparing the version [2] of Ada with its preliminary specification [1], one finds that its feasibility for real-time programming has been impaired by abolishing several significant features. Thus, the initiate statement was deleted. Tasks are immediately activated when their declarations have been elaborated. The once assigned priorities may not be dynamically changed. Semaphores and signals as predefined synchronisation mechanisms have been relinquished; and finally, the cumulative processing time of tasks is not available any longer.

As far as actual implementations are concerned, the situation is quite different for the six languages considered here. As summarised in [12], there is a vast number of Ada compilers already available or presently under development. The target systems include all major microcomputers. Various dialects of Fortran and PL/1 with a broad range of different real-time capabilities are on the market. Suitable for consideration in a language comparison, however, are only the corresponding proposals for standardisation [9,30] comprising the experience with former realisations and providing the widest spectra of features. Of the subroutine package constituting Industrial Real-Time Fortran only subsets have experimentally been implemented yet. For extended PL/1, unfortunately, no implementations could be ascertained. HAL/S compilers exist for IBM/360 and /370 systems and LTR can be employed on SEMS' line of Mitra computers. A survey of the available Pearl systems as communicated by [46] is given in Table 2.

Table 2. Target Computers of Available Pearl Systems		
AEG 80-20	Intel 8086	Siemens R10
ATM 80-05HD	Litef LRI 432	Siemens R20
ATM 80-10	LSI 11	Siemens R30
ATM 80-30	Micronova	Siemens R40
DP 1000	Motorola 68000	Siemens 310
DP 1500	Mudas 432	Siemens 330
EPR 1300	Mulby 3	Siemens 404/3
EPR 1500	Nord 10	Siemens 7000 Series
SDR 1300	Nord 100	VAX-11 Series
MPR 1300	PCS CADMUS 9000	Z 80
HP 1000 F	PDP 11 Series	Z 8000
HP 3000	RDC System	PC-AT Compatibles

2.1.4. Synopsis and Discussion of Further Real-Time Features to be Implemented

When comparing the requirements for real-time languages and systems that were outlined in section 2.1.1 with the capabilities of existing languages, it becomes obvious that various elements especially important for the production of reliable software are still missing or are only rudimentarily present in available languages. We shall discuss these elements and give arguments for their necessity in the sequel. To provide an easy survey, they are also listed in Table 3.

Table 3. Desirable Real-Time Features
- Application oriented synchronisation constructions
- Surveillance of the occurrences of events within time windows
- Surveillance of the sequences in which events occur
- Time-out of synchronisation operations
- Time-out of resource claims
- Availability of current task and resource stati
- Inherent prevention of deadlocks
- Feasible scheduling algorithms
- Early detection and handling of overload
- Determination of entire and residual task run-times
- Task oriented look-ahead virtual storage management
- Accurate real-time
- Exact timing of operations
- Application oriented simulation regarding the operating system overhead
- Interrupt simulation and recording
- Event recording
- Tracing
- Usage of only static features if necessary

They can be divided into three groups, the first of which comprises constructions both easing the formulation of frequent applications or serving to supervise the stati of tasks and resources as well as the duration of synchronisation and resource claim operations. The second group consists of desirable operating system services that should be provided for the purpose of reliable and foreseeable software performance. To be able to control these features several language elements need to be introduced. Finally, software verification measures are collected in the third group of features. Their utilisation only requires a few control statements.

Despite their fundamental significance in real-time applications, none of the languages considered allows to specify completion deadlines for task executions. But, as already stressed in section 2.1.2, processors ought to be scheduled using algorithms capable of guaranteeing the observation of the tasks' deadlines. This goal can generally not be achieved by priority schemes under control of the programmer as supported by most languages and operating systems. Such scheduling algorithms also allow early detection of whether it will be possible to process a task set in time. Otherwise, parts of the workload have to be removed. In order to carry this through in an orderly and predictable manner, it is desirable that the language allows a specification of which tasks could be terminated or at least be replaced by ones with shorter run-times when required by an emergency or an overload situation. The implementation of due dates observing scheduling algorithms must be supported by hardware and language elements for the determination of the tasks' run-times or upper bounds for them and for the updating of residual run-times necessary before the tasks' final completion. In this connection the stealing of memory cycles by direct memory access devices becomes a problem. But, in [22] it was shown that it can be solved in order to provide task scheduling schemes meeting the requirements of a reliable and foreseeable software behaviour in hard real-time environments.

To provide information about date and time, existing operating systems and thus also real-time languages completely rely on interval timers under control of a corresponding interrupt servicing routine. Since the processor load by the latter must be kept reasonably small, the resolution of the time information remains rather poor. The unpredictable operating system overhead introduces further inaccuracies which become more intolerable the longer systems run after restarts. Therefore, highly accurate real-time clocks should be provided as hardware components. The attention of the supervising software will be initiated by an interrupt to be generated when the actual time equals the next critical moment that has been loaded into a comparison register. Thus, the requirements of scientific and military applications also with regard to very short time intervals can be met. Provisions should be made in the language to put tasks in a wait state just before performing certain operations, e.g. inputs. When the clock's interrupt signal can be directly utilised to resume the execution of tasks waiting in such a manner, the accurate timing of single operations is made possible. In this way, for example, measurements can be carried out at precisely equidistant or Gaussian points.

Based on this timing hardware, several time related surveillance features should be able to be expressed within a language that otherwise would have had to be explicitly programmed. So in control applications it needs to be guaranteed that events occur within given time frames or in predefined sequences. As can be seen in Table 1, Ada and PL/1 already provide a time-out feature for synchronisation operations. More than that, the process of claiming resources and assigning them to tasks must be supervised in a real-time environment. In this respect in [37] the statement, which is also true for other languages, is found that Ada multitasking seems to assume that a calling task requesting a resource can wait until that resource is available.

Now, a series of features for supporting the verification of software and the predictability of its execution shall be discussed. A step towards enabling a technical safety approval of real-time programs is to create the possibility for application oriented simulation. This would also allow to determine the adequate computer capacity needed for a certain application. In contrast to the language specification [30], which reads that state transitions (of tasks) are performed "instantly", i.e. in zero time, the processing time for the operating system overhead has to be taken into consideration in the course of a simulation. In the test phase interrupts must also be simulated and protocoled to check whether a task set can cope with the requirements of a hard real-time environment. On the other hand, all events occurring during the execution of such a task set are to be recorded for the subsequent error analysis. This feature is also useful in the later routine application of a software package where it can ease the post mortem error search. To guarantee a predictable program execution, every task should be fully expressible in terms of static language elements wherever necessary. It has already been mentioned that requested resources must be granted within predefined time frames. To give urgent tasks the possibility to elude busy resources or to initiate alternative actions, the present stati of resources must be available for questioning. Finally, it is to be stressed once more that a real-time language ought to make the occurrence of deadlocks impossible by an a priori prevention scheme.

2.2. Proposal for an Extension of Pearl

The comparison of real-time languages, which was carried out in the preceding section, revealed that Pearl meets best the demand pattern as established there. However, a number of problem oriented language constructions especially with regard to expressing the programs' exact time behaviour appear to be missing. The purpose of this section is therefore to take remedial measures by defining here appropriate extensions of Pearl. The leading idea behind all the proposals, that will be outlined, is to facilitate reliable and predictable program execution, this being a prerequisite for the safety approval of software in hard real-time environments. The elements that seem to be missing or only rudimentarily present in Pearl were already discussed in section 2.1, where their necessity was also substantiated.

2.2.1. Proposal of Additional Language Elements for Pearl

2.2.1.1. Protection of Resources and Temporal Surveillance of Synchronisation Operations

For the purpose of securing data integrity when several tasks access the same resource, Pearl only provides the basic means of semaphores and bolts. Applying them for the locking of resources veils the nature of the operation to be performed, since there is no obvious and verifiable relation between resource and synchroniser. Furthermore, programming errors become possible by not releasing requested resources, that cannot be detected by the compiler due to the missing syntactical connection between the mutually inverse synchronisation operations.

The access to shared objects, i.e. to shared variables and dations, can be protected with the help of implicit, "invisible" bolts to be generated by the compiler. For instructing it to do so in the case of shared basic objects, arrays, and structures, we introduce the optional attribute

SHARED

as part of the pertaining declaration syntax. This feature has been taken over from the dataway synchronisation providing such a control upon opening. Since synchronisers are constituents of shared objects, the prevailing rules need to be observed. So they must be declared on module level. As data types of shared variables, array elements, and structure components, respectively, only basic ones are admitted, because sharing others either leads to difficulties or is meaningless. For encapsulating the access to protected resources and to enforce the release of synchronisers, we introduce a LOCK statement similar to the one described in [9] and having a time-out clause that was also proposed in [6] in a different context:

```
LOCK synchronisation-clause-list [NONPREEMPTIVELY]
        [timeout-clause] [exectime-clause]
PERFORM statement-string
UNLOCK;
```

with

```
timeout-clause::=
        TIMEOUT {IN duration-expression | AT clock-expression}
        OUTTIME statement-string FIN ,
exectime-clause::=
        EXECTIMEBOUND duration-expression ,
synchronisation-clause::=
        semaphore-expression-list |
        EXCLUSIVE(sync-object-expression-list) |
        SHARED(sync-object-expression-list)
```

and

sync-object::=bolt | shared-variable | dation

The task executing a LOCK statement waits until the listed shared objects can be requested in the specified way. By providing a TIMEOUT attribute, the waiting time can be limited. If the lock cannot be carried through before the time limit is exceeded, the statements of the OUTTIME clause will be executed. Otherwise control passes to the statement sequence of the PERFORM clause as soon as the implied request operations become possible. The corresponding releases will be automatically performed upon reaching UNLOCK, or when terminating the construction with the help of the

QUIT;

statement. In order to free seized resources on behalf of increasing the program efficiency as early as possible, the instruction

UNLOCK semaphore-expression-list | sync-object-expression-list;

can be applied already before dynamically reaching the end of the surrounding LOCK statement, where the remaining releases will be carried through. The optional EXECTIME clause is introduced to enhance the predictability and safety of real-time systems. It limits the time, during which a task is in a critical region. Thus, it is prevented that programming errors resulting in infinite loops can cause a blockage of the whole system. In order to handle a violation of a loop's execution time bound, a system signal must be introduced. The optional attribute NONPREEMPTIVELY serves for the improvement of performance. By specifying it, the operating system is instructed not to pre-empt the execution of the LOCK statement due to reasons of the applied processor scheduling strategy. Thus, superfluous and time-consuming context-switching operations can be saved in the case where a more urgent task requesting one of the locked resources commences execution before termination of the LOCK statement. For shared objects of type dation, their reference in a synchronisation clause of a LOCK statement shall be equivalent to executing an OPEN-CLOSE statement pair with a corresponding dataway synchronisation control. Except for their appearance in synchronisation clauses, shared objects may only be referenced within the framework of LOCK statements. They must be locked for exclusive access if the reference is used in any assignment context or if they are passed as parameters to procedures with the identical mechanism.

The above described LOCK feature should replace Pearl's six unstructured synchronisation statements. Only the USING clause of the ACTIVATE tasking operation complies with structured programming, since it implies the release of the seized semaphore upon termination of the activated task. However, like the LOCK statement, the USING clause must be endowed with an optional time-out clause specifying an alternative action, because in hard real-time environments it is not always possible to wait indefinitely until a corresponding activation operation is permitted to be performed:

USING expression
[TIMEOUT {IN duration-expression | AT clock-expression}
OUTTIME statement-string FIN]

2.2.1.2. Additional Monadic Operators

In this section we shall define several intrinsic functions providing status information on tasks and synchronisers. All functions yield results of type fixed. Given a certain model of task states together with an appropriate numbering of the latter, the

TSTATE task-identifier

operator returns the number of the parameter's actual status. The current value of a semaphore is to be made available by applying the operator

SVALUE sema-identifier.

In order to interrogate the stati of bolt synchronisers, we introduce the function

BVALUE bolt-identifier

returning the values 0 or -1 in the unreserved or exclusive access states, respectively, or the number of tasks having shared reading access. With this function also the stati of implicit bolts can be questioned, when applying it in conjunction with the operator

SYNC shared-object

referencing its argument's implicit bolt.

2.2.1.3. Surveillance of the Occurrence of Events

For the surveillance whether and in which sequence events occur we shall propose in this section a new language feature. In this context we use a wider notion of events to summarise

- interrupts,
- signals,
- time events,
- status transfers of synchronisers and tasks, and
- the assumption of certain relations of shared variables to given values.

These events may be stated according to the following syntax rules:

```
event::=
        WHEN interrupt-expression |
        ON signal-expression |
        AT clock-expression |
        AFTER duration-expression |
        status-function-call relational-operator expression |
        shared-variable-reference relational-operator expression |
        bit-type-shared-variable-reference
```

where status-functions are the ones introduced in the previous section. In case of the last three of the above alternatives the events are raised when the corresponding Boolean expressions turn true. Now the new language element is defined by

EXPECT alternative-string FIN;

with

alternative::=AWAIT event-list DO statement-string.

When the program flow of a task monitoring events reaches an EXPECT block, the expressions contained in the event specifications are evaluated and the results stored, and then the task is put into a wait state until any one of the events mentioned in the AWAIT clauses occurs. Then the statement string following the associated DO keyword will be executed. In case several events listed in different AWAIT clauses occur together, the corresponding DO clauses will be performed in the sequence they are written down. When the operations responding to the just occurred event(s) are executed, the task is again put into the wait state expecting further events. In order to leave the described construction after finishing a surveillance function and to transfer control behind the block end FIN, the

QUIT;

statement has to be applied. When this has been done or when it has been left otherwise, there will be no further reaction to the events mentioned in the EXPECT feature. Nesting RESUME statements, ON reactions, or other EXPECTs into the alternatives is not meaningful, since routines in which this appears to be necessary can also be formulated employing one EXPECT structure only. The applications for which a scheduled RELEASE statement has been required elsewhere [20,44] can be programmed using the element described above.

2.2.2. Additional Operating System Features to be Supported by Pearl

2.2.2.1. Inherent Prevention of Deadlocks

Employing the LOCK language element introduced earlier for claiming resources, it already becomes possible for the compiler to verify the correct application of two well-known deadlock prevention schemes, which may be requested by stating a corresponding pragma. According to the resource releasing procedure, all required shared resources must be reserved en bloc before entering the critical région where they are used. This can easily be accomplished by mentioning them in the synchronisation-clause-lists of the LOCK statement. The compiler has only to examine now if no further resource requests appear in the PERFORM clause to ensure a deadlock free operation. In order to apply the resource hierarchical method, an ordering of the shared objects needs to be declared. An appropriate facility for use on the module level, when this deadlock prevention scheme shall be applied, is introduced as

RESOURCE HIERARCHY sync-object1 hierarchy-clause-string;

with

sync-object1::=semaphore | bolt | shared-object

and

hierarchy-clause::= >sync-object1 .

Nesting of LOCK statements can then be allowed as long as the sequence in which the resources are claimed complies with the predefined hierarchical ordering.

2.2.2.2. Exact Timing of Operations

Although Pearl allows to specify time schedules for tasking operations, one cannot be sure when they actually take place. Since this situation is unacceptable in many applications, a language element appears to be necessary to request the punctual execution of tasking operations, i.e. the critical moments need to be actively awaited. For that purpose an optional

EXACTLY

attribute to be placed in time schedules will be used.

2.2.2.3. Application of Feasible Scheduling Algorithms

It has already been stressed before that processors ought to be scheduled employing procedures capable of guaranteeing the observation of strict deadlines usually set for the execution of tasks in hard real-time environments. This goal, however, can generally not be achieved by priority schemes under control of the programmer as supported by most operating systems and languages such as Pearl. The implementation of feasible scheduling algorithms, like the deadline driven one optimal for single processor systems, needs to be accompanied by language elements for specifying the due dates and for determining total and residual run-times, or at least upper bounds thereof, required by tasks to reach their completion. Furthermore, such scheduling algorithms allow the early detection of the possibility for processing a task set in time. Otherwise, parts of the workload have to be discharged. In order to carry this through in an orderly and predictable manner, it should be possible to state in the source program which tasks could be terminated or at least be replaced by ones with shorter run-times when required by an emergency or an overload situation. The above mentioned outlines the objectives of this section.

We begin with replacing the PRIORITY clause by an optional deadline of the form

DUE AFTER duration-expression

in task declarations and in the ACTIVATE, CONTINUE, and RESUME statements. When the condition for a task's (re-) activation is fulfilled, this duration is added to the actual time yielding the task's due date. As additional parameter the deadline driven algorithm requires the task's (residual) run-time which is stated in its declaration in the form

RUNTIME duration-expression | SYSTEM.

In general, the given duration can only be an upper bound for the required processing time. If the latter is not known, by inserting the keyword SYSTEM here, the compiler is instructed to supply it according to the method outlined below. For use by the scheduling algorithm three variables have to be allocated for each task, whose task control block may be denoted here by T. The due date will be stored in

T.DUE,

an object of type clock. The two further variables have the type duration and must be continuously updated while the task is being executed:

T.TIME

is initially set to zero and contains the accumulated execution time, whereas

T.RES

initialised with the RUNTIME parameter is decremented to provide the residual time interval to complete the task properly. When the due dates and execution times of tasks are a priori available, a feasible scheduling algorithm is able to detect whether a task set given at a certain moment can be executed meeting the specified deadlines. Otherwise an overload situation must be handled. In order to carry this through in an orderly and foreseeable manner, all interrupts will be masked and all tasks will be terminated and schedules for further activations of them deleted whose declarations do not contain the

KEEP

attribute, that needs to be introduced as an option. Then the remaining tasks together with the emergency tasks scheduled for the event of an overload condition will be processed.

Now the determination of a task's overall execution time requires more attention. As long as the program flow is strictly sequential, there is no problem, since only the processing times of the single instructions need to be summed up. In all other cases one has to carry out an appropriate estimation. As far as IF and CASE statements and procedure calls are concerned, the execution time of the maximum length path through these constructions will be feasible. The number of times the instructions in a REPEAT statement are performed generally depends on the values of variables and is therefore a priori not determined. In order to enable the run-time estimation also for this feature, we augment its syntax by the following clause already demanded in [18]:

MAXLOOP fixed-literal.

If the number of iterations exceeds the limit set with this clause, a system signal should be raised to be treated by an appropriate ON statement. At first, this restriction seems to be a serious drawback. But, on the other hand, it enhances the reliability being essential for real-time software, because faulty programming cannot lead to infinite loops and thereby to system hang-ups. The span between the times a synchronisation operation requesting resources is reached and finally carried through is generally undetermined. By extending the statements of this kind with TIMEOUT clauses also these language features become subject to run-time estimations. Since the execution of ON statements is not part of the normal program flow, they cannot be regarded in the course of run-time calculations. However, for these purposes ON elements can be considered as independent tasks activated by interrupts that immediately suspend the tasks in which they are defined and continue the latter as soon as they terminate. Thus, it becomes possible to estimate the workload requirements for handling ON statements with the methods of worst-case simulation to be discussed in the subsequent section. A similar argument applies for treating EXPECT blocks. At first, the initial part of the task surrounding an EXPECT feature is performed finishing with activating all alternative actions of the latter as separate tasks scheduled for the occurrence of the events specified in the respective WHEN clauses. Here also actions can be specified to be performed in case certain time limits are exceeded. Then, the given time conditions can be utilised for the run-time estimation. Finally, the task's continuation will be scheduled for the end of the EXPECT element. From the viewpoint of run-time determination, most concern is caused by the GOTO statement that is added to its harmfulness resulting from the fact that its unrestricted use may produce difficult to survey and hence error-prone programs. This has been perceived at an early stage and has given rise to the development of structured programming. Since this discipline has revealed that in the majority of cases the application of jumps can be avoided provided appropriate structured language elements are available - such as in Pearl - we can confine the usage of GOTO statements to the purpose of leaving loops, blocks, and other constructions when some code is to be skipped. Thus, both the software reliability is enhanced and the run-time estimation is enabled, because jumps can be simply disregarded. The above discussion has shown that an exact calculation of a task's run-time will be an exceptional case. Hence, we have to content ourselves with estimations. But, it will often be possible to improve the estimation of a task's residual run-time in the course of its execution, e.g. when control reaches a point where two alternative program sequences of different length join again. To this end the statement

```
UPDATE task-identifier.RES:=duration-expression;
```

is introduced for setting the residual run-time to a new and lower value.

2.2.2.4. Support of Task Oriented Virtual Storage Management

The task oriented virtual storage management scheme, which will be discussed in detail in section 3.2, requires one language feature for its support. To enable the determination of future task (re-) activations in the case of interrupt driven schedules, the user must supply an average occurrence frequency for each of them. This can be achieved by augmenting the interrupt definition in the system division with a corresponding optional attribute:

```
INTERVAL duration-literal.
```

2.2.3. Software Verification Features

2.2.3.1. Tracing and Event Recording

For the purpose of controlling whether a real-time software package works in the intended way, it is necessary to record intermediate results and the occurrence of events influencing the single activities. The first feature to be provided in this context is tracing being not typical for process control systems. Hence, we can refer here to other languages like Fortran and LTR where statements were defined instructing the compiler to generate additional code for writing specified traces into certain files. Comparable statements need to be introduced into Pearl. In order to avoid frequent changes in the source code, a compiler option appears helpful that selects whether the tracing control statements are to be considered or to be treated as comments. When the behaviour of a real-time system is to be understood, it must be known when the events determining the state transfers of tasks have occurred. The kind of events to be considered here are:

a) interrupts, signals, and changes of masking states,
b) state transfers of tasks and synchronisation variables,
c) reaching and actual execution of tasking and synchronisation operations.

These events or specified subsets thereof should be recorded on a mass storage device also during routine operation to enable the post mortem analysis of software malfunctions. As we shall see later on, when simulations are carried out in the test phase such files represent the corresponding output requiring no further specification. The aim of the source code oriented debugging system described in [38] is more to support the error removal process than the approval of a finished software package. Its services are requested outside the language interactively or in a test plan. Besides the above mentioned features, it also provides the usage of breakpoints as well as the possibility to change data and to execute certain statements interactively.

2.2.3.2. Restriction to Static Language Features

In hard real-time environments the application of dynamic language elements appears questionable, since it introduces unpredictability with respect to capacity and time requirements making the work of a scheduling algorithm impossible. Thence, for the sake of reliability, the usage of variable array dimensions and of recursive procedure calls should - at least optionally - be suppressed.

2.2.3.3. Application Oriented Simulation

Given the possibilities for event recording and tracing as outlined above, no additional features need to be introduced in Pearl to facilitate the simulation of application software, because the language already contains the statements necessary to generate external and internal events, viz. TRIGGER and INDUCE. A program simulation would now be carried out as follows. The formal description of the requirements or of a benchmark test for a program is laid down by writing a test routine generating according to worst-case conditions the events the software package is acting upon. If need be, appropriate test data are provided as inputs. In case these data cannot be read in from the original devices, they could be made available by especially written interfaces. Then, the test program and the software to be verified are jointly processed under control of the operating system that is also to be applied in a routine environment. When tracing and event recording were specified, all the results a simulation is expected to yield are automatically provided. Since the described simulation method takes place under very realistic conditions, it should thus fulfill the requirements of a technical safety approval. In general, the time consumption of a simulation will be greater or equal to the one of the actual process. Hence, it is necessary to stop the system clock always when the test routines are being executed. The overall time requirements, however, can even be reduced setting the system time to the next scheduled critical moment when the processor turns idle. These are the only additional functions a simulation monitor could provide. Naturally, also a faster processor of the same kind may be applied instead of the target system.

2.2.4. Synopsis of the Proposed Pearl Extensions

In order to give a complete survey on the various language extensions and alterations proposed above in a mainly informal manner, their exact production rules are collected in the sequel. The syntax description uses the symbols | and [ ] to denote alternatives and optional parts, respectively. For reasons of easier readability the following two suffixes are employed:

-list indicates one or more elements of appropriate type separated by commas and
-string indicates repetition of elements of appropriate type.

Not defined non-terminal symbols are either known from the original Pearl syntax or their meaning is evident from the previous informal description. Where for simplicity reasons the non-terminal "expression" is mentioned, its correct type has to be respected.

```
Monadic operators:
        BVALUE
        SVALUE
        SYNC
        TSTATE

basic-type-declare-clause [SHARED]
array-declare-clause [SHARED]
structure-declare-clause [SHARED]
task-declaration::=
        task-identifiers:TASK
        [DUE AFTER duration-expression RUNTIME {duration-expression | SYSTEM}]
        [GLOBAL] [RESIDENT] [KEEP];
        [definitions]
        [statement-string]
        END;

repeat-statement::=
        [FOR identifier]
        [FROM  expression]
        [BY    expression]
        [TO    expression]
        [WHILE expression]
        REPEAT MAXLOOP fixed-literal [;]
        [declaration-string]
        [statement-string]
        END;

quit-statement::=QUIT;

schedule::=
        AT expression [interval [duration]] [EXACTLY] |
        [WHEN expression] [AFTER expression] [ALL expression [duration]] [EXACTLY]
interval::=EVERY expression | ALL expression
duration::=UNTIL expression | DURING expression
timeout-clause::=
        TIMEOUT {IN duration-expression | AT clock-expression} OUTTIME statement-string FIN

due-clause::=DUE AFTER duration-expression

expect-statement::=EXPECT alternative-string FIN;

alternative::=AWAIT event-list DO statement-string

event::=
        WHEN interrupt-expression |
        ON signal-expression |
        AT clock-expression |
        AFTER duration-expression |
        status-function-call relational-operator expression |
        shared-variable-reference relational-operator expression |
        bit-type-shared-variable-reference

activate-statement::=
        [schedule-list] ACTIVATE task-identifier1 [due-clause]
        [USING expression [TIMEOUT {IN duration-expression | AT clock-expression}
         OUTTIME task-identifier2]];

continue-statement::=
        [schedule-list] CONTINUE [task-identifier] [due-clause];

resume-statement::=schedule-list RESUME [due-clause];

update-statement::=
        UPDATE task-identifier.RES:=duration-expression;

lock-statement::=
        LOCK synchronisation-clause-list [NONPREEMPTIVELY]
                [timeout-clause] [exectime-clause]
        PERFORM statement-string
        UNLOCK;

synchronisation-clause::=
        semaphore-expression-list |
        EXCLUSIVE(sync-object-expression-list) |
        SHARED(sync-object-expression-list)

sync-object::=bolt | shared-variable | dation

exectime-clause::=EXECTIMEBOUND duration-expression

unlock-statement::=
        UNLOCK semaphore-expression-list | sync-object-expression-list;

resource-hierarchy-definition::=
        RESOURCE HIERARCHY sync-object1 hierarchy-clause-string;

sync-object1::=semaphore | bolt | shared-object

hierarchy-clause::= >sync-object1

frequency-attribute::=INTERVAL duration-literal
```

3. Derivation of a Suitable Multiprocessor Architecture

3.1. Implications of Deadline Driven Scheduling

The fundamental condition that a process control system employed in a hard real-time environment is expected to fulfill is to carry out all tasks within predefined time frames, provided this is actually possible. Algorithms generating appropriate schedules for all task sets executable under observation of the given due dates are called feasible, and several have been identified in the literature [24,25,27,31,32]. A few of them deal with task sets whose elements can all be started immediately, whereas others operate on task sets for which precedence relations are given. The latter may even observe certain conditions being too restrictive for offering universal applicability. But it is also unrealistic to expect the existence of partial orders between the members of the task sets in any case, since they cannot explicitly be specified by available process control languages, and tasks may be activated by external events. Thus, the situation generally prevailing in real-time data processing is that at any time there is a number of runnable tasks competing for the assignment of a processor. This task state is entered by explicit activation, continuation, or after releasing synchronisers. For scheduling such "free" task sets the response time driven and the minimum slack time algorithms for single processor systems as well as a modification of the former strategy for multiprocessors are mentioned in [25], where their feasibility is shown. It is characteristic for the state-of-the-art to note that both presently employed scheduling methods based on either fixed, or on user modifiable priorities, are not feasible. Unfortunately, the minimum slack time algorithm is only of theoretical interest, since it is pre-emptive and requires processor sharing when several tasks have the same slack time. The latter can, for instance, be accomplished by time slicing with a very small time constant. Both properties imply frequent context-switching operations degrading the system performance by unproductive overhead. Furthermore, the processor sharing may cause synchronisation difficulties with regard to the resources requested by the single tasks. In contrast to this, the deadline driven algorithm is non-pre-emptive, at least as long as no further task becomes runnable. If the number of pre-emptions enforced by a scheduling procedure is considered as a selection criterion, this algorithm turns out to be optimal [24]. Even when further tasks turn runnable during the execution of a free task set to which they are added, this assignment scheme maintains its properties and then generates optimal pre-emptive schedules [31,32]. Transferred to multiprocessors, however, it is not even feasible any more. An extension of the algorithm, namely the one first stated in [27] and modified in [25], achieves the feasibility, but by sacrificing the non-pre-emptivity and at the cost of much higher complexity. This fact does not suggest utilising symmetric multiprocessor systems. In order to obtain, for structural decisions, a better understanding of the schedules that the algorithms generate, we should consider the following example.

Let a set T of six tasks be given at t=0, each of which is characterised by the tupel (Deadline, Required execution time), to be processed on a symmetric 3-processor system:

$$T=\{T1=(5,4), T2=(6,3), T3=(7,4), T4=(12,8), T5=(13,8), T6=(15,12)\} .$$

The schedule provided by the considered strategy is displayed in Figure 2 in form of a Gantt diagram. This example reveals that the scheduling process required 5 pre-emptions and corresponding context-switching operations, that may even result in repeated program loading, if the different processors do not use common memories. The diagram also shows that two processors are idle for 6 time units before the task set is completely executed. Since several processors cannot simultaneously work on one task, it is impossible to level the load and to reduce the task set's overall response time. When scheduling the same task set according to increasing deadlines for a single processor system three times faster, we obtain the Gantt diagram given in Figure 3. Here one task is uninterruptedly executed after the other. Thus, time consuming context-switchings are saved and at any time, essentially, there needs to be only one program in main storage. Furthermore, the handling of the entire task set is finished earlier than in the case of the comparable multiprocessor. Since in theoretical considerations the overhead is usually - but unrealistically - neglected, the overall execution time proportion is further shifted in favour of the single processor structure. Hence, the factor for speeding up a 1-processor system in order to become equivalent to an m-processor as far as performance is concerned will generally be considerably less than m. Another advantage of strictly sequential task execution is that synchronisation conflicts which may give rise to waiting and processor idle times are prevented.

An important part of a feasible scheduling algorithm is the examination whether a given free task set will be executable within the prescribed time frames. For any time t, $0 \le t < \infty$, and any task T with deadline $t_z > t$, let

$a(t) = t_z - t$ be its response time,
$l(t)$ the (residual) execution time required before completion, and
$s(t) = a(t) - l(t)$ its slack-time (margin).

Then, necessary and sufficient conditions [24,27] that a task set, indexed according to increasing response times of its n elements, can be carried through meeting all deadlines are,

for $m=1$:

$$a_k \ge \sum_{i=1}^{k} l_i, \; k=1,\dots,n, \tag{1}$$

and for $m>1$:

$$a_k \ge \frac{1}{m}\Big[\sum_{i=1}^{k} l_i + \sum_{i=k+1}^{n} max(0, a_k - s_i)\Big], \; k=m,\dots,n-m+1, \tag{2}$$

$$a_k \ge \frac{1}{n-k+1}\Big[\sum_{i=1}^{k} l_i + \sum_{i=k+1}^{n} max(0, a_k - s_i) - \sum_{i=n-m+1}^{k-1} a_i\Big], \; k=n-m+2,\dots,n, \tag{3}$$

For $k=1,\dots,m-1$ eq. (2) must be valid, except if there are j tasks with $a_k > s_i$ for $k < i \le n$ and $j+k < m$ then

$$a_k \ge \frac{1}{k+j}\Big[\sum_{i=1}^{k} l_i + \sum_{i=k+1}^{n} max(0, a_k - s_i)\Big] \tag{4}$$

must be fulfilled.

Comparing (1) with the set (2,3,4) of inequalities, it is quite obvious that the complexity of performing the processability examination is higher by far for the case m>1. The like also holds with regard to the actual processor assignment scheme, since for m=1 the task with the shortest response time is always being executed. In contrast to this, the relations of task margins to the response times of other tasks already assigned to processors need to be observed, and the algorithm has also to be called when a non-running task loses its margin. Hence, the procedure must keep track of the time events when the margins of non-assigned tasks vanish, or when a task's margin becomes equal to the response time of another executing one. This feature adds considerably to the already high complexity of the algorithm.

According to the above arguments, it proves to be favourable to structure real-time computer systems essentially in the form of single processors. This concept also covers a set of interconnected uniprocessors, each of which is dedicated to a certain partial application within the entire process to be controlled. Then, in the ideal case that no further task becomes runnable before the presently executed free task set is completed, the whole processing is done in a fully sequential manner, one task after another. Thus, unproductive context-switching operations are saved and deadlocks are impossible, and hence do not need to be handled. Unfortunately, such a situation is not realistic; it points, however, to the direction the development of new structures should follow. The goal ought to be to maintain the strictly sequential handling of task sets imposed by the deadline driven scheduling algorithm as far as possible. For this purpose the processor(s) need(s) to be relieved from the frequent interruptions of the normal program flow caused by external and internal events that need to be handled, although they only seldom lead to an immediate (re-)activation of a task.

3.2. Task Oriented Virtual Storage Management

The deadline driven scheduling algorithm orders the elements of free task sets according to increasing response times. Since the tasks are processed in this sequence, the latter also implies an ordering of the corresponding storage accesses to program code and data by increasing forward distance. This observation suggests that one should base a virtual storage administration scheme on task objects as paging elements. Compared with customary paging schemes, this approach better reflects the software structure, because the task is the basic unit of program flow and related data in real-time environments. Owing to the requirements of typical process control applications, one can assume that the storage demands of tasks are comparatively small. Without restriction of generality this storage size can be bounded, since auxiliary tasks may be defined to cope with the rare cases of larger storage needs. By reason of the fast reaction time to be provided and the generally short execution times of tasks, it is feasible to load the entire task code into main storage each time it is needed. Thus, parts of the available storage will temporarily remain unused, but this is justified by the advantages the method offers and by the still dropping prices for storage blocks.

In the sequel the details of a task oriented virtual storage administration procedure and of an enhanced algorithm will now be described.

The main storage is divided into an area for the supervisor, or only resident parts thereof, and for shared data structures not subject to paging, and into $K \geq 2$ page frames. The size of the latter corresponds to the maximum storage allowance for tasks. Since the identity of the used frames is irrelevant when changing pages, the assignment problem does not arise.

Let now $R_t := \{T_1, \ldots, T_{n_t}\}$ be the list of tasks ready for execution at the time t, which was ordered according to increasing response times by the deadline driven scheduling algorithm. Then, loading the subset $B := \{T_i \mid i=1,...,min(K,n_t)\}$ of R_t into main storage represents the solution of the moving-in problem. When a currently storage-resident task leaves the ready state, its page frame becomes available again. Upon termination it is simply released. Otherwise the page needs to be written back to mass storage. We shall see later, that such a page might remain in main storage when a refined algorithm is employed. A free frame is occupied by the first task in the ready list not yet loaded into main storage. In accordance with the scheduling algorithm, the task T_1 is always being processed. If the initial subset B of R_t varies differently as by termination of T_1 when it is completely carried through, i. e. if an element of B is superseded by another newly arrived one with shorter response time, the replacement problem must be solved. If the cardinality of B before the arrival of the new element was K, this is achieved by removing its final task. The latter possesses the longest forward access distance of all tasks in B. Hence, the replacement algorithm is optimal. Only in rare cases does the processor have to wait for the completion of a page replacement process, because in general there are tasks with tighter deadlines to be executed first. The two above-stated methods for solving the moving-in and the replacement problems, respectively, are look-ahead algorithms representing an application oriented realisation of working sets given in the form of task objects.

Selecting the value 2 for the parameter K may already be quite reasonable, since T_1 is executed essentially without interruptions, in accordance with deadline driven scheduling, and during its processing time the next task T_2 could be paged in. Ideally, the code of T_2 should be available upon completion of T_1. The actual choice of K depends on several factors:

- the number of I/O channels available for paging,
- the transfer time,
- the average execution time of tasks, and
- the frequency of suspension of the running task due to I/O and synchronisation with the possibility to utilise waiting times for the processing of other tasks.

For any given hardware configuration and any task system, an appropriate value for the parameter K can be determined by simulation, taking the minimisation of the waiting time for loading required pages as optimality criterion.

The above-outlined storage administration scheme only takes the ready tasks into account, and in relation to them the replacement algorithm's optimality is to be understood. But for utilisation in a more sophisticated look-ahead algorithm, information is available in process control systems specifying when presently non-ready tasks will be (re-)activated. When the latter event occurs, these tasks may supersede others, that became runnable earlier, from front positions in the ready list implying a change in the elements of the subset *B*. Hence, storage is assigned to the first *K* tasks in a list ordered with regard to ascending deadlines comprising

- ready tasks,
- buffered task activations, and
- the next scheduled activations and continuations of tasks.

The deadlines required for comparison purposes are already known in the two former cases. For cyclically scheduled (re-)activations, they can be calculated from the next critical moments specified by the schedules and the relative time conditions assigned to the pertaining tasks. To facilitate a similar determination of a future task (re-)activation in case of interrupt driven schedules, the user must supply an average occurrence frequency for each of them. Assuming this information to be given, we shall now state the refined look-ahead algorithm for task oriented virtual storage management in the form of a procedure. We commence with describing the data structures involved.

Let for any task known within a real-time application a task control block TCB[I], I=1,...,N, be given. The TCB entries relevant to virtual storage management are the following. The Boolean variable RES indicates whether a task is permanently held in main storage due to a corresponding programmer's request. The task's segment resides in the block with the number PAGE on mass storage, and if it is loaded, the number of the used page frame is stored in FRA. The latter variable contains the value -1 otherwise. Furthermore, in any TCB there are three items of type clock: TCOND, TACT, and TCONT. The first of them states the deadline in case the task is ready. The two others give the task's time conditions relative to the next - not yet buffered - activation and to the forthcoming continuation if the task is suspended. In Table 4 the circumstances are detailed under which these three variables are updated.

Table 4. Updating of the Various Task Deadlines

Event	Assignments
Initial state	TCOND:=TACT:=TCONT:= ∞
Activation buffering	buffer T+TR as TB[*]
Transfer to ready state	TCOND:= first buffered TB[*]
Task termination	TCOND:= first buffered TB[*] or ∞
Annihilation of schedules	TACT:=TCONT:= ∞
Task suspension	TCOND:= ∞
Task continuation	TCOND:=T+TR , TCONT:= ∞
Setting up and fulfillment of activation schedules:	
- start of cyclic schedule	TACT:= start time + TR
- prolongation of cyc. sch.	TACT:=T + interval + TR
- exhaustion of cyc. sch.	TACT:= ∞
- interrupt driven schedule	TACT:=T + INT + delay + TR
Setting up and fulfillment of continuation schedules:	
- temporal schedule	TCONT:= start time + TR
- interrupt driven schedule	TCONT:=T + INT + delay + TR

In the given expressions T stands for the actual time and TR for the relative response time contained in the corresponding activation or continuation statements, respectively. Finally, in the case of interrupt driven schedules, INT stands for the average time interval between the interrupt occurrences, calculated as

$$\left[\sum_i (\textit{mean time between occurrences of } i\text{-th interrupt})^{-1}\right]^{-1}$$

where the summation is extented over all interrupts mentioned in a schedule. Upon setting up such a schedule, only 0.5*INT is to be entered into the corresponding expressions.

After these preparations we can now formulate the virtual storage management scheme as a procedure.

```
proc([1:K] bool) VSADMIN = ([1:K] bool FOC) :
 co The entries of the Boolean array FOC[1:K] indicate whether
   a page frame is occupied and are initialised with false.
   Declaration of auxiliary variables: co;
 int I,J,L,R,S, [1:N] int FI, clock A,Z, [1:N] clock FZ;
 L:=R:=S:=0; A:=INF; co Let INF be a very large constant co;
 for I from 1 by 1 to N do
  if TCB[I].RES then R:=R+1 else
   if TCB[I].TCOND<A then A:=TCB[I].TCOND; L:=I fi;
   Z:=min(TCB[I].TCOND,TCB[I].TACT,TCB[I].TCONT);
   J:=S; S:=S+1;
   while J>0 and Z<FZ[J] do
    FI[J+1]:=FI[J]; FZ[J+1]:=FZ[J]; J:=J-1
   od;
   FI[J+1]:=I; FZ[J+1]:=Z
  fi
 od;
 J:=min(S,K-R); S:=S+1; R:=1;
 if L>0 and J<S then co The ready task with the next deadline is joined with the subset B. co;
  I:=1;
  while I<J and FI[I]≠L do I:=I+1 od;
  if I≥J then FI[J]:=L fi
 fi;
 for I from 1 by 1 to J do
  if TCB[FI[I]].FRA=-1 then
   while R≤K and FOC[R] do R:=R+1 od;
   if R≤K then FOC[R]:=true; TCB[FI[I]].FRA:=R
        else S:=S-1;
            while TCB[FI[S]].FRA=-1 do S:=S-1 od;
            save(TCB[FI[S]].FRA,TCB[FI[S]].PAGE);
            co Write task segment back to mass storage co;
            TCB[FI[I]].FRA:=TCB[FI[S]].FRA;
            TCB[FI[S]].FRA:=-1
   fi;
   load(TCB[FI[I]].FRA,TCB[FI[I]].PAGE);
   co Load task segment from mass storage into page frame co
  fi
 od
```

Some features of high-level process control programming languages such as Pearl can be used to provide directives for the storage management. System data and shared objects with GLOBAL scope should be placed together with the supervisor in the non-paged storage area. The task attribute RESIDENT is to be translated into setting the variable RES of the corresponding TCB. If this attribute is available in a language, the compiler must ensure that the number of simultaneously resident tasks will not exceed a certain fraction of K. The MODULE concept in connection with permanent residency can be employed to gather shared variables and procedures as well as heavily used small tasks in one page. The only feature of the here proposed virtual storage administration scheme not supported by a construct in any available high-level language is the indication of average occurrence intervals for interrupts. But this can easily be achieved by augmenting the interrupt declaration syntax with the optional attribute INTERVAL, which was introduced in section 2.2.

3.3. Implications of the Layer Structure of Real-Time Operating Systems

In this section we want to compare the features of the auxiliary processor outlined above with the general structure of a process control computer operating system as described in [10]. The real-time typical constituents of such a supervisor are interrupt handling, task management, communication, and synchronisation, as well as time administration, input/output routines, and an operator interface. One distinguishes between the nucleus and the shell of a supervisor comprising operating system processes of the first and second kind, respectively. The latter programs are handled in the same way as user tasks under control of the task management, whereas the former are activated by interrupts.

The auxiliary processor introduced here will be dedicated to the execution of these processes of the first kind. Hence, it will be the carrier of the system functions event, time, and task management, communication, and synchronisation. In certain application areas it may be necessary to perform some user event reactions with the same speed as processes of the first kind. These could then be implemented in the unit's secondary level in microprogrammed form. Since in real-time environments there is no intrinsic difference between user and system processes of the second kind, the functions of the supervisor shell, viz. data exchange with peripherals and file management, will be provided in the form of tasks or subroutines running on the (a) general processor. Thus, with regard to [10], the approach proposed in this paper constitutes a physical implementation of the layer model for real-time operating systems, and represents a clear - because physical - separation between nucleus and shell of the program package implemented on a process control computer.

This separation yields a number of improvements as against conventional processor structures. By providing a special device for the handling of all events, unnecessary context-switchings can be avoided and the normal program flow will only be 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. Therefore, the tasks will mostly be executed in a sequential manner, reducing the number of occasions when tasks are hindered to become running due to the observation of deadlock prevention measures. 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 functions of the nucleus to specialised hardware will contribute to enhancing reliability and efficiency essential in real-time applications.

3.4. Outline of the Architecture

In the preceding chapters and sections, with an integral view of real-time computers' application profiles in mind, it was investigated how the underlying hardware had to be structured. Single processor systems cooperating with devices specialised in carrying through operating system nuclei turned out to be the most advantageous approach. Hence, the concept represents an "orthogonal extension" [15] of the classical von Neumann-structure predominantly employed for process control applications. A better term, however, for characterising the proposed alterations is "outboard migration" of operating system support functions, to be carried out reliably, efficiently, and inconspicuously, into specialised hardware and firmware set-ups. Such a migration of functions is feasible, as was shown in [52], since it leaves invariant the execution sequence of a task set imposed by a given synchronisation scheme.

It is the purpose of this section to outline the structure and the functions of such an auxiliary processor. After compiling the services to be provided by the unit, they are assigned to three different reaction levels. These vary with regard to implementation, speed, and complexity. The working methods of the three functional levels are subsequently sketched. The detailed description of the unit's hardware modules and of the algorithms employed, however, will be the topic of the following chapter.

The asymmetric multiprocessor architecture, which is to be introduced here, can either be employed as a stand-alone device or as a node in a distributed system. It consists of one auxiliary processor each and one or more general task processors. It is assumed that the latter cannot be interchanged, because process control computers are usually connected to a specific part of a technical process. Hence, their functions depend on these subprocesses and cannot be moved to another computer. From the viewpoint of a task scheduling algorithm, the different general processors can then be treated as a set of unrelated uniprocessors. The input/output devices of the system are connected to the task processors, whereas all interrupt lines are wired to the auxiliary processor.

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. With presently available hardware, this 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 tasks may prohibit their interruptibility 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. 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.

Table 5. Requirements and Function Assignment for an Auxiliary Processor
1. *Hardware level* - Accurate real-time management based on a high resolution clock - Exact timing of operations as an option - Separate programmable interrupt generator for software simulation purposes - Event representation by storage element, latch for time of occurrence, and counter of lost arrivals - Synchroniser representation - Shared variable representation
2. *Level of primary reaction* - Recognition of events, i.e. of interrupts, signals, time events, status transfers of synchronisers, and value changes of shared variables - Commencement of secondary reactions - Recording of events for error tracking purposes - Management of time schedules and critical moments
3. *Level of secondary reaction* - Response time driven processor scheduling with overload handling - Task oriented virtual storage management - Execution of (secondary) event reactions, especially of tasking operations - Synchroniser management - Shared variable management - Acceptance of requests - Initiation of processor activities

According to the above argument, an auxiliary processor should fulfill the requirements of providing a separate, independently working event recognition mechanism capable of commencing a primary reaction to an event within a predefined, guaranteed short time frame. A means for achieving this is to structure the features of an auxiliary processor, to be described in the sequel, into three levels. The functions the single levels are to provide are compiled in Table 5. In general, it can be said that the unit will execute the nucleus of an operating system developed according to the special needs of the real-time programming language Pearl.

The unit's basic level comprises various specific hardware elements required to fulfill the demands for accuracy, predictability, and speed. All time dependent features rely on an accurate real-time clock. In order to keep the quantisation error and the number of time events to be processed as low as possible, a signal will only be raised when a moment is reached for which a certain action is scheduled. To generate this signal, a register is compared with the actual state of the clock. Additional hardware connected to the clock will facilitate the accurate timing of certain operations and the continuous updating of running tasks' accumulated execution times as well as the residual time spans required to reach their final completion. In the software verification phase external entities entering a program execution need to be simulated. Beside test data, that can be provided on input devices, interrupts have to be generated. By charging this task and other testing features to a separate unit, which can be programmed to produce various external conditions including interrupt patterns and to perform surveillance functions, a simulation process will deliver exact data on the system's time requirements. Since such a device is only needed temporarily, it could be designed as a removable hardware extension. Likewise for verification purposes, but also for the tracking of malfunctions in routine operation, there will be an elaborate hardware module for each event. This comprises a single bit latch for the signal itself and another one for the time of its occurrence and, a counter to record the number of signal arrivals before it is serviced. For the implementation of synchronisers, i.e. of semaphores and bolts, and of variables shared amongst several tasks, the unit will provide dedicated storage space. Connected to each such memory location is a corresponding event module, since the release of synchronisers and the value change of shared variables are also events whose occurrence reactions are scheduled.

The purpose of the auxiliary processor's second level is the servicing of the various events in form of a primary reaction. From the above discussion it already becomes evident what the notion event comprises: interrupts from external sources and from the clock comparator, signals, and the synchroniser and shared variable state transfers. In order to guarantee an upper bound for the reaction time, the recognition of these events is carried through by continuous cyclic interrogation of the corresponding storage elements. This method is similar to the one employed in Programmable Logic Control (PLC) devices, and was already used in the method of synchronous programming [33,37] 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. This implies that the complexity of the functions to be performed by the primary reaction level must be kept low. In the course of the polling process, the arrival time latches and the counters are read out and these data are saved for future reference. Then, the whole module will be reset. The occurrence of a time signal requires more service than that of other events. So, time schedules need to be handled by calculating the next critical moments and by checking if they have been exhausted. After determining the minimum of all these critical points in time, it will be loaded into the clock's comparison register. Upon completion of each interrogation cycle, information is passed over to the unit's level designed for secondary reaction. These data specify the set of schedules actually being fulfilled.

It is obvious from the above that the primary reaction level does not perform all event servicing functions. Instead, only fast event recognition and closely connected operations are carried out here. The reason, why the scheduled event reactions are not immediately initiated, is that these tasks still need to be submitted to a feasible processor assignment scheme. Thus, the complexity requirements for the processor working on this level are quite moderate. Since it also does not need to be freely programmable, it can be realised as a rather simple, fully microprogrammed device guaranteeing a high speed of operation. The separation of the operating system kernel functions into a set of rather simple ones, requiring fast reaction, and another set, whose elements have higher complexity, corresponds to the layer structure of a nucleus as discussed in [10]. Therefore, these functions are assigned to a fast, but simple primary reaction device and a more complex secondary reaction processor.

It has already been stated that fulfilled schedules are notified to the auxiliary processor's highest level. Here those tasks and other operations associated with them will be determined. Subsequently, the operating system's internal event reactions, especially tasking operations, will be executed. In the course of task activations and continuations, the response time driven scheduling algorithm will be called to determine if the new free task set can be processed, meeting all given deadlines, and to also define the sequence of processor assignments. In case an overload situation is recognised, correspondingly marked tasks will be terminated and other ones coping with this event will be activated. In connection with the processor scheduling, a task oriented virtual storage administration scheme as described above is carried through. Furthermore, the secondary level manages synchronisers and shared variables. So, it concurs with synchronisation requests when they cannot be granted immediately. Then resumption schedules for the suspended tasks need to be handled. When a new value has been written into a shared variable, the unit checks whether it now fulfills a given relation to a prescribed one, in which case scheduled operations are initiated. Finally, on the secondary level all communications with the remaining system elements of higher complexity than event recognition are processed. Parameters describing requests are accepted and inserted into the corresponding internal data structures. As a result of executing requested operations, processor activities may be terminated and others activated.

There are many possibilities to realise the data exchange between the general processor(s) and the auxiliary processor as well as within the latter. 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(s) and the primary level to the secondary level. The processing of the data read out of the second one has the precedence to guarantee the system's reaction time. For the storage of system and internal data, e.g. task control blocks, and shared variables the unit provides a common memory area directly accessible by the different system components. The shared objects' integrity is protected by associated synchronisers located in the unit, and the access rights are being surveyed by an appropriate hardware mechanism. In case the device works in a single processor environment, the common storage could be implemented as a two-port memory or as a certain block of main storage dedicated to communication purposes. Then the auxiliary processor itself had to be realised with an DMA interface. When in cooperation with a multiprocessor system, which may also be locally distributed, it should best be integrated into the message switching unit. Generally, the leading idea for selecting a certain means of data exchange with the auxiliary processor must be to achieve high speed combined with simplicity in order not to produce new bottle-necks. The diagram given in Figure 4 specifies the general set-up of the here proposed architecture and all connecting lines used to transfer data, control information, and signals.

3.5. Comparison with other Architectures

Now, for comparison purposes, we shall consider some different approaches, which were found in the literature and which are aiming into similar directions as the here proposed design, viz. Honeywell series 60 level 64 computer, IBM System /38 and SWARD, Bellmac-32, Mesa, Intel iAPX-432 and iAPX-86/88, Control Data Corporation 6000, 7600, and Cyber series, Cray-1, Texas Instruments ASC, Symbol processor, Bell Laboratories' Signaling and Scheduling Processor, and the supplementary processor for operating system functions described in [51,53]. These examples can be subdivided into two groups, either providing operating system support as architectural features or even possessing separate processors to accommodate the operating system or essential parts thereof. In these structures similar objectives have been realised. It is investigated to what extent they match the here proposed design.

An early representative of the former group is Honeywell's series 60 level 64 computer [8]. Here the machine supports the notion of processes with special instructions and by performing the context-switching operation when the processor is assigned to a different process. Furthermore, a semaphore mechanism is available for synchronisation purposes. Specialised on data base applications, the IBM System /38 [49] manages tasks according to priorities by corresponding hardware features and microcode. Communication and synchronisation between tasks is accomplished by a queuing structure.

Two more recent developments, the Bellmac-32 [11] and the Mesa [28] processors, also exhibit process oriented architectures. To organise the synchronisation of concurrent processes and of shared resources,}i monitors and condition variables are implemented in the latter. The processor is dispatched to the ready processes according to priorities. The instruction set contains elements for moving the process state blocks between various queues maintained in the system and associated with monitors, condition variables, and the ready state. An interrupt mechanism and a process switching support is also found in the

Bellmac-32. Furthermore, this approach provides hardware support for task rendezvous and assists exception handling. The scheduling, however, is carried through by the software level of the operating system.

The developments SWARD of IBM and iAPX-432 [39] of Intel are two further examples for supporting operating system features architecturally. For the former these are process management, synchronisation, communication, storage administration, and dispatching. Process interaction in the form of the Ada language's rendezvous is realised by port objects. In general, however, the architecture is neither oriented towards a certain language nor to real-time applications as is obvious by the lack of interrupt facilities. The concept of the iAPX-432 shows many resemblances with the one of SWARD, but strongly supports Ada language constructs such as expecting one of several messages by a process. Furthermore, this design integrates several real-time features. So objects can be locked and there are indivisible storage access operations. Communication is accomplished by send/receive queues managed according to the first-in-first-out, priority, or deadline policies. The low-level process scheduling is performed by hardware and firmware. Here the deadline algorithm can be employed, too. The timing facility relying on a hardware support for delays is not very accurate. Interrupts are indirectly handled using the message mechanism.

As final representatives of this group we consider Intel's iAPX-86/88 systems [23], where the functions of a real-time operating system nucleus are implemented in hardware and in a separate Operating System Firmware component, that also contains the interrupt logic and simple timing facilities. For synchronisation purposes the mailbox and semaphore concepts are realised. The system supports multitasking applications and schedules the task processing according to priorities. The architecture of the two microcomputers was developed aiming towards embedded systems.

We now turn to the second group of approaches where operating system functions have been migrated to separate processors. An early example for this is Control Data Corporation's 6000 series of computer systems [49]. A very fast central processor unit is supported by ten independent peripheral processor units (PPU's) mainly dedicated to perform I/O operations. One PPU, however, accommodates the operating system and thus controls the entire system. The leading idea for this approach was the optimal utilisation of the CPU for arithmetic purposes in a multiprogramming batch environment. The succeeding models of the 7600 and Cyber series and descendants like the Cray-1 or Texas Instruments' ASC have comparable architectures [49].

Another approach providing no support for real-time facilities is the Symbol computer [39] being controlled by a system supervisor. This is an interrupt driven master processor performing work queue administration, paging, and control table management.

In [4] a transaction oriented multi-microprocessor architecture with hardware support for communication and scheduling to be applied as a network node is described. Here the operating system functions scheduling, synchronisation, and interprocess communication are comprised in a separate unit, the Signaling and Scheduling Processor (SSP). Furthermore, the SSP handles all internal and external communications, the dispatching, and the resource assignment. All signaling is carried through using hardware queues. Larger data sets, however, are exchanged via a shared memory. The architecture enhances system performance by allowing its execution units to work without interference.

The approach, which appears to be the closest match with the here proposed ideas, is Tempelmeier's supplementary processor for operating system functions [51,53]. It is intended to increase the performance of real-time computers by migrating most functions of the operating system kernel to a separate processor. The immediate interrupt recognition and reaction, however, is still located in the general processor, which mainly performs the user tasks and operating system processes of second kind. It was shown [52,54], that the architecture yields a considerable improvement of task response times and interrupt reaction times. This was achieved by providing parallel processors for user tasks and operating system functions, and by almost reducing to zero the time intervals for which external interrupts need to be disabled for purposes of ensuring the integrity of operating system data structures. The architecture is also an implementation of the layer model for real-time operating systems [10], and achieves a simple operating system structure by the clear separation from the user tasks.

For completeness purposes, we shall finally consider some other examples of outboard migration found in real-time systems where the guide-line has been the support of special application oriented features, whereas the operating system remained in a classical von Neumann CPU. So, high computational performance shall be achieved by so-called real-time supersystems. In order to cope with the arithmetic requirements of certain application areas such as signal and image processing in the framework of embedded and satellite or missile borne systems, respectively, distributed architectures have been considered [7,50]. These systems are tailored to perform their evaluations within reasonable time-limits under the conditions imposed by the locations of deployment. Hence, the architecture of general purpose process control computers, which support the observation of strict deadlines, is not the scope of this development.

In a similar manner, special devices, cp. e.g. [42], have been built for carrying through certain time-critical algorithms. By implementing them in firmware the desired speed increases were achieved. These units are attached as front-end processors to conventional mini- or microcomputers, being too slow to cope with the corresponding functions themselves.

The above considerations reveal that various approaches have already been made to support operating system concepts by architectural measures. As far as real-time features are concerned, it can be concluded that only some of those have been migrated which were known from previous operating systems. No attempt, however, has been made to utilise hardware and firmware facilities for the implementation of typical hard real-time support features that cannot otherwise be realised. Although architectures were developed to meet some language constructs, especially of Ada, there was no integral effort yet to built a system according to the special requirements of a process control language suited for the application engineer and of a corresponding operating system. That is the motivation for the attempt aiming into this direction which is to be worked out in detail in the remainder of this paper.

4. Design of an Auxiliary Processor

In the previous chapter, the concept of an auxiliary processor was described whose function it is to accommodate the kernel, i.e. the time-critical part, of a real-time operating system especially tailored for the real-time programming language Pearl as extended in section 2.2. Such a unit shall cooperate as system controller with one or more conventional processors in process control computers employed in hard real-time environments. The device is composed of three levels, which perform the functions as assigned in Table 5. Partly, these functions are called from the application programs and thus also constitute the operating system processor's user interface. The subject of this chapter is to describe in detail the hardware modules of the proposed auxiliary processor and the algorithms executed in two separate subprocessors as primary and secondary event reactions.

4.1. Description of the Hardware Composition

The single hardware components of the auxiliary processor are all grouped around a common internal dataway, the bus. The access to it and further on to the various storage elements is synchronised by a bus controller. The latter coordinates the bus activities of the primary and secondary reaction processors as well as of the external general processing unit(s) and a peripheral for storing protocol data, and surveys the distinct access rights. Since a bus controller is a widely used hardware component, it does not require further discussion here.

In order to prevent excessive overhead connected with writer/reader synchronisations, additional data communication channels are provided for the transmission of operation parameters from the primary level and the attached general processor(s) to the secondary level in the form of first-in-first-out memories.

In the sequel we shall now specify the different functional units arranged along the dataway.

4.1.1. Time Dependent Elements

All time related features, as shown in Figure 5, are based on a quartz controlled clock with a separate current source to maintain a correct real-time even if the system power supply breaks down. The frequency generator provides three different clock pulse trains f, f', and f'', the first of which normally triggers the clock. This is realised in form of a divided, individually addressable counter yielding day, hour, minute, second, and an appropriate subdivision thereof, selected to keep the quantisation error negligibly small. 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. It is fed into a storage module that will be detailed later in the context of recognition devices for events. The algorithms performed as reaction to this signal are then the subject of the next section.

The purpose of a second register/comparator combination is to produce continuation signals for transmission to the external processor(s), where they start the execution of operations at precisely given points in time. The continuation signal originates from gating the comparator output with the one of an 1-bit storage location that can be set and cleared via the bus. The comparator pulse automatically loads the register with a bit pattern equivalent to infinity and also resets the enable flip-flop if propagating through the gate. Before a processor requests the exact timing of an operation, it performs all necessary preparations and then transfers the corresponding task into a wait state by disabling its system clock. This takes place not more than *eps* time units before the critical moment, whose time is loaded into the comparator register together with setting the masking flip-flop associated with the pertaining processor. As can be seen from Figure 6, the continuation signal finally resumes the task execution by enabling the clock frequency. If *eps* is an installation parameter, there will never be more than one pending operation scheduled for a precise time. The constance of *eps* also avoids the necessity of mutual synchronisations in case a multiprocessor is being controlled.

In order to obtain exact results on program performance and behaviour, the software verification process is architecturally supported by a testing module. Since this is only temporarily needed during the lifetime of a real-time computer system, it can have the form of a plug-in module. The device consists of an interrupt generator and of a further clock comparator/register combination. For each interrupt source to be simulated, the unit handles a time schedule. Thus, the simulation results will not be falsified by the additional workload that would be imposed on the auxiliary processor if it had to take care of these schedules, too. Upon the occurrence of the comparator signal, the unit triggers the associated interrupts via the internal dataway. Then the register is loaded with the next critical moment. The interrupt generator's working method does not need to be further described here, because it is quite similar to the primary reaction to time events, that will be discussed later. However, two functions modifying the system clock during the test phase must be mentioned. If required by the simulation program, the clock can be stopped by disabling the frequency generator's three output lines. Furthermore, the unit controls the multiplexer, feeding either the frequency *f* or *f'* into the clock. Normally, *f* is put through. But, if no task is being executed during a simulation run, the clock may be triggered by the considerably higher frequency *f'*, in order to reduce the corresponding time requirements. The occurrence of any of the three comparator signals will automatically restore the original multiplexer setting. The interrupt generation unit detects an idle state of the external processor(s) by questioning the processor state flip-flop(s). Their contents gate the frequency *f* counting the running tasks' accumulated execution times up and the residual times down, which are required before the tasks will terminate. These counters are necessary as hardware support for the implementation of feasible scheduling algorithms, viz. the deadline driven one for single processor systems and the one first stated in [27] and modified in [25] for multiprocessors. As many flip-flop/counter combinations must be provided as there are processors in the system, since this is the maximum number of running tasks.

4.1.2. Event Recognition Modules

In the course of this hardware definition, we distinguish between four kinds of events for which recognition modules need to be provided:

- the time signal,
- external interrupts,
- status transfers of synchronisers, and
- the assumption of certain relations of shared variables to given values.

The general structure of such a module is displayed in Figure 7.

A signal generated by one of the different sources, whose internal composition will be discussed later, sets the event storage flip-flop and triggers the counter of event occurrences. Furthermore, the actual time is latched when the output of the former turns high. Before flip-flop and counter will have been serviced and cleared, subsequent incoming signals will only increase the value held in the counter. Its purpose is to also recognise those signals arriving so closely after each other that they cannot be serviced. This feature will mainly be used in the verification phase of an application for detecting performance bottlenecks.

The source of the time signal is the first comparator of a register with the real-time clock as shown in Figure 5. Hence, its output is simply connected with a storage element and a counter.

Interrupts may be raised by external devices or internally for simulation purposes, e.g. by the Pearl statement TRIGGER. Since the latter can be accomplished via the internal dataway, an or-composition of this with an externally connected signal line serves as signal source in case of interrupts.

Accessible to all system components, the auxiliary processor provides a common storage area for shared and for synchronisation objects. The synchroniser type semaphore can be reduced to the type bolt and, therefore, only one implementation needs to be considered. A bolt is represented as a pair of integer variables, the first of which expresses the state of the object. The second one gives the bolt's enter range, or the maximum number of requests that can be granted at any one time in case of a semaphore representation, respectively. According to the hardware implementation of a bolt as shown in Figure 8, two signals are generated. They are raised when the status assumes the respective values *0* and *enter range-1*. Thus, two event recognition modules are associated with each synchroniser. For the fast execution of locking operations the status signals are also accessible as a bit vector.

For the description of the signal generation facility connected to shared variables we refer to Figure 9. The addresses of those shared objects are stored into the address variables for which the assumption of certain states is being awaited. When in the course of a writing access to the common storage area the data address is available on the bus, what can be detected by observing the Read/Write and Address/Data bus lines, respectively, the comparators match it with the contents of the address registers. In case of equality, an event is signaled to the associated recognition module. All further operations will be performed on the primary reaction level.

4.2. Primary Event Reaction

Before we can commence discussing the algorithms running in the primary reaction processor handling the various kinds of events, the time management requires some attention, since it performs a wider and more specific range of operations. We begin with describing a general form of time schedules implying a considerable simplification, before turning to the data structures and the procedures, which will be formulated in the process control language Pearl.

4.2.1. Representation of Time Schedules

Since the previously introduced hardware will allow a precise and reliable time management, the distinction between the Pearl language elements ALL and EVERY and between UNTIL and DURING may be dropped. Hence, the various forms of time schedules can be converted to a common one reading as

AT {clock-expression | [interrupt-expression]+duration-expression1}
EVERY duration-expression2 DURING duration-expression3

where [...] stands for the point in time when the specified interrupt occurs and the duration-expressions yield non-negative values. This language construct is also representable by a tripel

$T(n)=(t(n),dt,rt(n))$, $n\geq 0$, with
$t(n+1):=t(n)+dt$, $rt(n+1):=rt(n)-dt$, $n\geq 0$,
t(0)=clock-expression | [interrupt-expression]+duration-expression1
dt=duration-expression2, and
rt(0)=duration-expression3.

Upon reaching the time $t=t(n)$, $n\geq 1$, the two calculations in the above equation will be carried out in parallel. The schedule is worked off if $rt(n+1)<0$ holds. Otherwise, $t(n+1)$ will be handled as a future critical moment.

4.2.2. Algorithms and Data Structures of the Time Management

Let $n>0$ be a configuration parameter. The just introduced time schedule tripels are stored in the 1-dimensional data structure $T=\{(t(i),dt(i),rt(i)) \mid i=0,...,n\}$. We lay down that $dt(i)=0$ shall mean that the corresponding $T(i)$ contains no valid schedule. Then, the set of critical moments for which certain operations are scheduled is given in unordered form by $\{t(i) \mid dt(i)=0 \text{ and } i=0,...,n\}$. The stati of the elements of the four 1-bit-arrays p, q, g, and e of length n+1 signify:

p(i)= L, T(i) will be fulfilled with the next time signal and t(i) is already the next but one critical moment of T(i),
0, otherwise

q(i)= L, T(i) will be fulfilled with the next but one time signal,
0, otherwise

g(i)= L, T(i) is associated with an operation to be timed precisely,
0, otherwise

and

e(i)= L, T(i) is exhausted,
0, otherwise.

Finally, let zk1 and zk2 be the next and the next but one, respectively, critical points in time of all schedules. The value of the variable zk1 is for a time t<zk1 equal to the one held in vr, the clock's first comparator register.

Upon reaching the time zk1, i.e. with recognising the time signal, the following procedure will be performed. It first updates the comparison register of the clock. By saving p as vector r, the information is provided which the event processing algorithm needs to initiate those operations being associated with the schedules now fulfilled. The exhausted ones of the latter are marked. Then, the schedules are evaluated that will be fulfilled at the next critical moment, to yield new next but one points of time. Eventually, the actual content of the vector q and the minimum of all valid t(i), i=0,...,n, are determined, in order to be able to redefine vr immediately after the occurrence of the next time signal. This proceeding minimises the delay between the comparator signal and the reloading of the corresponding register.

```
update: PROCEDURE;
    vr:=zk1:=zk2; r:=p; e:=(n+1)'0'B;
    FOR i FROM 0 BY 1 TO n REPEAT;
     IF p(i) AND rt(i) LT 0 THEN
      dt(i):=0; e(i):='1'B;
     FIN;
    END;
    p:=q;
    FOR i FROM 0 BY 1 TO n REPEAT;
     IF p(i) THEN
      t(i):=t(i)+dt(i); rt(i):=rt(i)-dt(i);
     FIN;
    END;
    CALL minimum;
    END;
```

with

```
minimum: PROCEDURE;
    zk2:=infinity; q:=(n+1)'0'B;
    FOR i FROM 0 BY 1 TO n REPEAT;
     IF rt(i) GE 0 AND dt(i) GT 0 THEN
      IF t(i) LE zk2 THEN
       IF t(i) LT zk2 THEN
        zk2:=t(i); q:=(n+1)'0'B;
       FIN;
       q(i):='1'B;
      FIN;
     FIN;
    END;
    END;
```

where "infinity" stands for an appropriately selected large constant and "(repeat) bit-literal" serves as abbreviation for an explicit loop when setting a bit array. The complexity of the above procedures is of the order O(n+1). It is possible to speed them up by performing the operations of the loops in "update" in parallel on n+1 array components. In order that the routines can work properly, several variables need to be initialised as follows:

```
init: PROCEDURE;
    FOR i FROM 0 BY 1 TO n REPEAT;
     dt(i):=0; p(i):=q(i):=g(i):=e(i):='0'B;
    END;
    vr:=zk1:=zk2:=infinity;
    END;
```

For deleting a time schedule T(i), $0 \le i \le n$, only three assignments are necessary:

```
delete: PROCEDURE(i FIXED);
     dt(i):=0; p(i):=q(i):='0'B;
     END;
```

The price for this extremely simple solution of complexity O(1) is that possibly time signals may be generated, although no element of p or q, respectively, is set.

Before the secondary reaction level requests the primary one to perform the inverse operation of inserting a new schedule S=(ts,dts,rts,ex) into T, it has checked the validity of the transmitted parameters and has determined the index i of an unoccupied array element where the components of S will be stored. Then, according to the value of ts, the variables vr, zk1, zk2, p, and q, respectively, will if necessary be updated.

```
insert: PROCEDURE(ts CLOCK, (dts,rts) DURATION, ex BIT, in FIXED);
     t(in):=ts; dt(in):=dts; rt(in):=rts; g(in):=ex;
     IF ts LT zk1
      THEN vr:=zk1:=ts;
         FOR i FROM 0 BY 1 TO n REPEAT;
          IF p(i) THEN
           t(i):=t(i)-dt(i); rt(i):=rt(i)+dt(i);
          FIN;
         END;
         p:=(n+1)'0'B; p(in):='1'B;
         t(in):=ts+dts; rt(in):=rts-dts;
         CALL minimum;
      ELSE
       IF ts EQ zk1
        THEN p(i):='1'B; t(i):=ts+dts; rt(i):=rts-dts;
           CALL minimum;
        ELSE
         IF ts LT zk2
          THEN zk2:=ts; q:=(n+1)'0'B; q(in):='1'B;
          ELSE
           IF ts EQ zk2 THEN q(in):='1'B; FIN;
         FIN;
       FIN;
     FIN;
     END;
```

In case the loop in the above instruction sequence can be converted to parallel operations, e.g. for determining which components of p have the value '1'B, the insert routine's complexity can be reduced from O(n+1) to O(1). Since the THEN clauses are partly identical, there are some possibilities for code compression desirable for a microprogrammed implementation. When inserting or deleting schedules the actual time and the pertaining parameters have finally to be written into the protocol file.

4.2.3. Algorithms and Data Structures of the Event Management

As already previously pointed out, the set of events to which the unit must react is composed of
- the time signal z,
- the external interrupts u(i), i=1,...,card(U), under observation of associated masking bits m(i),
- synchronisation events s(i), i=1,...,card(S), and
- the assumption of certain relations of shared variables to given values v(i), i=1,...,card(V),

i.e. $E=\{z\}\cup U\cup S\cup V$ with m=card(E)=1+card(U)+card(S)+card(V). At any point in time there is a set of scheduled activities $A=\{a(i) \mid i=0,1,...\}$. Since a time schedule may exist for each $a\in A$, the maximum possible number of them to be handled equals the dimension n+1 of T described in the last section. The elements of A may be
- tasking and synchronisation operations in the sense of Pearl,
- the start of program units reacting to ON-exceptions,
- the start of the alternatives in EXPECT-statements,
- the start of routines handling the exceptions of temporal surveillance features,
- the call of the processor scheduling strategy, and
- the start of routines when shared variables have assumed given value relations.

Between the two sets E and A there exists a relation

$R\subset E\times A$: eRa iff the activity a must be executed upon the occurrence of the event e, $e\in E$ and $a\in A$.

In the case of e=z only those elements of $R\subset\{(z,a(i) \mid i=0,...,n\}$ may be considered for which currently r(i)='1'B holds, i=0,...,n. For use in the event reaction algorithm to be stated in the sequel, the relation R will be implemented as a two-dimensional array "rel" of bits. This primary reaction is activated by each of the clock pulses arriving with the frequency *f'*, being an appropriate part of the basic system clock *f*. Hence, the recognition time of an event does not exceed 1/*f'*. In a loop polling all m event recognition modules it is detected whether and which events have occurred. First, the corresponding time latches and counters are read out and protocoled together with the event identification. Then, the event storage flip-flops and the occurrence counters will be reset:

```
protocol: PROCEDURE(i FIXED);
    WRITE i, tl(i), cr(i) TO protocol buffer;
    ff(i):='0'B; cr(i):=0;
    END;
```

For the time signal the procedure "update" will subsequently be executed. It provides the vectors r and e stating which schedules are presently being fulfilled together with the exhausted ones. After recording them, the componentwise logical conjugation of the array r with the first row of the relation R will be carried out to determine those time dependent activities that must now be performed. With the result the vector g is masked to mark those activities requiring precise timing. Interrupts are only be considered if an associated masking bit is set, which will also be protocoled. The masking data do not require special flip-flops for storage. Instead, they can be held in the unit's working storage area. The primary interrupt handling concludes with the Boolean disjunction of the array r with the row of R associated with the event just considered. Thus, those activities are determined that must be subsequently carried out by the secondary reaction processor. Synchronisation and shared variable events are treated in an analogous manner. In the latter case, however, it will be immediately examined whether the new value, that was just stored into a shared variable, fulfills a given relation to a prescribed quantity. This will be accomplished by calling the procedure "compare". Its parameters are an index and the arrays ad, rl, and v representing the shared variables' addresses, the arithmetic or logical relations, and the comparison values, respectively. In case the Boolean procedure detects that the shared variable in question has assumed the expected value relationship, a message is written into the protocol file and r is appropriately updated. Linked to the first shared variable are the insertion and deletion operations of time schedules, that are handled as described in the next section. After completion of a polling cycle, the vector r identifying the elements of A to be performed is written together with the actual time to the secondary level processor. The task "primary reaction" has a complexity of the order O(n+m+1).

```
WHEN f' ACTIVATE primary reaction;

with

primary reaction: TASK;
 r:=(n+1)'0'B; t:=clock;
 IF ff(1)
  THEN CALL protocol(1); CALL update;
      WRITE r,e TO protocol buffer;
      r:=r AND rel(1,0:n); gm:=g AND r;
 FIN;
 FOR i FROM 2 BY 1 TO UPB u +1 REPEAT;
  IF ff(i)
   THEN CALL protocol(i);
       IF mk(i)
        THEN WRITE mk(i) TO protocol buffer;
            r:=r OR rel(i,0:n);
       FIN;
  FIN;
 END;
 FOR i FROM UPB u +2 BY 1 TO UPB u +UPB s +1 REPEAT;
  IF ff(i)
   THEN CALL protocol(i); r:=r OR rel(i,0:n);
  FIN;
 END;
 i:=UPB u +UPB s +2;
 IF ff(i)
  THEN CALL protocol(i);
      IF CONT ad(1) EQ 1
       THEN CALL insert(ts,dts,rts,ex,in);
           WRITE ts,dts,rts,ex,in TO protocol buffer;
       ELSE
        IF in GE 0 THEN
         CALL delete(in); WRITE in TO protocol buffer;
        ELSE
         FOR in FROM 0 BY 1 TO n REPEAT;
          IF dv(in) THEN
           CALL delete(in); WRITE in TO protocol buffer;
          FIN;
         END;
       FIN;
      FIN;
      CONT ad(1):=0;
 FIN;
 FOR i FROM UPB u +UPB s +3 BY 1 TO
 UPB u +UPB s +UPB v +1 REPEAT;
  IF ff(i)
   THEN CALL protocol(i); k:=i-UPB u -UPB s -2;
       IF compare(k,ad,rl,v)
        THEN WRITE ad(k),v(k) TO protocol buffer;
            r:=r OR rel(i,0:n);
       FIN;
  FIN;
 END;
```

```
WRITE t,r,e,gm TO secondary level;
/* Push data into input fifo of the secondary level processor. */
END;
```

and

```
compare: PROCEDURE(k FIXED, (ad,rl,v) ( ) FIXED IDENTICAL) RETURNS(BIT);
    CASE rl(k)
     ALT b:=CONT ad(k) EQ v(k);
     ALT b:=CONT ad(k) NE v(k); ...
     /* and other appropriate comparisons */ ...
     OUT b:='0'b;
    FIN;
    RETURNS(b);
    END;
```

4.2.4. Implementation of Further Features

As a means of internal event communication Pearl provides objects of type SIGNAL. In the above described hardware set-up no special support for them has been assigned, because signals can be easily implemented employing the shared variable feature. Thus, a signal will be associated with a shared object of type BIT. Raising or INDUC(E)ing it then reduces to the operation of setting the variable, and the signal reaction will be scheduled as an activity to be commenced when the corresponding variable assumes the value '1'B.

As the program given in the last section shows, the insertion and deletion of time schedules into the internal data structure is also linked to a shared variable assuming certain values. When the latter has been set by the secondary level processor to 1 or 2 for insertion or deletion, respectively, during the next event polling cycle the corresponding parameters are fetched from locations in the common memory, the requested operation is performed, possibly even several times in case of a delete, and the shared variable is finally reset to zero.

For certain applications it may be necessary to know in the user program which particular events gave rise to an activity. Information on the just occurred events can be provided by copying the stati of the flip-flops ff into a bit vector during execution of the task "primary reaction" and by writing it to the secondary level, too.

The verification of the reliable software performance, especially with regard to the temporal requirements, is an aspect of fundamental importance for real-time data processing. That is the reason why all data necessary to reconstruct the occurrence of events and the connected operations are protocoled by the above stated procedures. The thus recorded information is essential during the implementation and acceptance phase of an application, but also valuable for analysing later software malfunctions. Physically the protocol information is written into a buffer being part of the common memory. As already mentioned, there is a separate device independently controlling the formatting into readable form and the data transmission to a mass storage medium, when the buffer is filled to a certain extend. Further duties of this device could be selecting only a specified subset of all protocol data, when they are not entirely needed any more.

4.3. Secondary Event Reaction

Proceeding from the requirements of the high-level language Pearl, first the individual functions of the secondary reaction level processor will be derived. Then, its main control programs for accepting operation parameters and initiating the requested services will be given. After describing the single entries of the task control blocks managed by the unit, its various functions will be discussed in detail and stated in form of Pearl routines. Their formulation, however, will involve only low-level features suggesting a microprogrammed implementation.

4.3.1. Compilation of Functions to be Performed

In Pearl, as well as in other real-time programming languages, concurrent processing is realised by the task concept. For the operating system nucleus, and hence for the corresponding auxiliary processor, this implies that the following functions must be provided:
- initialisation of a control block for each task upon elaboration of its declaration,
- scheduling and execution of the tasking operations ACTIVATE, TERMINATE, normal termination of a task, PREVENT, SUSPEND, CONTINUE, and RESUME,
- buffering of task activations,
- scheduling and execution of task resumptions in connection with synchronisation operations and EXPECT statements,
- prevention of EXPECT schedules, and finally
- scheduling and initiation of ON reactions.

Since there will generally be more than one ready task, the processor dispatching must be organised. To this end, the feasible deadline driven scheduling algorithm is called in the course of the above mentioned operations' execution. This processor assignment scheme handles also overload situations as early as possible and calls for its part the virtual storage management. The latter employs a look-ahead algorithm as described in section 3.2, allocating page frames for the code of complete tasks and utilising information provided by the response time driven dispatcher and extracted from the schedules of task (re-) activations. All these operating system features are controlled by the language elements introduced into Pearl in section 2.2. The same holds for the precise timing of tasking operations, requiring some additional service from the secondary level processor. The assumption of certain value relations by shared variables is a class of events occurring in schedules. The primary reaction level in connection with special hardware modules performs the necessary surveillance of the shared variables. Providing parameters to this feature and controlling it is a further function to be carried through by the auxiliary processor's highest level.

In addition to the above mentioned functions directly initiated from language constructs in the application programs, there are other ones supporting the execution of the former. In detail, these comprise the interpretation and processing of operation parameters, the validation of time schedules, and the assignment and pre-emption of the general processor. The most complex task, however, is the management of schedules, especially of those temporal ones whose initial times depend upon interrupts, and of their relation to activities to be performed when the corresponding events occur.

4.3.2. The Control Programs

We begin our considerations of the procedures executed by the auxiliary processor's secondary reaction level (SRL) with introducing some data structures and stating two routines accepting service request parameters and initiating their elaboration.

As already mentioned earlier, operation parameters are transmitted from the attached general processor(s) and the unit's primary reaction level (PRL) via two first-in-first-out memories to the SRL in order to prevent excessive overhead connected with writer/reader synchronisations. For the purpose of guaranteeing the system's high reaction speed accepting PRL data has the precedence. The latter specify the activities associated with the just occurred events. These activities are represented by one or more parameter sets for the individual SRL functions stored in the array b. The various objects involved - which were partly already introduced and discussed in section 4.2 - and their interdependence are displayed in Figure 10. The parameters arriving from the general processor through the corresponding fifo are first brought into a temporary buffer for further treatment. Each parameter set consists of a word specifying the function to be performed followed by data words required by this routine. Their number may be derived from the function identifier by calling the procedure "parno".

We shall now state the control programs in an implementation oriented form. In order to avoid context-switching operations in connection with processing the data originating from the two fifos, SRL is equipped with two register sets. If the one associated with PRL is active, then the data-available-interrupt of the corresponding fifo is disabled. It will be enabled upon switching to the other register set. The main task cyclically calls the virtual storage administration procedure "vsadmin", which was stated in section 3.2. The parameter of the latter is the Boolean array foc, whose entries indicate whether the corresponding page frames are occupied. Initially, the elements of foc are cleared. The procedure call "protocol" appearing at various locations in the sequel serves as an abbreviation of writing appropriate data into the protocol file.

```
MODULE(SRL control programs);
 PROBLEM;
  prl service: TASK PRIORITY 1;
         DISABLE prl fifo data available interrupt;
         Switch context to register set 1;
         WHILE prl fifo data available REPEAT;
          READ t,r,e,gm FROM prl fifo;
          CALL protocol;
          FOR i FROM 0 BY 1 TO n REPEAT;
           IF r(i) THEN
            j:=1;
            WHILE b(j,i) NE NIL REPEAT;
             Execute function identified by b(j,i);
             j:=j+parno(b(j,i));
            END;
           FIN;
          END;
         END;
         Switch context to register set 2;
         /* Thus the infinite loop in the other task is continued. */
         ENABLE prl fifo data available interrupt;
         END;
  main: TASK PRIORITY 2;
         WHEN prl fifo data available interrupt ACTIVATE prl service;
         ENABLE prl fifo data available interrupt;
    wait: WHILE processor fifo empty REPEAT;
          CALL vsadmin(foc);
         END;
         READ tb(1) FROM processor fifo;
         j:=parno(tb(1));
         FOR i FROM 2 BY 1 TO j REPEAT;
          READ tb(i) FROM processor fifo;
         END;
         Execute function identified by tb(1);
         CALL protocol;
         GOTO wait;
         END;
MODEND;
```

4.3.3. Description of the Task Control Blocks

It was already mentioned that upon declaration a control block is initiated for each task. For this purpose an array tcb(1:cb), cb≥1, is allocated in the common storage consisting of records as defined in Table 6.

Table 6. Task Control Block Layout

Entry	Data Type	Initial Value	Meaning
tid	CHARACTER(8)	d	Task identifier
res	BIT	d	'1'B, if task declared with RESIDENT attribute
keep	BIT	d	'1'B, if task declared with KEEP attribute
trsp	DURATION	d	Task's response time
trun	DURATION	d	Task's maximum run time
onm	(1:m) BIT	d	Mask of signals occurring in task's ON statements
fsa	FIXED	0	Continuation address after completing ON reaction
sigid	FIXED	0	SIGNAL identification for ON reaction
aon	FIXED	l	Start address of general ON reaction
sta	FIXED	l	Task code start address
page	FIXED	l	Page number of task code on mass storage
frame	FIXED	-1	Number of page frame where task resides; -1 otherwise
reg	(0:creg-1) FIXED	0	Register status of processor
bolt	(1:cs) BIT	zero	'1'B, if bolt seized by task
tl	DURATION	0	Maximum residual run time
rt	DURATION	0	Accumulated run time
tcrit	CLOCK	∞	*Moment of next exactly timed operation; ∞ otherwise*
int	DURATION	∞	Average activation interval of interrupt driven schedules
tcond	CLOCK	∞	Deadline if task is ready; ∞ otherwise
tact	CLOCK	∞	Time condition relative to next not yet buffered activation
tcont	CLOCK	∞	Time condition relative to forthcoming activation
ba	FIXED	0	Number of buffered activations; 0≤ba≤mba
ring1	FIXED	mba-1	1. Index for administration of circular buffer of buffered activations
ring2	FIXED	mba-1	2. Index for administration of circular buffer of buffered activations
ta	(0:mba-1) CLOCK	∞	Time condition
using	(0:mba-1) FIXED	0	Pointer to semaphore mentioned in USING clause
ato	(0:mba-1) CLOCK	∞	Timeout condition of USING synchronisation
alt	(0:mba-1) FIXED	0	Tcb index of alternative task mentioned in USING clause
opt	FIXED	0	Parameter for synchronisation resume
exp	(0:n) BIT	zero	Parameter for EXPECT statement processing
ready	BIT	0	'1'B, if task is contained in ready list
run	BIT	0	'1'B, if task uses the processor
susp	BIT	0	'1'B, if task suspended explicitly or by EXPECT
sync	BIT	0	'1'B, if task suspended for synchronisation
em	(0:n) BIT	zero	Mask of task's EXPECT schedules contained in rel
acs	(0:n) BIT	zero	Mask of task's ACTIVATE schedules contained in rel
tes	(0:n) BIT	zero	Mask of task's TERMINATE schedules contained in rel
prs	(0:n) BIT	zero	Mask of task's PREVENT schedules contained in rel
sus	(0:n) BIT	zero	Mask of task's SUSPEND schedules contained in rel
cos	(0:n) BIT	zero	Mask of task's CONTINUE schedules contained in rel, also used for RESUME schedules
tos	FIXED	0	Column index in rel of timeout schedule
sys	FIXED	0	*Column index in rel of synchronisation resume schedule*

In its initial value column the abbreviations d and l stand for appropriate data to be extracted from the task declarations or to be provided be the linkage editor, ∞ for a very large constant, and zero for a vector of '0'B entries, respectively. As array bounds three installation parameters are mentioned in the table, viz. mba, creg, and cs, standing for the maximum number of permitted buffered task activations, for the number of the general processor's internal registers, and for the amount of synchroniser hardware representations, respectively. The elements of the bit arrays with lengths n+1 correspond to the ones in the vectors r, e, and gm. The meaning of the single tcb entries will become clear in detail from the context in which they are applied as described in the next section. There, like a tcb field, the quantity "active" is interrogated being an abbreviation for the expression

ready OR run OR susp OR sync.

4.3.4. Algorithms of the Operating System Nucleus

In this section we shall describe the routines called by the control programs to perform the SRL functions compiled earlier. For easy readability they are formulated as Pearl procedures. By transforming the procedure names into one word quantities, however, and appending the appropriate procedure parameters the data sets describing the requested services are formed. We already know them as contents of the arrays b and tb from section 4.3.2.

Since they have no connection with the other SRL algorithms, we commence with two procedures initialising and terminating the surveillance of shared variables. For the meaning of the mentioned arrays we refer to the procedure "compare" in section 4.2.3.

```
initsurv: PROCEDURE((a,op,w) FIXED);
    /* Parameter description:
      a : address of shared variable,
      op: relational operator specification,
      w : value */
    i:=1;
    WHILE ad(i) NE NIL AND ad(i) NE a REPEAT;
     i:=i+1;
    END;
    ad(i):=a; rl(i):=op; v(i):=w;
    CALL protocol;
    END;
```

and

```
termsurv: PROCEDURE(a FIXED);
    i:=1;
    WHILE ad(i) NE a REPEAT; i:=i+1; END;
    ad(i):=NIL;
    CALL protocol;
    END;
```

We now turn to the routines that put the first task of the ready list, i.e. the most urgent one, into the running state or perform the reverse operation, respectively. Before pre-empting the processor, the following procedure waits as long as the non-interruption bit is set. This bit may be set by a task if a short critical region is to be executed without disturbance. The tcb index of the task to which the processor is presently assigned is stored in the variable "ass". If the processor is idle its state as well as the value of "ass" are zero. The further objects occurring in the following processor withdrawal procedure were introduced in sections 2.2.2.3. and 4.1. Here and in the sequel the parameter i always gives the tcb index of the task to which the function is applied.

```
preempt: PROCEDURE(i FIXED);
   IF ass EQ i THEN
    WHILE non interruption bit REPEAT; END;
    processor state:='0'B;
    tcb(i).rt:=t time; tcb(i).tl:=t res;
    FOR j FROM 0 BY 1 TO creg-1 REPEAT;
```

```
        tcb(i).reg(j):=processor register(j);
       END;
       tcb(i).run:='0'B; ass:=0; CALL protocol;
      FIN;
      END;
```

The array ti contains the tcb indices of the tasks ready for execution and ordered according to increasing deadlines. The actual number of the ready list's elements is available as ct. Accordingly, the processor assignment routine reads as follows.

```
procadm: PROCEDURE;
      IF ass NE ti(1) THEN
       IF ass NE 0 THEN CALL preempt(ass); FIN;
       ass:=ti(1); tcb(ass).run:='1'B;
       IF tcb(ass).frame EQ -1 THEN
        CALL vsadmin(foc);
       FIN;
       FOR j FROM 0 BY 1 TO creg-1 REPEAT;
        processor register(j):=tcb(ass).reg(j);
       END;
       t time:=tcb(ass).rt; t res:=tcb(ass).tl;
       IF tcb(ass).tcrit NE ∞ THEN
        /* Preparations for task (re-) start at precisely given time */
        comparison register 2:=tcb(ass).tcrit;
        tcb(ass).tcrit:=∞;
        continuation signal mask:='1'B;
       FIN;
       processor state:='1'B;
       CALL protocol;
      FIN;
      END;
```

When a task has turned ready for execution the following procedure is invoked. It inserts the task's tcb index into the ready list. By calling the routine "schedule" it is checked whether the actual free task set can be executed meeting all given deadlines. If need be, an overload situation is handled by terminating all tasks that were not declared with a KEEP attribute and by preventing any further (re-) activations. All interrupts are disabled by resetting the mask mk and the overload signal is set. For the occurrence of this event special tasks coping with the situation can be scheduled.

```
toexec: PROCEDURE(i FIXED);
     tcb(i).ready:='1'B;
     tcb(i).tcond:=tcb(i).ta(tcb(i).ring1);
     k:=ct; ct:=ct+1;
     WHILE tcb(i).ta(tcb(i).ring1) LT
      tcb(ti(k)).ta(tcb(ti(k)).ring1) AND k GT 0 REPEAT;
      ti(k+1):=ti(k); k:=k-1;
     END;
     ti(k+1):=i;
     CALL schedule;
     CALL protocol;
     END;
```

with

```
schedule: PROCEDURE;
      IF ass NE 0 THEN tcb(ass).tl:=t res; FIN;
      s:=0; zg:='0'B; t:=clock;
```

```
          FOR k FROM 1 BY 1 TO ct REPEAT;
           s:=s+tcb(ti(k)).tl;
           IF tcb(ti(k)).ta(tcb(ti(k)).ring1)-t LT s THEN
            GOTO out;
           FIN;
          END;
          zg:='1'B;
      out: IF zg THEN CALL procadm; ELSE
           FOR k FROM 1 BY 1 TO ct REPEAT;
            IF NOT tcb(ti(k)).keep THEN
             CALL execprev(ti(k)) /* Prevent task ti(k) */;
             CALL execterm(ti(k)) /* Terminate task ti(k)*/;
            FIN;
           END;
           mk:=zero;
           sv(2):='1'B
           /* Raising of signal associated with an overload situation */;
          FIN;
          CALL protocol;
          END;
```

The inverse operation of removing a task from the ready list is performed by

```
     backexec: PROCEDURE(i FIXED);
          IF tcb(i).ready THEN
           IF tcb(i).run THEN CALL preempt(i); FIN;
           tcb(i).ready:='0'B; l:=0;
           FOR k FROM 1 BY 1 TO ct REPEAT;
            IF i NE ti(k) THEN l:=l+1; ti(l):=ti(k); FIN;
           END;
           ct:=l; ti(l+1):=0;
           CALL procadm;
           CALL protocol;
          FIN;
          END;
```

The next procedure to be stated here prepares the schedules of tasking operations and inserts these data into the corresponding data structures. First, an old schedule is deleted when a new one for the same task and the same operation is to be inserted. In case a resume operation is requested, the pertaining task is immediately suspended and a continuation is scheduled. This applies to the RESUME tasking statement as well as to the future task reactivations connected to EXPECT and synchronisation operations. The parameters of the operations to be performed upon fulfillment of their schedules are read in from the processor fifo, their validity is checked, and then they are stored into the array b. New values for the tcb entries required by the storage administration scheme are determined. In the course of this, the array ivl is used giving for each interrupt the mean time between its occurrences. Finally, data pointing to the associated schedules are stored in the task's tcb. There is only one schedule common to all tasks linking the signals to an ON reaction described later. This schedule always resides in rel(1:m,0). To simplify the formulation of the following procedure, the tcb entries acs(.), tes(.), prs(.), sus(.), and cos(.) are addressed as sch(1:5,.), respectively.

```
     schtop: PROCEDURE((i,s) FIXED);
         /* Parameter description:
          i: tcb index,
          s: operation selector:
                        1=ACTIVATE,
                        2=TERMINATE,
                        3=PREVENT,
                4=SUSPEND,
                5=CONTINUE,
                6=RESUME,
                7=EXPECT-Resume,
                8=Synchronisation-Resume,
                10=ON */
IF s EQ 10 THEN
 b(1,0):=s; b(2,0):=NIL;
 READ rel(1:m,0) FROM processor fifo; GOTO return;
FIN;
l:=IF s NE 6 THEN s ELSE 5 FIN;
IF l LE 5 THEN
 IF tcb(i).sch(l,0:n) NE zero THEN
  hv:=zero;
  FOR j FROM 0 BY 1 TO n REPEAT;
   IF tcb(i).sch(l,j) THEN
    hv(j):=rel(1,j); rel(1:m,j):=zero; b(1,j):=NIL;
   FIN;
  END;
  IF hv NE zero THEN in:=-1; dv:=hv; sv(1):=2; FIN;
  tcb(i).sch(l,0:n):=zero;
 FIN;
 IF l EQ 1 THEN tcb(i).tact:=tcb(i).int:=∞; FIN;
 IF l EQ 5 THEN tcb(i).tcont:=∞; FIN;
FIN;
IF s GE 6 AND s LE 8 THEN
 CALL execsusp(i) /* Suspend task i */;
 IF s EQ 6 THEN s:=5; FIN;
 IF s EQ 8 THEN
  tcb(i).susp:='0'B; tcb(i).sync:='1'B;
 FIN;
FIN;
READ l FROM processor fifo;
k:=1; hv:=zero; tvs:=∞; dti:=0;
FOR j FROM 1 BY 1 TO l REPEAT;
 WHILE k LE n AND b(1,k) NE NIL REPEAT; k:=k+1; END;
 IF k GT n THEN GOTO return; FIN;
 /* No more space for schedule storage */;
 READ rel(1:m,k) FROM processor fifo;
 kk:=1; hv(k):=ei:='1'B; tcomp:=0;
 IF NOT rel(1,k) THEN tcomp:=clock ELSE
  READ ts FROM processor fifo
   /* This variable and dt0 use the same storage location. */;
  READ dts FROM processor fifo;
  READ rts FROM processor fifo;
  READ q FROM processor fifo; ex:=q AND '01'B;
  IF (q AND '10'B) EQ '00'B THEN
   IF ts LE clock OR dts LE 0 OR rts LT 0 THEN
    hv(k):=ei:='0'B; rel(1:m,k):=zero;
   ELSE tcomp:=ts; in:=k; sv(1):=1; FIN;
  ELSE
   IF dt0 LT 0 OR dts LE 0 OR rts LT 0 THEN
    hv(k):=ei:='0'B; rel(1:m,k):=zero;
   FIN;
```

```
          /* Parameter preparation for the second stage of a two-stage schedule */
          rel(1,k):='0'B; b(1,k):=11; b(2,k):=dt0;
          b(3,k):=dts; b(4,k):=rts; b(5,k):=ex; b(6,k):=k;
          tcomp:=dt0+clock; kk:=7;
         FIN;
        FIN;
        b(kk,k):=IF s EQ 8 AND rel(1,k) THEN 9 ELSE s FIN;
        kk:=kk+1; b(kk,k):=i;
        IF s EQ 1 OR s EQ 5 THEN
        /* Input of further parameters for activation and continuation operations */
         kk:=kk+1; READ b(kk,k) FROM processor fifo;
         tcomp:=tcomp+b(kk,k);
         IF s EQ 1 THEN
          FOR ll FROM 1 BY 1 TO 4 REPEAT;
           kk:=kk+1; READ b(kk,k) FROM processor fifo;
          END;
         FIN;
         IF ei THEN
          IF rel(2:m,k) NE zero THEN
           dta:=0;
           FOR ll FROM 1 BY 1 TO cinterrupt REPEAT;
            IF rel(ll+interrupt displacement,k) THEN
             dta:=dta+1/ivl(ll);
            FIN;
           END;
           IF s EQ 1 THEN dti:=dti+dta; FIN;
           tcomp:=tcomp+0.5/dta;
          FIN;
          tvs:=min(tvs,tcomp);
         FIN;
        FIN;
        kk:=IF ei THEN kk+1 ELSE 1 FIN; b(kk,k):=NIL;
       END;
       IF s LE 5 THEN
        tcb(i).sch(s,0:n):=hv;
        IF s EQ 1 THEN
         tcb(i).tact:=tvs;
         IF dti NE 0 THEN tcb(i).int:=1/dti; FIN;
        FIN;
        IF s EQ 5 THEN tcb(i).tcont:=tvs; FIN;
       ELSE
        IF s EQ 7 THEN tcb(i).em:=hv; ELSE
         j:=1; WHILE NOT hv(j) REPEAT; j:=j+1; END;
         tcb(i).sys:=j; j:=j+1;
         WHILE NOT hv(j) REPEAT; j:=j+1; END;
         tcb(i).tos:=j;
        FIN;
       FIN;
   return: CALL protocol;
       END;
```

The routine reacting upon the fulfillment of two-stage schedules reads as follows. It initiates the execution of the scheduled operation when a corresponding event has occurred. In case an interrupt was detected, the associated cyclic time schedule is newly set up.

```
secstage: PROCEDURE((dt0,dts,rts) DURATION,ex BIT,k FIXED);
        IF ev(1) THEN CALL procedure identified by b(7,k)
          with parameters b(j,k), j=8,... ; FIN;
        IF ev AND '01...1'B NE zero THEN
         in:=k; sv(1):=1; rel(1,k):='1'B; ts:=clock+dt0;
         IF b(7,k) EQ 1 THEN
          tcb(b(8,k)).tact:=ts+b(9,k);
         FIN;
         IF b(7,k) EQ 5 THEN
          tcb(b(8,k)).tcont:=ts+b(9,k);
         FIN;
         /* Wait one PRL cycle. */; in:=k; sv(1):=2;
         IF dt0 EQ 0 THEN CALL procedure identified by
           b(7,k) with parameters b(j,k), j=8,... ; FIN;
        FIN;
        CALL protocol;
        END;
```

The procedures that will be described in the sequel carry through those operations whose parameters and links to schedules were processed by the last two ones.

The first routine to be stated performs the SRL reaction to signals, and all necessary preparations required by the application program to branch into an appropriate ON sequence. Then the processor is restarted. The index of the program counter within the processor's register file is here designated as pc.

```
execon: PROCEDURE;
      IF ass NE 0 THEN
       hv:=ev AND tcb(ass).onm;
       IF hv NE zero THEN
        tcb(ass).fsa:=processor register(pc);
        processor register(pc):=tcb(ass).aon;
        j:=1;
        WHILE NOT hv(j+signal displacement) REPEAT;
         j:=j+1; END;
        tcb(ass).sigid:=j;
        processor continuation signal:='1'B;
       FIN;
       CALL protocol;
      FIN;
      END;
```

The next procedure serves for removing all schedules set up within the framework of an EXPECT statement.

```
prevexp: PROCEDURE(i FIXED);
        hv:=tcb(i).em AND rel(1,0:n);
        FOR j FROM 1 BY 1 TO n REPEAT;
         IF tcb(i).em(j) THEN
          tcb(i).em(j):='0'B; rel(1:m,j):=zero;
          b(1,j):=NIL;
         FIN;
        END;
        IF hv NE zero THEN in:=-1; dv:=hv; sv(1):=2; FIN;
        CALL protocol;
        END;
```

Upon occurrence of one of the events mentioned in an EXPECT statement's alternative, the following procedure will resume the execution of the task containing this language construct.

```
resexp: PROCEDURE(i FIXED);
     hv:=tcb(i).em AND e;
     IF hv NE zero THEN in:=-1; dv:=hv; sv(1):=2; FIN;
     hv:=tcb(i).em AND r; tcb(i).exp:=tcb(i).exp OR hv;
     IF tcb(i).susp AND NOT tcb(i).ready AND
       NOT tcb(i).run AND NOT tcb(i).sync THEN
      tcb(i).susp:='0'B; CALL toexec(i);
     FIN;
     CALL protocol;
     END;
```

When a task or an activation operation is suspended for synchronisation two reactions may be scheduled: the check whether the required resources can now be claimed and the timeout handling. For the former the bit vector "boltevent" is examined in appropriate positions comprising the status outputs of the synchroniser hardware representations.

```
testsync: PROCEDURE(i FIXED);
      IF rel(1:m,tcb(i).sys) AND boltevent EXOR
        rel(1:m,tcb(i).sys) EQ zero THEN
       tcb(i).opt:=tcb(i).opt+1;
       CALL protocol; CALL ressync(i);
      FIN;
      END;
```

and

```
timeout: PROCEDURE(i FIXED);
     tcb(i).opt:=tcb(i).opt+2;
     CALL protocol; CALL ressync(i);
     END;
```

with

```
ressync: PROCEDURE(i FIXED);
     tcb(i).sync:='0'B; in:=tcb(i).tos; sv(1):=2;
     rel(1:m,tcb(i).sys):=rel(1:m,tcb(i).tos):=zero;
     b(1,tcb(i).sys):=b(1,tcb(i).tos):=NIL;
     tcb(i).sys:=tcb(i).tos:=0;
     IF NOT tcb(i).susp THEN
      IF tcb(i).opt LE 2 THEN CALL toexec(i);
                 ELSE CALL prepexec(i);
      FIN;
     FIN;
     CALL protocol;
     END;
```

Finally, we turn now to the processing of the actual tasking operations and commence our considerations with the treatment of schedule prevention. According to the semantics of Pearl, any schedule for tasking operations connected to a certain task and eventual buffered activations of the latter are deleted.

```
execprev: PROCEDURE(i FIXED);
      tcb(i).ba:=0; tcb(i).tact:=tcb(i).tcont:=∞;
      hv:=tcb(i).acs OR tcb(i).tes OR tcb(i).prs OR
        tcb(i).sus OR tcb(i).cos;
      FOR j FROM 1 BY 1 TO n REPEAT;
       IF hv(j) THEN
        hv(j):=rel(1,j); rel(1:m,j):=zero; b(1,j):=NIL;
```

```
     FIN;
    END;
    IF hv NE zero THEN in:=-1; dv:=hv; sv(1):=2; FIN;
    tcb(i).acs:=tcb(i).tes:=tcb(i).prs:=tcb(i).sus:=tcb(i).cos:=zero;
    CALL protocol;
    END;
```

In order to simplify the forthcoming procedures, we introduce the following subroutine for the annihilation of exhausted time schedules.

```
annexsch: PROCEDURE(bc() BIT IDENTICAL);
    /* bc is an array of type (0:n) BIT. */
    hv:=bc AND e;
    IF hv NE zero THEN
     in:=-1; dv:=hv; sv(1):=2;
     FOR j FROM 1 BY 1 TO n REPEAT;
      IF hv(j) THEN
       rel(1,j):='0'B;
       IF rel(1:m,j) EQ zero THEN
        b(1,j):=NIL; bc(j):='0'B;
       FIN;
      FIN;
     END;
    FIN;
    CALL protocol;
    END;
```

Employing this routine, the procedure performing task suspension can be easily formulated.

```
execsusp: PROCEDURE(i FIXED);
    IF tcb(i).active THEN
     CALL backexec(i);
     tcb(i).susp:='1'B; tcb(i).tcond:=∞;
     CALL annexsch(tcb(i).sus); CALL protocol;
    FIN;
    END;
```

The inverse tasking operation, viz. continuation, is carried through by the next procedure. Here and later on in the activation program, the parameters for the operation's precise timing are set if this feature is requested. The actual operation takes place *eps* time units after the corresponding premature time event.

```
execcont: PROCEDURE(i FIXED, tc DURATION);
    /* Parameter description:
      i : tcb index,
      tc: response time */
    IF tcb(i).susp AND tcb(i).em EQ zero THEN
     tcb(i).susp:='0'B; tcb(i).tcont:=∞;
     tcb(i).ta(tcb(i).ring1):=tcb(i).tcond:=clock+tc;
     av:=tcb(i).cos AND gm; CALL annexsch(tcb(i).cos);
     IF NOT tcb(i).sync THEN
      IF av NE zero THEN
       tcb(i).tcrit:=clock+eps;
       tcb(i).ta(tcb(i).ring1):=tcb(i).tcrit+tcb(i).tl;
      FIN;
      CALL toexec(i);
     FIN;
     CALL protocol;
    FIN;
    END;
```

When a task is to be activated, this request and the corresponding parameters are first buffered. If there are no former task activations to be processed, the task execution is prepared and initiated.

```
execact: PROCEDURE(i FIXED, tc DURATION, usg FIXED,
        tor DURATION, toa CLOCK, ii FIXED);
    /* Parameter description:
      i : tcb index of task,
      tc : response time,
      usg: pointer to semaphore mentioned in USING clause,
      tor: relative timeout condition,
      toa: absolute timeout condition,
      ii : tcb index of alternative task */
    IF tcb(i).ba LT mba THEN
     tcb(i).ba:=tcb(i).ba+1;
     tcb(i).ring2:=tcb(i).ring2+1 REM mba;
     tcb(i).ta(tcb(i).ring2):=tvs:=clock+tc;
     tcb(i).using(tcb(i).ring2):=usg;
     tcb(i).ato(tcb(i).ring2):=IF toa NE 0 THEN toa ELSE clock+tor FIN;
     tcb(i).alt(tcb(i).ring2):=ii;
     av:=tcb(i).acs AND gm; tcb(i).tact:=∞;
     FOR j FROM 1 BY 1 TO n REPEAT;
      IF tcb(i).acs(j) AND NOT e(j) THEN
       tcb(i).tact:=min(tcb(i).tact,tvs+
         IF rel(1,j) THEN dt(j) ELSE tcb(i).int FIN);
      FIN;
     END;
     CALL annexsch(tcb(i).acs);
     IF NOT tcb(i).active THEN CALL prepexec(i); FIN;
    FIN;
    CALL protocol;
    END;
```

with the following auxiliary procedure taking care of the synchronisation implied by the presence of an USING clause:

```
prepexec: PROCEDURE(i FIXED);
    j:=tcb(i).using(tcb(i).ring1);
    IF j GT 0 THEN
     IF tcb(i).opt EQ 0 THEN
      IF bolt status(j) EQ 0 THEN
       bolt status(j):=-1; tcb(i).bolt(j):='1'B; j:=i;
      ELSE
       k:=1; ts:=tcb(i).ato(tcb(i).ring1);
       WHILE k LE n AND b(1,k) NE NIL REPEAT;
        k:=k+1;
       END;
       l:=k+1;
       WHILE l LE n AND b(1,l) NE NIL REPEAT;
        l:=l+1;
       END;
       IF l GT n OR ts LE clock THEN GOTO return; END;
       tcb(i).sync:='1'B; tcb(i).opt:=2;
       b(1,k):=8; b(1,l):=9;
       b(2,k):=b(2,l):=i; b(3,k):=b(3,l):=NIL;
       rel(j+sync displacement,k):=rel(1,l):='1'B;
       tcb(i).sys:=k; tcb(i).tos:=l;
       dts:=1; rts:=0; ex:='0'B; in:=l; sv(1):=1;
```

```
                FIN;
               ELSE
                av:=zero;
                IF tcb(i).opt EQ 3 THEN
                 bolt status(j):=-1; tcb(i).bolt(j):='1'B; j:=i;
                ELSE j:=tcb(i).alt(tcb(i).ring1); FIN;
                tcb(i).opt:=0;
               FIN;
              ELSE j:=i;
              FIN;
              IF tcb(i).opt EQ 0 THEN
               tcb(j).rt:=0; tcb(j).tl:=tcb(j).trun;
               tcb(j).reg(pc):=tcb(j).sta;
               IF av NE zero THEN
                tcb(j).tcrit:=clock+eps;
                tcb(j).ta(tcb(j).ring1):=tcb(j).tcrit+tcb(j).tl;
               FIN;
               CALL toexec(j);
              FIN;
        return: CALL protocol;
              END;
```

Upon reaching its normal end, each task requests the execution of the following SRL procedure that removes the task from the ready list and puts the next buffered activation - if any - into the ready state.

```
        execend: PROCEDURE(i FIXED);
              CALL backexec(i); tcb(i).tcond:=∞;
              j:=tcb(i).using(tcb(i).ring1);
              IF j GT 0 THEN
               bolt status(j):=0; tcb(i).bolt(j):='0'B;
              FIN;
              tcb(i).ring1:=tcb(i).ring1+1 REM mba;
              tcb(i).ba:=tcb(i).ba-1;
              IF tcb(i).ba GT 0 THEN
               av:=zero; CALL prepexec(i);
              FIN;
              CALL protocol;
              END;
```

Additionally to the above actions, in the course of handling a TERMINATE tasking operation, eventual synchronisation, timeout, and EXPECT schedules are deleted and all synchronisers seized by the task including an USING semaphore are released.

```
        exectem: PROCEDURE(i FIXED);
              IF tcb(i).active THEN
               CALL backexec(i); CALL annexsch(tcb(i).tes);
               tcb(i).susp:=tcb(i).sync:='0'B;
               tcb(i).tcond:=∞;
               IF tcb(i).sys NE 0 THEN
                rel(1:m,tcb(i).sys):=zero;
                b(1,tcb(i).sys):=NIL; tcb(i).sys:=0;
               FIN;
               IF tcb(i).tos NE 0 THEN
                in:=tcb(i).tos; sv(1):=2;
                rel(1:m,tcb(i).tos):=zero;
                b(1,tcb(i).tos):=NIL; tcb(i).tos:=0;
               FIN;
```

```
 CALL prevexp(i);
 FOR j FROM 1 BY 1 TO cs REPEAT;
  IF tcb(i).bolt(j) THEN
   tcb(i).bolt(j):='0'B;
   IF bolt status(j) EQ -1 THEN bolt status(j):=0;
    ELSE bolt status(j):=bolt status(j)-1; FIN;
  FIN;
 END;
 tcb(i).ring1:=tcb(i).ring1+1 REM mba;
 tcb(i).ba:=tcb(i).ba-1;
 IF tcb(i).ba GT 0 THEN
  av:=zero; CALL prepexec(i);
 FIN;
 CALL protocol;
FIN;
END;
```

5. Evaluation of the Architecture

5.1. Implementation of Extended Pearl

It is the subject of this section to detail the additional steps a compiler for extended Pearl has to take, when preparing programs for execution on the considered process control computer. This comprises compile time checks, the processing of options, and mainly the generation of special code and task control data. Particularly, the translation of language elements into calling sequences of the operating system processor and into direct access to its hardware components is described. For several language constructs their interaction with functions of the auxiliary processor and their reduction to program sequences involving only low-level language features are outlined. Employing these elaborations, the corresponding machine code generation becomes easily possible.

5.1.1. Compiler Functions

We commence our considerations on the additional compiler activities by treating the pragmas being intended for the selection of various options. Thus, the compiler can be instructed to flag dynamic language features, when their usage is to be prevented for security reasons. In the test phase of a program many verification language elements including TRIGGER and INDUCE statements will be inserted in it to control its execution and to examine its performance. Hence, in order to avoid the removal of these statements and to retain them for later use instead, it is selected with a pragma whether they are to be honoured or to be regarded as comments. Different syntax check procedures with respect to the LOCK synchronisation statement are invoked by a further pragma. The available alternatives are the application of either the resource releasing or of the resource hierarchical deadlock prevention scheme. The latter allows the nesting of LOCK statements, provided the sequence of resource requests complies with a predefined resource hierarchy. The remaining pragmas concern the compiler's code generation phase. So, it is determined whether the just considered program module shall be executed on the general processor or under the test monitor in the auxiliary processor's interrupt generator, where it produces temporal request patterns for the testing of other software. The test monitor is a supervisor tailored for this purpose. In case the general processor turns idle, it redefines the system time to the next scheduled moment in order to reduce the time requirements of simulation runs. Another pragma specifies the extent of the event recording desired, and has to be converted by the compiler into parameters for the protocol hardware.

Now we turn to the discussion of some compiler functions to be carried out during the processing of declarations and specifications. Upon elaboration of a task declaration, a corresponding control block is allocated and initialised, provided the maximum number of tasks that can be simultaneously handled is not exceeded. Interfaces are treated in the same way as tasks. The encountered interrupts and synchronisers are assigned to the corresponding hardware elements being part of the auxiliary processor. Accordingly, object names are transformed into hardware addresses. In the course of this, information from the SYSTEM division is utilised, and interrupt array references are reduced to lists of single interrupt references. When shared variables are declared appropriate storage space is allocated within the common memory. Furthermore, for synchronising the access to them, each shared variable is associated with an implicitly defined bolt. The application of the SYNC monadic operator to a shared variable is replaced by a reference to its implicit bolt. The system or user defined objects of type SIGNAL are represented as shared one-bit variables. They do not require protection by implicit bolts. The mentioned storage and hardware element assignments are subject to capacity restrictions whose observance naturally needs to be checked. The data contained in the INTERVAL attributes of interrupt specifications are gathered for later use in the array ivl also located in the common memory. The task attributes KEEP and RESIDENT give rise to corresponding task control block entries. Here also the maximum task run times must be recorded. They are either provided by the user or estimated by the compiler according to the method outlined in section 2.2.2.3. In this context, the program load times need to be regarded.

In order to subject a program to the virtual storage administration scheme described in section 3.2, its object code generated by the compiler must be divided into pages. Tasks and procedures with the RESIDENT or REENT attributes, respectively, as well as objects of GLOBAL scope are placed in those pages that remain permanently in storage. However, their maximum number may again not exceed a certain limit.

5.1.2. Run-Time Features

In this section we shall discuss the implementation of the new language constructs as introduced in section 2.2 as well as the realisation of some other statements that require communication with the operating system processor. We start our considerations with some simple features.

The auxiliary processor's hardware level is addressed by the following instructions and functions. In order to determine values for the monadic operators NOW, BVALUE, SVALUE, and TSTATE, giving the actual time, the stati of bolts and semaphores as well as of tasks, the clock register, the status registers of the synchronisers' hardware representations, and appropriate task control block entries are directly read out, respectively. On the other hand, the UPDATE statement is converted into writing a new value into the down counter T.RES determining the executing task's maximum residual run time. Certain one-bit storage locations in the common memory serve for masking the interrupt lines. They are set and reset by ENABLE or DISABLE statements. TRIGGERing an interrupt is performed by a writing access to the corresponding interrupt recognition hardware element. In a similar way the INDUCE statement is realised, viz. a logical one is written into the shared variable associated with the signal to be raised. If this variable was subjected to surveillance for the assumption of the true value, the scheduled event reaction will be initiated by the auxiliary processor.

For the implementation of the MAXLOOP clause in the repeat statement the compiler generates a counting of the performed iterations, and for the case of an overflow a conditional branch to the raising of the pertaining system defined signal.

When an activity is to be scheduled for later execution, corresponding parameters need to be transmitted to the auxiliary processor as we shall see later. In the course of preparing these parameters for a time schedule given with the EXACTLY attribute, an associated indicator is set and the installation parameter *eps* is subtracted from the initial time of the schedule. Furthermore, as time condition the sum of the (residual) maximum task run-time and *eps* is supplied, causing the deadline driven scheduling algorithm to put the task into the running state immediately. All other preparations are carried through by the operating system processor within the reaction time span of length *eps* before the hardware finally starts the task execution at the exactly specified moment. No provisions are to be made within the code of the task.

Since the occurrence of an interrupt is unforeseeable, the implementation of WHEN reactions without delay is impossible, unless the event is expected with the program loaded and the processor in the idle state. However, this possibility shall not be considered, because it is inefficient and the system cannot perform any other tasks. If this feature is needed nevertheless, the external hardware generating the interrupt must do that a little earlier. Then it can be realised with the help of a two stage schedule having the EXACTLY attribute.

In the sequel we shall discuss the replacement of an EXPECT statement by a piece of code easily tangible by the compiler. The syntax of the EXPECT statement was introduced by the following production rules:

expect-statement::=EXPECT alternative-string FIN;

with

alternative::=AWAIT event-list DO statement-string .

We consider now an EXPECT statement appearing in a task whose control block is stored under the index i in the array tcb. By transmitting data describing the event lists to the auxiliary processor, the later execution of the statement body is scheduled. The single lists or alternatives correspond with the set bits in the array tcb(i).em(0:n). Upon occurrence of one or more mentioned events, the alternatives to be carried through then are marked in the bit array tcb(i).exp(0:n) by the operating system processor. Thereafter the task is resumed and all marked alternatives are executed in the sequence they were originally written down. At the end of this, the task is either suspended or control branches to those alternatives which were marked in the meantime. Thus, an EXPECT statement can be replaced by the following equivalent sequence.

```
/* Schedule an EXPECT-Resume operation for the occurrence of any event
contained in
   event-list1, event-list2,...; this implies the suspension of the task */;
begin: hv:=tcb(i).exp; tcb(i).exp:=zero;
     /* zero stands for a vector with appropriately many '0'B components */;
     j:=0;
     FOR k FROM 1 BY 1 TO n REPEAT;
      IF tcb(i).em(k) THEN
      j:=j+1;
      IF hv(k) THEN
       CASE j
        ALT statement-string1;
        ALT statement-string2;
        ...
       FIN;
      FIN;
     FIN;
     END;
     IF tcb(i).exp EQ zero THEN SUSPEND; FIN; GOTO begin;
end:  /* Request the operating system to prevent the task's EXPECT schedule */;
```

The QUIT instructions appearing within the statement strings are replaced by jumps to the label "end", where the operating system processor routine prevexp is called for the prevention of the above EXPECT-resume schedule.

A structured synchronisation feature was defined as follows:

```
lock-statement::=
        LOCK synchronisation-clause-list [NONPREEMPTIVELY]
         [TIMEOUT {IN duration-expression | AT clock-expression}
          OUTTIME statement-string FIN]
        PERFORM statement-string UNLOCK;

with

synchronisation-clause::=
        semaphore-expression-list |
        EXCLUSIVE(sync-object-expression-list) |
        SHARED(sync-object-expression-list)

and

sync-object::=bolt | shared-object   .
```

Let i again be the tcb index of that task's control block in which a LOCK statement occurs. First, the synchronisation clause list is converted to entries in the array lv(1:cs,1:2) of bits according to

lv(j,1):='1'B, if exclusive access to the protected object is requested,
lv(j,2):='1'B, if shared access to the protected object is requested,
lv(j,*):='0'B, otherwise,

for j=1,...,cs, where cs designates the number of synchroniser hardware representations. Their status outputs are available as the matrix "boltevent" having the same structure as lv. The comparison of them yields whether the synchronisers can be claimed as requested. The indicator nia is set to 1 in case the NONPREEMPTIVELY attribute is present and to the value 0 otherwise. It is used in the course of programming the non-interruption hardware register, signifying that the running task may not be pre-empted if it holds the value '1'B. The setting of this bit given in the sequel refers to the application of the resource releasing deadlock prevention method, i.e. there may be no nesting of LOCK statements. When the resource hierarchical algorithm is employed, the bit's programming as shown in form of comments is slightly more complex and requires an integer variable nni with initial value 0 as a further tcb entry. In the following piece of code replacing the LOCK statement, the tcb fields "opt" and "bolt" are used to control the flow of processing and to mark claimed bolts, respectively. When seizing a synchroniser, appropriate new values are written into its status register.

```
non interruption bit:='1'B;
IF lv AND boltevent EXOR lv EQ zero THEN tcb(i).opt:=1;
  ELSE
   non interruption bit:='0'B;
   /* or: non interruption bit:=tcb(i).nni GT 0; */;
   /* Parameter preparation and transmission to the operating system
processor for scheduling
      the PERFORM and OUTTIME clauses; this implies task suspension */;
FIN;
IF tcb(i).opt EQ 1 THEN
 non interruption bit:='1'B;
 FOR j FROM 1 BY 1 TO cs REPEAT;
  IF lv(j,1) OR lv(j,2) THEN
   tcb(i).bolt(j):='1'B;
   IF lv(j,1) THEN bolt status(j):=-1;
          ELSE bolt status(j):=bolt status(j)+1;
   FIN;
  FIN;
 END;
 tcb(i).opt:=0;
 IF nia EQ 0 THEN non interruption bit:='0'B; FIN;
 /* or: tcb(i).nni:=tcb(i).nni+nia; non interruption bit:=tcb(i).nni GT 0; */;
 /* Execution of the PERFORM clause;
    here appearing QUIT statements are replaced by jumps to "unlock" */;
unlock: FOR j FROM 1 BY 1 TO cs REPEAT;
      IF lv(j,1) OR lv(j,2) THEN
       tcb(i).bolt(j):='0'B;
       IF lv(j,1) THEN bolt status(j):=0;
              ELSE bolt status(j):=bolt status(j)-1;
       FIN;
      FIN;
     END;
     non interruption bit:='0'B;
     /* or: tcb(i).nni:=tcb(i).nni-nia; non interruption bit:=tcb(i).nni GT 0; */;
ELSE tcb(i).opt:=0; /* Execution of OUTTIME clause */;
FIN;
```

Within the PERFORM clause seized resources can already be released before reaching the LOCK statement's end by employing an UNLOCK instruction defined as

unlock-statement::=UNLOCK semaphore-expression-list | sync-object-expression-list; .

The lists of mentioned synchronisers are again converted to entries of an array uv(1:cs,1:2) whose meaning is analogous to that of the lv elements. Accordingly, the UNLOCK replacement reads as follows.

```
FOR j FROM 1 BY 1 TO cs REPEAT;
 IF uv(j,1) OR uv(j,2) THEN
  tcb(i).bolt(j):='0'B;
  IF uv(j,1) THEN lv(j,1):='0'B; bolt status(j):=0;
          ELSE lv(j,2):='0'B; bolt status(j):=bolt status(j)-1;
  FIN;
 FIN;
END;
```

Owing to the close interaction between operating system processor and application programs in handling ON reactions, this feature requires attention here, too. When a signal is raised from a task, the auxiliary processor's scheduled reaction is activated via the signal hardware and the general processor ceases to execute instructions. This is necessary in order to prevent further operations on undefined data, e.g. in case of a divide check. The operating system processor now loads the actual content of the program address counter as continuation address, and the identification of the occurred signal into the task's tcb fields fsa and sigid, respectively. Then, the start address of the general ON reaction within the application program is written from the tcb entry aon into the program address register and the task execution is restarted. As shown in the sequel, this reaction routine commences with saving the continuation address and the signal identification in the two stacks cad and sid, in order to enable the handling of nested ON requests. The procedures "push", "pop", and "top" perform the necessary stack operations. With each signal a Boolean lock variable is associated to prevent the nested treatment of the same signal. In case the reaction to a signal leads to the raising of the same one, the operating system is called to treat this event. Normally, the ON reaction proceeds with resetting the occurred signal, invoking the corresponding handling procedure, and returning to the address where the program flow was interrupted. The single handling procedures are contained within the bodies of the blocks where they are defined as not explicitly called subroutines. In the course of processing the ON statements appearing in an application program, the starting addresses of these procedures must be loaded into the array rpa in accordance with the block structure.

```
CALL push(cad,tcb(i).fsa); CALL push(sid,tcb(i).sigid);
IF lock(top(sid)) THEN CALL pop(cad); CALL pop(sid); INDUCE system reaction signal;
 ELSE
  lock(top(sid)):='1'B; signal(top(sid)):='0'B;
  CALL CONT rpa(top(sid)); lock(top(sid)):='0'B;
  next:=top(cad); CALL pop(cad); CALL pop(sid);
  GOTO CONT next;
FIN;
```

As final topic relevant to the implementation of extended Pearl on the considered process control computer architecture, the calls for operating system services remain to be discussed. These calls are always performed by loading appropriate parameter sets into a first-in-first-out memory linking the general with the auxiliary processor. The first element of a parameter set determines the requested function and is followed by a variable number of specific data. The meaning of the mentioned data will be described in the sequel.

When the surveillance of a shared variable is to be commenced or terminated its address is provided. In case of the former operation also a relational operator and a corresponding comparison value are specified.

The above stated ON reaction, being part of the application tasks, is initiated by an operating system procedure scheduled upon user program start for the occurrence of all possible signals. This general routine has no parameters.

The majority of operating system functions require as sole parameter the index of that element in the array tcb where the control block of the task is stored to which the operation applies. Among these functions are the routines carrying out the tasking statements PREVENT, SUSPEND, TERMINATE, and the normal task end. After requesting the latter, a task must cease its execution by stopping the processor.

Then, we have the procedures resuming a task upon occurrence of one of the events mentioned in an EXPECT statement's alternative and preventing the corresponding schedules, respectively. Additionally to the tcb index, the service routine performing task CONTINU(E)ation requires as parameter the response time, from which the task's time condition is calculated. The same data must be provided when converting an ACTIVATE statement to a calling sequence. To take care of an USING synchronisation clause, however, further parameters are necessary: the pointer to the mentioned semaphore, a relative and an absolute timeout condition, and the tcb index of the task to be activated when the waiting time exceeds. If no USING clause is present, the pointer must assume the value 0. The other parameters are then irrelevant. When the absolute timeout parameter is 0, the required value is calculated by adding the relative timeout condition to the actual time.

The most complex parameter sets have to be generated for the purpose of scheduling operating system services for future execution. They commence again with the tcb index of the pertaining task, followed by a selector with the values 10 for the ON reaction and 1 to 8 for the operations ACTIVATE, TERMINATE, PREVENT, SUSPEND, CONTINUE, RESUME, EXPECT-resume, and synchronisation-resume, respectively. As further parameter for scheduling the ON reaction, a bit vector is supplied enabling the recognition of all signals that are mentioned in the tasks of the running program. In case of the other options, after the selector a number is transmitted specifying the amount of elementary schedules that will follow. Each of them consists of a bit vector indicating all those events whose occurrences shall cause the execution of an operating system function. If the bit corresponding to the time event is set in this vector, four additional data items describing a time schedule must be provided. The first one is either the schedule's initial value or the displacement to be added to the occurrence time of an interrupt yielding such a value. After that follow the interval between time events, the validity duration of the schedule, and two further bits. The first of these specifies that a two-stage schedule is present, i.e. a time displacement was given, whereas the other one requests exact timing of the operation. For reasons of easing the operating system processor's operation in the case of the tasking statements CONTINUE, RESUME, and ACTIVATE, after each elementary schedule a response time, and if need be also USING clause parameters as discussed above, must be provided. When scheduling a synchronisation-resume, always two elementary schedules are transmitted with which the auxiliary processor associates two different reactions. The first one is the check whether the synchronisation request has turned possible after one or more synchronisers were released and the other one initiates the handling of the timeout condition.

5.2. Qualitative Evaluation

In the course of evaluating the here proposed computer architecture, some qualitative considerations will be sufficient, because a quantitative evaluation based on analytic modeling was already carried out by Tempelmeier in [52,54,55].

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. There the objective of further development is to increase the processing speed. For real-time systems, however, 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 costs of a two processor system as proposed in this paper are higher than those 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, like the non-timely execution of a scheduled task or the loss of an interrupt, the price for an auxiliary processor will be almost negligible.

In a conventional real-time computer every interrupt causes a considerable overhead. This is especially unproductive, if the context contained in large register sets needs to be saved and later reloaded. In most installations, 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. The here proposed architecture uses simple hardware support to provide the same functions based on a more accurate timing. By migrating the operating system nucleus to a special device, the context-switching operations, which are necessary to be performed in the general processor, are reduced to the minimum. The latter is only determined by the structure of the application and the employed task scheduling strategy. As was mentioned already before, the deadline driven scheduling is optimal with respect to the number of pre-emptions it causes.

In his dissertation [47], Schrott points out the problem of prolonged reaction times caused by long phases, during which the operating system [cp. also 10] disables the recognition of interrupts. 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. The here proposed architecture solves this problem by distributing the intrinsically independent functions of event recognition, task administration, and task execution to separate units.

In the early days of real-time data processing the fundamental requirements of timeliness and simultaneousness [33] 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" [37]. Thus, a predictable software behaviour could be realised and the observation of the time conditions could be guaranteed. Later, the method of synchronous programming 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 auxiliary 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. The latter is heavily influenced by the software organisation and here especially by the operating system. Its overhead has effects both on the task response times and the interrupt reaction times. The former depend on the overall computing speed of a system and on the operating system overhead, which is 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 proposed 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.

5.3. Outlook

In order to evaluate the feasibility of the here proposed auxiliary processor more thoroughly, presently a simulation program is being developed. Also, a prototype is under construction. All hardware features as mentioned in section 4.1 have been built. Since speed is not the objective of a prototype, but its easy realisation and the provision of a comfortable environment to allow for frequent software modifications, the primary and secondary reaction levels are modeled by one personal computer each. The first one of them communicates via a digital interface with a local bus, to which all hardware features are connected. The fifo data transfer between the two reaction levels is realised by a standard serial communication link. The routines stated in chapter 4 could be directly implemented on the personal computers by applying a corresponding Pearl compiler, which recently became available for PCs.

The development of the auxiliary processor is a contribution to a project on "Future Generation Process Control Computers for Use in Hard Real-Time Environments", carried out in cooperation with Prof. T. Ichikawa, Information Systems, Faculty of Engineering, Hiroshima University, Japan. Its objective is to realise an architecture for process control computers employed in time-critical applications characterised by providing inherent reliability. In Hiroshima a high-level language oriented descriptor architecture named SPRING [41,57,58] was developed, which comprises a specially tailored operating system. After successful check-out, the prototype of the auxiliary processor will be integrated into SPRING to form an asymmetrical multiprocessor structure. It will serve as a fast event recognition device and will perform all time-critical operating system and task scheduling functions.

"In conclusion, our trial could ... be recognised as a significant step towards the realisation of advanced architectures with adequate role-sharing between hardware and software" [41].

References

[1] Preliminary Ada Reference Manual. And: J.D. Ichbiah et al.: Rationale for the Design of the Ada Programming Language. ACM SIGPLAN Notices 14,6 Parts A and B, June 1979

[2] Ada Reference Manual (July 1980). In: H. Ledgard: Ada - An Introduction. New York-Heidelberg-Berlin: Springer 1981

[3] National Standards Institute, Inc., ANSI/MIL-STD-1815A-1983. Lecture Notes in Computer Science 155, Berlin-Heidelberg-New York-Tokyo: Springer-Verlag 1983

[4] S.R. Ahuja, A. Asthana: A Multi-Microprocessor Architecture with Hardware Support for Communication and Scheduling. ACM SIGPLAN Notices 17,4,205-209, April 1982

[5] U. Ammann: Vergleich einiger Konzepte moderner Echtzeitsprachen. 6. Fachtagung der GI ueber Programmiersprachen und Programmentwicklung. Darmstadt 1980. pp. 1-18. Informatik-Fachberichte 25. Berlin-Heidelberg-New York: Springer 1980

[6] C. Andres, A. Fleischmann, P. Holleczek, W. Muehlbauer: Ein Pearl-Testsystem zum Einsatz in verteilten Systemen. Pearl-Rundschau 3,5,247-250, 1982

[7] R.G. Arnold, R.O. Berg, J.W. Thomas: A Modular Approach to Real-Time Supersystems. IEEE Transactions on Computers, C-31,5,385-398, May 1982

[8] T.D. Atkinson et al.: Modern Central Processor Architecture. Proc. of the IEEE 63,6,863-870, June 1975

[9] R. Barnes: A Working Definition Of The Proposed Extensions For PL/1 Real-Time Applications. ACM SIGPLAN Notices 14,10,77-99, October 1979

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

[11] A.D. Berenbaum, M.W. Condry, P.M. Lu: The Operating System and Language Support Features of the BELLMAC-32 Microprocessor. ACM SIGPLAN Notices 17,4,30-38, April 1982

[12] B.M. Brosgol et al.: Matrix of Ada Language Implementations. ACM Ada Letters II,3,71-76, November 1982, as updated in II,4,136-143, January 1983, and in II,5,97-98, March 1983

[13] G.E. Brown, R.H. Eckhouse, Jr., R.P. Goldberg: Operating System Enhancement Through Microprogramming. ACM SIGMICRO Newsletter 7,1,28-33, 1976

[14] P.J. Brunner, H. Boesmann, A. Tarabout, W. Werum: Universelles PEARL-Betriebssystem. KFK-PDV 55, Karlsruhe, 1976

[15] G. Chroust: Orthogonal Extensions in Microprogrammed Multiprocessor Systems - A Chance for Increased Firmware Usage. EUROMICRO Journal 6,2,104-110, 1980

[16] DIN 44300: Informationsverarbeitung, Nr. 161 (Realzeitbetrieb), Maerz 1972

[17] DIN 66253: Programmiersprache Pearl, Teil 1 Basic Pearl, Vornorm, Juli 1981; Teil 2 Full Pearl, Norm, Oktober 1982

[18] W. Ehrenberger: Softwarezuverlaessigkeit und Programmiersprache. Pearl-Rundschau 3,2,49-55, 1982

[19] B.F. Eichenauer: Prozessprogrammiersprachen und Portabilitaet. 5. Fachtagung der GI ueber Programmiersprachen. Braunschweig 1978. pp. 9-27. Informatik-Fachberichte 12. Berlin-Heidelberg-New York: Springer 1978

[20] A. Ghassemi: Untersuchung der Eignung der Prozessprogrammiersprache PEARL zur Automatisierung von Folgeprozessen. PhD Thesis. Universitaet Stuttgart 1978

[21] W.K. Giloi: Grundlagen, Operationsprinzipien und Strukturen von innovativen Rechnerarchitekturen. GI - 8. Jahrestagung. Berlin 1978. pp. 274-307. Informatik-Fachberichte 16. Berlin-Heidelberg-New York: Springer 1978

[22] W.A. Halang: On Methods for Direct Memory Access Without Cycle Stealing. Microprocessing and Microprogramming 17, 5, May 1986

[23] G. Heider: Let Operating Systems Aid in Component Designs. Computer Design 21,9,151-160, September 1982

[24] R. Henn: Deterministische Modelle fuer die Prozessorzuteilung in einer harten Realzeit-Umgebung. PhD Thesis. Technical University Munich 1975

[25] R. Henn: Zeitgerechte Prozessorzuteilung in einer harten Realzeit-Umgebung. GI - 6. Jahrestagung. pp. 343-359. Informatik-Fachberichte 5. Berlin-Heidelberg: Springer-Verlag 1976

[26] R. Henn: Antwortzeitgesteuerte Prozessorzuteilung unter strengen Zeitbedingungen. Computing 19, 209-220, 1978

[27] H.H. Johnson, M. Maddison: Deadline Scheduling for a Real-Time Multiprocessor. Eurocomp. Conf. Proceedings, 1974, pp. 139-153

[28] R.K. Johnson, J.D. Wick: An Overview of the Mesa Processor Architecture. ACM SIGPLAN Notices 17,4,20-29, April 1982

[29] A. Kappatsch: Full Pearl Language Description. PDV-Bericht KFK-PDV 130. GfK Karlsruhe 1977

[30] W. Kneis (Ed.): Draft Standard on Industrial Real-Time Fortran. International Purdue Workshop on Industrial Computer Systems. ACM SIGPLAN Notices 16,7,45-60, 1981

[31] J. Labetoulle: Ordonnancement des Processus Temps Reel sur une ressource pre-emptive. These de 3me cycle, Universite Paris VI, 1974

[32] J. Labetoulle: Real Time Scheduling in a Multiprocessor Environment. IRIA Laboria, Rocquencourt, 1976

[33] R. Lauber: Prozessautomatisierung I. Berlin-Heidelberg-New York: Springer-Verlag 1976

[34] R. Lauber: Prozessautomatisierung und Informatik. GI - 8. Jahrestagung. Berlin 1978. pp. 381-394. Informatik-Fachberichte 16. Berlin-Heidelberg-New York: Springer 1978

[35] C.L. Liu, J.W. Layland: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. JACM 20, 46-61, 1973

[36] LTR Reference Manual. Compagnie d'informatique militaire, spatiale et aeronautique, Velizy, October 1979

[37] L. MacLaren: Evolving Toward Ada in Real Time Systems. ACM SIGPLAN Notices 15,11,146-155, November 1980

[38] K. Mangold: Ein quellenbezogenes Testsystem fuer PEARL auf einem Prozessrechner. PEARL-Rundschau 2, 6, 3-7, Dezember 1981

[39] G.J. Myers: Advances in Computer Architecture, 2. Ed., New York: John Wiley & Sons 1982

[40] P.M. Newbold et al.: HAL/S Language Specification. Intermetrics Inc., Report No. IR-61-5, November 1974

[41] N. Nishi, H. Tsubotani, T. Ichikawa: SPRING: A High Level Language Architecture in Ada Environment. Proceedings of the IEEE Computer Society's International Computer Software and Applications Conference (COMPSAC), 7-11 November 1983, pp. 373-377

[42] Periphere Computer Systeme GmbH: KE-Handbuch, Munich, 1981

[43] R. Roessler, K. Schenk (Eds.): A Comparison of the Properties of the Programming Languages Algol 68, Camac-IML, Coral 66, PAS 1, Pearl, PL/1, Procol, RTL/2 in Relation to Real-Time Programming. International Purdue Workshop on Industrial Computer Systems. Physics Institute of the University of Erlangen. Nuremberg 1975

[44] R. Roessler: Betriebssystemstrategien zur Bewaeltigung von Zeitproblemen in der Prozessautomatisierung. PhD Thesis. Universitaet Stuttgart 1979

[45] H. Sandmayr: A Comparison of Languages: CORAL, PASCAL, PEARL, ADA and ESL. Computers in Industry 2, 123-132, 1981

[46] V. Scheub (Secretary of the Pearl Association): Private communication, May 1985

[47] G. Schrott: Ein Zuteilungsmodell fuer Multiprozessor-Echtzeitsysteme. PhD Thesis. Technical University Munich 1986

[48] A. Schwald, R. Baumann: Pearl im Vergleich mit anderen Echtzeitsprachen. Proceedings of the 'Aussprachetag Pearl'. PDV-Bericht KFK-PDV 110. GfK Karlsruhe, Maerz 1977

[49] D.P. Siewiorek, C.G. Bell, A. Newell: Computer Structures: Principles and Examples. New York: McGraw-Hill 1982

[50] E.E. Swartzlander Jr., B.K. Gilbert: Supersystems: Technology and Architecture. IEEE Transactions on Computers, C-31,5,399-409, May 1982

[51] T. Tempelmeier: A Supplementary Processor for Operating System Functions. 1979 IFAC/IFIP Workshop on Real Time Programming. Smolenice, 18-20 June 1979

[52] T. Tempelmeier: Antwortzeitverhalten eines Echtzeit-Rechensystems bei Auslagerung des Betriebssystemkerns auf einen eigenen Prozessor. PhD Thesis. Technical University Munich 1980

[53] T. Tempelmeier: Auslagerung eines Echtzeitbetriebsssytems auf einen eigenen Prozessor. Proc. of Fachtagung Prozessrechner 1981, 196-205

[54] T. Tempelmeier: Antwortzeitverhalten eines Echtzeit-Rechensystems bei Auslagerung des Betriebssystemkerns auf einen eigenen Prozessor. Teil 2: Messergebnisse. Report TUM-I8201, Technical University Munich 1982

[55] T. Tempelmeier: Operating System Processors in Real-Time Systems - Performance Analysis and Measurement. Computer Performance, Vol. 5, No. 2, 121-127, June 1984

[56] T. Tempelmeier: Response Times of a Real-Time System with Microprogrammed Operating System Kernel. Report TUM-I8508, Technical University Munich 1985

[57] H. Tsubotani, N. Monden, M. Tanaka, T. Ichikawa: A Computing System to Support Development of Reliable Software. IEEE 1984 Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems. pp. 232-237

[58] H. Tsubotani, N. Monden, M. Tanaka, T. Ichikawa: A High Level Language-Based Computing Environment to Support Production and Execution of Reliable Programs. IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, 134-146, February 1986

[59] N. Wirth: Modula: A Language for Modular Multiprogramming. Software Practice & Experience 7, 3-35, 1977

[60] N. Wirth: Programming in Modula-2. Berlin-Heidelberg-New York: Springer 1982

Figure 1. The basic concept

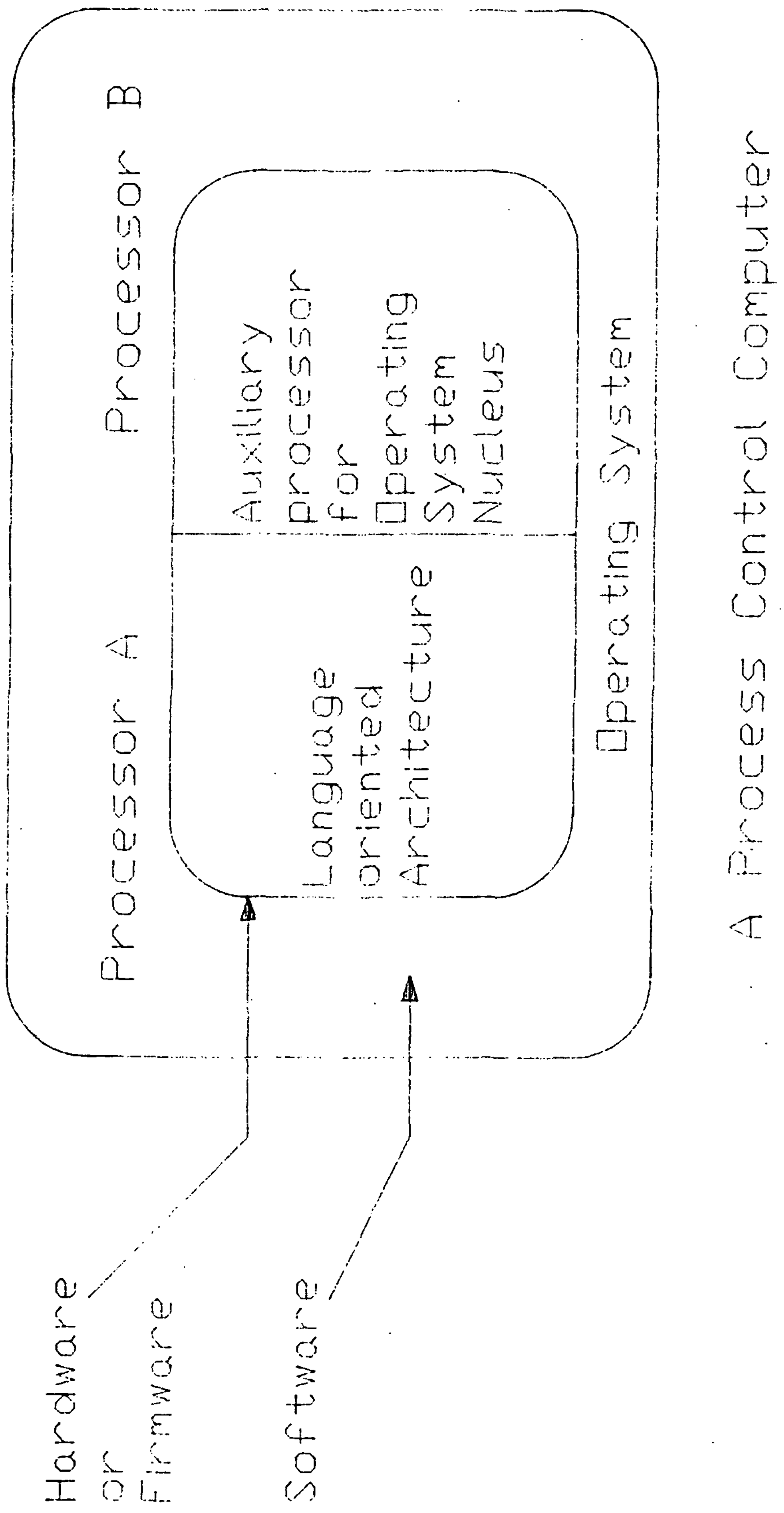
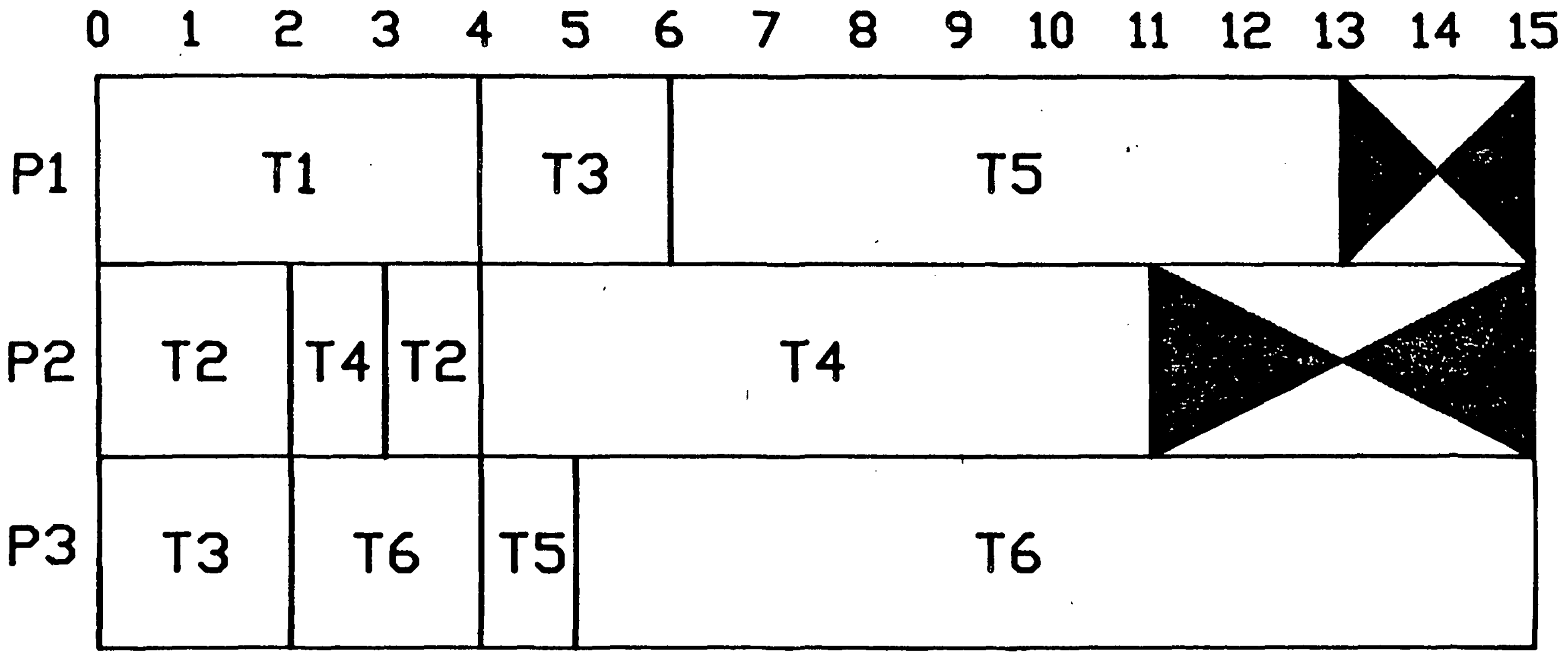

A Process Control Computer
Processor A
Processor B
Language oriented Architecture
Auxiliary processor for Operating System Nucleus
Operating System
Hardware or Firmware
Software

Figure 2. Gantt diagram of a feasible schedule for a symmetric 3-processor system



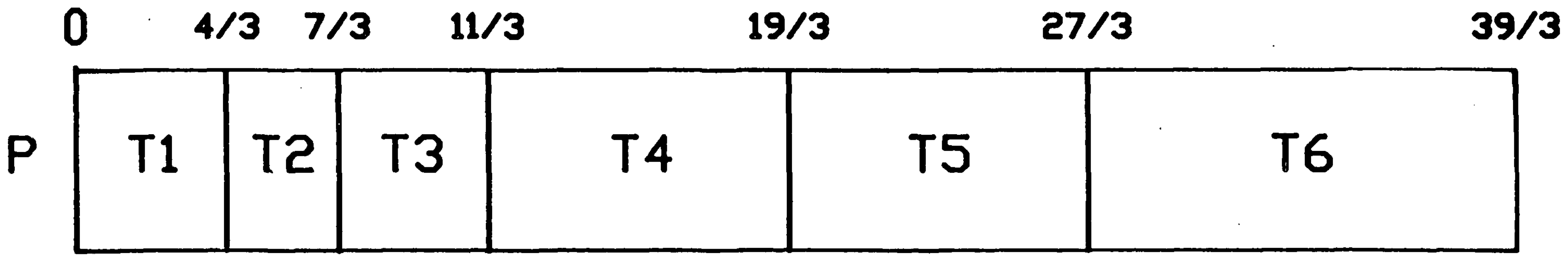


Figure 3. Gantt diagram of a feasible schedule for a single processor system

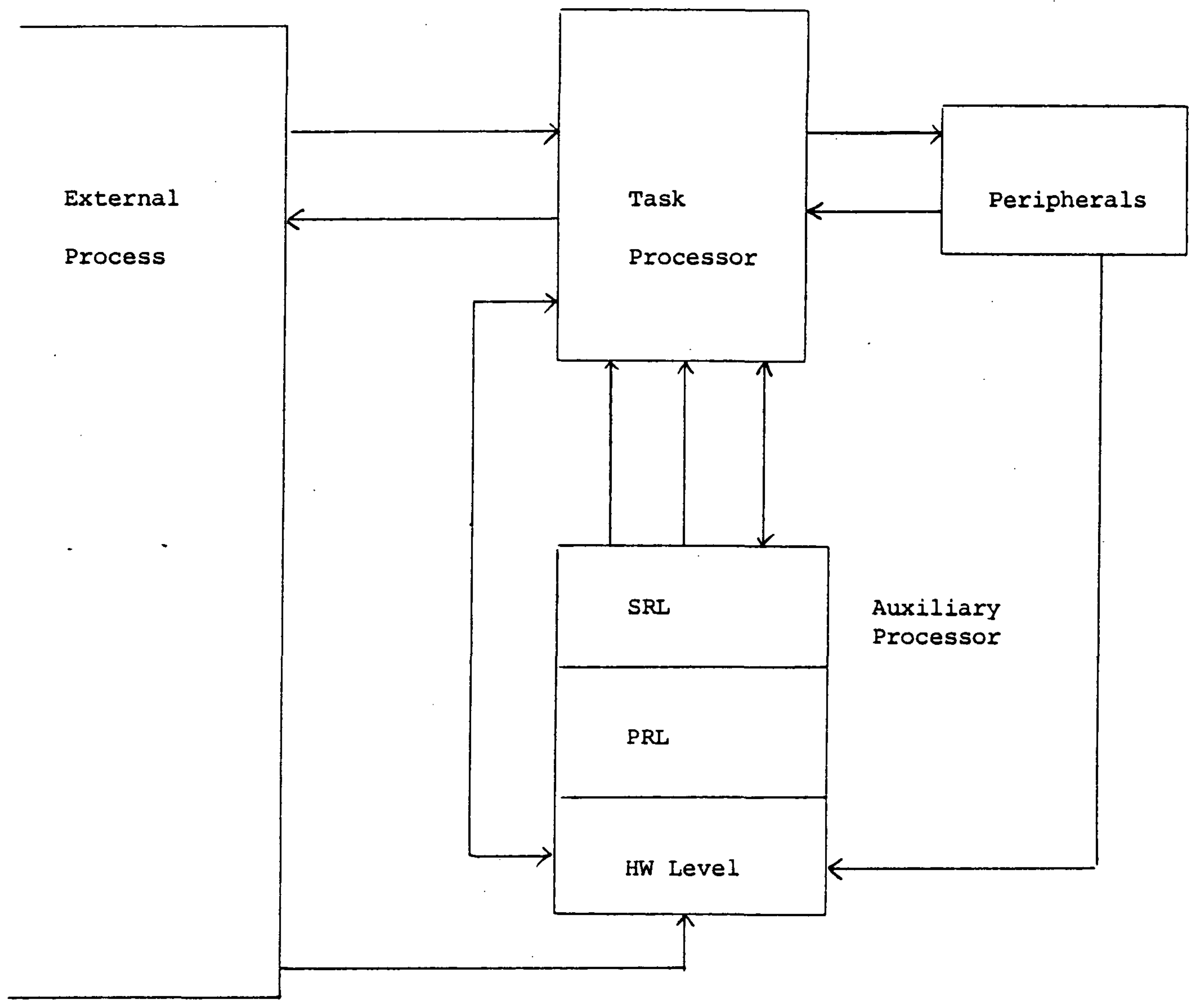


Figure 4. General Configuration and Interconnections

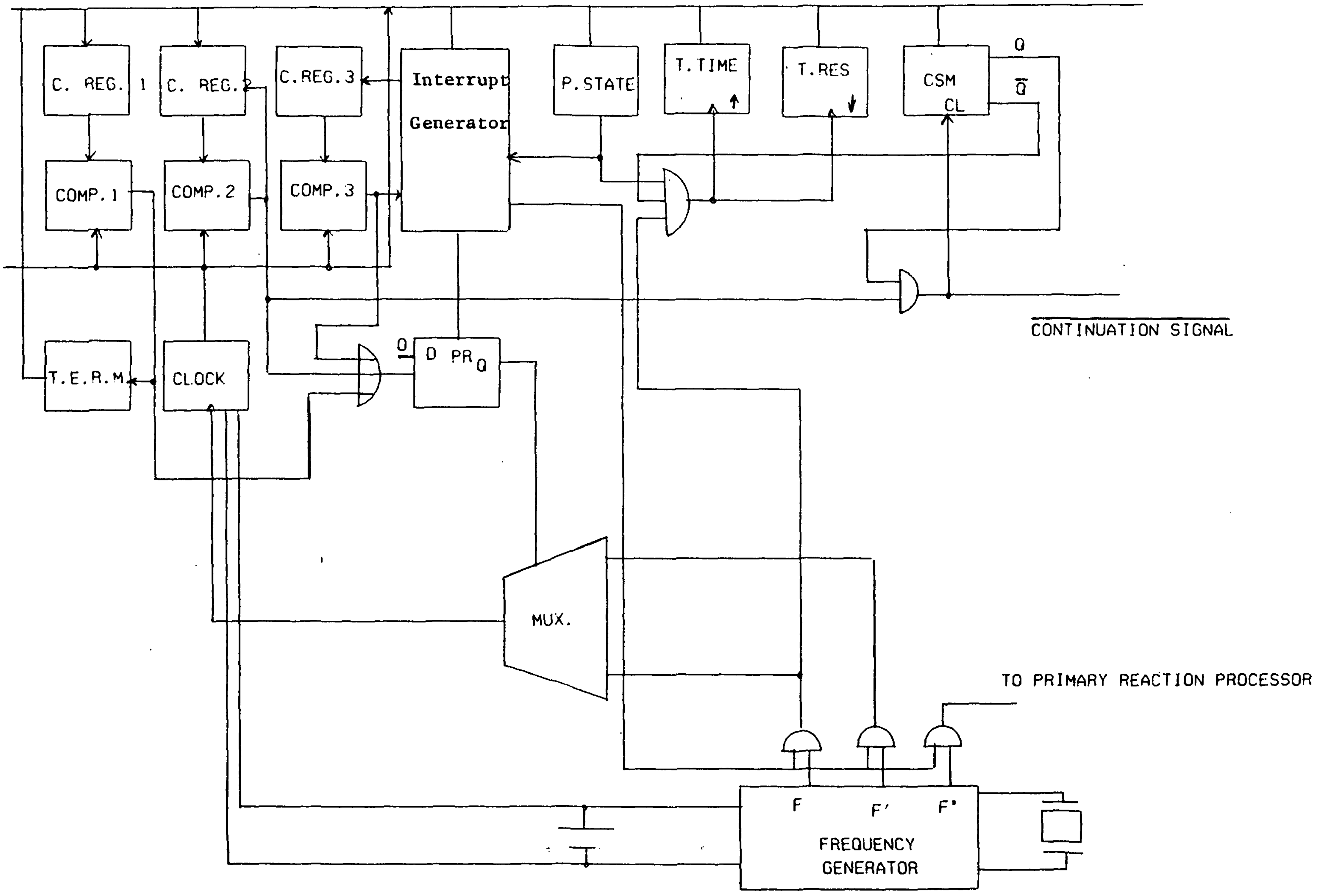


Figure 5. Time Related Hardware Components

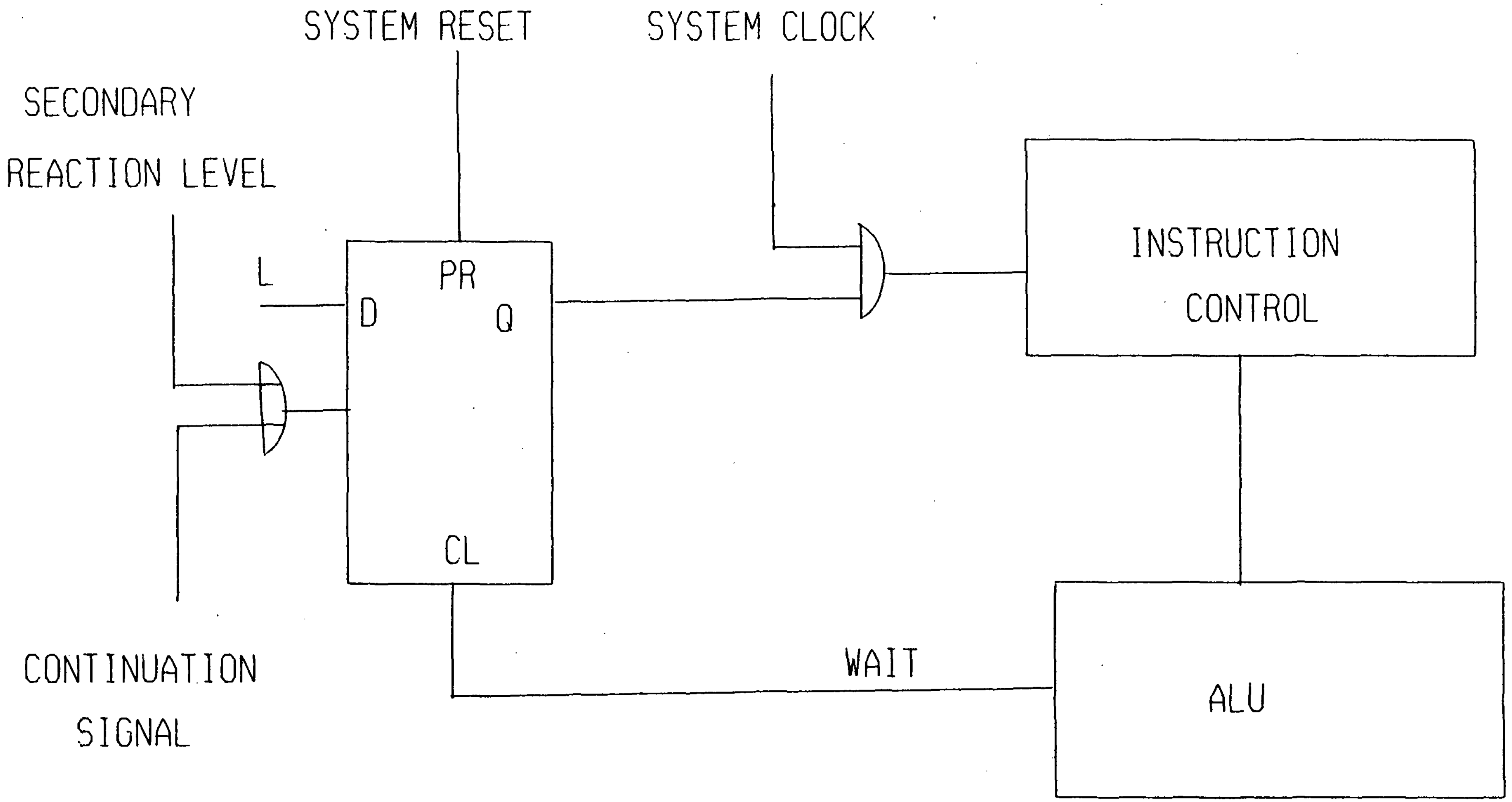


FIGURE 6. EXACT TIMING FEATURE IN A GENERALPROCESSOR

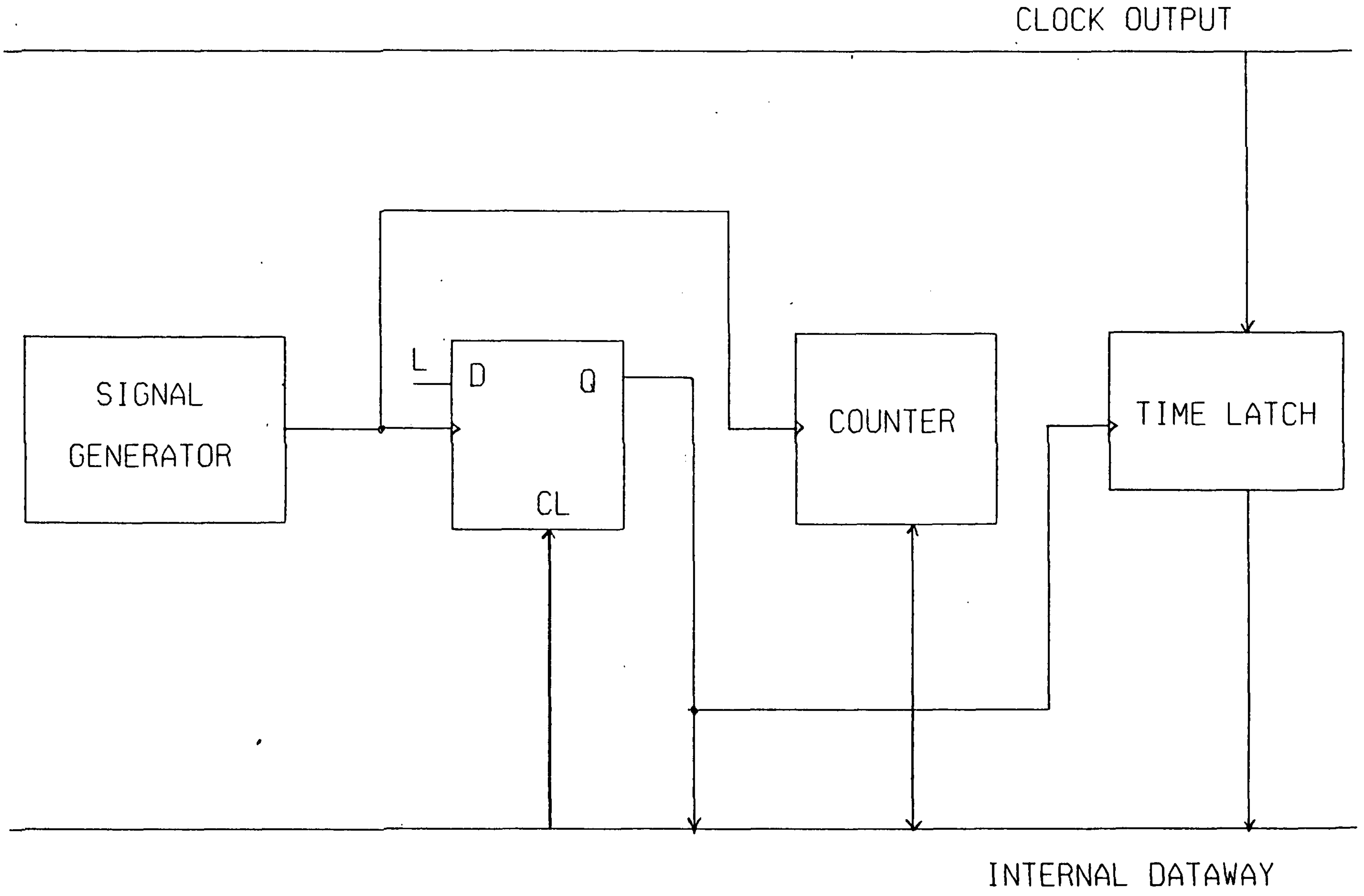


FIGURE 7. EVENT RECOGNITION MODULE

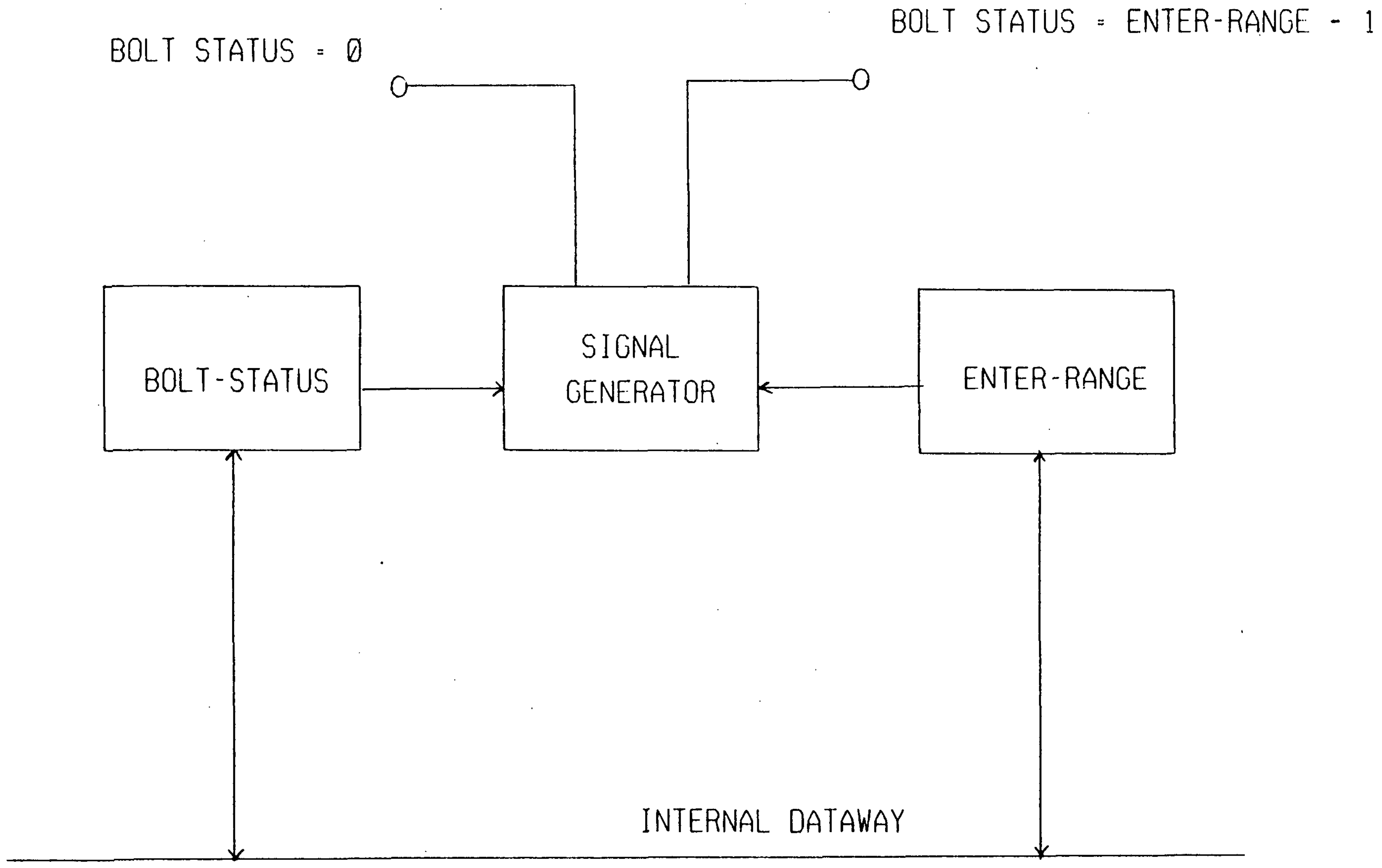


FIGURE 8. HARDWARE IMPLEMENTATION OF BOLT SYNCHRONISER

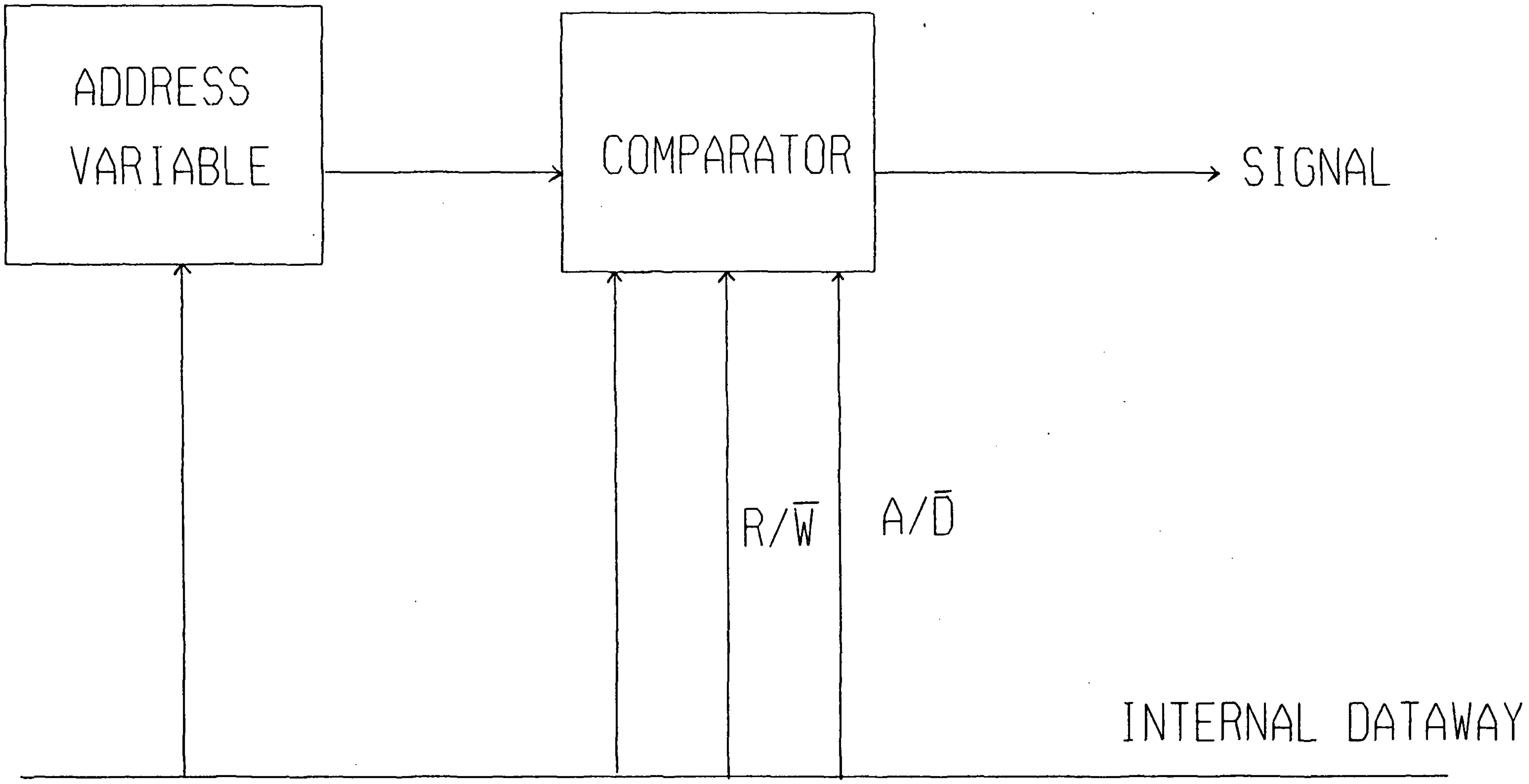


FIGURE 9. HARDWARE TO DETECT VALUE CHANGES OF

SHARED OBJECTS

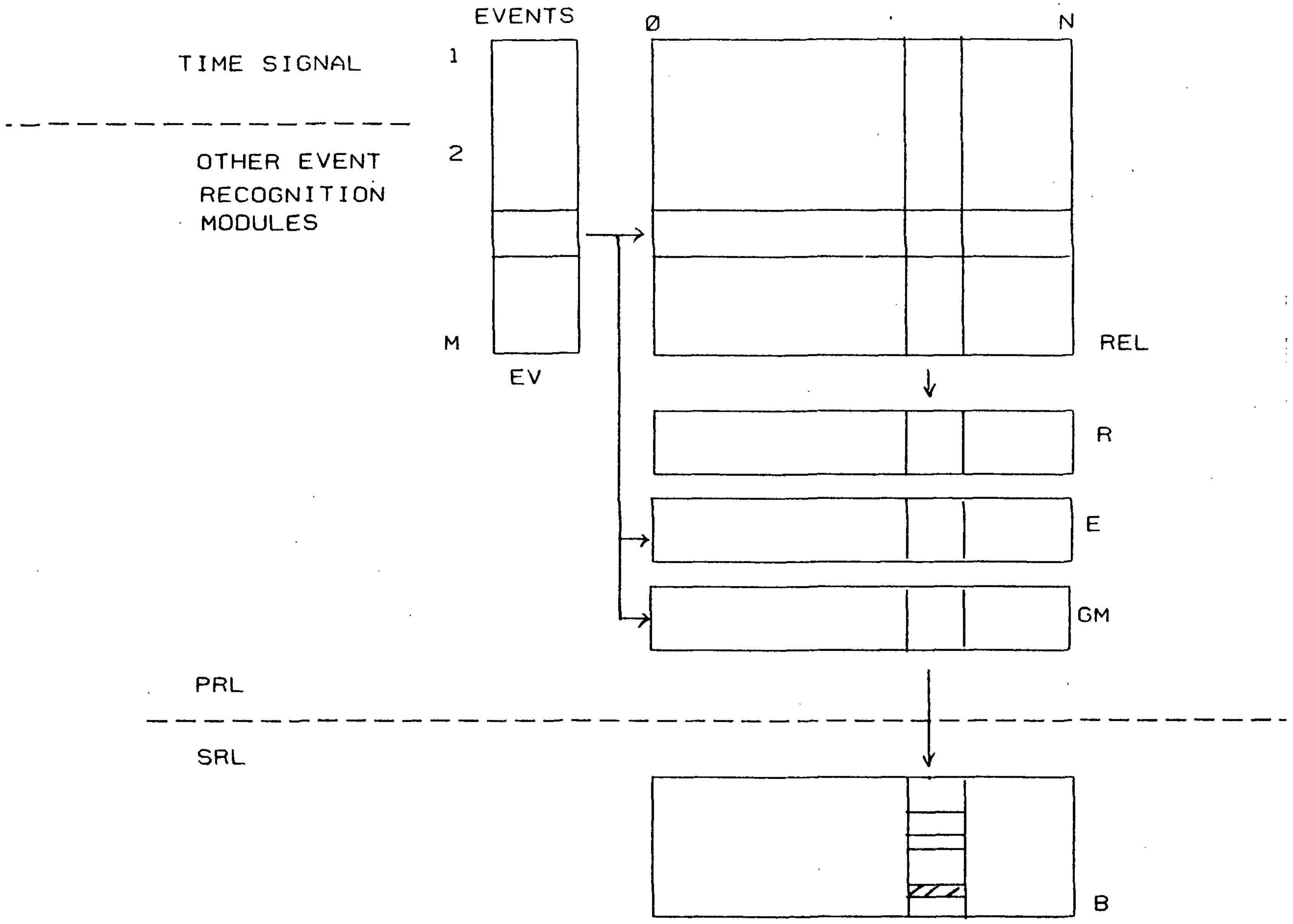


Figure 10. Logical Connection of Objects Maintained by the Auxiliary Processor