The Real-Time Operating System of MARS

A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz

Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/E182
A-1040 Wien
Austria

ABSTRACT

This paper gives a short overview of the architecture of the distributed real-time system MARS (*MA*intainable *R*eal-Time *S*ystem) and describes the design and implementation of its operating system. The main purpose of the MARS kernel is to achieve a timely execution of hard real-time tasks and to provide an efficient communication mechanism suitable for distributed real-time systems.

Edited by:
Johannes Reisinger
<reising@vmars.uucp>

Vienna, 1988-10-13

1. Introduction

A system that has to respond to external and internal stimuli within a specified interval of time is called a real-time system. Depending on the consequences of missing a deadline, soft and hard real-time systems are distinguished. Whereas in soft real-time systems the averages of the response times are of importance, in hard real-time systems the maximum response time must be guaranteed in all anticipated operating conditions.

The design goal of a real-time system for process-control is not (only) high performance but determinism and predictability of the system behavior even under peak load. The operating system has to assure that the timing constraints specified during the design are kept at run-time. It has to be supported by tools for the creation and evaluation (dependability, timing, etc.) of the design of a real-time system.

In the world of hard real-time systems it is possible to distinguish between event driven and periodic (time driven) systems. In an event driven system the system's activities are initiated by external events. In order to avoid the loss of events, event based systems must exercise explicit flow-control over the message source. Since it is in general not possible to exercise such an explicit flow-control over the controlled object, we decided to follow the second approach, that is a periodic system.

In a periodic system implicit flow control is exercised by the cycle times of the computational processes. It is thus possible to design a system for the specified peak-load and to use messages which contain state observations.

Since we could not find a real time operating system which supports our design methodology, we were forced to design and implement our own operating system.

This paper is organized as follows. First, an overview of our real-time system architecture, the MARS System [1], is given. In section three hardware and software structure of MARS components are described. In the main part of the paper, the design and implementation of the operating system kernel is discussed. The support of the kernel by system tasks is presented in section five. Section six outlines the communication primitives provided to the application programmer. Finally, the most important and innovative aspects of the MARS operating system are summarized.

2. The Architecture of MARS

MARS is a fault-tolerant distributed real-time system architecture for process control applications [1]. It is intended to be used in industrial real-time systems (e.g. a rolling mill, railway control systems, etc.) where hard deadlines are imposed by the controlled environment.

One characteristic feature of MARS distinguishing it from other distributed systems (e.g. the V system [2], Accent [3], Chorus [4]) is the completely deterministic behavior of the system even under peak load conditions, i.e. when all possible stimuli occur with their maximum specified frequency. It is one of the basic concepts in MARS to design hard real-time systems for peak load conditions. To achieve this determinism, MARS is strictly time driven and periodic.

MARS uses a transaction model to describe the activities of a real-time system. A transaction is a sequence of interrelated actions transferring the system from one consistent state to another. Viewed on a lower level, an action itself may be a transaction constituted of more primitive actions. A transaction is triggered by a stimulus and produces a response. If the corresponding response has to be generated within a given interval after the transaction's stimulus, the transaction is called a real-time transaction.

A MARS configuration consists of a set of clusters with a high inner connectivity. Each cluster is composed of several components interconnected by a synchronous real-time bus, the so called MARS-bus. A component is a self-contained computer, including the application software. It is a hardware/software unit of given functionality and performance [5]. A set of real-time tasks and an identical copy of the MARS operating system kernel are executed on each component. A typical target system for MARS is outlined in fig. 2.1.

Communication among tasks and components is realized by the exchange of state-messages (see also section 4.1) with a validity time. As soon as the validity time of a state-message expires, the message is discarded by the operating system. This measure is taken because the validity of real-time information depends on its correctness not only in the value domain but also in the time domain [6]. State-messages are the only means of communication between hard real time tasks in MARS.

All MARS components have access to a common global time base, the system time, with known synchronization accuracy. The system time is provided by the architecture and achieved cooperatively by the operating system (synchronization task) and a VLSI clock synchronization unit (CSU) [7]. It is used for reasoning on the validity of real-time information, for error-detection, for controlling the access to the real-time bus, and to discard redundant information.

The fault-tolerance of MARS is based on self-checking components running in active redundancy and the multiple sending of messages on the real-time bus. A high error-detection coverage is achieved by the use of software error-detection mechanisms at the operating system level and hardware mechanisms inherent in the processor. MARS-components fail silently, i.e. they either operate as intended or do not produce any results.

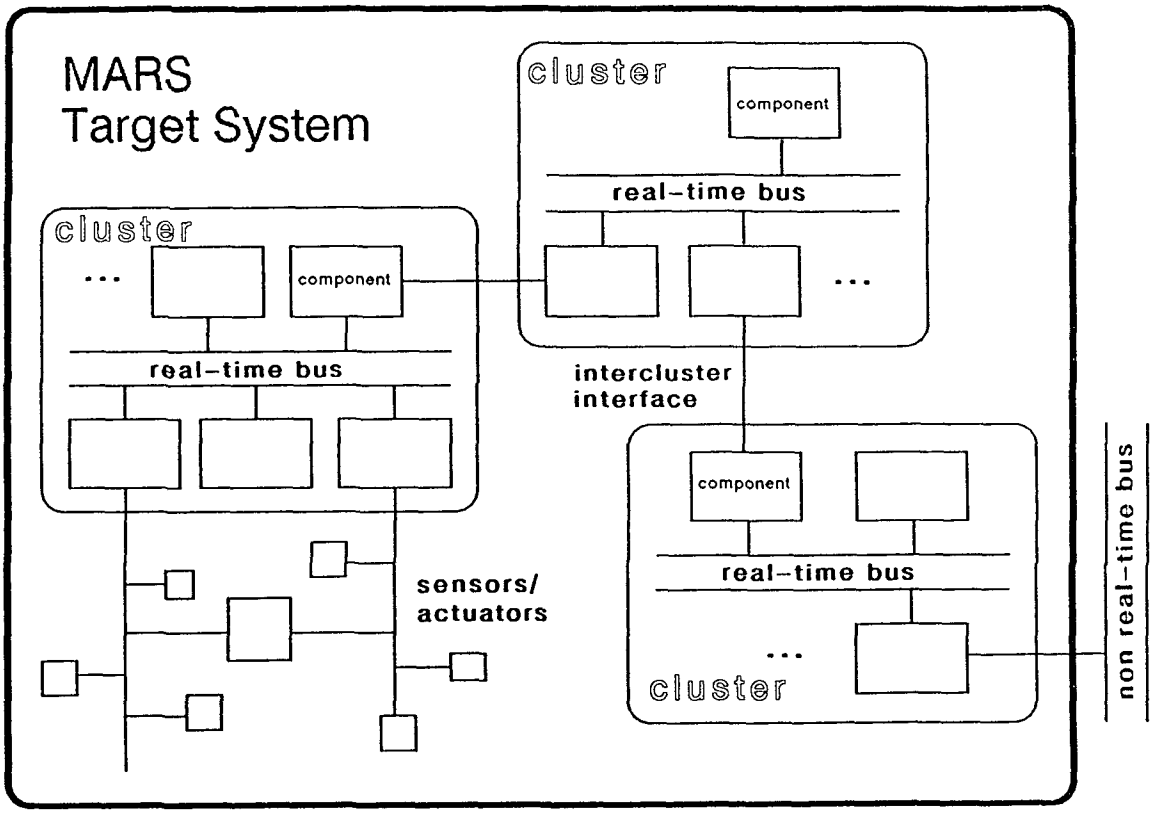

Fig. 2.1: MARS Target System

Maintainability and extensibility of MARS are based on the clustering of components. Redundant components may be removed from a running cluster (e.g. for repair) and reintegrated later. Moreover, existing components can be expanded into a cluster by converting the component in the original cluster into an interface component showing the same I/O behavior as the old component but forwarding all messages to the new cluster. The new cluster can be designed independently from the rest of the system as long as the I/O characteristics of the interface component remain unchanged.

A more detailed description of MARS can be found in the literature [1,5,6,7].

3. Structure of a MARS Component

A MARS component is a functional hardware/software unit. Clusters are built of a set of homogeneous components simplifying maintenance and the replacement of faulty components. Figure 3.1 outlines hard- and software of the current MARS prototype component.

The hardware of the experimental MARS component is a slightly modified standard single-board computer originally designed as an intelligent communication controller in a UNIX machine. It consists of a Motorola 68000 CPU (10 MHz) with access to 512 kB of RAM and is provided with a Local Area

Network Controller for Ethernet (LANCE), the Clock Synchronization Unit (CSU) [7], a custom designed chip, two RS-232 interfaces, and one Small Computer System Interface (SCSI).

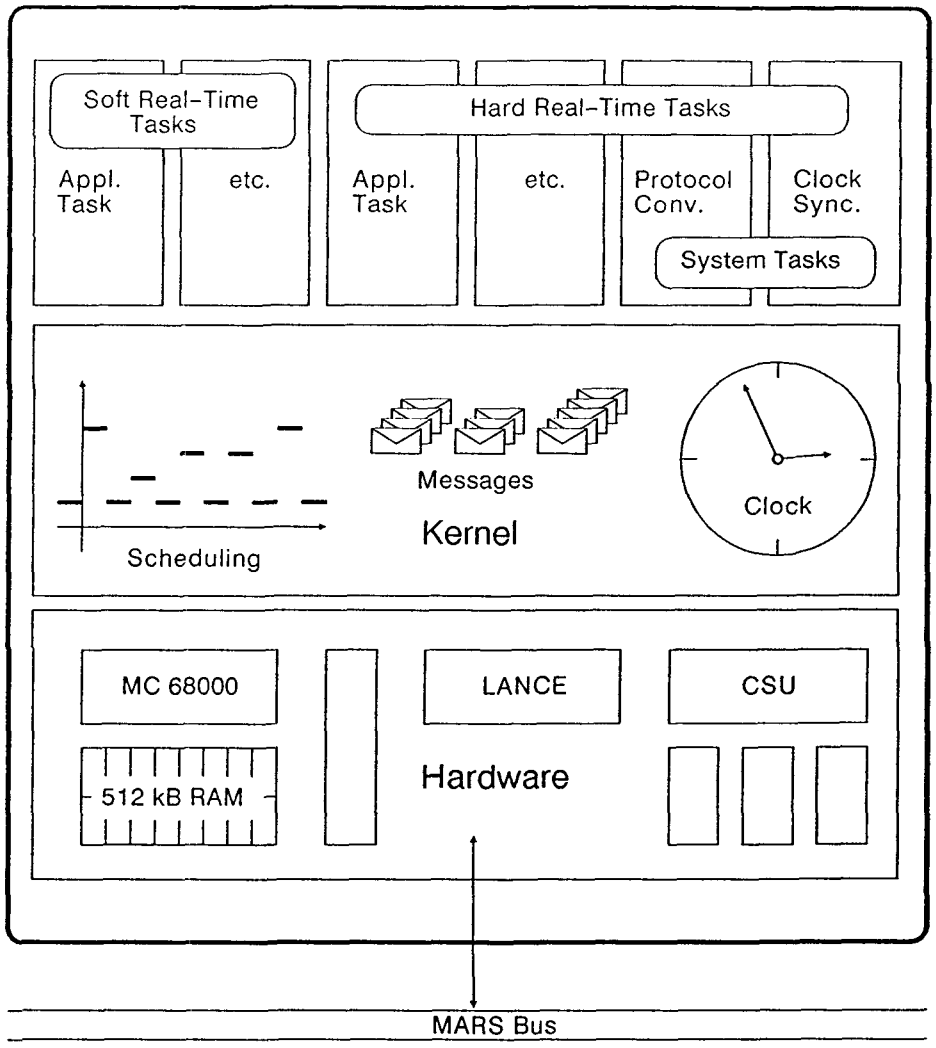

Figure 3.1: Hardware and Software Structure
of the Current MARS Component

The software residing in a MARS component can be split into the following three classes:

(1) *Operating System Kernel:*
The kernel consists of the entire code running in supervisor mode on the CPU. Its primary goals are resource management (CPU, memory, bus, etc.) and hardware transparency.

(2) *Hard Real-Time Tasks (HRT-Tasks):*
HRT-tasks are cyclic tasks receiving, processing, and sending messages. Receiving a message may be interpreted as the stimulus, sending a message as the response of the task. Each instance of a task has to be completed before a given deadline. Reaction time and latency of HRT-tasks are deterministic and specified during the design of the system. Most HRT-tasks are *application tasks*, but some of them are *system tasks* performing specific, hardware independent functions of the operating system, e.g. time synchronization, protocol conversions, etc.

(3) *Soft Real-Time Tasks (SRT-Tasks):*
All tasks which are not subject to strict deadlines are called *soft real-time tasks*. Usually a SRT-task is an acyclic task utilizing the idle time of the CPU in low load situations.

All syntactic hardware details are hidden within the kernel. Application tasks and system tasks access the kernel only by means of defined system calls [8]. Kernel data cannot be accessed directly. Most of the code of the operating system, except some parts of the kernel, is implemented in the programming language C. Porting MARS to a new hardware requires the specific parts of the kernel to be adapted, but the functionality of the system calls remains unchanged.

4. Operating System Kernel

The MARS operating system kernel has been designed from scratch, since presently no comparable kernel exists that meets our requirements in predictability and flexibility. As a consequence of the totally new kernel design, we were free to implement each part of the kernel (message passing, interrupt handling, scheduling, etc.) without any restrictions, such as those caused by the adaptation of an existing operating system.

4.1. Message Passing

State-Messages

A uniform message passing mechanism is applied for communication among tasks, components, clusters, and peripherals. Messages are identified by a clusterwide *unique name* referring to the semantic contents and the data type of the message. All messages are sent as periodic state-messages. State-messages are used to exchange information about the state of the environment or about an internal state as it has been observed at a given point in time and is assumed to hold for a certain interval of time. State-messages are not consumed when read, i.e. a state-message can be read an arbitrary number of times by an arbitrary number of tasks. One message of a sequence of messages carrying the same name is an *instance* of a message. Only one instance of a given message can be valid at a time.

Structure of a Message

MARS messages have a constant length, a standard header, and a standard trailer. Besides the LAN dependend standard header (destination address, source address, etc.), additional information such as the name of the message and several time fields are contained in the header. The time fields include the validity time and the observation time of the information contained in the message [9] as well as the send and receive time stamped into the message by the CSU upon physically sending or receiving the message. The trailer basically contains a checksum. The structure of the user-data is defined by the application programmer. Its size is predefined since MARS messages have a constant length. All objects for message passing adhere to this common structure, the MARS message structure. Data from peripheral devices such as sensors or actuators have to be converted to the fixed message format in an appropriate interface.

Buffer Management and Implicit Flow-Control

Because of the state-semantics of MARS-messages, the number of message buffers required is static and an implicit flow-control is achieved. Each time a new version of a message is received, the previous message will be overwritten and the state described in the message is updated. The total number of message-buffers required in the component can be calculated from the number of messages to be received in the component and the sum of the internal buffers required by each task.

The buffer management is carried out by the operating system kernel. There is no time consuming copying of messages between the kernel's and the task's message buffers (and vice versa). Only a pointer to the message is delivered upon reception of a message. Since messages describe real-time entities that cannot be altered by tasks, messages are kept in read-only buffers of the operating system, where they can be read by several tasks simultaneously. The number of message buffers owned by each task is determined at compile time. By convention, each task has to allocate the maximum number of buffers it needs during its initialization phase. If a system-call consumes a buffer (e.g. sending of a message), it always returns a new buffer to the task that is used exclusively by this task (i.e. it is not read-only and may be used for sending another message); if a system-call returns a pointer to a buffer (e.g. receiving of a message), the task has to return an old or unused buffer to the buffer manager (upon receiving a message, the returned buffer usually is the older version of the state-message to be received). This concept allows to statically and even automatically compute the number of buffers required by each component.

The message exchange itself is asynchronous, i.e. sender and receiver do not have to wait for each other. Nevertheless, the time when a message is sent is predefined by a pre run-time scheduler. Consequently, sender and receiver are synchronized. In MARS, there is no need for explicit flow control.

If the sender is activated more frequently than the receiver, the state is updated faster than read, but no buffer overflow will occur because the latest instance of a state-message replaces the previous one with the same name.

TDMA and Message Scheduling

MARS messages have to be sent via the Ethernet for communication between components. The LAN-controller on each board is primarily designed for use with a CSMA/CD protocol which does not qualify for the MARS system due to its unpredictable message delay. Thus, a TDMA-protocol (Time Division Multiple Access) is used to provide a collision-free access to the Ethernet even under peak-load conditions.

Since the clocks in the MARS components are synchronized (as described in section 5.1), the overhead of the TDMA-protocol is very low. The space between two slots need to be no longer than the synchronization precision of about 10 microseconds. The disadvantage of the TDMA-protocol is its inflexibility resulting in inefficiency under low-load conditions, because the sending capacity of a component cannot exceed a fixed limit (approximately the network capacity divided by the number of components in the cluster) even if no other component in the cluster has to send messages. But the major design principle of MARS is reliability and predictability even under peak-load conditions which TDMA satisfies best.

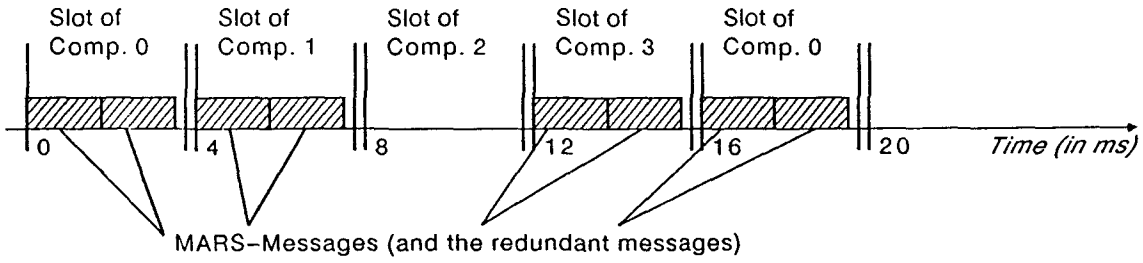

Figure 4.1: Timing of the MARS-Bus
(for a cluster consisting of four components)

To make the reliability of the message transfer comparable to the reliability of a MARS component, each MARS-message is sent twice (Figure 4.1).

In MARS two classes of messages are distinguished. HRT-messages are statically scheduled. At most one HRT-message is assigned to each TDMA-slot. A HRT message can only be sent in the slot reserved for it. If a HRT-message is not available at the slot it should be sent or the slot is unused by design, then this slot may be used by a SRT-message. SRT-messages are not guaranteed to be sent before any deadline. They are used to transmit asynchronous data, for example the core image of a new component, or archival data that should be stored on a disk outside the real-time cluster.

Error-Detection in the Time Domain

HRT-tasks are periodic tasks communicating exclusively by the exchange of periodic messages. The periodicity of the messages and the message scheduling allow the receiver to detect communication errors and component failures in the time domain by the absence of messages that should have been sent at specified times.

Time Stamping

Each MARS-message received via the Ethernet, contains two data-fields for the sending and receiving time-stamps. The CSU allows a very precise time-stamping mechanism (with an accuracy of about three microseconds). This accuracy is achieved by a close cooperation between the CSU and the LANCE.

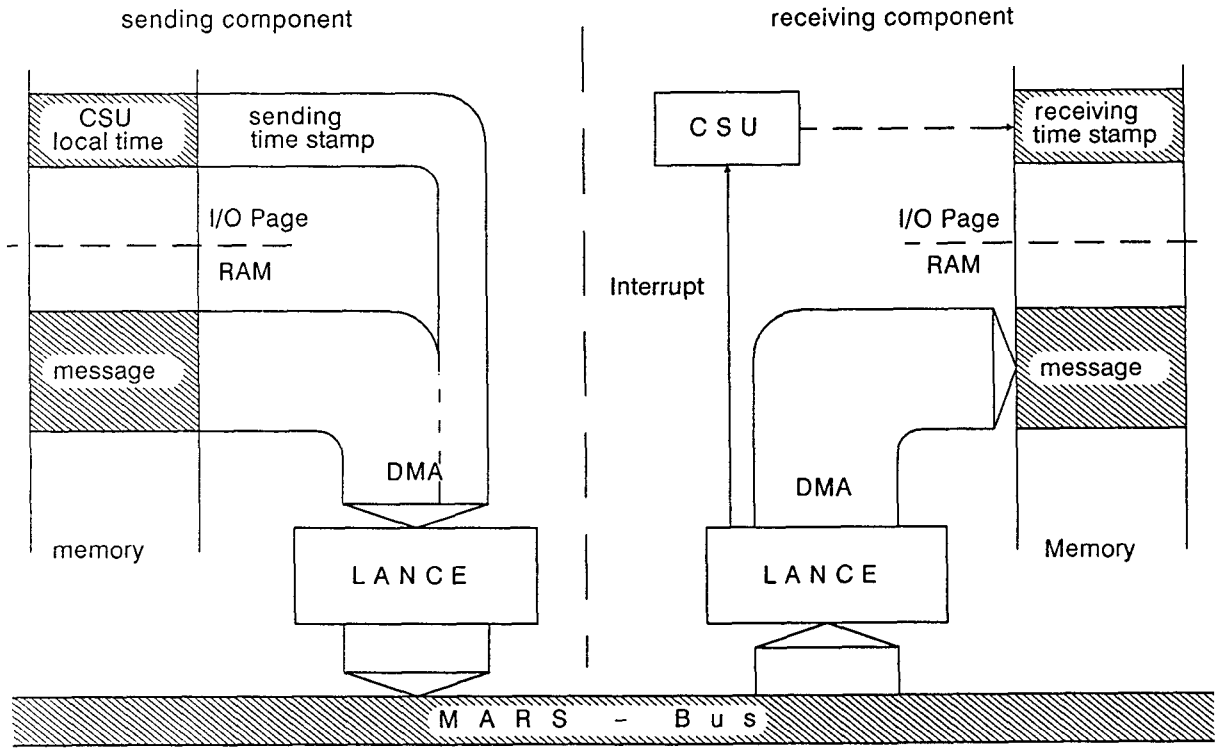

Figure 4.2: Time Stamp Mechanism in MARS

Figure 4.2 schematically shows the time-stamp mechanism for messages in MARS. Whenever a message should be sent it is placed in a buffer by the CPU and the LANCE is started. The LANCE transmits the data to the MARS bus using DMA after access to the bus has been granted. The LANCE is capable of packaging several memory fragments continuously into one message. The last fragment of each message is a memory mapped real-time register of the CSU which is accessed exactly at the time of sending. At the receiver, the LANCE

issues an interrupt immediately after a message has arrived. This interrupt is directed to the CSU for the generation of the receiver time-stamp. At the next clock-interrupt both time-stamps (sender time-stamp and receiver time-stamp) are copied to the message header. The time-stamps are used especially for clock synchronization.

4.2. Interrupt handling

MARS is designed to meet real-time deadlines even under anticipated high load conditions. This goal is achieved by a consequent realization of a strictly periodic and deterministic system behavior within all levels of software. Thus, only one interrupt, the periodic clock interrupt, is allowed. The interaction with peripheral devices is not interrupt-driven but realized by polling.

The interrupt handler is split into two sections activated with different frequencies. The first section (written in assembler for efficiency reasons) is carried out every millisecond. The second section (written in C) is executed every eight milliseconds, immediately following the first part of the interrupt (the first and the second section will be referred to as the minor and the major interrupt handlers in this paper). The minor interrupt handler may suspend any system-call, therefore this routine must be carefully coded. The major interrupt handler must not suspend any system-call, i.e. its activation will be delayed until the end of the system-call and handled immediatley after the termination of the call.

The concept of two synchronous interrupt handlers activated with different frequencies offers the possibility to split a device driver into two parts, on the one hand the time-critical part (mainly for polling the hardware), and on the other hand a part, which updates the system data structures upon sending or receiving data from the time-critical part of a device handler. This technique improves also the consistency between MARS-components (especially redundant components), since state-changes of messages are only performed in the major interrupt handler, which is synchronized throughout the system.

4.3. Task Scheduling

In the MARS-System, a static scheduling approach is pursued [10, 1]. The scheduling is performed off-line considering the maximum execution times of tasks, their cooperation by message exchange, as well as the assignment of messages to TDMA-slots. Tables, produced by the off-line scheduler, are linked to the core-image of each individual component. Dynamic task scheduling can be avoided because MARS is strictly periodic, the activation sequence of MARS tasks is predefined, and tasks need not be generated dynamically. This approach minimizes the runtime overhead for making scheduling decisions.

The scheduling lists generated by the pre-runtime scheduler determine

the points in time when task-switches are required. Start- and endpoints of tasks are recorded in the lists. Upon reaching a start-point, the corresponding task is activated. Each task has to release the CPU by itself (using the system call *suspend*) before the global time reaches the end-point specified for this activation of the task in the scheduling list otherwise an error (time-limit exceeded) will be detected by the kernel. Alarm-tasks deviate from this scheduling mechanism. They are not periodically activated but only by an explicit scheduling switch as described later.

Task scheduling is performed by the major interrupt handler (every 8 ms) according to the scheduling tables calculated by the pre-runtime scheduler. At the beginning of the execution of the major interrupt handler all registers of the task currently running are saved on the task's system stack (no matter wether scheduling decisions will be made). Therefore, a task switch is performed by simply changing the task pointer to the task descriptor of the new task making it to the current task. At the end of the major interrupt handler, the registers of the current task are restored.

Suspension of a Task

A task may release the CPU with the system call *suspend* [8] either until the next invocation of the major interrupt handler or until the task's next start-point in the scheduling list depending on the parameter of suspend. Releasing the CPU until the next invocation of the major interrupt handler is useful when a task is waiting for the arrival of a message because a new message can be recognized by the kernel only in the major interrupt handler. Releasing the CPU until the next start-point signals the kernel that the calling task has finished all calculations to be carried out since its last activation. The kernel can then use the time until the activation of the next HRT-task to execute SRT-tasks.

Scheduling Switch

Task scheduling can be adapted to different situations in spite of the fact that scheduling is static in MARS. During the design-phase, the developers of MARS-applications may plan different task sets to be active in different phases of system operation. Separate schedules can be provided for startup, alarm handling, and several states of the application. The nature of startup, alarm handling, and a few other tasks is acyclic, thus it has to be possible to combine cyclic and noncyclic schedules in the same application.

The change between two schedules may be caused either by an explicit system-call (*sswitch*) in an application task or by the reception of a message associated with a scheduling switch. The message-list in each component contains an indication whether the reception of a message triggers a scheduling switch or not.

Two types of scheduling switches can be distinguished. When performing

a consistent scheduling switch, each task is guaranteed to remain in a consistent state with the environment and the other tasks. Because of this requirement, a consistent switch may be performed only at predefined points in time determined during the design-stage. The other switch, namely the immediate, cannot preserve consistency, but it guarantees that switching will be performed as soon as possible, i.e. at the next invocation of the major interrupt handler. Therefore, the maximum delay of an immediate switch is limited to eight milliseconds. The immediate switch should only be used in emergency cases, where a fast response to a possibly catastrophic situation is necessary.

4.4. Device Drivers

Allowing each device to interrupt the CPU at will leads to an unpredictable CPU load during run-time. A priority scheme for interrupts gives advantage to high priority devices while low priority devices might starve for CPU causing missed deadlines in consequence. Such priority interrupt mechanisms are not suitable for MARS.

Consequently, in MARS all interrupts to the CPU are disabled except for the clock interrupt from the CSU. Because the interrupts from other peripheral devices, even from the local network controller, are disabled these devices have to be polled periodically within the clock interrupt handler.

A device driver in MARS consists of one or more low-level parts and one or more high-level parts. The activities of all low-level drivers are initiated by the clock interrupt routine. The low-level parts can only perform small and frequent actions which must not block in their execution. Most actions of the low-level drivers are performing device polling. The high-level parts consist of the device specific code executed within a system call. Most high-level drivers report or set device specific parameters, e.g. the type field of an Ethernet message or the baud rate of an RS-232 line.

5. System Tasks

Operating system functions that do not have to be realized within the kernel are implemented in so-called system tasks. The clock synchronization task is described as an example of a system task in the following section. Other system tasks are concerned with the initialization of the components and the cold-start of a cluster.

5.1. Clock Synchronization

Real-time data often correspond to physical quantities, e.g. temperature, volume, speed, etc., and their dependencies might be modeled with differential equations. Real-time data may refer to specified points in world time, e.g. a start occurs at 03:15:20 pm UTC. Any computer control system must be related to the controlled objects of the "real world", and so

especially the fundamental measure "time" needs to be calibrated.

In MARS, the International Atomic Time TAI [11], has been chosen as a reference. TAI is a strictly chronoscopic time measure and does not suffer from switching seconds due to irregularities in the rotation of the earth. It differs a known integral number of seconds from the world time UTC (e.g. in spring 1988: TAI - UTC = +24.0 s).

Computer clocks are based on simple quartz oscillators with a typical drift rate of $10^{-4} \cdots 10^{-6}$. Free running computer clocks diverge tenths of microseconds each second. A periodic clock synchronization is needed to adjust computer clocks to each other and to world time.

Internal Synchronization

In MARS, time synchronization is based on message exchange to avoid special hardware links for time signal propagation. The measurement error in reading the time of one component by another component is called *reading error*. Without any hardware support the reading error for time in messages is 1 ms or more depending on protocol complexity and interrupt latency. Due to the special VLSI chip CSU (Clock Synchronization Unit) [7] with a clock resolution of 1 μs and its cooperation with the network controller chip LANCE, the reading error is reduced to 4 μs in MARS.

Each component records the time differences to the other components periodically. Based on this information a correction term for the local clock is calculated with the *Fault-Tolerant Average Algorithm* (FTA) [12, 13]. In FTA, an ensemble of N clocks may include up to k faulty clocks. The local clock differences d_i are sorted by value. The k lowest and the k highest values are discarded. The arithmetic average of the remaining values is the new correction value c for the local clock:

$$c = \frac{1}{N-2k} \sum_{k+1}^{N-k} d_i$$

If the drift rate of a quartz is less than $5 * 10^{-6}$ and the resynchronization interval is 1 s, then the internal synchronization tolerating a single byzantine clock fault in an ensemble of more than four clocks is always better than 30 μs [7]. The experimental evaluation has shown, that even in case of anticipated faults the deviation of good clocks does not exceed 6 μs [14].

External Synchronization

The ensemble of local clocks is kept together by internal synchronization. The calibration with world time is done by external synchronization. Long wave radio signals provide an economic access to the UTC time standard. Since the seconds of UTC and TAI are phase synchronous and the time difference of UTC and TAI are published in advance by the BIH*, any UTC

source can be used as a source for TAI.

Special signal modulation and signal processing techniques [15] allow the receipt of standard time with an absolute precision better than 100 μs.

Each MARS cluster contains a component with access to a time standard, to measure the deviation between the MARS system-time and the world time. An appropriate rate correction value is broadcast effecting the speed of all internal clocks, independently of corrections by the internal clock synchronization.

Time Adjustment

The time correction must not change the chronoscopic behavior of the local system clocks. An instantaneous change of the local clock would lead to errors in running measurements and would disturb the periodic schedules. Thus a continuous time adjustment is supported by the CSU chip.

The local time correction value can be split into three terms:

- the constant, known quartz drift of the individual component,
- the correction term due to the internal synchronization algorithm,
- and the drift of the clock ensemble to world time.

The CSU chip has a rate correction and state correction register. The state correction register is cleared by the CSU after the correction has been applied while the rate correction register remains constant until a new value is inserted. The known quartz drift of an individual component is the initial value for the rate register. The internal time synchronization affects only the state register. The observed drift to world time is added to the rate registers of all CSUs in all components.

The CSU chip derives its time values from a 10 MHz quartz. The continuous time adjustment is achieved by inserting or suppressing some cycles spread over a second. These "corrected" cycles are the basis for the system clock, the time stamp mechanism with the LANCE (see also figure 4.2), and the periodic CPU clock interrupt. So, all observable time values in a component are kept calibrated.

6. C/MARS Language Constructs

The C programming language (used to implement MARS tasks) is enhanced by additional language constructs to provide primitives enabling the application programmer to implement cooperating real-time tasks. The MARS kernel offers predictability, maintainability, and fault-tolerance. The

---

• Bureau International de l'Heure

high-level C/MARS language [16] interfaces these features to an easy to use but nevertheless powerful communication mechanism for distributed programs. C/MARS language constructs are available for communication and the support of the execution-time analysis of the real-time tasks required to achieve a guaranteed response time of HRT-tasks. In the following only the language constructs for communication are described briefly. The execution time analysis will be discussed in a following paper.

The INPUT statement can be composed of a number of INPUT-constructs specifying the receipt of one message. INPUT-constructs can be combined by OR- and AND-operators allowing alternate inputs and the acceptance of more than one message at the same time.

A FILTER-construct can optionally be added to each INPUT-construct. The receipt of a message can then be made dependent on the fulfillment of a condition implemented in a filter function specified in the FILTER-construct. The contents of a newly arrived message can be such a condition (e.g. the message is accepted only if the temperature described in the message is above a certain value).

Additionally, a timeout can be specified to limit the time the application task is waiting for a message to arrive. The timeout may be set to zero making the INPUT-statement non-blocking. Optionally, a separate block of statements can be defined that is executed upon the occurrence of an error during the execution of the INPUT-statement.

The second C/MARS language construct for communication is the non-blocking OUTPUT statement. An identical OUTPUT statement is used regardless whether the message is passed internally or transmission takes place using the MARS-bus, the parallel port, or the serial port. The destination of a message is specified during the design depending on the allocation of the application tasks to the components of a cluster. No receiver is specified by the sending task.

The OUTPUT statement allows the specification of a validity time and an observation time. The message is valid from the point of observation until the validity time given as an absolute point in time. Executing an INPUT statement on a message of which the validity time has already expired has the same result as if no message were available.

7. Summary

The MARS project is concerned with the development of a distributed fault-tolerant real-time system that meets given requirements of reliability and timeliness by design. The main principles of MARS have been described briefly and the operating system of the current MARS prototype has been discussed in more detail in this paper.

The MARS operating system fully supports the architecture of MARS and guarantees the meeting of the timing requirements of the application, even in peak load situations. Hard real-time tasks are scheduled by a time-rigid scheduling policy minimizing the dynamic scheduling overhead and guaranteeing that hard real-time tasks are completed within their deadlines under all foreseen circumstances.

The communication system is based on an efficient implementation of the state-message mechanism solving flow-control problems implicitly. TDMA is used as a medium access protocol to the local area network connecting MARS components in a cluster to provide a collision free access and to allow the specification of an upper bound on the communication delay.

The availability of a synchronized system time with a bounded maximum deviation is an important prerequisite for building a distributed real-time system. In MARS, a continuous correction of the local times in a cluster is achieved by the cooperation of a VLSI clock synchronization chip and the clock synchronization task.

Based on the implementation of the MARS operating system, an experimental system of 16 components has been established. It will serve as a testbed for experiments in the areas of real-time and fault-tolerant systems. Currently the implementation of a membership protocol is investigated.

8. References

1. H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach," *accepted for publication in IEEE Micro*, (Feb. 1989).

2. D. R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, vol. 1, no. 2, pp. 19-42, (Apr. 1984).

3. R. Fitzgerald and R. F. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems*, vol. 4, no. 2, pp. 147-177, (May 1986).

4. H. Zimmermann, J. S. Banino, A. Caristan, M. Guillemont, and G. Morisset, "Basic Concepts for the Support of Distributed Systems: The Chorus Approach," *Proc. 2nd Conf. on Distributed Computing Systems*, pp. 60-66, Paris, (Apr. 1981).

5. H. Kopetz and W. Merker, "The Architecture of MARS," *15th Fault-Tolerant Computing Symposium*, pp. 274-279, Ann Arbor, Michigan, (June 1985).

6. H. Kopetz, "Design Principles for Fault-Tolerant Real-Time Systems," *19th Hawaii Conference*, vol. II, pp. 53-62, (1986).

7. H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Transactions on Computers*, vol. 36, no. 8, pp. 933-940, (Aug. 1987).

8. W. Schwabl, "MARS System Calls," Research Report 2/88, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria (May 1988).

9. H. Kopetz and K. Kim, "Consistency Constraints in Distributed Real-Time Systems," *Proc. of the 8th IFAC DCCS Workshop*, (to be published by Pergamon Press in 1989), Switzerland, (Sept. 1988).

10. C. Koza, "Scheduling of Hard Real-Time Tasks in the Fault-Tolerant Distributed Real-Time System MARS," *Proc. 4th IEEE Workshop on Real-Time Operating Systems*, pp. 31-36, Cambridge, Massachusetts, (July 1987).

11. G. Becker, "Die Sekunde," *PTB-Mitteilungen*, vol. 85, pp. 14-28, Physikalisch-Technische Bundesanstalt, Braunschweig, BRD, (Jan. 1975).

12. D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, "Reaching Approximate Agreement in the Presence of Faults," *Proc. Third Symp. Reliability in Distributed Software & Database Systems*, pp. 145-154, (Oct. 1983).

13. J. Lundelius and N. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," *Proc. Third ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*, pp. 75-88, Vancouver, Canada, (Aug. 1984).

14. W. Schwabl, "Der Einfluß zufälliger und systematischer Fehler auf die Uhrensynchronisation in verteilten Echtzeitsystemen," Dissertation, Technisch-Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria (Oct. 1988).

15. P. Hetzel, "Zeitübertragung auf Langwelle durch amplitudenmodulierte Zeitsignale und pseudozufällige Umtastung der Trägerphase," Dissertation, Fakultät Fertigungstechnik, Universität Stuttgart, BRD (Jul. 1987).

16. M. Pflügl, A. Damm, and W. Schwabl, "Interprocess Communication in MARS," Research Report 6/88 (accepted at: ITG/GI Conference on Communication in Distributed Systems, Stuttgart, Feb. 1989), Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria (June 1988).