Inside the MARS Operating System*

J. Reisinger

Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A–1040 Wien, Austria/Europe

Tel.: +43/222/58801–8173
Fax: +43/222/569149
E–mail: joe@vmars.uucp

Version 1.0; Dec. 21, 1989

Abstract

This paper describes the design and implementation of the MARS[1] Operating System. The MARS Operating System has been developed at the Technical University of Vienna. It is designed for Hard Real–Time (HRT) Applications. The aim of this paper is to describe major concepts of the operating system as well as some interesting details of the implementation. A special section of the paper deals with future changes and extensions of the operating system, especially with a Soft Real–Time (SRT) part, which will be added to the System.

1. MAintainable Real–time System

* For INTERNAL use only

Contents

1 Introduction 3

2 The MARS System Kernel 3
2.1 Overall Concepts 4
2.1.1 Minor Interrupt Handler 5
2.1.2 Major Interrupt Handler 6
2.1.3 System Calls 7
2.2 Message Handling System 8
2.2.1 Timestamping 13
2.2.2 Buffer Allocation 15
2.3 Run Time Scheduler 15
2.4 Alarm Handling 18
2.5 Clock Functions 19
2.6 Timing Profile 20
2.7 Error Handler 21
2.8 Device Drivers 22
2.8.1 Raw Ethernet Driver 23
2.8.2 Serial Line Driver 23
2.8.3 SCSI Driver 24

3 System Tasks 24
3.1 Start Task 24
3.2 Clock Synchronization Task 25

4 Library Functions 25

5 Soft Real–Time Kernel 26

6 Summary 27
6.1 A Short Look into the Future 28

References 29

List of Figures

Figure 1: Structure of a MARS node 3
Figure 2: Sample of an interrupt sequence 7
Figure 3: Structure of a MARS message 14
Figure 4: Double chained list of MARS tasks 16

1 Introduction

The MARS Operating System has been developed at the Technical University of Vienna to provide a run time environment for the MARS System. MARS is a hard real–time system, which focuses on achieving a high degree of reliability, safety, predictability and maintainability [Kop85], [Kop89], [Dam89].

The hardware supporting the MARS System consists of several single–board computers, the so called MARS nodes. They were built on the basis of an ICC[2] card, normally used in a CADMUS computer for communication purposes. A MARS node is composed of a 68000 processor, 64 kB ROM, 512 kB RAM, two serial interfaces, an SCSI bus, a LANCE[3], and a CSU[4] (see figure 1). The MARS nodes are linked together via an Ethernet cable [Met76] to form a MARS Cluster. MARS Clusters are interconnected by an SCSI[5] bus.

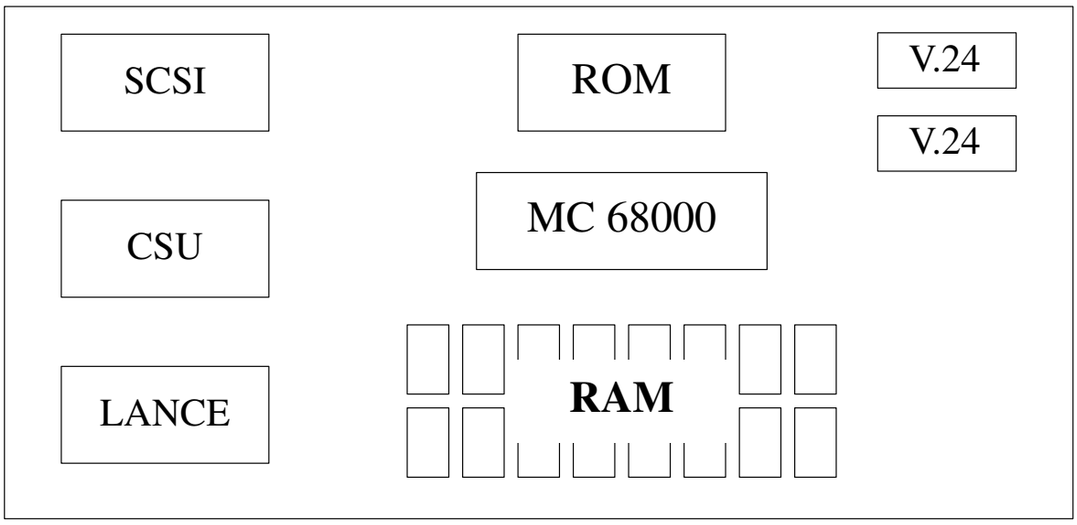

Figure 1: Structure of a MARS node

The Read Only Memory contains a monitor and downloading software, which allows downloading from a host computer to each of the processing nodes.

2 The MARS System Kernel

To achieve a high degree of reliability, it is of great importance to keep the operating system kernel as small as possible. This simplifies testing and debugging the kernel.

2. Intelligent Communications Controller
3. Local Area Network Controller for Ethernet
4. Clock Synchronization Unit
5. Small Computer System Interface

Although there is the possibility to execute soft real–time tasks on a component, or to use a MARS node as an interface component for a MARS cluster, all design decisions were made with the intention to support the hard real–time applications as much as possible. This led to some system features which appear to be uncomfortable in their use, and some other features which increase reliability at the expense of system performance.

All these disadvantages can be eliminated at a second thought. You can improve system performance by increasing the number of components in your system or by using improved hardware; you can increase programming comfort by providing powerful tools for the programmer, but there is no way to improve reliability other than to make each design decision with the intention to maximalize reliability. This consideration directed the whole design of the MARS operating system, especially of the MARS system kernel.

Many tasks executed during run–time in conventional operating systems, are already carried out during the design stage of a MARS application [Sen88]. The considerable amount of work, which is carried out off–line, is made possible by the *periodic nature* of the MARS system. Every component may send a message at the Ethernet cable at fixed, predefined points in time, because the communication protocol in MARS is a slightly modified TDMA[6] protocol. Not only the bus access, but also task scheduling is performed in a cyclic manner. Each HRT[7] task will be started at fixed equidistant points in time, which permits task scheduling to be carried out off–line.

Off–line task scheduling allows to guarantee a predefined amount of CPU time for each task. Together with an offline execution time analysis of each task, this leads to the certainty, that each task finishes within its time slot, and therefore guaranteed response times can be computed during the design and implementation phase of an application.

Like task scheduling, several other tasks, such as buffer allocation or bus scheduling are determined statically in MARS to achieve a greater degree of reliability and to keep the kernel small. Currently the MARS kernel consists of about 3000 lines of C source code and 1000 lines of assembly source code (only 300 lines of the assembly code are ‘sophisticated’ code, the rest consists of ‘silly’ code such as entry points for the system calls or refresh of dynamic RAM).

2.1 Overall Concepts

In a hard real time application each task has to meet strict deadlines. To fulfil this requirement, it is *absolutely* necessary to know

- the CPU time each task may consume in the worst case and
- the CPU time consumed by the operating system (the operating system overhead)

Neither calculating the maximum execution time of application tasks [Pus89] nor going into details about computing the operating system overhead is the aim of this paper. Instead,

6. Time Division Multiple Access
7. Hard Real Time

the mechanisms, which are used in MARS to bound the operating system overhead, are presented here.

To achieve a high degree of determinism (in the time domain) of the MARS System Kernel, asynchronous events have to be avoided whenever possible. Asynchronous events, which can be found in every conventional operating system, are interrupts, generated by several sources: clock, serial interface, timeout, ethernet, external busses, …

Another source of indeterminism is direct memory access (DMA) by peripheral devices. DMA decreases the effective CPU performance due to the 'stealing' of cycles from the main processor.

Unpredictable timing behavior also results from programming constructs in the kernel, which are not boundable. Such constructs are searching of lists of variable size or handling of peripheral requests, where the number of requests is not limited. Even if there is an upper bound for the execution time of such constructs (a list cannot be larger than the memory related to the operating system, the number of operating system buffers limits the number of received messages, …), in general this upper bound leads to a worst case execution time, which is orders of magnitude longer than the actual execution time, and therefore of no use.

In MARS only one interrupt – the periodical clock interrupt, which occurs every millisecond – is provided. Every request or timeout caused by a peripheral device is not realized by asynchronous interrupts but by polling. This concept simplifies the calculation of an upper bound for the operating system overhead. Instead of having several, possibly nested asynchronous interrupts, only one periodic interrupt is to be taken into account. Therefore the overhead is bounded by *execution time of interrupt handler / period of clock interrupt*. Otherwise, the maximum frequency of each interrupt must be used when calculating the system overhead, leading to a significantly higher percentage of CPU time used by the kernel in the worst case.

The DMA overhead in our system results from two peripheral devices: Ethernet and SCSI. With the use of the Ethernet in TDMA mode, the number of DMA cycles 'stolen' by the Ethernet controller is limited by the number of CPU words contained in a message divided by the length of a TDMA slot. The same holds for the SCSI bus, because a simple type of TDMA is used there too.

The code of the MARS operating system kernel does not contain any unbounded loop. In most cases the boundary is an integer, otherwise the boundary can be derived by the semantics of the application. For example, there is a loop which handles all Ethernet messages received since the last interrupt. Because the number of TDMA slots between two interrupts is fixed, and in each TDMA slot only one message can be transmitted, a boundary for this loop which results from the semantics of the system is given.

2.1.1 Minor Interrupt Handler

Although only one interrupt is permitted, two different interrupt handlers are part of the kernel. The first, the so called '*minor interrupt handler*', is executed whenever a clock inter-

rupt occurs (i.e. every millisecond). The '*major interrupt handler*' is executed with a lower frequency. It is started every eight milliseconds, immediately following the minor interrupt handler. There are two reasons for this implementation:

- Several periodic tasks within the system kernel are not required to be executed every millisecond, such as task scheduling, message handling, … On the other hand, several tasks *must* be performed with a higher frequency, such as polling the serial lines, dynamic memory refresh, … The more conventional way is, to perform all these functions within one interrupt handler, doing the less frequent tasks whenever necessary (e. g. handling a message immediately after the detection of the arrival of a message). The disadvantage of this method is the difficulty to bound the maximum execution time of the interrupt handler, since bounding it with the sum of the maximum execution times for each function, that may be performed in the interrupt, would lead to times which are much longer than the effective ones.
- The computation power of the current MARS nodes is too low to perform every periodic function with a frequency of one millisecond.

The following tasks are performed in the minor interrupt handler:

- Refreshing of dynamic RAM. To make the MARS nodes as predictable as possible, memory refresh is not done by hardware (since the refresh logic steals cycles from the CPU). For this reason, the operating system has to execute a sequence of 'no–operation' statements periodically.
- Timestamping of incoming messages (see chapter 2.2.1).
- Delayed sending of Ethernet messages (see chapter 2.2).
- Polling of the serial lines (see chapter 2.8.2).
- Starting of the major interrupt handler each eight milliseconds.

2.1.2 Major Interrupt Handler

The major interrupt handler performs functions, which have to be executed periodically, but there is no need for doing them every millisecond. In general, the tasks of the major interrupt handler require more time and are more complex than the tasks performed within the minor interrupt handler for the following two reasons:

- The major interrupt handler is executed with a much lower frequency than the minor interrupt handler, therefore its tasks have more time available than the ones of the minor interrupt handler. Because the major interrupt handler may be interrupted by the minor interrupt handler, the upper bound for its maximum execution time is 8 ms – (8 * $MAXT_{minor}$), where $MAXT_{minor}$ is the maximum execution time of the minor interrupt handler. Needless to say, that this theoretical upper bound should not be reached in practice, because this would decrease the CPU availability for application processes to zero.

- The minor interrupt handler is written completely in assembler code to increase speed, while the major interrupt handler is coded in C allowing more complex programming constructs with less effort.

A fundamental difference between the two interrupt handlers is the fact, that the minor interrupt handler may interrupt any system call, while the major one may not. Immediately before calling the major interrupt handler, a task within the minor interrupt handler checks, whether a system call is in progress or not. If not, the major interrupt handler is executed at once, otherwise it will be executed immediately after the current system call has finished (see figure 2).

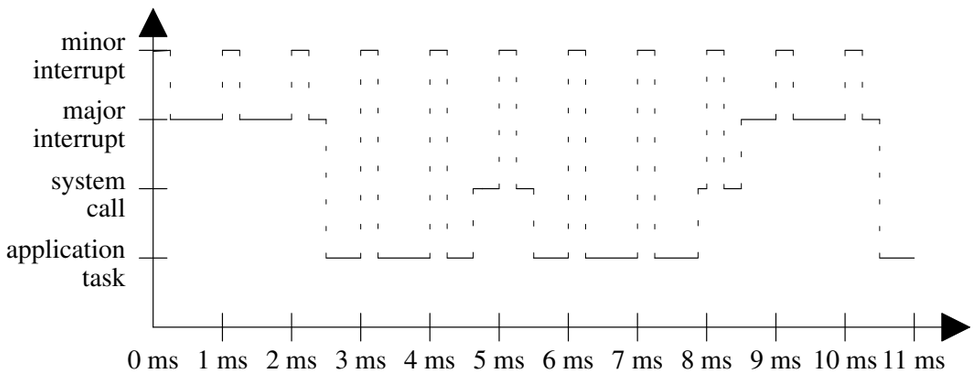

Figure 2: Sample of an interrupt sequence

The following tasks are carried out in the major interrupt handler:

- Handling of the system clock (see chapter 2.5).
- Performing the task scheduling (see chapter 2.3).
- Handling of messages to and from the serial interfaces (see chapter 2.8.2).
- Sending of Ethernet messages in TDMA mode (see chapter 2.2).
- Polling for received Ethernet messages (see chapter 2.2).

2.1.3 System Calls

Besides the interrupt handler there is another important part in an operating system – the interface to the application tasks, represented by the system calls. MARS system calls have to meet special requirements, to hold the constraints given by the principles of the MARS system, such as its high degree of predictability. A MARS system call *must not* block, and because a system call can delay the invocation of the major interrupt handler, the maximum of the maximum execution times of every system call must be subtracted from the time which is available for the major interrupt handler.

The set of the MARS system calls differs from the one in other operating systems, such as UNIX. Due to the static structure and the message orientation of the MARS system there is no need for system calls for task creation, dynamic memory allocation or sending of asynchronous signals. On the other hand, there is a lot of new system calls for error notification, message buffer handling, message handling, …

To facilitate the task of writing applications for the MARS system, compatibility to UNIX is attempted whenever possible. This is done by emulating UNIX system calls with library functions (see chapter 4). This method also makes the development of a testing environment for MARS on UNIX systems easier, at least an environment for the functional testing of application tasks (testing the correct interaction of the tasks would be more complex).

The following groups of system calls are currently implemented in MARS:

- Message handling (see chapter 2.2).
- Buffer handling (see chapter 2.2.2).
- Task handling (see chapter 2.3).
- Handling of system clock (see chapter 2.5).
- Handling of raw devices (see chapter 2.8).
- Error notification (see chapter 2.7).

2.2 Message Handling System

MARS is a message based system. Messages are the only mechanism for communication among tasks as well as for synchronization between tasks. Other communication primitives, such as signals, semaphores or shared memory are not supported by the MARS kernel.

To simplify message handling, MARS messages have a fixed length and a fixed header, independent of the method used to send the message. This concept allows sending of one message at different media without trouble. Consequently, each intelligent peripheral device should send and receive MARS messages to simplify the communication with peripheral devices.

State Message Approach

In real time systems (as well as in conventional computer systems, which provide messages) there are two different message types: event messages and state messages. The semantics of these two kinds of messages is rather different. Event messages are used to describe processes by reporting only changes in the process, while state messages send the state of a process periodically.

Obviously, an event message can transmit more information in general, because a message must be sent only when the state of the process changes, not periodically as in the other approach. In hard real time systems, however, the state message concept has major advantages:

- A hard real time system has to be designed for maximum load conditions. In this case preventing the message handling system from crashing due to overload when using event messages requires much effort. In some cases, it would be rather impossible to limit message transmission and simultaneously guarantee, that all important messages reach their destination. A state message oriented system has a constant load level in high as well as in low load conditions. Therefore, no effort is required to limit message transmission in the case of high load conditions, leading to a much less complex message handling system. One may say, that event messages lead to a finer granularity describing state changes, since state messages are sent only at fixed, predefined intervals. But the amount of information, which can be processed, depends not only on the amount of transmitted messages, but also on the processing capacity of the tasks. If the message sending intervals are adjusted to the task cycles, using event messages leads to no advantages.
- A state message based system provides implicit flow control. Since the arrival of a new state message invalidates the old state by definition, the number of valid messages at any point in time is limited by the number of processes, whose states are to be transmitted.
- It is much easier to use redundant processing nodes, which produce equivalent messages in the case of no fault occurrences. In event message based systems, the operating system has to distinguish between a redundant message, describing the same state change as the previous message, and a message which describes a new state change. In a state message based system, the new message overwrites the previous one, no matter if it is a redundant one or it describes a new state.
- Because state messages are sent at predefined points in time, the detection of a message loss is very simple: If at any point in time, when the arrival of a message is expected, the message does not arrive, it is considered as having got lost. This detection is almost immediately after the point in time when the message loss occurs. In event message based systems, a special protocol must be executed in order to detect message losses. This leads to a communication overhead and to a higher delay in detecting a message loss.

Addressing

MARS messages do not have destination addresses. Instead, addressing is realized by message names, which identify messages. Two messages of the same name are considered to be redundant, or the second message is assumed to be a newer instance of an old one. Each component has to decide which messages to receive. Theoretically, any component is able to re-

ceive any message (This makes the implementation of maintenance components easy, since the component can easily observe the entire message traffic).

Devices for Message Passing

There is one restriction to this concept. Some devices support only local messages, which cannot be received by each component in a MARS cluster, but only by the component, which is connected to the peripheral device. The devices, to which MARS messages can be sent to, are:

Ethernet: The Ethernet is the device, which is used for communication among MARS nodes (and therefore it is also called the '*MARS bus*'). MARS doesn't use the Ethernet in CSMA/CD[8] mode as usual, but it uses the TDMA protocol. The advantages of this protocol for MARS are:

- Each component knows in advance the exact points in time when it can send a message, thus leading to a higher predictability of the communication system.
- The network throughput is constant. Using CSMA/CD, the throughput will be decreased under high load conditions.
- Because of the absence of collisions, the reliability in the message transfer is high.
- Very short response times can be achieved by coordinating task– and bus–scheduling.

To improve reliability, each message is sent twice at the MARS bus in one TDMA slot. This method does not increase the response times of tasks significantly, and the reliability in message transfer reaches the reliability of the MARS nodes (this fact was underlined by extensive experiments).

Two kinds of Ethernet messages are provided in MARS: Hard real time and soft real time messages. Each HRT message is assigned to one or more TDMA slots, where it is to be sent. This assignment is done during the design phase, and an application task does not have to know the slot, in which it has to send a specific message. No contention between HRT messages of different names in a component occurs, because each TDMA slot is occupied by exactly one message. Some slots may remain free. These slots are reserved for soft real time messages. The operating system cannot guarantee maximal transmission times for them. SRT messages are sent in First Come First Served order.

The length of a TDMA slot is 4 ms, while the major interrupt handler is called every 8 ms. Because sending of Ethernet messages is initiated during a major interrupt, there is a mechanism in MARS, called '*delayed sending*', which allows to initiate the sending of a message, which is actually sent 4 ms later. In this case the major interrupt handler prepares the message to send, and sets a flag as an indication. The minor interrupt, which is invoked 4 ms later, only has to notify the LANCE to send the previously prepared message. Since in MARS redundant components have the TDMA slots 2n and 2n + 1 (where

8. Carrier Sense Multiple Access with Collision Detection

n is an integer), the delayed sending supports the concept, that two redundant components should produce exactly the same output (the point of commitment is the same for both redundant components, namely the invocation of the major interrupt handler).

RAM: Communication among tasks, which reside on the same component, is carried out by so called '*internal messages*'. Unlike other operating systems, which support signals, semaphores, shared memory, …, messages are the only communication primitives of tasks in MARS. The advantages of this concept are:

- Simplicity. An operating system, which supports different communication methods, is more complex than one, which supports only one method.
- Transparency. The communication between tasks, which reside at the same component, is done with the same programming constructs as the communication between tasks at different MARS nodes.

Serial interfaces: Screen output and console input is also done by messages in MARS. Writing a character, a line or an entire page to the screen is realized by sending a message to the corresponding device. Input from a keyboard (or any peripheral unit, which is connected with the serial port) is fetched by reading MARS messages. There are two serial lines in MARS: The console and the auxiliary port.

SCSI: The SCSI bus is used to connect a MARS component to:

- a MARS component belonging to another MARS cluster ('*cluster path*').
- a MARS component used as an interface component. The interface component may be used to communicate with a host computer (via the Ethernet port of the interface component) or with the controlled environment, …
- mass storage, such as a harddisk. This offers a way to store process data on a non–volatile storage without transmitting it to a host computer.

Each MARS message may be sent to each combination of the above devices concurrently. The devices, to which a message is to be sent, are statically fixed by the design system. An application task should be the same, no matter if the messages, that it produces, are sent to the Ethernet, the console or any other device (or any combination of them).

An Example: Normally, there is a message with a fixed name used for console output. If one wants to provide only one terminal for an entire MARS cluster, the console output messages at all other components have to be defined to be written to Ethernet instead as to the console (they must be considered during the computation of the bus–schedule). If there is a task at the terminal console, which sends all these messages to its terminal, the output of all other nodes is put to this terminal without having made any change to the tasks itself.

Structure of a MARS Message

As mentioned above, all MARS messages have a fixed length and a fixed structure. In the current implementation, the length of a message is equal to the maximum length of an Ethernet

message, that is 1518 Bytes (the maximum data amount with the minimum computation overhead). A message consists of a header, the data area and a trailer (see figure 3).

The message trailer contains internal system data (CRC, information for timestamping). The structure of the message header is as follows:

- Ethernet destination address (6 bytes). This address is the same in each MARS message, which is sent at the Ethernet (not used when sending to other devices).
- Ethernet source address (6 bytes). It is set to the logical component number (= number of TDMA slot) multiplied by two.
- Ethernet type field (2 bytes). The type of a standard MARS message is 0x8300. Future enhancements (such as test messages, …) may use type fields in the range 0x8301 … 0x830f.

  Destination address, source address and type field form the standard Ethernet message header. The following fields are MARS specific enhancements.
- Message name (2 bytes).
- Key fields (2 bytes). There may be some messages, whose improper usage would be disastrous for the entire cluster (e.g. a message, which causes an emergency system stop). To force the programmer to be conscious of this fact, such a message may be equipped with a key. If so, a task *must* set the key field to the predefined key value to enable the transmission of the message.
- Type of an alarm scheduling switch (see chapter 2.4).
- Identifier of the alarm schedule (see chapter 2.4).
- Validity time (8 bytes). Since a MARS message is a state message, and states are changing, a message is valid only up to the point in time given by the validity time. After this time has expired, no task will be able to read the message. The maximum of the validity times of all messages is the period a component has to wait until it has a consistent view of all messages.
- Observation time (8 bytes). The time, when the user data in the message has been observed (in the controlled environment).
- Sending time (8 bytes). The value of the local clock upon sending the message.
- Receiving time (8 bytes). The value of the local clock when receiving the message. The difference between sending time and receiving time is used for clock synchronization.
- Length of user data (2 bytes). This field is reserved for application tasks.

2.2.1 Timestamping

Each MARS message, which is sent at the Ethernet, is provided with two timestamps. Each node is provided with a special chip, the clock synchronization unit CSU [Kop87], to carry

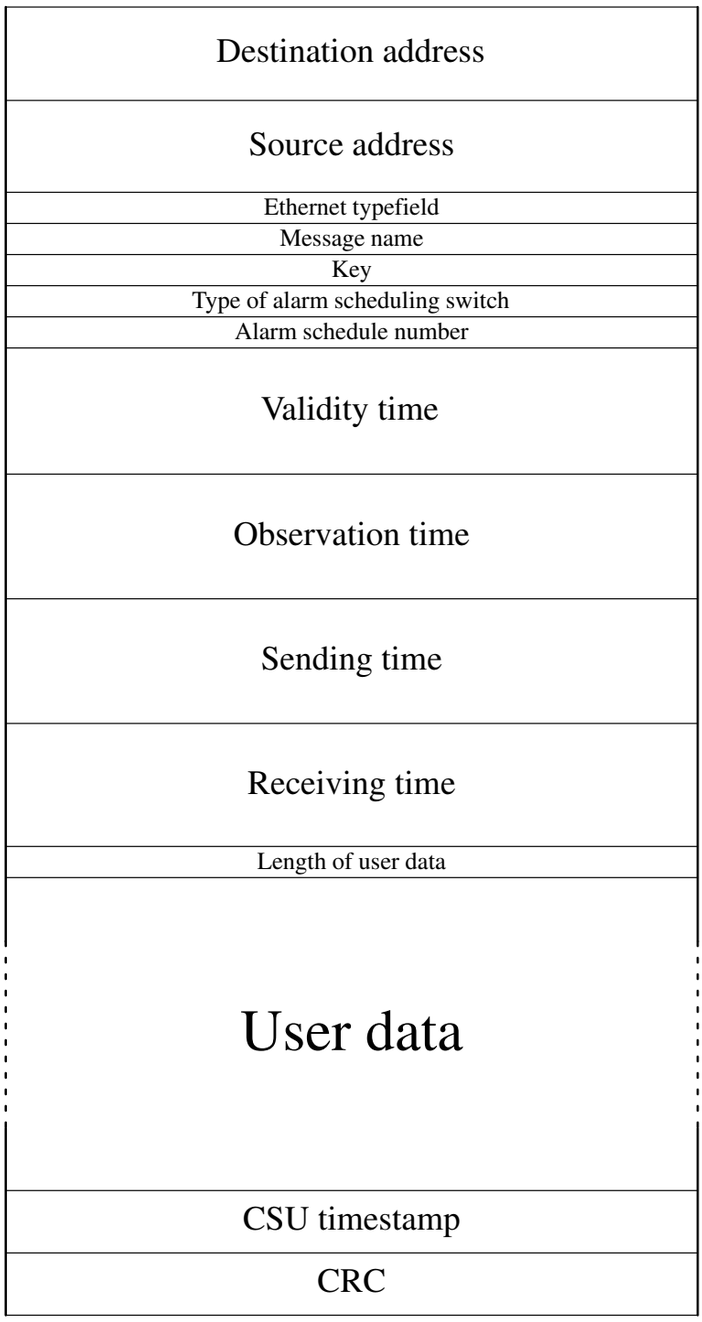

Figure 3: Structure of a MARS message

out a very precise timestamping mechanism. The CSU makes a clock with a resolution of 1 µs available. The timestamps of a MARS message are:

Sending timestamp: When sending a message, the last four bytes of the message will be reserved for the current value of the local clock. If the message handling system simply copies the value of the local clock into the message prior to sending it, the sending time would be rather inaccurate, since the time period between writing the time into the message and sending the message at the Ethernet will vary in a specified range. The MARS components use an improved method. The LANCE chip supports *split sending*, that is, to form a message in a way, that the segments of the message are located at different memory locations. In MARS, the first segment of an Ethernet message is fetched from the ordinary message buffer, and the second segment, which has a length of four bytes, is fetched directly from the CSU registers. Therefore the sending time is read just before sending it at the Ethernet, and the variability of timestamping is reduced to the granularity of the local clock. The receiver of the message copies the time into the field 'sending time' in the message header.

Receiving timestamp: The CSU has a special register (the *timestamp* register), to which the value of the local clock is copied, when a specific interrupt occurs. This interrupt is generated by the LANCE after receiving a message. Each time, the minor interrupt handler is called, it checks if a message has been received in the last millisecond. If so, it copies the value of the timestamp register into the field 'receiving time' in the message header. It is impossible to receive two MARS messages between two invocations of the minor interrupt handler, because the transmission of a MARS message lasts more than one millisecond (the length of a message is 1518 bytes = 12144 bits; the Ethernet is capable of transmitting up to 10000 bits per millisecond).

The variability of the difference between sending and receiving timestamp is very low. It consists of the granularity of the CSU when sending and receiving the message, and the variability in message transmission, which can be reduced significantly, if the physical distances between the processing nodes are taken into account. The other portion of the difference between the two times is fixed. Experimental results have shown, that the fixed part of the difference is about 38 µs, while the variable part is about 2 µs.

2.2.2 Buffer Allocation

Message buffers are managed by the operating system. This concept makes avoidance of message copying possible. To hand over a message to a task, the operating system is not forced to copy the contents of the message to a local buffer of the task, instead, it simply passes a pointer to the message buffer to the task. The consequence of this method is, that each message received by a task has the status 'read only', since several tasks may receive the same message.

The maximum amount of buffers needed by an application *must* be calculated *prior* to system operation, because having too few buffers either leads to the loss of messages or to the

violation of the timing conditions, both may be disastrous in a real time system. The amount of buffers is calculated from:

- The number of buffers used by the kernel (this is a fixed amount)
- The maximum number of components in a cluster (equal to the length of the TDMA cycle), because the latest message from each component is resident within the operating system (e.g. for clock synchronization).
- The number of messages of different names to be sent from this component. Messages, which are to be sent, are distinguished by their names. At most one message per message name may be waiting for transmission at a point in time.
- Two times the number of messages of different names to be received in this component. The two latest instances of each message are stored within the kernel.
- The number of buffers required by the application tasks. The number of buffers used by an application task does not change during runtime. The amount of buffers needed by each task is requested during the initialization phase of this task. Each system call, which fetches a buffer from the operating system (such as the reception of a message), must return an unused buffer to the operating system. On the other hand, a system call which hands over a buffer to the kernel (such as sending of a message) gets an empty buffer from the operating system, in exchange.

2.3 Run Time Scheduler

Because of the static nature of task scheduling in MARS, the run time scheduler is, in contrast to run time schedulers in other operating systems, rather simple. Three conditions, which may cause a task switch in MARS, can be distinguished:

- The major interrupt handler detects, that a context switch has to be made. The off line scheduler creates a list, which contains the points in time, when a task switch has to be done. There are two types of entries in this list:

  Start a task: The named task is started immediately. The currently running task, which may be a HRT task or the '*fill task*', is preempted. The fill task is a non–HRT task, running whenever no HRT task is scheduled. Figure 4 shows the internal representation of the currently active and preempted tasks. Starting task 1 at a point in time, when no other HRT task is active, corresponds to a change from situation I to situation II. If another HRT task, say task 2, is active, the activation of task 1 corresponds to a change from situation III to situation IV. In this case, task 2 has to wait until task 1 is suspended.

  End a task: If there is an end task entry in the scheduling list, the corresponding task *must* have finished its execution cycle at this point of time. If not, an error has occurred, since the maximum execution time analysis has to guarantee, that each

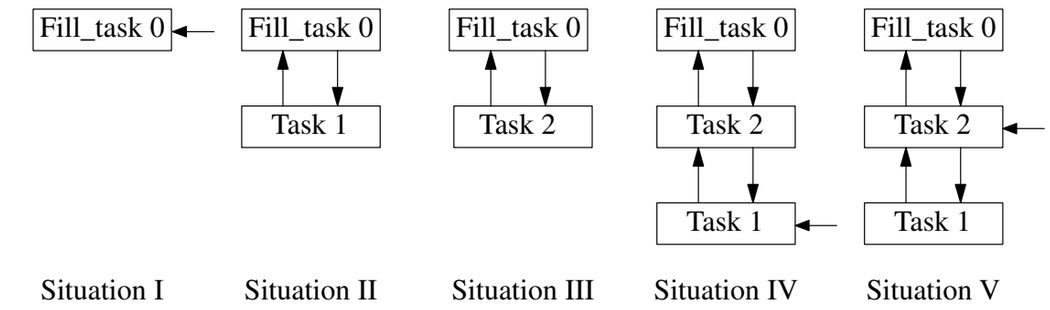

Figure 4: Double chained list of MARS tasks

task finishes its work within its cycle. If a task does not meet its deadline, either an error has occurred in calculating the maximum execution time of the task or an error in the hardware (or in the operating system) has occurred.

The scheduling list is processed periodically. Task scheduling and bus scheduling both have the same cycle.

Apart from the scheduling list, there is another list for the task scheduling within the kernel, the task list. It contains an entry for each task, which is composed of:

- The current value of the system stack pointer of the task. Each MARS task has its own user and system stack. Each time a task is preempted, its registers, its user stack pointer and its status word are pushed onto its own system stack, then the system stack pointer is saved in the task descriptor. Since this action is also done prior to invoking the major interrupt handler (where a context switch might occur), changing to another task is done by merely changing the pointer to the active task descriptor.
- The task identification number.
- The structure of the core image of the task (size and location of text, data and bss segments, pointer to the user– and system stacks, segment checksum…).
- A pointer to a task, which is preempted by this task (this is valid only when the task is active or preempted and when the task is not the fill task).

  - A pointer to the task, which preempts the task (this is valid only when the task is preempted now).

- The task calls *suspend(S_CYCLE)*. This system call causes the suspension of the task for the duration of the current task cycle (until the next invocation of the task). The call must be executed *before* the major interrupt handler reaches an end–task entry in the scheduling list (see chapter 2.1.2). In figure 4 this call is represented by the change from situation II to situation I, respectively from situation IV to situation III. A *suspend(S_CYCLE)* executed by the fill task will be ignored.
- The task calls *suspend(S_TICK)*. The execution of the task will be suspended until the next invocation of the major interrupt handler (it may be used for avoiding busy waiting). In this case, the chaining of tasks, as shown in figure 4, does not change, but the pointer to the active task is changed to the task, which is preempted by the current task, which called *suspend(S_TICK)* (situation V).

Scheduling Switch

In a real time system, there are different states, which must be handled by different schedules. Some examples are:

- System initialization
- Integration (or reintegration) of components
- Alarm handling
- Different states in normal activity (such as reduced operation, for example during the night)

If there were only one static schedule for these situations, this would be extremely difficult to handle, and the run time overhead would be very high (since the maximum execution time calculation must use the longest path in each task causing some overhead in the functions, which do not use this longest path).

MARS provides the *scheduling switch* mechanism to handle such situations. If a task executes the *schedswitch* system call, the current scheduling list is replaced by another list (it may use the same tasks with other start– and endpoints, it may replace only one task, but it also may replace the complete task set). The scheduling lists are calculated prior to system operation, and they reside within the kernel. Two different ways to perform a scheduling switch can be distinguished:

Immediate: This scheduling switch causes the operating system to start the new schedule at once. In the new schedule, the task, which is to be invoked next, is started immediately. Then the new schedule continues normally. Because an immediate scheduling switch cannot guarantee a consistent transition from the previous schedule to the actual one, this call is used only in emergency situations (e.g. for an emergency stop). After handl-

ing the emergency situation, the system usually should perform a warmstart, to restore a consistent state.

Consistent: The consistent scheduling switch is used to perform a well–regulated transition to a new schedule. After executing this system call, normal operation of the old schedule continues until a predefined transition point is reached (one transition point is always the beginning of a new component cycle). After reaching such a point, a switch to the new schedule is performed. Because the transition points are scheduled prior to system operation, the consistent scheduling switch guarantees a consistent transition.

FCFS Schedule

There are some special situations, where the periodic scheduling strategy of MARS is not appropriate. These situations are, for example:

- The initialization phase. The tasks have to be initialized (reserve message buffers,…).
- Alarm scheduling. Alarm handling (e.g. turn off a machine out of operation) is not cyclic due to its nature.

To handle these situations, MARS offers the possibility to carry out a schedule in FCFS[9] order instead of cyclic order. The last task of the FCFS schedule *must* execute a scheduling switch to another schedule (if not, the fill task will run forever).

2.4 Alarm Handling

A real time system has to handle some emergency cases as fast as possible, no matter if normal work can be resumed afterwards or not. E.g., when performing an emergency stop, the controlled environment should assume a safe state as fast as possible. After reaching the safe state, and removing the cause of the alarm, a system restart may be performed.

In MARS, alarm handling is always carried out by switching to a schedule, which handles the emergency situation. There are three methods, to initiate a scheduling switch in the entire cluster:

- Any HRT message may cause a scheduling switch. In each component there is a list, which contains the names of all HRT messages and the number of the schedule to which to switch after the reception of the message (most entries are empty, which means no scheduling switch). If a message causes a scheduling switch in one component, there is no need that the message causes scheduling switches in all other components. Also, the numbers of the schedules, which are executed after a scheduling switch in two different components, need not be the same.

  Since a HRT message of a specific name may be sent only at fixed, predefined points in time, and the scheduling switch is associated with a specific message name, the latency times of alarms are rather long, but consistency is preserved when using this method.

9. First Come, First Serve

- The number of an alarm schedule may be transmitted within each message in a field within the header. Because an alarm may cause different scheduling switches in different components, the number of the alarm schedule is not equal to the number of the schedule to which to switch in each component after reception of the message, but each components provides a list for changing the global number of an alarm schedule to the number of a local schedule.

  Using this method, the scheduling switch is done as fast as possible. Because no HRT message is lost, it may be possible to continue normal work after alarm handling.

- An alarm message is sent. Since the alarm message occupies a TDMA slot normally used by a HRT message, there is no way to return to normal system operation after sending this message. Therefore, an alarm message should be sent only in cases, which require immediate actions, and the costs of handling the alarm too late exceed the costs of a system shutdown. The advantage of this method is its short latency and the fact, that data describing the emergency situation may be sent within the alarm message.

2.5 Clock Functions

There are several reasons, why real time systems must have global time bases:

- A representation of the political time is needed to synchronize system operation with the environment (e.g. to start and stop the controlled machines at fixed points in time in the morning or in the evening).
- The length of a period of time has to be measured (e.g. to detect the end of the validity time of a message).
- Synchronization of processing nodes in a distributed real time system.
- Communication between nodes on the basis of a TDMA protocol.

  …

The administration of the global time base is strongly supported by hardware, especially by the CSU chip. Inside the CSU there is a counter, which makes a time base with a granularity of 1 µs available. Since the counting range is limited to one million, a software counter which counts seconds must be provided. A major problem in clock adjustment is, that adapting of clocks normally is not done continuously (either the clock is stopped for a period of time or some clock ticks are skipped). The CSU clock can be adjusted smoothly (each correction is spread over a whole second). There are two correction factors, which can be set by a system call:

State correction factor: This correction is done once, in the next second after the factor is written into the CSU.

Rate correction factor: This correction is done every second following the second the factor was written into the CSU. This is useful to correct permanent clock drifts.

For the initial setting of the clock a system call is provided, it should be performed *only* during the initialization phase of the component (calling this routine during normal operation may cause unexpected results).

Besides the real time clock, there are some 'logical' clocks, which are used for bus and task scheduling:

tdma_tick contains the number of the current TDMA slot.

comp_tick contains the current component tick, it cyclically counts up to the length of the component cycle (= the smallest common multiple of all task cycles in a component).

cluster_tick contains the current cluster tick, it cyclically counts up to the length of the cluster cycle (= the smallest common multiple of all component cycles in a cluster).

2.6 Timing Profile

To gain a general overview of a component's behavior, it is necessary to determine the periods of time consumed by each task as well as by the operating system. Acquiring these times may be useful to improve the maximum execution time analysis of tasks. There are two ways to do this:

- If the measured execution time exceeds the calculated execution time, the maximum execution time analysis has failed (needless to say that this should never happen).
- If the measured execution time is much smaller than the calculated one, programmers are forced to repeat the task design to reduce the maximum execution time (obviously, this will not be successful in any case, but the chance to reduce the maximum execution time is better for a task, which actually consumes 10 % of its maximum execution time than for a task which consumes 80 %.

For each task, there are three times, which are stored within the task control block: The accumulated CPU–time, the duration of the current task cycle and the length of the longest task cycle.

To obtain a global view of the CPU–time distribution, the time consumed by the system kernel has to be measured as well. There are two conditions, under which a component sets itself into kernel mode:

- A task invokes a system call. The time required by the system call is added to the task's time and *not* to the kernel time.
- An hardware interrupt occurs. Afterwards, the minor interrupt handler and sometimes the major interrupt handler is invoked. There are two separate counters, which sum up the CPU–time of both handlers. For the verification of the maximum execution time analysis not only the sums of the CPU–time, but also the maximum duration of interrupt handling is determined.

2.7 Error Handler

Since the MARS nodes are self–checking units, error handling is rather simple: components are switched off when a severe error is detected. Errors causing a component to switch off by itself are:

- Processor exceptions. They can be caused by faulty hardware, such as an erroneous RAM chip (or another IC), or by a bug in the system software (which may result in a division by zero, illegal address, …).
- A task has exceeded its CPU slot time. Because the maximum execution time analysis is carried out offline, and scheduling is made accordingly, exceeding the execution time of a task is a fatal error (either the maximum execution time analysis has failed, the operating system has failed, or a hardware error has occurred).
- Failure of a peripheral chip, e.g. a fatal error of the LANCE causes the processor node to switch off by itself.
- A double failure in receiving a message. When a message *and* its redundant copy get lost, consistency with the redundant MARS node (and the whole system) can not be guaranteed further on.

When a fatal error had been detected, the cause of the error, as far as detectable, is written to the console and to a fixed memory location for further failure analysis. Error report to a maintenance component via the Ethernet is avoided, because after a fatal error the further behavior of the node is not specified. Therefore no message should be sent on the Ethernet.

Besides the fatal errors, there are some non–fatal errors, which do not cause a component to switch off by itself. All these errors concern the loss of a single Ethernet message. The LANCE reports different reasons for Ethernet failures (Checksum error, framing error, …). These errors are stored in a ringbuffer within the system kernel. An error reporting task can access them via a system call and send them to the operator console for statistical processing.

2.8 Device Drivers

Reading from and writing to devices is done by messages in MARS. There are several message types according to the accessible devices of a MARS node (ETH, CON, AUX, SCSI). Two different ways to access a MARS device can be distinguished:

- Using a MARS message.
  - The bytes, which are to be sent, are written into an ordinary MARS message, whose destination is the device, to which the message is to be sent (or any combination of devices).
  - Reading from a device is done by reading a MARS message, whose source is the corresponding device.

- Using a raw message.
  - The unstructured data is written into a message buffer, and the '*output*' system call is used to send the data.
  - The '*input*' system call returns an buffer with unstructured data from the corresponding device.

  When using raw messages, there is no possibility to send a message to different devices at the same time.

A MARS device driver itself usually consists of two different parts:

- The minor part is called from the minor interrupt handler. It is used to access the peripheral chips (reading is done by polling the chips). The data to be sent are actually written to the device, and the read data is written into system buffers. This part must be written completely in assembler code, since the entire minor interrupt handler is coded in assembler language (to avoid saving of registers).
- The major part is called from the major interrupt handler. It carries out all tasks to make a message available to the application tasks, respectively to prepare a message to be sent to a device. When reading from a character device, it is used to construct MARS messages from the data received in the minor part. When sending data to a character device, the data is extracted from the MARS message and made available to the minor part of the device driver. This part may be written in a higher level language.

2.8.1 Raw Ethernet Driver

In a MARS cluster, the Ethernet is used for MARS messages in TDMA mode. Using a component outside a MARS cluster, e.g. as interface component, requires the operating system to provide the facility to send messages which do not conform to the MARS message structure ('*Raw Ethernet messages*'). In this case, the CSMA/CD protocol may be used instead of TDMA, and repeatedly sending messages is avoided. Since distinction between MARS and non–MARS messages is made only by the Ethernet type field, raw messages may be sent on the MARS bus in TDMA mode. This mechanism allows to implement a gateway to a general purpose Ethernet using a component, which forwards the Ethernet messages to the MARS bus in TDMA mode.

2.8.2 Serial Line Driver

Each MARS component supports two serial interfaces, which are normally used for terminal I/O, but can be used for several other tasks, such as downloading or connecting two MARS nodes of different clusters. Like any other device, the serial lines may be used to transmit MARS messages as well as raw messages. Although MARS messages have a fixed length, unused bytes of a message are not transmitted when using the serial interface (since this inter-

face is very slow compared with the Ethernet or the SCSI). Because normally the serial interface is connected to a terminal, the message header is not transmitted (this would be stuff for the terminal). The result of having neither a fixed message length nor a message header with the actual length of the user data is, that there is no possibility to detect the end of a MARS message. Instead, the operating system supports two methods to determine the end of a MARS message:

- A forwarding character is received. This character may be set by the *device control* system call (the default is the return character, so each line, which is typed at a terminal becomes a separate MARS message).
- A predefined character limit is reached. Normally this limit is the length of the user data area of a MARS message. Setting this limit to 1 results in transmitting each character as soon as typed (using 1 as limit provides an easy way to implement echoing of characters at the terminal).

The minor part of the device driver (executed within the minor interrupt handler) accesses the SCC[10] chip. Since this is done 1000 times per second, and the maximum transfer rate of the serial interface with a standard configuration (9600 baud, 8 bits, 1 startbit, 1 stopbit) is 960 bytes per second, a MARS node drives both serial lines with the *maximum* possible speed under any circumstances.

The major part of the driver (executed within the major interrupt handler) communicates with the minor part via small buffers. It determines the end of a message and builds up MARS messages using the data obtained from the minor device driver.

2.8.3 SCSI Driver

There are two possibilities to use the SCSI bus within a MARS system:

- To access peripheral devices, e.g. harddisks. Because a harddisk (or any other standard device) cannot receive MARS messages, communication is done by raw messages.
- To communicate with other MARS nodes. In this case it is appropriate to use MARS messages for communication.

3 System Tasks

There are several tasks within an operating system, which have to be performed to guarantee correct system operation, but which need not be part of the system kernel. Because one design goal in MARS was to keep the kernel small, such functions are not implemented as parts of the kernel but as so called '*system tasks*', which are scheduled like ordinary application tasks. There are two differences to an application task:

10. Serial Communications Controller

- There are some system calls, which are restricted to system tasks (setting the system time, setting task table entries,…).
- The system tasks are the same within each application. They need not be recompiled with each new application.

Another reason for performing some functions within system tasks instead of within the clock interrupt is, that some operating system functions have to be performed with a lower frequency than the frequency of the execution of the interrupt handlers (for example, the clock synchronization has to be done every ten seconds, whereas the major interrupt handler is called every eight milliseconds).

3.1 Start Task

Immediately after its initialization a MARS node does not have any information concerning its environment (such as the number of components in the cluster, the political time, …). The start task integrates the component into a MARS cluster and into its environment. The start task is executed immediately after the initialization phases of the application tasks (which contains only statements with local effect, such as fetching of message buffers, getting the component name, …; it must not contain statements to send or receive messages or to read the system clock).

Immediately after invocation, the start task attempts to read the external time. There are one or two components in each cluster, which are connected to an external radio clock (PZF), which provides the world–wide synchronized time UTC[11] with an accuracy of 100 µs. A component, which is connected to a PZF reads the time and prompts the user to input the start time of the system. After it, a start message, which contains the current time and the start time of the system, is repeatedly broadcasted until the system start time is reached.

If a component determines, that it is not connected to a PZF, it attempts to receive a start message. When the start message is received, the node sets its local time according to the received value and sends an acknowledge (which is also done repeatedly until the system start time is reached). By reading the start– and acknowledge messages each component determines the configuration of the cluster.

Upon reaching the system start time, a scheduling switch to the application schedule is performed, and the system is ready for normal operation.

3.2 Clock Synchronization Task

During the startup phase, a master synchronization (with the PZF–connected node as master) is performed to synchronize the component clocks. Because this method of synchronization is not fault–tolerant, it must be replaced by a better method during normal operation.

11. Universal Time Coordinated

MARS uses the FTA[12] [Kop87] algorithm to synchronize its clocks. There are no extra messages for clock synchronization, because each MARS message is provided with precise timestamps (sending– and receiving timestamps) (see chapter 2.5).

The clock synchronization task is scheduled every ten seconds. It fetches the latest message from each component, computes the FTA and adjusts the local time according to this value. Because the CSU is able to perform not only a state correction, but also a rate correction of the clock, each component observes the deviation of its own clock to the fault tolerant average of all clocks and attempts to determine the permanent rate failure of its local clock. Eliminating a permanent rate failure by setting the rate correction register of the CSU leads to a synchronization accuracy which is an order of magnitude better than simply performing state correction [Sch88].

To keep the ensemble of clocks synchronized with the external time, the PZF–connected clock broadcasts the deviation of the system time to the external time, and this value is used to modify the FTA to calculate a correction factor which takes the external time into consideration. For initial synchronization with the external time, each clock is tested once to determine its typical rate correction factor, which is used to initialize the rate correction register of the component. Deviations from this value mainly result from temperature effects.

4 Library Functions

The design of the MARS library functions aims at two major goals:

- To make more powerful functions available to the programmer than the system calls do. This is the general goal which each library function in any operating system attempts to achieve.
- To emulate UNIX system calls. Although the UNIX system call environment can not be emulated completely, implementing most of its system calls makes porting of UNIX software easier. After emulating the system calls, the whole UNIX library can be recompiled, which provides an environment to the programmer with which he is more familiar than with the pure MARS system call interface.

MARS library functions, which are not available in a UNIX system, are:

- Send a string to a terminal. Console output is done using state messages. When sending messages to a terminal without flow control, some messages may be lost (if a console–output message overwrites another message which is not be sent to the terminal yet). The *scr_out* library function computes the time needed to transmit a console message, and rejects any other terminal output until the previous message is completely sent.
- Receive a string from a terminal. A console message is received and the input data is extracted from the message.

12. Fault Tolerant Average

- Arithmetic functions on the internal MARS time format.
- Functions to convert the internal MARS time format to the UNIX format and vice versa.
- High level handling of MARS messages. This includes filter functions (a message, or a group of messages is only received if a given condition is true), functions for error handling, … (a more detailed description can be found in [Pfl89], [Dam89]).

All other library functions perform the same (or nearly the same) tasks as existing UNIX system calls or UNIX library functions.

5 Soft Real–Time Kernel

One of the major criticisms of the MARS system is, that due to its design for maximum load conditions, the percentage of machine utilization is rather low under normal load conditions. Although this fact should not prevent us to direct our major attention to the design for maximum load conditions, we attempt to utilize the unused CPU resources for non real time (or soft real time) purposes.

The CPU time, which is consumed neither by hard real time tasks nor by the operating system, is made available to the soft real time system, which is controlled by the SRT kernel. The SRT kernel differs from any other known operating system kernel. The main difference to other kernels is, that it does *not* operate in the CPU supervisor level, thus causing the following effects:

- A failure in the SRT kernel does not affect the HRT system. Therefore, the reliability of the SRT kernel need not to be as high as the reliability of the HRT system.
- The SRT kernel has no access to peripheral chips. The HRT system has to provide system calls, which allow SRT tasks to do their I/O.
- There are some tasks of an operating system kernel, such as task scheduling, which *must* be done in supervisor mode. To do this, the HRT kernel provides some low level system calls which allow the SRT kernel to perform such tasks. For example, there is a system call, which actually does a context switch. There is not implemented any scheduling strategy inside this function, but the SRT kernel computes the task, which is executed next, and then it calls the HRT kernel. The parameters of this system call are the task to be performed next, and the length of the time slice of the task. Then the task switch is performed and the SRT task starts its execution.

The CPU time, which is available to a SRT task depends on the time consumed by the HRT tasks and thus is not constant. Therefore, a fair scheduling strategy cannot be implemented without knowing the CPU time actually consumed by each SRT task. Therefore, the length of a timeslice of a SRT task is not given in units of real time but in units of actually consumed CPU time. Optionally the SRT kernel may determine an absolute point in time when the SRT kernel is to be invoked next (if a task with a higher priority is to be scheduled at that time).

There are three different ways to implement the SRT system:

- The SRT system is used to perform (soft) real time tasks. In this case, the design of the SRT kernel should follow an existing (soft) real time kernel, such as pSOS[13].
- The SRT system is used to perform non real time tasks. In this case, the design of the SRT kernel (better: non real time kernel) should follow any existing operating system, such as UNIX.
- The SRT system is used to perform both soft real time tasks and non real time tasks. In this case, the non real time system may consume the remaining CPU time of the SRT system (like the SRT system, which uses the remaining CPU time of the HRT system).

The design and implementation of the SRT kernel is currently in progress.

6 Summary

In this paper we described the design and implementation of the MARS operating system. Since the requirements on this operating system strongly deviate from all known existing systems, the design was done from scratch, and few concepts from other systems were used.

Because MARS is a distributed system, one of the major tasks of the operating system is communication between different components. Actually, the message handling system was the first part of the operating system, which has been developed, and the design of the other parts of the operating system was done with regard to the constraints given by the implementation of the message handling system.

Currently, the SRT kernel is under development. Efforts are made to separate the SRT– and HRT system in order not to decrease the reliability of the HRT system by the SRT system.

6.1 A Short Look into the Future

Simultaneously to the development of the MARS operating system software, the design of a new MARS hardware is undertaken. In cooperation with the ‘Institut für Elektrische Meßtechnik’ new MARS components are developed.

The main enhancements of the new nodes are:

- Use of a RISC[14] Processor, probably the Motorola 88000 [Mot88].
- Use of a separate I/O processor.
- Dual ported RAM for communication between the application– and the I/O processor.
- Two Ethernet interfaces (for redundant MARS buses).

13. pSOS is a real time, multiprocessing operating system kernel. It is a trademark of Software Components Group Inc.
14. Reduced Instruction Set Computer

- A general purpose I/O bus (e.g. VME bus, NUBUS, …).
- Massive use of hardware self checking mechanisms(watchdog timer, parity bits, signatures, …) [Dam88].

References

[Dam88] A. Damm. *Experimental Evaluation of Error–Detection– and Self–Checking Coverage of Components of a Distributed Real–Time System.*. PhD Thesis, Technisch Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria, Oct. 1988.

[Dam89] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The Operating System of MARS. *ACM Operating System Review*, 23(3):141–157, July 1989.

[Kop85] H. Kopetz and W. Merker. The Architecture of MARS. In *Proc. 15th Int. Symposium on Fault–Tolerant Computing*, pages 274–279, Ann Arbor, Michigan, June 1985.

[Kop87] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real–Time Systems. *IEEE Transactions on Computers*, 36(8):933–940, Aug. 1987.

[Kop89] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real Time Systems. *IEEE Transactions on Computers*, 36(8):933–940, Aug. 1987.

[Met76] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[Mot88] *Technical Summary: 32–Bit RISC Microprozessor MC88100*. Motorola Inc., Phoenix, Arizona, Dec. 1988.

[Pfl89] M. Pflügl, A. Damm, and W. Schwabl. Interprocess Communication in MARS. In *Proceedings of the ITG/GI Conference on Communication in Distributed Systems*, Gesellschaft für Informatik, Stuttgart, BRD, Feb. 1989.

[Pus89] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real/ Time Programs. *Real–Time Systems*, 1(2):159–176, Sep. 1989.

[Sch88] W. Schwabl. *Der Einfluß zufälliger und systematischer Fehler auf die Uhrensynchronisation in verteilten Echtzeitsystemen*. PhD Thesis, Technisch Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria, Oct. 1988.

[Sen88] Ch. Senft. A. Computer–Aided Design Environment for Distributed Real–Time Systems. In *IEEE Compeuro 88, System Design: Concepts, Methods and Tools*, pages 288–297, Brussels, Belgium, Apr. 1988.