CONCEPTS FOR REAL-TIME EXECUTION IN SAFETY-CRITICAL APPLICATIONS

Martin Skambraks *

* *Faculty of Electrical and Computer Engineering, FernUniversität, 58084 Hagen, Germany, E-mail: martin.skambraks@fernuni-hagen.de*

Abstract: Programmable electronic systems (PESs) that are currently employed in safety-critical applications follow either a strictly periodical or a task-based operating policy. This paper discusses these policies with regard to safety aspects and introduces a novel real-time execution concept that combines their advantages. The main characteristics of this new concept are quantisation of time into discrete execution intervals, and partitioning of tasks into execution blocks matching these intervals. This operating principle features task-based software execution without utilising interrupts, eases integration into a holistic safety concept, and conforms particularly well with the safety standard IEC 61508. *Copyright© 2005 IFAC.*

Keywords: Programmable electronic system, programmable controller, real-time operation, forward-recovery, safety criticality, safety licensing.

1. INTRODUCTION

Nowadays, programmable electronic systems (PESs) are employed in many application areas of vital importance. Although their use in safety-critical applications has become common practice within the last 20 years, safety-licensing of these combined software and hardware systems is still problematic. The problems arise less due to inevitable spontaneous physical failures which must be taken into account by design, as rather from the complexity of such systems, which causes an enormous effort for verification.

The safety standard IEC 61508 limits the complexity indirectly by restricting the use of some conventional processing methods. As an example, its design guidelines only permit *'limited use'* of interrupts and pointers in software for the safety integrity levels SIL3 and SIL4 (Part 3, Table B.1). For these safety classes, the standard also states that the use of formal methods for software verification and the avoidance of dynamic objects and variables is *'Highly Recommended'* (Part 3, Tables A.1 and B.1). The latter term denotes that *'if this technique ... is not used then the rationale behind not using it should be detailed during safety planning and agreed with the assessor'*. These guidelines, which at first instance sound incongruously fuzzy, denote indirectly that *design simplicity* is the key to safety.

Depending on the operating policy they follow, the systems that are currently employed in safety-critical applications can be categorised into two classes: *Periodically Operating PESs* and *Task-based PESs*. The first class complies perfectly with IEC 61508, but its field of application is limited to simple control tasks; the second class has a less restricted application field and a more problem-oriented programming style, but it requires more effort for safety-licensing. What is even more important, conventional task-based PESs actually do not comply with IEC 61508 for the two higher safety integrity levels, since they use interrupts to control the program flow.

In the past, safety-related PES concepts have rather evolved by adapting technologies originally developed for industrial use than by considering safety as a design criterion that requires a completely different design approach. The ProfiSafe fieldbus which bases on the ProfiBus technology is only an example. Considering the characteristics of the operating policies mentioned in the previous paragraph, the fundamental requirements for safety-related PESs can easily be summarised. Developing a new PES concept solely based on these requirements is certainly a better approach than originating from an industrial predecessor. This paper exemplifies this design strategy by introducing a novel PES concept which was developed by following the design policy "Progress is the road from the primitive via the complicated to the simple", which was stated by Kurt Biedenkopf in (Biedenkopf 1994).

The remainder of this paper is structured as follows. Section 2 categorises conventional PESs into two classes and discusses their benefits and drawbacks with regard to safety aspects. Based on this, the requirements for a real-time execution concept which perfectly suits safety-licensing in accordance with IEC 61508 are derived in section 3. The subsequent section introduces a novel PES concept that meets these requirements. First, feasibility aspects of the application software and the applied task state model are discussed. Then, task processing without the need for interrupts and its advantages regarding safety licensing are described. At the end of this section, structure and operating principle of the operating system, which conforms to the requirements of the real-time programming language PEARL, are explained. Section 6 covers the integration into a holistic safety concept. Finally, our current state of work and further plans are summarised.

2. CHARACTERISTICS OF AVAILABLE PES

The programmable electronic systems that are currently employed in safety-critical applications can be categorised into *periodically operating* and *task-based* ones, depending on the operating policy they follow.

2.1 Periodically Operating PESs

Periodically operating PESs execute their application-specific programs in processing cycles of constant, fixed duration. The program code is always processed completely within a cycle. The strictly cyclic operating principle facilitates condition-controlled branching merely on a restricted scale; a completely process-controlled program flow is not possible (Bolton 2003, Rabiee 2002). This operating policy does not only limit the field of application to simple control tasks, it also results in a not problem-oriented programming style. Typical representatives of this category are the programable logic controllers (PLCs), which are mostly programmed following the function block policy of IEC 61131. The latter is more problem-oriented, but causes further restriction of the field of application.

All tasks must be implemented in a way that complies with the 'global' cycle time; individual timing constraints are only considered at second instance. Extensive algorithms either cause long cycle periods, or they must be distributed over consecutive cycles. The former increases the system response time, the latter the complexity of the application program. Employing such a PES for several process tasks with extremely different or varying response times is difficult and also increases the complexity of the application program. This PES category typically does not process its program code in direct relation to legal time. Hence, additional effort is necessary to record all system activities relative to Universal Time Coordinated (UTC), and for synchronisation to external systems. Of course, this can be done by simple means, but it increases system complexity unnecessarily.

Nevertheless, the most essential advantage of this PES category is its inherent simplicity. As long as the application process does not lead to the increase in complexity discussed above, the architecture and the temporal execution behaviour of these systems as well as the application-specific software are of remarkably low complexity. This does not only minimise the effort for safety licensing, it also makes, in principle, this PES class suitable for applications of the highest safety class (SIL4). However, the systems that are currently available off-the-shelf are only certified up to SIL3, e. g., SIMATIC S7-414H (`www.siemens.com`).

2.2 Task-based PESs

Task-based PESs use interrupts to control software execution. This allows the program flow to be arbitrarily controlled by the process, and enables asynchronous processing of several tasks. On the one hand, the asynchronous operating principle increases system complexity, since special mechanisms for task synchronisation like, e. g., semaphores are necessary; on the other hand, this operating style makes this PES class more flexible and suitable for extensive control tasks (Shaw 2001). Thus, although asynchronous, task-based programming is more problem oriented than cyclically processed program code – and therefore easier to verify –, this PES category

causes more effort for safety-licensing. This is due to the high complexity of the hardware (especially the processor), the real-time operating system, and their interaction with the application software.

Typically, a very complex part of a processor is the logic for interrupt handling. This interrupt logic does not serve the actual program execution, but merely supports the operating system. Moreover, interrupt handling usually involves a (stack) pointer and a dynamic memory, although IEC 61508 restricts the use of pointers and dynamic objects in highly safety-critical applications.

The complexity of the operating system usually arises from performance reasons. In order to keep the response time of real-time systems short, most operating systems are subdivided into several layers (VDI 1982). The lowest layer – which is closest to the hardware – serves time-critical functions of low computational extent; more extensive functionalities are processed on higher layers. Minimising the computational load caused by the OS kernel functions is another opportunity to keep response times short. That is why mostly priority based scheduling policies are employed, although time based strategies, which cause higher computational load, suit the demands of real-time systems better. Provided the processor allocation is scheduled based on time at all, the time representation usually does not comply with the UTC standard. Thus, recording all system activities in reference to UTC, which is fundamental for most safety-critical applications, requires conversion of the system time to UTC-time. The necessity of such conversions as well as the layered OS structure mentioned above increases the complexity of real-time operating systems considerably.

The complexity caused by the interaction of processor, operating system and application software results from dividing the program code into interruptible and non-interruptible parts, the dependence of execution times on the process as well as the use of mechanisms for task synchronisation. These facts make proving the timeliness of the application software, which is fundamental for these asynchronous systems, either unacceptably expensive or even impossible. Although the use of interrupts does not perfectly comply with IEC 61508 for the two highest safety integrity levels, there are task-based real-time operating systems available that are certified for SIL3, e. g., OSE RTOS (`ww.ose.com`).

It is important to note that the SILs defined by IEC 61508 are measures of the safety of a given process; no individual product can carry a SIL rating. If a vendor claims his product to be certified for SIL3, this means that it is certified for use in a SIL3 environment (Smith and Simpson 2001).

3. DESIGN PRINCIPLES FOR SAFETY-RELATED PES CONCEPTS

The most appropriate design principle for safety-related components is *Simplicity* (Halang and Konakovsky 1999, Vogrin 2000). Design simplicity avoids engineering errors at first instance and eases safety-licensing at second instance. Following simplicity as major design guideline implies to avoid any processing policy or function that increases complexity unnecessarily. Considering the benefits and drawbacks discussed in the previous section, a PES concept for safety-critical applications must realise design simplicity in three aspects:

- Programming style,
- Temporal behaviour, and
- Architecture.

The programming style is simplified by focussing on the timing constraints of the process, rather as on restrictions imposed by the operating principle of the PES. Hence, the software should be organised in tasks. This problem-oriented concept must be supported on the architectural level to enable proving the feasibility by simple (formal) means.

The temporal behaviour is simplified by minimising asynchronous component interactions, e. g., through synchronisation. Additionally, interrupts, if not avoidable, must not affect the execution times of the tasks.

The architecture, which comprises hardware structure and operating system, is simplified by paring down to the essential. Identifying the essential is difficult, since not all dispensable components or processing methods are as obvious as, e. g., cache memories. Consider the following example. Timing constraints for a task are always defined by the process, and priority-based scheduling requires to map these constraints to priorities. Thus, avoiding this mapping by scheduling directly in accordance with the timing constraints is more essential. Paring down to the essential also implies to avoid multilayered structures for the operating system, and to avoid the distinction between internal and external (legal) time by processing all internal activities in direct relation to UTC. The techniques implemented to achieve fault-tolerance also need to be reduced to the essential, while still covering all failure possibilities.

These three simplicity aspects can be considered as the fundamental requirements for safety-related real-time PESs. In the sequel, a novel PES concept is introduced that exemplifies how it is possible to meet these requirements in every respect.

4. A NOVEL PES CONCEPT

The PES concept presented here bases on physical separation of real-time operating system and application processor. Time is quantised into *Execution Intervals*, and tasks are partitioned into *Execution Blocks* matching these intervals. This task execution concept renders the use of interrupts superfluous and conforms particularly well with IEC 61508. Since the concept does not require special mechanisms for task synchronisation, either, the number of functions that the operating system must provide is minimised, and the operating system is realisable in form of a digital logic circuit. This simplifies integration into a holistic safety concept, which unifies the three safety-related functions *failure detection*, *forward recovery at runtime* and *non-intrusive monitoring*. All system components operate periodically and synchronously, leading to a simple and easy to model temporal execution behaviour of the entire system.

4.1 Feasibility of Application Software

In order to ease proving the timely feasibility of application software, the PES supports a particularly simple task state model on the lowest possible level – the hardware level. The model bases on the *execution characteristics*: *worst case execution time* t_C, *maximum response time* t_B, and *minimum activation period* t_T. These parameters specify the time behaviour of each task, and enable to formally prove the feasibility of application software. Fig. 1 illustrates the model, which differs from other ones, e. g. (Halang and Stoyenko 1991), by the state *Suppressing*.

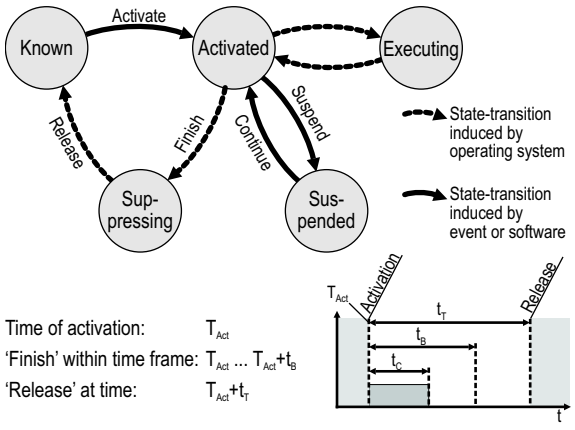

Fig. 1. The task state model employed

The minimum activation period equals the minimum delay between two task activations. Thus, this parameter limits the computational load indirectly. Only tasks in state *Known* can be activated. In case a task is completely processed before the time frame $T_{Act} \ldots T_{Act} + t_T$ elapses, it is transferred to the state *Suppressing*. The transfer back to the state *Known* is carried out at $T_{Act} + t_T$, enabling further activations. The state *Suspended* is provided for synchronisation purposes. Since application software is executed non-pre-emptively in intervals, this state can only be reached by passing the state *Activated*. That means, the suspension is invoked by the program while the task is in the state *Executing*, but the associated state transfer takes place at interval end.

Since the task state model is supported by hardware, the activation of a task is restricted to one instance at a time on the lowest possible level. This complies perfectly with the safety standard 61508, which forbids dynamic instantiation of objects for applications of highest safety criticality.

To sum up, the hardware supported *execution characteristics* allow to formally proof the feasibility of application software. Appropriate feasibility conditions are already known but not subject of this article. Publications that discuss proper feasibility constraints are, e. g., (Teixeira 1978, Sorenson and Hamacher 1975, Halang and Stoyenko 1991).

4.2 Task Execution without Interrupts

Proving the feasibility of application software for conventional real-time systems employing interrupts is problematic, since determining the worst-case execution times (WCET) of tasks is difficult. Each interrupt induces a context-switch of the processor which, in turn, influences the actual response times of all activated tasks. The distinction between interruptible and non-interruptible program parts aggravates the situation. The PES concept presented here avoids this problem by executing tasks without interrupts.

This is achieved by strict separation of an Application Processing Unit (APU) and an Real-Time Operating System Unit (RTOSU). The APU executes the application-specific program code. It consists of a processor in Harvard-architecture, i. e., with separate program and data memories. The RTOSU is responsible for task administration and processor scheduling. The latter is done in accordance with the Earliest-Deadline-First (EDF) scheduling policy. Henn showed in (Henn 1989) that this strategy will always lead to a feasible scheduling, provided timely execution is possible at all.

Time is quantised into discrete *execution intervals* and tasks are *partitioned* into a number of *execution blocks* each. The execution intervals have a fixed duration, and are defined for the syn-

chronously operating RTOSU and APU. The execution blocks have the following characteristics.

- Each execution block can be executed by the APU within a single execution interval.
- The execution of a block is not pre-emptable.
- Data exchange between execution blocks is only possible via the APU memory; the content of the processor registers is lost at the end of a block's execution.
- The execution blocks of a task are indexed for identification.
- The execution blocks of a task do not need to be executed in consecutive order. For each task, the RTOSU stores a parameter called *NextBlock* in the task list, which identifies the subsequent execution block.

At the beginning of each execution interval, the RTOSU outputs the *ID of Block to Execute*, which corresponds to the *NextBlock* parameter of the task that must be executed according to the scheduling algorithm. The APU reads this ID and processes the associated execution block. When the APU completes the block at the end of the execution interval, it outputs the *ID of Next Block*, which identifies the task's execution block that needs to be executed next. The RTOSU reads the ID and stores it in the task list as new NextBlock parameter. The flow chart in Fig. 2 illustrates this in more detail.

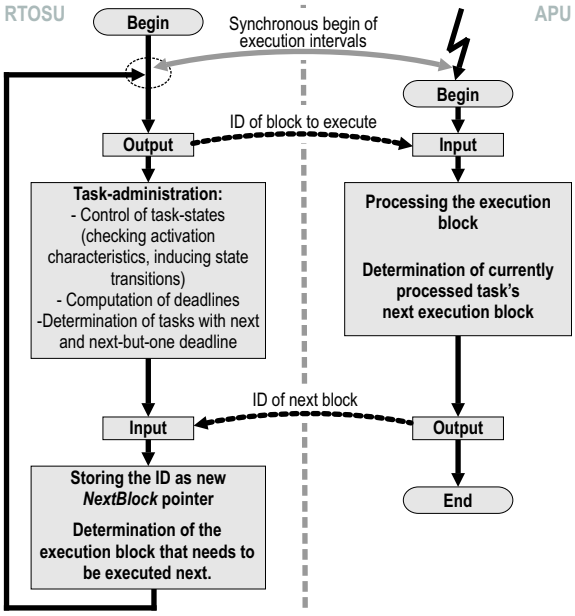

Fig. 2. Illustration of the operating principle without interrupts

If the executed block was a task's last one, i. e., if a task is completely executed, the APU outputs the block ID 'Nil'. In this case, the RTOSU sets the state of the associated task from *Activated* to *Suppressing*. Additionally, the RTOSU takes this completion into account to compute the *ID of Block to Execute* for the next execution interval. This is why the RTOSU does not only determine the task with the earliest, but also the task with the next-but-one deadline. This enables the RTOSU to immediately output the *NextBlock* identifier of the task with the next-but-one deadline, in case the task with the next deadline corresponds to the task just been processed *and* just been completed by the APU.

This operating principle renouncing the use of interrupts minimises the complexity of the APU processor, as no interrupts need to be supported. What is even more significant in terms of safety, the operating principle without interrupts also makes synchronisation mechanisms like semaphores superfluous, since each task has non-interruptible exclusive access to the processor during an execution interval. Tasks can communicate with each other via the data memory without the danger of interruptions while writing messages. Using variables stored in the data memory, mutual exclusive access to peripheral components can be realised by simple means. Alternating access of several tasks can be realised in a similar way, it only requires the ability of programmed task activation. The absence of RTOS-supported synchronisation mechanisms results in further simplification of the design. However, since mutually exclusive access is realisable, methods to prevent deadlocks need to be applied during software development.

In summary, this task execution concept without the necessity of interrupts simplifies the time behaviour as well as the hardware structure significantly, eases formal verification, and increases conformity to IEC 61508 for systems of highest safety-criticality (SIL3/SIL4). Of course, the proposed operating principle requires special compilation of the application software.

4.3 PEARL-based Hardware-RTOS

The RTOS of the proposed PES bases on the task concept of the **P**rocess and **E**xperiment **A**utomation **R**ealtime **L**anguage (PEARL), which was standardised by DIN 66253-2 (1998). One of the interesting features of this language is the direct notion of time (Hamuda and Tsai 2003). This enables exact and problem-oriented specification of timing conditions to activate, terminate, suspend, continue or resume tasks (GI 1998). A periodic task activation during a given time-frame is specified by

```
AT {clock-expression |
    [asynchronous-event-expression]+duration1}
EVERY duration2 DURING duration3
ACTIVATE task-name
```

This is the most general form of a task activation schedule (Halang and Stoyenko 1991). The hard-

ware implementation of the RTOSU inherently supports such activation plans. Therefore, each task is assigned a set of parameters, which facilitates configuration for various activation conditions. These *activation parameters* allow to specify, e. g., that a task is activated after the occurrence of an asynchronous signal and an further, pre-defined delay.

The proposed PES implements the EDF algorithm in hardware, rendering the differences between priority-based and EDF scheduling to be irrelevant in terms of computing time. Thus, there is no need for task priorities, and the time-based scheduling policy can be applied. The internal clock of the PES is permanently synchronised to UTC, which is the internationally standardised sole *legal* time reference. It is available worldwide via, for instance, GPS and GLONASS. The PES processes *all* internal actions with reference to UTC. Thus, there is only one time-base in an entire distributed system. This reduces the system complexity, since the problems related to different time bases are prevented.

On the hardware level, time instants are represented by 48 bits wide data words consisting of binary numbers for year, month etc., and a bit identifying whether a time value is *absolute* or *relative*. Relative time values are the results of adding (resp. subtracting) time values to (resp. from) absolute time values. In order to keep the hardware simple, the RTOSU performs these computations neglecting leap seconds and the different numbers of days in a month. These time irregularities are taken into account at the moment they occur by appropriate substraction operations. Due to the structure of the RTOS operations, this simple technique guarantees correct time processing.

A perfect real-time operating system would permanently check the activation conditions of all tasks, and inform about the task to be executed according to the scheduling policy. This demand for a continuous working pattern leads to the objective of implementing the RTOS in form of a digital logic circuit that processes the kernel algorithms for all tasks in parallel. Unfortunately, this would require an unacceptably large amount of logic gates, considering the extent of the RTOS kernel algorithms and the fact that a typical real-time application consists of some 10 to 50 tasks.

For this reason, the hardware architecture of the RTOS combines parallel and sequential processing. The kernel algorithms are structured in a way as to allow for parallel processing of the operations related to a single task, whereas all tasks are sequentially subjected to these operations. Fig. 3 illustrates this processing pattern. It shows the main parts of the Real-Time Operating System Unit (RTOSU), viz., the *Task Data Administration Unit (TDAU)* and the unit for *Activation Control and Scheduling (ACS)*.

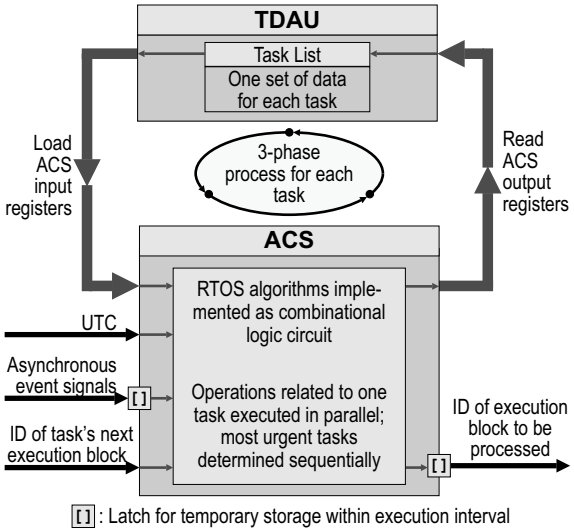

Fig. 3. Operating principle of the hardware RTOS; for each task, the 3-phase process is carried out once within an execution interval

The TDAU administrates a *Task List*, which contains a set of parameters for each task such as information about the current task states, the activation characteristics, and the execution properties. The task list has a static size, i. e., all tasks a certain application is consisting of must be registered with the RTOSU at set-up time. Thus, in conformance with the requirements of IEC 61508 for SIL4 applications, dynamic instantiation of tasks (resp. objects) is not supported. Instead, the activation characteristics of a task can be modified.

The TDAU co-operates closely with the ACS while sequentially processing all tasks within each execution interval. During this *Sequential Task Administration (STA)*, the TDAU initiates for each task a three-phase process:

(1) First, the TDAU accesses the Task List and transfers the entire data set to dedicated input registers of the ACS.
(2) Then the ACS processes the task data. This is carried out by a combinational logic circuit within one clock cycle.
(3) During the last phase, the TDAU reads the updated task data from the ACS and transfers them back to the Task List.

The ACS is responsible for the following operations:

(1) Checking the activation characteristics (this includes, e. g., checking time schedules and asynchronous occurrences),
(2) Supervising task state transitions,
(3) Computing deadlines,
(4) Generation of updated task parameters,

(5) Identifying the task with the earliest deadline and the one next in line,
(6) Output of the *ID of Block to Execute*.

The first four operations are separately executable for each task. Therefore, they are performed in parallel by a combinational digital circuit. The latter, which implements the most substantial and most critical functionality of an RTOS, is inherent simple – especially in comparison to a conventional software implementation.

The fifth item requires comparing the deadlines of all activated tasks. This is carried out sequentially, while the IDs of the two most urgent tasks are temporarily stored within the iterations of the 3-phase process. The ID of the execution block that needs to be processed in the subsequent interval is output at the end of an execution interval, after the APU submitted an ID to the RTOSU.

5. SAFETY-RELATED INTEGRATION

The operating strategies of the PES concept described in the previous section aim to avoid design errors by simplifying both architecture and operating principle. In order to guarantee safe and reliable operation, these strategies must be integrated into a holistic safety concept, which also takes spontaneous hardware failures into account and does not ruin simplicity.

The standard IEC 61508 recommends various techniques to reduce the influence of hardware failures. Some of these techniques are, e. g., redundant memory banks or multiple processors combined with majority voting. However, these techniques usually cover only few failure sources, and a combination of several techniques is necessary to cope with all failure possibilities. This increases system complexity significantly. Thus, applying these techniques would infringe the strategy of design for simplicity.

This is why a rather unusual but more integrative approach was taken to reduce the influence of hardware failures. In a redundant configuration, each PES outputs a *Serial Data Stream (SDS)* that provides full information about the internal processing states. This SDS concept unifies the following three safety functions.

Detection of processing errors: Each PES can detect processing errors by comparing its SDS with the SDSs of the other PESs.

Forward recovery at runtime: In case a PES is affected by a transient hardware fault, the SDSs of the redundant PESs enable to copy the internal state and to resume processing at runtime.

Recording process activities: The SDSs can be utilised to record the system's execution behaviour for later program flow analysis.

The redundant PES instances operate periodically in *Module Cycles*, which are synchronised to UTC. Thus, there is no need for an additional global clock signal. These module cycles define the begin of the execution intervals. Fig. 4 illustrates this.

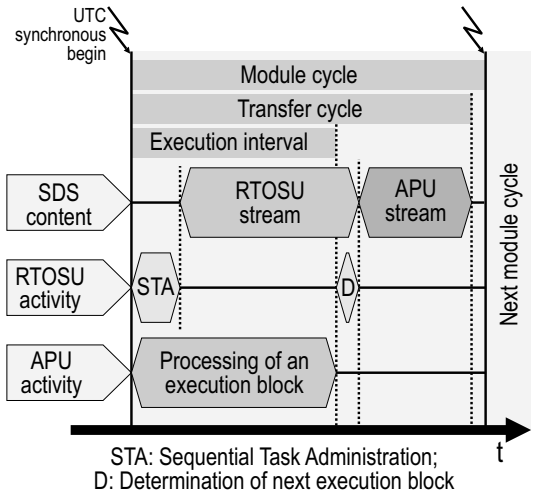

Fig. 4. Module cycles, which are synchronised to UTC, define the begin of execution intervals and transfer cycles

5.1 The SDS Concept

Each PES creates a *Serial Data Stream (SDS)*. This SDS serially transfers all information that is necessary to determine the exact state of the PES at the beginning of a module cycle. The SDS is organised in *Transfer Cycles* which match the module cycles. This causes redundant PES instances to output their SDSs synchronously.

The state of a PES is defined by the content of all its storage devices, e. g., registers, latches and memory blocks. Certainly, complete transfer of this content via the SDS in each transfer cycle would enable error detection and runtime recovery. However, this would either restrict the maximum amount of storage devices in the PES, require an unacceptably high data transfer rate, or limit the minimum duration of a transfer cycle and – as a consequence – of a module cycle. That is why a complete state transfer is distributed over a number of consecutive cycles. During each transfer cycle, only the most recent data changes (resp. state changes) are transferred via the SDS. This concept minimises the required data transfer rate. An extension of this concept guarantees complete state transfer within a predefined number of transfer cycles.

Due to the operating principle of the PES, the SDS is composed of the *RTOSU Stream* and

the *APU Stream*. The RTOSU Stream is transferred first, followed by the APU Stream. Fig. 4 shows the SDS content in relation to the RTOSU and APU activities. At the beginning of each module cycle, the RTOSU starts the *Sequential Task Administration (STA)*, and the APU starts processing one execution block. At the time the RTOSU finishes the STA, the processing inside the RTOSU is almost complete, only the next execution block still needs to be determined. Thus, the state of nearly all storage devices inside the RTOSU is fixed for the rest of the module cycle and the RTOSU Stream can be generated.

When the APU finishes processing an execution block, it outputs an identifier. This identifier either points to the next execution block of the task or signalises that the task has been processed completely. The RTOSU uses this identifier to determine the execution block that needs to be processed in the next execution interval. This identifier and the ID of the next execution block is inserted at the end of the RTOSU Stream. Subsequently, the APU Stream starts.

The APU stream terminates a short time before the next cycle begins. Thus, there is a small time gap between the end of the transfer cycle and the end of the module cycle. This gap is necessary to guarantee evaluation of the received SDSs before the next module cycle begins. The time gap must compensate the SDS transfer delays and the (slight) synchronisation differences between redundant PES instances.

Detection of processing errors:
Errors are detected at bit level. A PES is denoted as erroneous, if one bit of the SDS differs from its associated majority value. Therefore, each PES includes a *Majority Voter* which compares the SDS of all redundant PES instances. The majority voter does not only compare the bits of all data streams to determine the majority, it also takes the validity of each stream into account. An SDS is denoted as 'invalid' if, e. g., the associated PES is in 'Error State' or if the reception fails. The majority voter uses only *valid* streams to determine the majority. If a stream differs once (resp. in one bit), the majority voter does not use this stream for subsequent evaluation during the rest of the transfer cycle. In case a PES's own SDS does not equal the majority voter's output during the whole transfer cycle, it is transferred to 'Error State'. The PES stays in this state until the recovery process completes successfully.

Forward recovery at runtime:
When a PES has been transferred to 'Error State', the recovery process is automatically initiated in the subsequent module cycle. This process can be divided into the two phases

- *State Equalisation* and
- *(Waiting for) Resumption.*

State equalisation refers to equalising the internal state of a PES to the state of other running instances. The internal state of a PES is defined by the content of all its storage elements (latches, registers, memories). This includes the memory content of the APU as well as the register content of the RTOSU (e. g., task parameters). The main problem of state equalisation at runtime is that the internal state of the running modules is not static. While the internal state is copied, the state changes continuously. Due to the physical separation of real-time operating system and application processing, the runtime equalisation is divided into the two parts

- equalisation of the APU state and
- equalisation of the RTOSU state.

The PES can only resume its processing when the internal state is totally equivalent to the state of the (still running) redundant PES instances. The applied equalisation method guarantees this equivalence only at distinct points in time. Thus, a PES in 'Error State' must wait for such a moment to induce the resumption, which must take place at the beginning of a module cycle.

5.2 Equalisation of the APU State

In the proposed PES concept, execution blocks are executed without interruption. Thus, context-switches occur only at the beginning of a module cycle. These context switches do not require to save the register content of the processor, since using these registers to store data between module cycles is not allowed. Hence, equalising the APU state does not require to transfer any APU register contents. As long as the resumption takes place at begin of a module cycle, equalising the memory content is absolutely sufficient.

The memory of the APU is subdivided into *Program Memory*, *Help Memory* and *Data Memory*. The *Program Memory* contains the software code, it supports only reading access (ROM). The purpose of the *Help Memory* is to store intermediate results during an execution interval; its content is lost at the end of each execution interval. Data that need to be accessed in multiple execution blocks must be stored in the *Data Memory*. Only the data memory needs to be copied for state equalisation of the APU.

The data are transferred in serial form. With modern techniques it is possible to achieve data transfer rates of several Mbits per second. Nevertheless, the data transfer rates will always be lower than the speed at which the APU can process

data. In order to limit the frequency of data word changes in the data memory, the number of write accesses allowed during each execution interval is limited. All write accesses to the data memory are recorded (addresses and associated data words) and subsequently output in form of a serial data stream called *APU Stream*.

The recovery process requires that the complete data memory content is transferred via the APU Streams within a pre-defined time frame. This is not automatically the case, since only the write accesses induced by the application-specific software are added to the APU Stream, and the software might not perform write accesses to all data memory cells within this time frame. To overcome this problem, the APU uses *Idle Execution Intervals* to sequentially read and write back all cells of the data memory. This usage of idle execution intervals guarantees complete recovery of the data memory content. The time for recovery depends on the size of the data memory and the occurrence frequency of idle execution intervals, which must be specified at setup time of a system.

The point in time at which all memory cells are at least once 'updated' is automatically detected. From that point in time on, the PES in error state and the remaining PESs have at the beginning of each module cycle identical contents in their data memories.

5.3 Equalisation of the RTOSU State

The internal state of the RTOSU is defined by the content of all its storage devices: the registers (or latches) and memory blocks.

Equalisation of the register content:
Most registers of the RTOSU are solely used to temporarily store data within a module cycle; they do not exchange information from one module cycle to the next. The content of these registers does not need to be copied during the recovery process, since it is irrelevant as long as resumption takes place at the beginning of a module cycle. However, some registers are utilised to exchange information from one module cycle to the next and must, therefore, be considered in terms of state equalisation. Fortunately, most of them (e. g., the *ID of Next Block Latch*, in which the APU writes the ID of the task's next execution block) have no meaningful content after idle execution intervals. Thus, resuming the processing after an idle interval is the simplest solution for state equalisation.

Only two circuit parts of the RTOSU contain registers that have meaningful content after idle intervals. These are the *ID of Block to Execute* latch, which signalises the execution block that needs to be processed by the APU, and the *Task Info Unit*, which enables reading access to important task data (e. g. current task state). The *ID of Block to Execute* can change at the end of any module cycle and, hence, needs to be transferred via the SDS within each transfer cycle. The register content of the *Task Info* unit is derived from the current (resp. updated) task list data during the sequential task administration of each module cycle. The content of this register set does not need to be transferred via SDS, since it can be derived from the recovered task list data.

Equalisation of the Task List memory:
The memory block that stores the Task List is the only memory block that needs to be recovered in terms of state equalisation. Equalising this memory is more complex than equalising the APU memory content, because the frequency of *RTOSU task data changes* (in the following only 'task data changes') cannot be limited like the write accesses to the data memory of the APU. Every task can perform a state transition at any time, and independent from the states of the other tasks. Thus, in theory, it is possible that all tasks change their state from 'known' to 'activated' within the same module cycle. This would cause a huge amount of data changes (storing activation times, modifications of the activation schedules). Transferring such a huge amount of data within one cycle t_C to achieve state equalisation is not desirable, since it would either require a very high data transfer bandwidth, restrict the minimum cycle duration t_C, or limit the number of manageable tasks.

The amount of data that must be transferred within one module cycle can be reduced by taking the task characteristics into account. It is not possible to predict the time instants of task data changes (resp. state changes), but the maximum number of task data changes is known in advance. The activation of a task itself causes a data change, but also each 'Suspend' operation that the software code of a task contains. Since the maximum activation frequency of each task is indirectly given by the *minimum activation period*, the maximum rate of RTOS data changes is limited, computable and constant.

The 'Age-Variable' concept:
Some Task List data are changed quite frequently, while others are changed only seldom. If one data value changes multiple times during the equalisation process, only its last value needs to be transferred, since the old values are obsolete. This characteristic is useful to minimise the required data transfer rate. Therefore, an integer variable is assigned to each RTOSU memory data word that represents the age of the data value. These integers are subsequently referred to as *Age* variables.

By default, the *age* variables are set to zero. Each time the associated data word is modified, the integer value is set to the maximum representable value. If the age value does not equal zero or '1', these variables are decremented by one at the beginning of each module cycle. Thus, the lowest integer values (except zero) identify the 'oldest' modified values. During each cycle, a subset of the oldest data words is inserted in the SDS and the associated age variables is set to zero.

This operating policy restricts the transfer to data words that are modified. An extension is necessary to facilitate transfer of all RTOS data words within a reasonable time frame. The extension must take into account, that all redundant PES instances must generate identical SDSs, otherwise majority voting methods cannot be applied to check the streams received. This requires a strategy that transfers the remaining, not modified data words of redundant SDS instances at the same points in time. Since all PESs are synchronised to UTC, this can be achieved by setting all age variables to their maximum representable value at UTC-synchronous time instants.

With the proposed extension, the policy of inserting a subset of the oldest data words in the SDS will lead to a total transfer of the RTOSU data, provided the frequency of RTOSU data word changes is sufficiently low. Therefore, the number of RTOSU data words that the SDS includes within each transfer cycle must be higher than the average number of changed data words.

Each time an RTOSU data word is added to the SDS, the associated age variable is set to zero. The complete transfer is signaled by all age variables being zero. This situation is suitable to resume the processing of the execution module.

5.4 Resumption

The state equalisation of the APU data memory and the RTOSU data is initiated after a processing error has been detected. Within a predictable period of time after the PES state has been denoted as erroneous, the identity of the data memory content will be provided at the beginning of each module cycle. Thus, from that time on, the PES must only wait until the content of the RTOSU storage devices has been completely recovered.

The state equalisation of the RTOSU also completes within a predictable period of time. In contrast to the equalisation of the APU data memory, this does not necessarily mean that this state identity is also reached in all subsequent module cycles. It is most probable, but if, e. g., too many task state transitions happen within one module cycle, the amount of RTOSU data changes is too large to be transferred via the SDS within one transfer cycle and, consequently, state identity cannot be maintained in the subsequent cycle. Moreover, total state equalisation of all RTOSU storage devices can only be reached after an idle execution interval. Because of this fact, the occurrence frequency of idle execution intervals must be selected high enough to guarantee the coincidence of state equivalence and an idle execution interval within a given time frame.

At the moment the identity of the data memory content and the RTOSU data is reached after an idle execution interval, the recovery process is complete and the PES can resume processing.

5.5 Communication Technique

For communication *one* interface is used which does not only suit the needs to exchange the serial data streams between the redundant PES instances, but also supports data exchange with the process periphery (e. g. sensors, actuators). This results in low wiring expenses. The communication technique bases on the fieldbus 'Interbus S', because of its low hardware requirements and inherent simplicity (Baginski and Müller 1998). All system nodes, i. e., the redundant PES instances as well as sensors and actuators, are connected to a ring and data are transferred from node to node as in a shift register. The data transfer is organised in *transfer cycles* that match the UTC-synchronous module cycles. Each data word is shifted through all nodes of the ring within a cycle.

Conventional ring-based communication techniques increase safety by using both possible data transfer directions. This approach cannot guarantee system availability in case of more than one ring interruption or device outage. That is why the proposed PES makes use of a different approach, which bases on a second (i. e., third) communication ring. The output of a node is not only connected to the next node in the ring, but also to the next-but-one node. The additional wires are arranged in a second communication ring which is *physically separated* from the first one. This second ring is subsequently called *Reserve Communication Ring (RCR)*. Similarly, the first communication ring is named *Primary Communication Ring (PCR)*. Fig. 5 illustrates the connection scheme.

With the RCR, communication does not fail even if one communication wire is cut or a node fails completely. This 'multi-ring' technique allows scalable safety by adding further rings and, moreover, can even be combined with the bi-directional method. Fig. 5 illustrates the connection scheme.

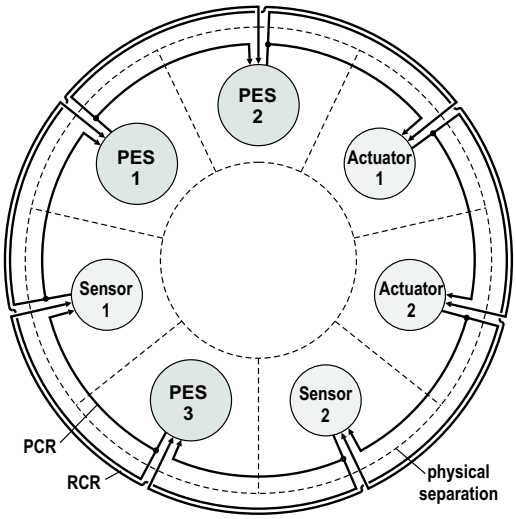

Fig. 5. Connection scheme with three redundant PESs

Although all nodes are physically connected to one ring, the communication method employs multiple *transfer channels*. That means, each physical wire transfers multiple channels. There is one channel for general I/O communication and one channel for each SDS. This multi-channel technique causes only small delays to shift the SDS through all nodes, since the transmitted data must only be repeated by the PES instances passed. Fig. 6 illustrates this multi-channel technique for a PES configuration with three redundant instances.

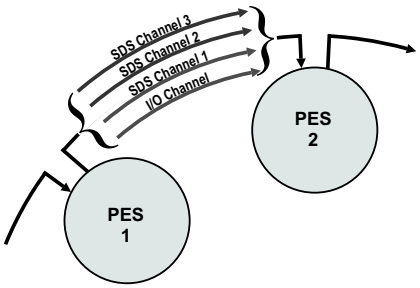

Fig. 6. Multi-channel technique; since each PES generates an SDS, three SDS channels are necessary for a PES configuration with three redundant instances

5.6 Hardware Implementation

The SDS concept as well as the periodic communication method have been realised in form of a digital logic circuit. In contrast to most other task-based PESs, the proposed architecture even realises forward recovery in hardware. This hardware-oriented approach simplifies the PES architecture, since the RTOSU and the APU have also been realised in hardware and, hence, no complex software/hardware interfaces between system components are necessary. Furthermore, the temporal behaviour of the entire system is simplified, since all PES components operate periodically and synchronously. All PES functionalities ,e. g., task administration, forward recovery, are realised in hardware and the PES software consists only of application specific program code. This clear distinction between architectural components of the PES and the application specific software minimises the effort for safety-licensing, once the PES has been certified for use in a safety-related environment.

6. CONCLUSION

The programmable electronic systems (PESs) that are currently employed in safety-critical applications have been sub-divided into two classes. *Periodically operating PESs* operate in cycles of constant duration and process their program code always completely within each cycle. They suit the demands for safety-licensing best, since their architecture, temporal behaviour and application specific software are of remarkable low complexity. However, their field of application is limited to simple control tasks. *Task-based PESs* use interrupts to control software execution. On the one hand, they support a more problem-oriented programming style and have a less restricted field of application, on the other hand, their inherent complexity causes huge effort for safety licensing. Moreover, since interrupts play a major role in conventional task-based systems, this PES category actually does not comply with the safety standard IEC 61508, which restricts the use of interrupts for the two higher safety integrity levels SIL 3 and SIL 4.

Based on the benefits and drawbacks of the two PES classes, the ideal characteristics for a safety-related programmable real-time system were derived and mapped to requirements for the programming style, the temporal behaviour and the system architecture. Then, a novel PES concept was introduced that meets these requirements in every respect and combines the advantages of both conventional PES categories. What distinguishes this concept from other ones is that 'Design for Simplicity' was followed as the major development guideline. The simplicity achieved eases verification and – more importantly – lowers the cost for safety licensing.

Simplicity of design is achieved by strictly separating operating system and application processor. The operating system realises the task-processing strategies of the real-time programming language PEARL in form of a digital logic circuit. This enables task scheduling in direct relation to **U**niversal **T**ime **C**o-ordinated (UTC). Time is

quantised into *Execution Intervals*, and tasks are partitioned into *Execution Blocks* matching these intervals. This operating principle renders the use of interrupts superfluous. As a result, not only the processor architecture is simplified, but also the time behaviour of the total system. The hardware implementation of the RTOS ensures short response times without utilising a complex multilayered structure nor minimising the computational effort by applying a primitive priority-based scheduling algorithm. Instead, the kernel algorithms follow the time-based Earliest-Deadline-First (EDF) policy.

Many techniques were developed in the past to increase the reliability and availability of programmable electronic systems. Unfortunately, most of these techniques cover only a small amount of failure sources and various techniques need to be combined in safety-related systems, causing the design complexity to increase significantly. The proposed system integrates error detection, forward recovery, and non-intrusive monitoring in a unified concept. This functional unification, which bases on exchanging Serial Data Streams between redundant PES instances, leads to a hardware design with minimum complexity.

The proposed PES concept is completed by an inherently simple communication strategy, which serves both data exchange between redundant PES instances and with the process peripherals. A 'multi-ring' technique combines high, scalable reliability and low wiring expenses with an especially simple structure. Like the application processing, the communication is performed in discrete intervals.

All system components operate periodically and synchronously, leading to a simple and easy to model temporal behaviour of the entire system. As a result, formal verification of the application-specific software is simplified. In comparison to conventional cyclically operating PES, the field of application of the proposed PES concept is larger, since it is capable of task-based software execution with process-dependent program lows.

So far, a VHDL description has been prepared that realises, to a large extent, the proposed PES concept as a System-on-Chip. The APU builds up on a soft-processor compatible to the well known 8051 processor from Intel, which is provided as circuit description by Oregano Systems. The VHDL design has extensively been tested by simulation and, then, implemented in a field programmable gate array. Our current work focuses on completing this prototype. Later, the 8051 soft-processor is planned to be replaced by a processor core that is formally proven correct. Such a core is currently under development.

The article covered only the fundamental operating principles and how they simplify the design. Various aspects like, e. g., the exact synchronisation of redundant PES instances, the handling of leap seconds and leap days, the data transfer via SDS, detection of processing failures in a redundant configuration, forward recovery at runtime, etc. could not be discussed in detail. These problems have all been solved in particularly simple ways but describing them would exceed the page limit for this article. The interested reader is invited to contact the author for further information.

REFERENCES

Baginski, A. and M. Müller (1998). *Interbus*. Hüthig Verlag. Heidelberg.

Biedenkopf, K. (1994). Komplexität und Kompliziertheit. *Informatik Spektrum* **17**, 82–86.

Bolton, W. (2003). *Programmable Logic Controllers*. Elsevier Books. Oxford.

GI, Technical Committee 4.4.2 (1998). *PEARL90 Language Report*. `www.real-time.de`. Bonn.

Halang, W. A. and A. D. Stoyenko (1991). *Constructing Predictable Real Time Systems*. Kluwer Academic Publishers. Boston.

Halang, W. A. and R. Konakovsky (1999). *Sicherheitsgerichtete Echtzeitsysteme*. Oldenburg Verlag. Munich.

Hamuda, G. and G. Tsai (2003). Formal specification of a real-time operating system's component. *Real-Time Programming*. pp. 19 – 24. Elsevier. Oxford.

Henn, R. K. J. (1989). Feasible processor allocation in a hard-real-time environment. *Real-Time Systems* **1**, 77 – 93.

Rabiee, M. (2002). *Programmable Logic Controllers: Hardware and Programming*. Ingram. New Orleans.

Shaw, A. C. (2001). *Real-Time Systems and Software*. John Wiley. New York.

Smith, D. J. and K. G. Simpson (2001). *Functional Safety*. Butterworth-Heinemann. Oxford.

Sorenson, P.G. and V.C. Hamacher (1975). A real-time system design methodology. *INFOR* **13**(1), 1 – 18.

Teixeira, T.J. (1978). Static priority interrupt scheduling. Proc. *7th Texas Conference on Computing Systems*. pp. 5.13 – 5.18.

VDI (1982). *Richtlinie VDI/VDE 3554: Funktionelle Beschreibung von Prozessrechner-Betriebssystemen*. Beuth Verlag. Berlin-Cologne.

Vogrin, Peter (2000). *Safety Licenable and High Speed Programmable Digital Controllers Providing Any Required Control Behaviour*. VDI Verlag. Düsseldorf.