MASTER

The real-time EMPS kernel : memory management and suitability for EPEP/PhyDAS

Marissen, R.J.

Award date:
1994

The real-time EMPS kernel: Memory Management and suitability for EPEP/PhyDAS

Rob Marissen

September 14, 1994 NF/FTL 9404

Technische Universiteit Eindhoven
Faculteit Technische Natuurkunde
Vakgroep Fysische Informatica

Afstudeerdocent: prof. dr. ir. K.Kopinga
Technische begeleiding: ing. L.A.H.M. van Houten

Abstract

The EMPS system is a multi-processor system designed for two different application areas, namely (i) real-time data processing and control of physics experiments, and (ii) dependable distributed computing. The hardware of the EMPS system is built around a VME/VSB computer bus and computer modules based on the MC68030 processor. A short description of the EMPS hardware can be found in chapter 1. The emphasis of the rest of the chapters is on the software of the system, which is called the EMPS kernel.

Multi-processor systems require another approach to writing system software than single processor systems do. Chapter 2 deals with the particular multi-processor related parts of the kernel. Also, some more practical problems associated with the multi-tasking properties of the kernel are addressed.

In a system that serves multiple purposes, the system software must be flexible in order to accommodate different needs. One of the parts of the kernel in which this flexibility is expressed, is the handling of memory resources, often referred to as memory management. The MC68030 processor, which is used in the EMPS system, offers hardware support for memory management in the form of a Memory Management Unit (MMU). Extensions that have been added to the memory management in the EMPS kernel, and the use of the MMU are discussed in chapter 3.

The performance and real-time behavior of kernel services is important when the EMPS system is used to control physics experiments. In chapter 4, the results of performance tests are presented. The performance degradation caused by the particular way interrupts are handled in the EMPS kernel, as well as some suggestions for improvements, are also discussed.

In chapter 5, the results of the previous chapters are discussed.

In the appendixes, the kernel functions that can be called from application programs are described. Information about the most important data structures inside the kernel, as well as other relevant data, is also given.

Contents

Chapter 1

Introduction

The Eindhoven Multi Processor System (EMPS) has been designed at the Eindhoven University of Technology for two different application areas, namely data processing and control of physics experiments, and dependable distributed computing. The EMPS comprises both the hardware and the software of this multi-processor system.

In this chapter, an overview of the EMPS hardware is given, followed by some characteristics of the EMPS software and a section that focuses on the use of the EMPS for control of physics experiments. The development status of the software is also dealt with.

1.1 EMPS hardware

Multi-processor systems can be classified by the way information is exchanged between processors. Two types of communication can be distinguished:

- A system in which processors communicate via shared memory: a tightly coupled system.
- A system in which processors can only communicate via other means, e.g. a local area network: a loosely coupled system.

In the EMPS system, both types are combined, thus forming a hierarchy of communication channels. The system consists of *nodes*, which contain a number of processor-, memory-, and other modules. Within each node, shared memory can be used for communication. Communication between nodes uses either a local area network or fast (20 Mbits/s) serial transputer links which reside on each processor module.

The local area network, which is called PhyLAN, is also used to load the EMPS software when the system is started. The EMPS software and other files are supplied through PhyLAN via a fileserver. In the development stage of the EMPS software the fileserver is a PC, as shown is figure 1.1. Because the actual development of the software is done on a another computer, namely a SUN workstation, the PC in turn is connected to this workstation using

a different local area network, ETHERNET. Hence, the PC is only a link between the SUN workstation and the EMPS system.

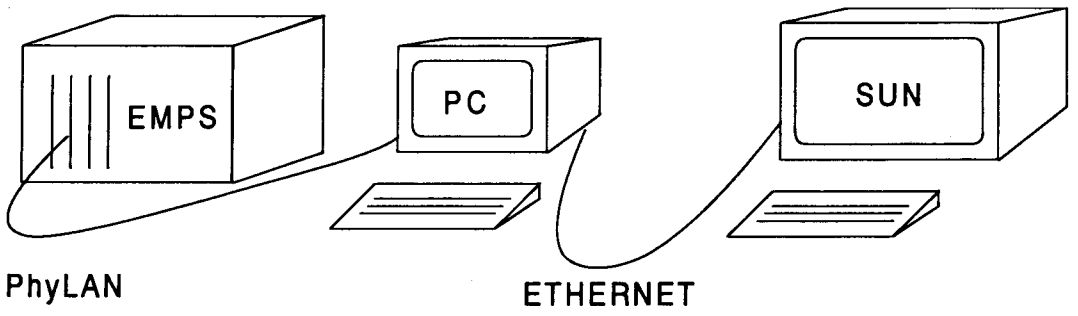

Figure 1.1: *The EMPS is connected via PhyLAN to its fileserver, a PC. The PC is connected via ETHERNET to a SUN workstation.*

1.1.1 The VME/VSB bus

The hardware of the EMPS system is built around a VME/VSB computer bus. The VSB is an extension of the VME bus, but can be considered as an independent bus. Most of the modules that are used in the EMPS system, can be accessed via both the VME bus and the VSB bus. In multi-processor systems the shared bus often forms a bottleneck. The availability of two separate buses helps to avoid this. Processor modules, memory modules, and others are connected to the bus by inserting them in *slots*, of which a maximum of 20 are available in each node.

In figure 1.2 the bus architecture is shown. All modules within one node can be accessed via the VME bus, whereas the VSB bus is split into groups of 5 slots, called *clusters*. The VSB bus can only be used to access modules that belong to the same cluster. The division of a node into clusters further helps to eliminate the occurrence of a communication bottleneck, because access to shared memory within the same cluster using the VSB bus does not occupy the VSB bus of other clusters.

Both the VME and the VSB bus use 32 bits data and addresses. On the VME bus, data and addresses have separate sub-buses, whereas on the VSB bus, data and address lines are time-multiplexed.

Within the bus system, the first slot has a special function: it contains the *system controller*. The hardware of the system controller is almost the same as that of processor modules, except that it has extra facilities for bus arbitration. The software of the EMPS system is designed in such a way that system bus interrupts are handled exclusively by the system controller.

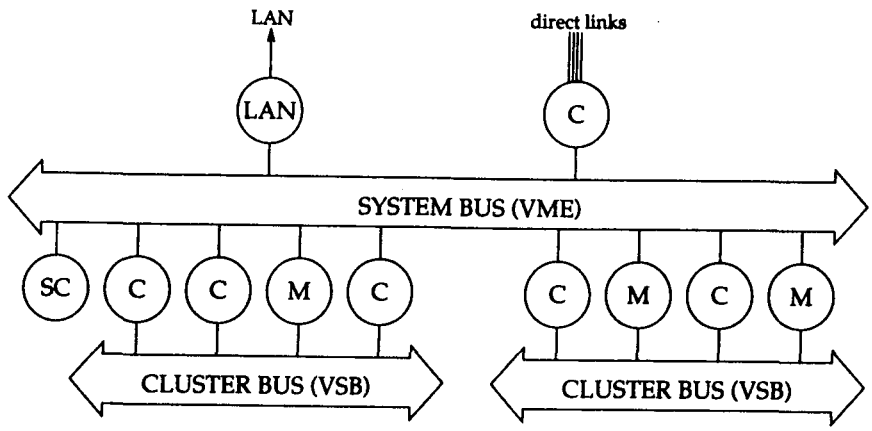

Figure 1.2: *The bus architecture of the EMPS system. The VME bus connects all modules in a node, whereas the VSB bus is split in clusters of 5 slots.*

1.1.2 The processor module

The heart of the processor module on which the EMPS system is based is formed by a Motorola MC68030 processor, that is upward object code compatible with the MC68000. Some of the features of the MC68030 are an on-chip memory management unit, data- and instruction caches, 32 bits data- and address buses, and 16 general purpose registers. The processor is clocked at 33 MHz and can be extended with a MC68882 floating point co-processor. Figure 1.3 shows a block diagram of the computer module.

Each processor module is equipped with 2 Mbyte static RAM and 512 kb EPROM. These memories are local to the processor module, and cannot be accessed by other modules. The RAM, also referred to as *private memory*, is used in applications that make extensive use of memory, e.g. program code. A monitor program is stored in the EPROM. At system start up, the monitor program is executed, and copies itself to the local RAM memory for faster access. The monitor program is used to download programs, i.e. the EMPS kernel, via the local area network.

A Dual Universal Asynchronous Receiver/Transmitter (DUART) provides two RS232 serial connections, e.g. for terminals. A programmable timer is also available; it is used to periodically generate interrupts at 20 ms intervals.

Transputer links offer a means for fast communication. Every processor module has four transputer links that transfer data at 20 Mbit/s each. The links can be used for communication between processors in the EMPS system,

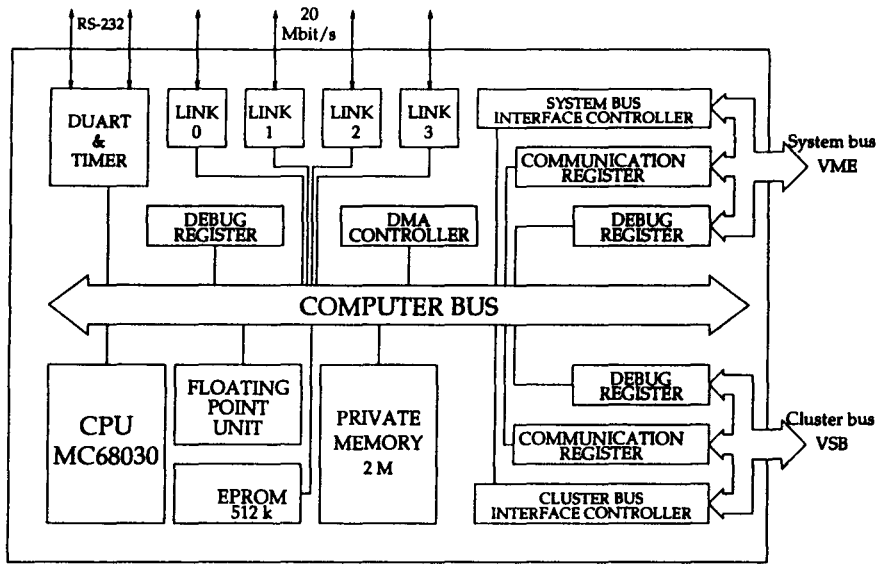

Figure 1.3: *A block diagram of the processor module.*

but are also used for fast transfer of data obtained from physics experiments from a computer module to a PC.

The processor modules in the EMPS system are equipped with a special facility for efficient communication between processors within a node in the form of *communication registers.* Communication in tightly coupled multiprocessor system is often done by writing a message into shared memory, and generating an interrupt at the target processor to attract its attention. The VME bus offers 7 interrupts for all modules together, which is insufficient to use this method efficiently. Communication registers offer a more efficient way of generating an interrupt at another processor, while simultaneously exchanging some information. The communication register of one processor can directly be written by another processor, by writing to a specific memory location. An interrupt is automatically generated at the target processor, which then can read the communication register. Thus, the generation of an interrupt at another processor requires nothing but a write action to a specific memory location. The processor module has a separate communication register for the VME and the VSB bus, respectively.

The address at which a communication register can be reached, depends on the slot of the processor module. The slot number is derived from the place within a cluster, which is obtained from geographical address signals supplied by the bus, and from the setting of two jumpers on the module, which are set to the number of the cluster. In appendix B, the address of

the VME communication register is listed for each slot number.

1.1.3 The memory module

The memory modules used in the EMPS system can contain 4 Mbyte to 64 Mbyte DRAM. Memory can be accessed via both the VME and the VSB bus, with cycle times of 500 ns and 600 ns, respectively. The memory modules can be accessed by all processors within one node and will be used, among others, as the shared memory that is used for communication between processors. It will also be referred to as *common memory.*

Access arbitration is arranged in 3 priorities: refresh cycles, which have the highest priority, VME memory accesses, which are handled when no refresh is pending, and VSB memory accesses, which have the lowest priority. Thus, if a memory module is accessed at the same time via the VME and VSB bus, VME access is handled first. This does not have a serious impact on the speed of VSB accesses, because the access time of the DRAM memory on the module is much smaller than the cycle time of the VME and VSB buses. A VSB memory access may therefore be started before the VME access cycle is completed, so that the accesses overlap.

Both the VME and the VSB buses support uninterrupted read-modify-write cycles, which are useful for some multi-processor functions, e.g. semaphore handling.

The address space of the memory module is determined by the slot in which the module is placed. The slot number is obtained in the same way as for the processor module. In appendix B the addresses of memory modules as a function of the slot number are listed. Whether a memory module is accessed via the VME bus or via the VSB bus, is determined by the address that is used.

1.1.4 The PhyLAN module

Each node in the EMPS system is equipped with a local area network controller that is used for communication with other computer systems. The local area network, which is called PhyLAN, has a transmission rate of 2.5 Mbit/s. The PhyLAN module has an on-board DMA controller to support fast data transfer to VME/VSB memory modules. Direct data transfer to the local memory of the computer modules is not possible, because this memory cannot be accessed via the VME/VSB bus.

In the future, a different networking capability will be added to the system in the form of an ETHERNET module.

1.1.5 The VME-PhyBUS converter

PhyBUS is the bus system of interface modules that are used to control physics experiments. The VME-PhyBUS converter forms the connection between the VME bus and the PhyBUS. This module contains a DMA controller for fast data transfer between the PhyBUS and the VME bus, and some logic to convert the 32 bit PhyBUS signals to variable size (8/16/32 bit) VME bus signals.

1.2 EMPS software

The system software of the EMPS, also called the EMPS kernel [dij 93], was designed to perform the basic functions needed by both the Department of Physics and the Department of Mathematics and Computer Science. These include multi-tasking facilities, interrupt handling, semaphores, and memory management, which are also present in single-processor kernels.

Support for multi-processor applications is given in the form of location transparent exchange of information between processes using mailboxes, and process migration. Facilities offered by the kernel are available uniformly at all processor modules in the EMPS system, since the program code of the kernel is exactly the same at all processors. However, access to specific objects is in some cases limited to one processor module, or to processors within one node (e.g. semaphores, memory).

For efficiency reasons, the EMPS kernel has been implemented in the C programming language. The kernel is developed using an Oasys cross compiler [oas 93] running on a SUN workstation.

1.3 Data processing and experiment control

At the time cheap computers first became available, increased use of computers in control of physics experiments resulted in a fast expansion of the different types of interfaces that were being developed for this purpose. It was realized that, in order to limit this growing number of different hardware designs, some form of standardization was required. Within the Department of Physics this has lead to the development of a general purpose Physics Data Acquisition System (PhyDAS).

1.3.1 PhyDAS

PhyDAS is the name of the system that was developed to control physics experiments and to acquire experimental data. It consists of a number of

data-acquisition and control interfaces, which are connected with an asynchronous bus system, called the PhyBUS. The PhyBUS is connected to a computer system that controls the interfaces, and that can have a different bus system. Since the life cycle of computers is generally much shorter than the life cycle of the interface modules, separating the interface bus from the computer bus enables cost-efficient upgrading to a new computer generation.

The PhyDAS concept was developed around 1979 and has since been used with three different computer generations: first DEC PDP11, followed by a system based on the Motorola 68000 processor, which is called the Micro-Giant, and more recently, the EMPS system. Currently, only the hardware part of the EMPS system is used to control the PhyDAS hardware. The software of the EMPS is still in development and will later be incorporated in the system. At this moment, the software that is used to control the PhyDAS interfaces is an "all in one" program, called EPEP.

1.3.2 EPEP

The system software that is used for the control of physics experiments, is an interpreter type of programming language, called EPEP (Eindhoven Program Editor and Processor). This language provides special features for process control in the form of a multi-tasking capability with event driven process switching. Furthermore, processes can have different priorities, reflecting the urgency of each process. EPEP has a command interpreter that can be used interactively while processes are being executed, which is useful for accessing data and parameters of an experiment that has already started, i.e. runs under program control.

The version of EPEP that is currently used in the Department of Physics is not only an interpreter, but also comprises the underlying operating system plus a program editor. All these functions are integrated in one program. Because EPEP was developed at a time that efficient use of computer resources was still a factor of primary importance, this program has been written in assembly language.

Because of the ongoing development of micro electronics, efficient use of resources, especially memory, has become less important. Maintainability and portability, on the other hand, are now factors of increased consideration. Therefore, the three functions of EPEP, (interpreter, editor, and operating system) will be split in separate parts, which are implemented in a higher, portable programming language, namely C.

One of main the goals of the EMPS kernel is to provide the basic functionality upon which the EPEP interpreter and editor will be built. This is reflected in the services offered by the kernel, which include the capabilities of the operating system part of EPEP, for instance multi-tasking with event driven process switching, different process priorities, and semaphores.

1.4 Overview

Low level multi-tasking facilities form the basis for all other kernel functions, including multi-processor services. Chapter 2 includes a description of multi-tasking concepts which are specific for the EMPS kernel. The multi-processor services that are concerned with handling interrupts are also described.

The original version of the EMPS kernel has a property that limits its usefulness: hard limits are placed on the amount of memory that can be used by application programs, because fixed-size data structures are used in memory management routines. To make memory management more flexible, this part of the kernel has been rewritten. In chapter 3, the memory management of the kernel is described.

The performance and real-time behavior of kernel routines is important when the EMPS system is used to control physics experiments. The results of performance tests are presented in chapter 4. Special attention is given to the performance degradation that is caused by interrupts, which are handled by dedicated Interrupt Service Processes in the EMPS kernel.

The results of the previous chapters are discussed in chapter 5.

Documentation about the original kernel was only available from an object oriented description and from sources code. From these sources, information that is important for extensions of the kernel, plus a description of kernel services that can be used in application programs has been gathered. This information is presented in the appendices.

Chapter 2

Multi-processor and multi-tasking aspects

A property of the EMPS system is its non-homogeneous communication topology, which is supported by kernel services. If processors are located in the same node, shared memory is used for communication, otherwise the PhyLAN is used. Which communication path is used, is transparent for application programs that use kernel services developed for this purpose. However, some other kernel services are not available transparently in different nodes, e.g. semaphores, can only be used within the same node. When the kernel is used as a basis for EPEP, it is not expected that more that one node will be needed in the near future. The description of multi-processor functions will therefore be limited to services that work within one node.

The basis of the EMPS kernel is formed by a set of low level functions that offer multi-tasking facilities. Interrupt driven task switching is one of these facilities. The handling of interrupts, which is done by dedicated processes, is described in section 2.2.

2.1 Multi-processor services

Mailbox communication and process migration are the services that make the kernel a multi-processor kernel. Mailbox communication offers communication between processes without the need that processes keep track of each others locations. Process migration provides the possibility to transfer a process between processors in a distributed system. It supports efficient use of computing resources, because the processing load can be balanced evenly over all processors in the system.

2.1.1 Mailbox communication

Location transparent communication is one of the goals of the EMPS kernel. It is achieved by using mailboxes, which allow transparent communication between processes, even if a process has been migrated to another processor.

When mailbox communication is used, processes send information to a mailbox instead of to another process. The kernel, which handles the mailbox, routes the information to its destination.

Setting up a communication path between processes using mailboxes consists of 2 steps:

1. Creating a mailbox. The mailbox data structure that is created, is used by the kernel to keep track of processes that are communicating. Only one mailbox must be created for all processes that communicate via that mailbox.

2. Attaching a process to a mailbox. This is done by connecting a data structure of type *PORT* to the mailbox. *PORT* is used by the process to communicate via the mailbox. Each process that communicates via a mailbox must connect a port to that mailbox.

Processes connected to the same mailbox can communicate by issuing send- or receive commands to the port through which they are attached to that mailbox.

Mailbox communication is location transparent, because processes send and receive information via the mailbox and have no knowledge of the location of the process with which they communicate. The actual process locations are stored in the mailbox data structure, and are updated by the kernel when a process migrates.

Mailboxes provide both a unicast service (*send* and *receive* primitives) and a Remote Procedure Call (RPC) service (*send request*, *receive request* and *send reply* primitives). When a process attaches to a mailbox, the port type determines whether the unicast service or the RPC service is provided. Details about the port types are included in appendix D.

The kernel also includes unicast and RPC services that are not location transparent. These services are completely separate from the mailbox services and are used internally in the kernel. They provide the same functionality as the mailbox services, but in a location dependent way.

The mailbox operations are very similar to the file I/O operations in the C language.

2.1.2 Process migration

Processes in the EMPS system can be migrated to another processor transparently, i.e. without any changes to their computation or communication. Process migration can be used to balance the processing load over all processor modules in the system.

Process migration is handled by migration server processes that are present at each processor in the EMPS system. To initiate migration, a message is

sent to the migration server process at the processor where the process that must be migrated is present. This message contains an identification of the process that will be migrated, plus the destination of this process. The migration request can be sent by any process on the system.

The migration server at the source processor detaches the process from its environment by removing it from the queue it is in and by locking the mailboxes to which it is connected. Next, it gathers all process information, e.g. process state, processor registers, code-, data-, and stack memory, and sends this to the migration server at the destination processor. The process is then removed from the source processor.

At the destination processor, a new process is created using the process information obtained from the source processor, and mailbox connections are redirected to the new process. The new process is then restarted.

A process can only be migrated if it meets the following demands:

- The process only communicates via mailboxes. Most services of the kernel are provided by server processes which can be reached via a mailbox, and can therefore be used by a process that is migrated.
- No dynamically allocated heap memory is used. Migration of heap memory is not supported.

Use of the EMPS process migration facility is less appropriate for load balancing within the EPEP interpreter. In EPEP, a process consists of EPEP instructions that are interpreted, whereas an EMPS process consists of MC68030 instructions. Some sort of mapping is needed such that a EPEP process corresponds to an EMPS process. In practical terms, this means that the interpreter will be an EMPS process that interprets EPEP instructions. When a new EPEP process is created, this will cause the creation of a new EMPS process, that again interprets EPEP instructions. If an EMPS process is created, only a reference to the program code, data, and heap is made. However, if a process is migrated, a copy of the code and data is made. This is undesirable, because it can cause multiple copies of the same code and data to be present on one processor if the process that has been migrated was part of a task including more processes. Also, within EPEP information is exchanged between processes through the memory they share. When the EMPS process migration facility is used, this is no longer possible.

2.2 Multi-tasking facilities

2.2.1 Tasks and processes

Programs that run on top of the kernel, such as EPEP, are called application programs or tasks. Within one task, several related processes can be defined.

The memory space for program code, data and heap are shared between all processes in one task. Each process within a task has its own stack memory.

2.2.2 Supervisor/user mode

The MC68030 processor supports protection of system facilities by offering two modes for program execution: the supervisor mode, which allows access to all facilities, and the user mode, which limits access. Each mode has its own stack pointer. In user mode, execution of certain instructions is prohibited and memory access can be limited to specific areas by using the MMU (chapter 3).

For some kernel functions, execution in supervisor mode is required. Switching between user mode and supervisor mode is a time consuming operation, because of the rather complicated passing of parameters between user mode and supervisor mode routines. Therefore, the whole kernel, including all system processes, execute in supervisor mode.

Application programs should not access privileged facilities, and execute in user mode. A switch from user mode to supervisor mode is made when a kernel routine is called. Parameters for this routine are put on the user stack, and must be copied to the supervisor stack before the kernel routine is invoked. The way a kernel function is called from an application program is described in appendix D.

2.2.3 Interrupt handling

Interrupts are generated by hardware devices to indicate that an event has occurred, e.g. a character was received from the terminal. For some hardware devices, the interrupt can have multiple causes. The DUART for example, contains a clock, 2 serial outputs, and 2 serial inputs, each of which can cause an interrupt. One physical interrupt source may thus be associated with several logical interrupts, each of which is associated with a logical device.

When an interrupt is activated, this leads to the starting of an Interrupt Service Routine (ISR). Each physical interrupt source has its own ISR. In the EMPS kernel, the ISR does not handle the interrupt itself. Instead, logical interrupts are handled by dedicated processes, called Interrupt Service Processes (ISPs). The ISR determines which logical device is the cause of the interrupt, and unblocks the ISP that corresponds to it.

The main reason for using ISPs to handle interrupts is that this offers location independent access to devices. A process that wants to perform an I/O operation sends a request message to the ISP that handles the device. Because messages can be sent to processes at any location in the system, location independence is achieved. A typical ISP consists of an infinite loop,

in which it waits for a request message, initializes the I/O device, waits for the interrupt that signals I/O completion, and handles the interrupts. When the ISP is associated with an input device, it sends a response message. An example of such an ISP is shown in figure 2.1.

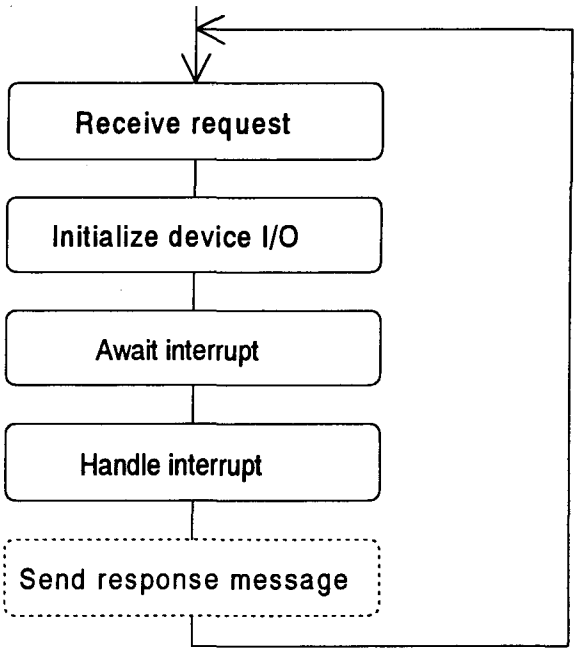

Figure 2.1: *A typical ISP consists of an infinite loop in which I/O related actions are performed.*

Apart from location independent device access, handling of interrupts by ISPs has the following advantages:

- **Mutual exclusion:** Mutual exclusion between processes that perform I/O operations on the same device is achieved automatically, because there is only one ISP that physically accesses the device.
- **Minimal stack use for handling interrupts:** Interrupts are handled on the supervisor stack of the currently executing process. When an ISR would handle device interrupts, sufficient stack memory would have to be allocated to each process to handle nested interrupts from all devices. Interrupt handling can involve calling of other routines which use the stack to store local variables. By using ISPs that handle interrupts, these routines use the stack memory of an ISP, so that stack use by ISRs is minimized.

  We like to note that for the MC68030 processor, this argument is not completely valid, because it has the possibility of using a separate stack

pointer for interrupt processing. By using this interrupt stack pointer, reserving extra stack memory in all processes is not necessary.

- **Buffered interrupts:** Access to some internal data structures in the kernel must not be shared by processes. Shielding these data structures with semaphores is not always possible (e.g. semaphore data structures cannot be shielded by semaphores). Therefore, interrupt driven task switching must be disabled during access to these data structures. If this is done by disabling interrupts completely, interrupts may be lost. With ISPs, activation of the ISP is postponed until access to the data structure is permitted. If an interrupt occurs while activation of the ISP is postponed, the I/O registers of the device that corresponds to the interrupt are stored in a buffer.

There are also some disadvantages in using ISPs:

- **Performance loss:** For every interrupt, a context switch is needed to activate the ISP. When the interrupt handling is completed, another context switch is needed to continue execution of a regular process. Some overhead is also created by the kernel routine that unblocks the ISP after an interrupt has occurred, and by the routine that blocks the ISP when interrupt handling is completed. In chapter 4, the performance loss is discussed.

- **Increased interrupt latency:** The context switch and other processing that occurs before the ISP is activated, increase the time between assertion of the interrupt and handling of the device.

Interrupt service processes always have a higher priority than other processes, and therefore they should only be used to perform actions that are directly related to the interrupt. Any subsequent actions, e.g. the numerical processing of data from the device, must be done by activating another process (e.g. by sending a message to a process or by signaling a semaphore).

Each logical device is associated with a data structure of type *DEVICE*, which contains information concerning the device, i.e. which ISP handles the device interrupts, an optional timeout routine, and the number of times that the device interrupt has occurred.

The following logical devices are connected with an interrupt service process:

- **Clock:** the clock ISP is activated at 20 ms intervals, and is used to start timeout routines and to restart processes that are waiting for a specified amount of time. The clock can also be used to divide the processor time over the processes which are ready to execute, by rotating the queue

of these processes. This is called time slicing. Time slicing may not always be desirable, and therefore the corresponding program code for time slicing is compiled conditionally. Compilation of the program code for time slicing is enabled with a `#define TIME_SLICE` statement.

- **Terminal input:** the terminal input ISP is activated when a character is received from the terminal.
- **Terminal output:** the terminal output ISP is only activated when two conditions are met: the DUART must be ready to send a character, and output must be available.
- **Communication register:** the communication register ISP is activated when another processor has written a message into the communication register of the processor module. It is used in communication between processors.
- **PhyLAN I/O:** the PhyLAN ISP is used to service the PhyLAN module. This interrupt is handled by the system controller (processor in slot 0).
- **Link input:** the link input ISP is activated when data was received via a transputer link on the processor module.
- **Link output:** the link output ISP is activated when sending data via a transputer link is completed.
- **PhyBUS interrupt:** contrary to the other interrupts, the PhyBUS interrupt cannot be handled completely within the kernel.

  The PhyBUS interrupt is a VME bus interrupt, which is generated by the VME-PhyBUS converter when one of the interfaces connected to the PhyBUS signals an interrupt. There can be a maximum of 16 interfaces acting as an interrupt source on the PhyBUS. Which interface caused the interrupt, is determined by the ISP that handles the PhyBUS interrupt.

  The configuration of the PhyBUS interfaces depends on the experiment for which they are used, and therefore the PhyBUS interrupt cannot be handled any further inside the kernel. This PhyBUS interrupt must somehow be handled by the software which controls the experiment, which will be an EPEP program. To activate a process that handles the PhyBUS interface which caused the interrupt, semaphores are used. For each of the 16 possible PhyBUS interrupt sources, a separate semaphore exists. When a PhyBUS interrupt occurs, the ISP performs a signal operation on the semaphore that corresponds to the so-called

interrupt bit of the interface from which the interrupt originates. The 16 PhyBUS semaphores are named *PhyBUS 0* through *PhyBUS 15.*

Because the PhyBUS related program code is only needed when the EMPS system is used to control the PhyBUS, it has been made optional. At compile time of the kernel, generation of the extra code for the PhyBUS can be enabled with a `#define PHYBUS` statement.

Chapter 3

Memory Management

One of the primary functions of every operating system is handling memory resources, which is called memory management. In the EMPS system this consists of two parts:

- Maintaining a list of memory locations that are occupied.
- Protection of memory areas, and separation of (logical) addresses used in programs from the (physical) addresses that are used to access hardware. This is done by using the MMU (Memory Management Unit) of the MC68030 processor.

These two parts are largely independent.

In many operating systems, memory management allows programs to access more memory than is physically available. This can be achieved on computer systems that contain an MMU, by mapping memory areas to secondary (disk) storage, and is completely transparent for application programs. In the EMPS kernel this technique is not used, because the delay caused by swapping memory areas to secondary storage is not compatible with the real-time requirements of the kernel.

3.1 Memory allocation

In any computer system that can have more than one program in memory simultaneously, the use of memory by each program must somehow be administered to avoid that different programs use the same memory area. This administration is one of the tasks of the kernel. To this end, it must keep information about occupied and free memory areas, which is only possible if a program invokes a kernel routine when it needs memory.

When a program needs a specified amount of memory, it issues a call to the kernel (e.g. in the C language the function *malloc* is used). The kernel then checks whether the required amount of memory is available and, if so, it registers the memory area as occupied and returns the start address of the area. If there is not sufficient free memory, some sort of error condition is

signaled. When the program no longer needs the memory, it issues another kernel call (in C, the function *free* is called) so that the memory can be added to the free area.

In the EMPS system, memory is divided into blocks of a fixed size, called the pagesize. The size of a memory area that is allocated is always an integer multiple of the pagesize. The reason for splitting memory into pages originates from the hardware of the MMU, which will be discussed in the following sections. In the current kernel version, the pagesize is 1 kbyte.

To register free and occupied pages, the EMPS kernel uses an array of bytes, which will be referred to as the allocation table. The index of an element of this array is equal to the number of the memory page that corresponds to it. A byte in the allocation table contains the value FREEPAGE if the corresponding page is free. If a page is occupied, it contains the value OCCUPIEDPAGE, FIRSTPAGE or LASTPAGE. An example of the allocation table is shown in figure 3.1.

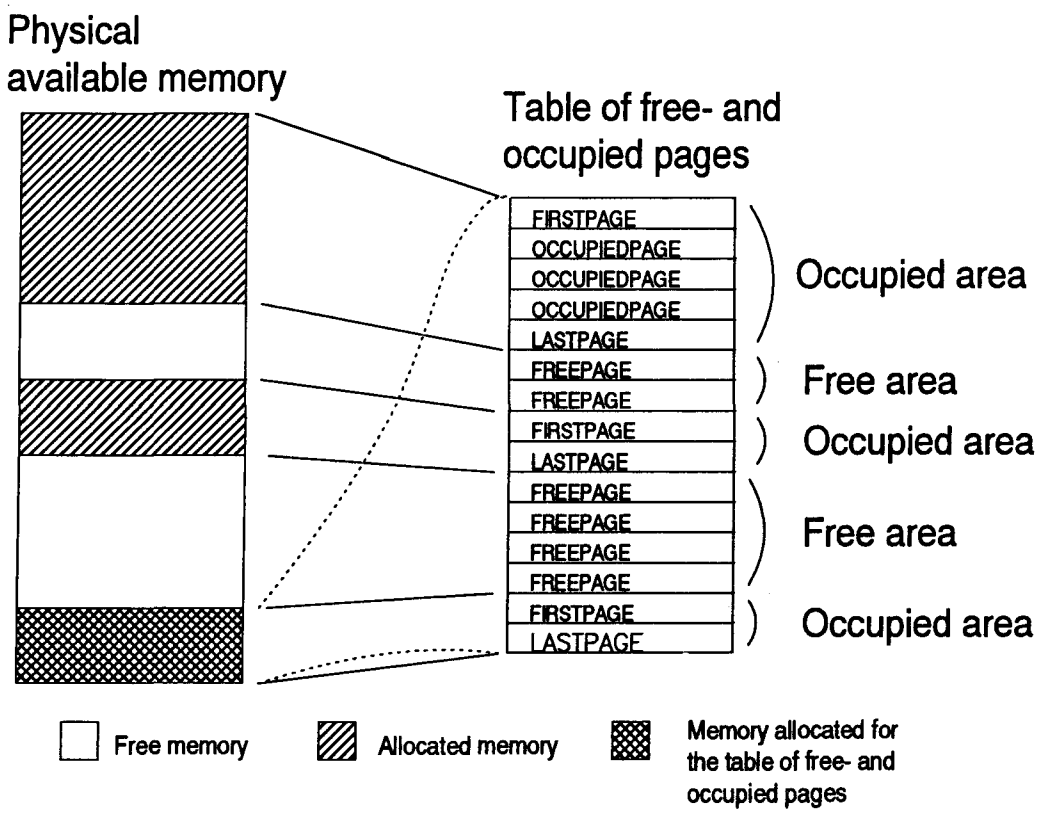

Figure 3.1: *A list of free- and occupied pages is used for memory allocation. The list is located at the end of the memory area to which it belongs.*

The lowest level memory allocation routine takes a parameter which contains the number of contiguous pages that is required. When memory must be allocated, the allocation table is searched for the requested number of contiguous free pages. The array is then modified to indicate that memory is occupied.

The method of memory allocation that is employed in the EMPS system is not used in most other systems, because searching an array for a number of contiguous pages is a slow operation. Several other ways of keeping track of memory use are described in [tan 87]. The most commonly used algorithms maintain a linked list of allocated and free memory areas, respectively. An advantage of such a method is that searching for a free memory area is faster. A drawback is that the size of the linked lists changes when memory is allocated, whereas the size of the allocation table always remains the same. A variable list size is more difficult to implement. When the EMPS system is used in control of physics experiments, the speed of memory allocation operations is not an important factor, because memory allocation will not be performed in time-critical program sections.

We like to note that other methods of memory allocation can be implemented in the EMPS kernel relatively easy, since the allocation part of the kernel is completely separated from the other memory management functions.

3.1.1 Implementation of memory allocation

The basic routines that the kernel has to supply are the allocation of a memory area, and the release of a previously allocated area. In the EMPS kernel, this is done by the low level functions *reserve_contiguous_pages* and *release_contiguous_pages*.

The memory space in the EMPS system consists of different areas: private memory, which is the on board memory of the processor module, and one or more common memory areas from memory modules. In general, memory areas from different memory modules are not contiguous; a gap of unused addresses exists between these areas.

For each contiguous memory area, a separate allocation table is maintained. The position of the table is always at the end of the memory area to which it belongs, as can be seen in figure 3.1.

Because multiple processes may be allocating or releasing memory, mutual exclusion is required. This would suggest using semaphores to shield access to the allocation table. However, memory used to store semaphore data structures is allocated dynamically in the initialization phase of the kernel, which means that the allocation table is accessed before semaphores can be used. To avoid the circular problem that results from this, the allocation algorithm should be made such that semaphores are only used for shielding of the table after the initialization of the semaphores.

In the EMPS kernel, no semaphores are used, but instead context switching is disabled when the allocation table is accessed. This is done by disabling the context switch for interrupt handling by ISPs, as explained in section 2.2.3. When the allocation table for common memory is accessed, disabling interrupts is not sufficient, because the table is shared between all processors. In this case the TAS (Test And Set) instruction of the MC68030 is used to test a busy flag: a processor that wants to access the table performs TAS instructions on the flag until the allocation table is available. When the allocation table becomes available, the flag is cleared. The TAS instruction cannot be interrupted by another processor.

3.2 MMU

Most modern microprocessors, including the MC68030, provide an MMU that can be used to support multi-tasking facilities and to make the software independent of the memory locations in hardware. The MMU translates a virtual (also called logical) address used by the software into a physical address which is used to access the hardware. In this section, the function of an MMU in general is described. Details of the MMU in the MC68030 processor and the way it is used in the kernel can be found in the subsequent sections.

3.2.1 Address translation with the MMU

Translation of a virtual address into a physical address is the most basic function of an MMU. In the translation process, not all bits of the virtual address are used. Instead, the memory space is divided into pages, as is shown in figure 3.2. A page is the smallest block of memory for which a translation can be made. An address is split in a page number plus a subaddress within the page. The page number is translated through the MMU page table, whereas the subaddress within the page is the same for the virtual- and the physical address.

In the example of figure 3.2, the pagesize is 1 kbyte, so 10 bits are used as a subaddress within the page. There are 32 address bits, so 22 bits are used to define the page number, giving $2^{22} = 4194304$ possible page numbers. If this example would represent reality, a table would be needed with a size equal to the number of pages. Clearly, such a large table is not economical. Therefore, MMUs always have a way to limit the translation table size that is required.

The translation table can be built in such a way that more than one virtual address translates to the same physical address, because different table entries can point to the same physical page.

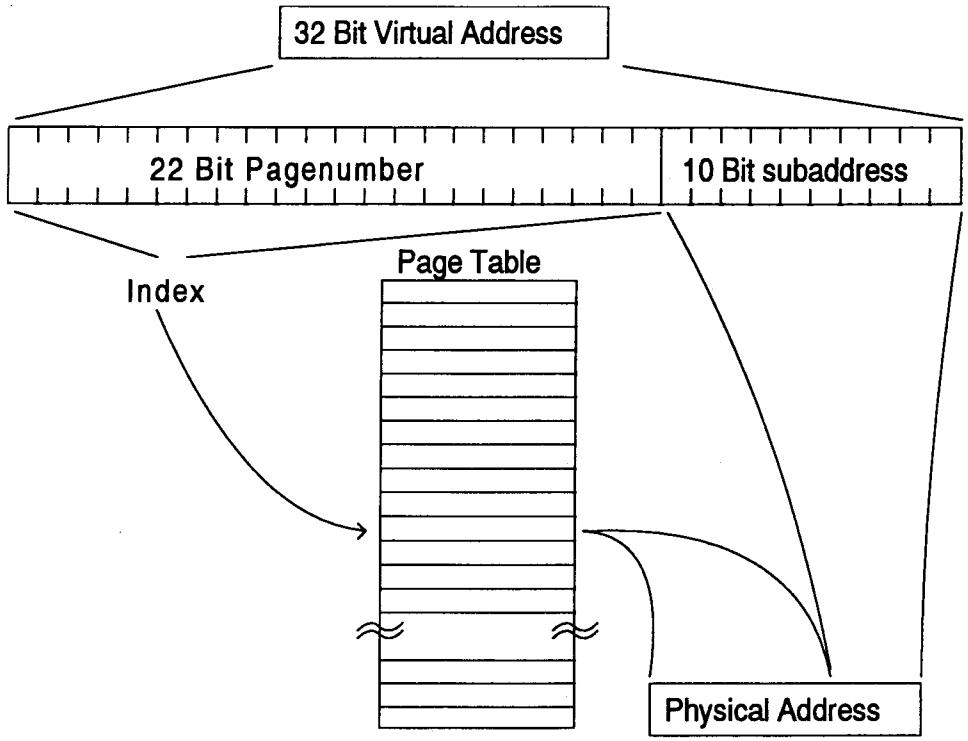

Figure 3.2: *The MMU uses a table to translate virtual addresses into physical addresses.*

Apart from translating addresses, an MMU can be used to protect memory areas from illegal access. For instance, it is possible to define a page to be read only, so that a write operation to a address within that page produces an exception which can be handled by the system software.

3.2.2 Use of the MMU in the kernel

The reasons for using the MMU in the EMPS system are:

- Efficiency of application programs. Every time an application program is loaded, it has the same virtual memory map, so absolute addressing can be used in the program code of the application. This is more efficient than position independent program code.
- Protection. The MMU can be used to prevent application programs from overwriting the address space of other programs.

In the EMPS system, a virtual memory map is created for each process. The memory map, which is shown in figure 3.3, consists of three parts:

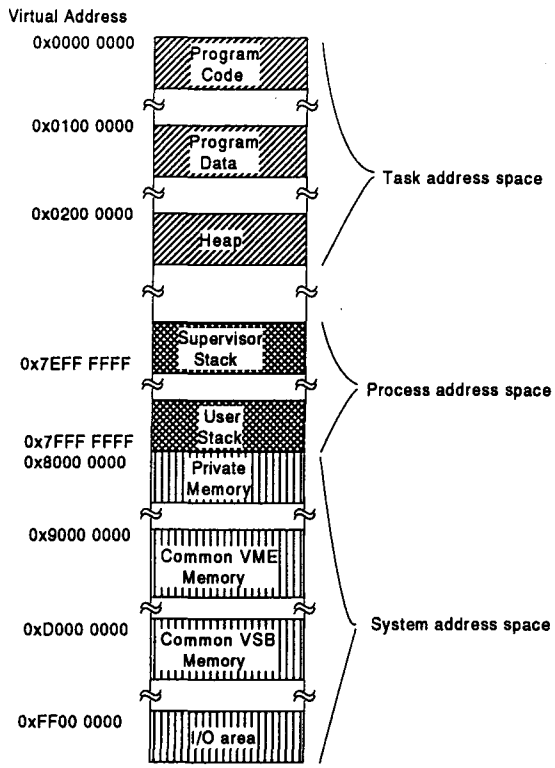

Figure 3.3: *A virtual memory map is created for each process. The process address space is owned by the process, whereas the task address space is shared by all processes in a task, and the system address space is shared by all processes on a processor.*

1. The system address space. This is the part of the memory that is shared between all processes running on one processor. In the system address space, all memory locations, plus all memory mapped I/O locations, can be reached. The program code and the data structures of the kernel are situated in this part of the virtual memory. In the kernel, it is necessary to emulate some MMU actions in software. For efficiency of this emulation, it is essential that a simple relation between a physical memory address and the corresponding system address exists. Therefore, the virtual memory map is created in such a way that the system address and the physical address differ by a constant offset.

   In the current kernel version, all processes have full access to most of the system address space. However, shielding of the system address space against write access by application programs is desirable. Protection of memory areas is discussed in section 3.5.4.

2. The task address space. A task consists of one or more related processes. The memory areas that are shared between all processes within a task are the code, data, and heap, which together form the task address space.

3. The process address space. Each process in a task has its own stack memory. A separate stack exists for user- and supervisor mode.

3.2.3 Process migration and memory management

Multiple processes within one task can use the data area of the task to exchange information. When one process in a task is migrated to another processor, this is no longer possible: the data area of the migrated process has become disjunct from the original data area.

Allowing only migration of processes that do not share the code, data, and heap areas with other processes (i.e. single process tasks) can solve this problem, but this limits migration possibilities. Instead, it is assumed that processes that are migrated only communicate via mailboxes. At the destination processor, a new task is created that contains a copy of the code, data, and stack areas. As mentioned in chapter 2, migration of heap memory is not supported. The migrated process is the only process that exists within the newly created task.

After a process has migrated, the stack memory that was used by the process at the source processor is released. If the migrated process was the only process in the task at the source processor, memory in the task address space (code and data areas) is also released.

3.3 The MMU of the MC68030

The MC68030 processor contains a very flexible Memory Management Unit [mot 90], which has many features that are not relevant for the kernel. Only the functional part of the MMU that is used will be described here.

For address translation, the MMU uses a set of tables that are organized in a tree structure: the Translation Table Tree. The tree is stored in the private memory of the processor module. An entry of the table is called a *descriptor*. In the kernel, the tree has 3 levels. The actual address translation can best be explained with an example.

Figure 3.4 shows how the virtual address 0x02001234 is translated using the translation table tree. In each level of the tree, a number of bits of the virtual address is used as an index for a table. In figure 3.4, the 2 most significant bits of the virtual address are used as an index of the table of the first level (A). Since both bits are 0, the first entry of this table is

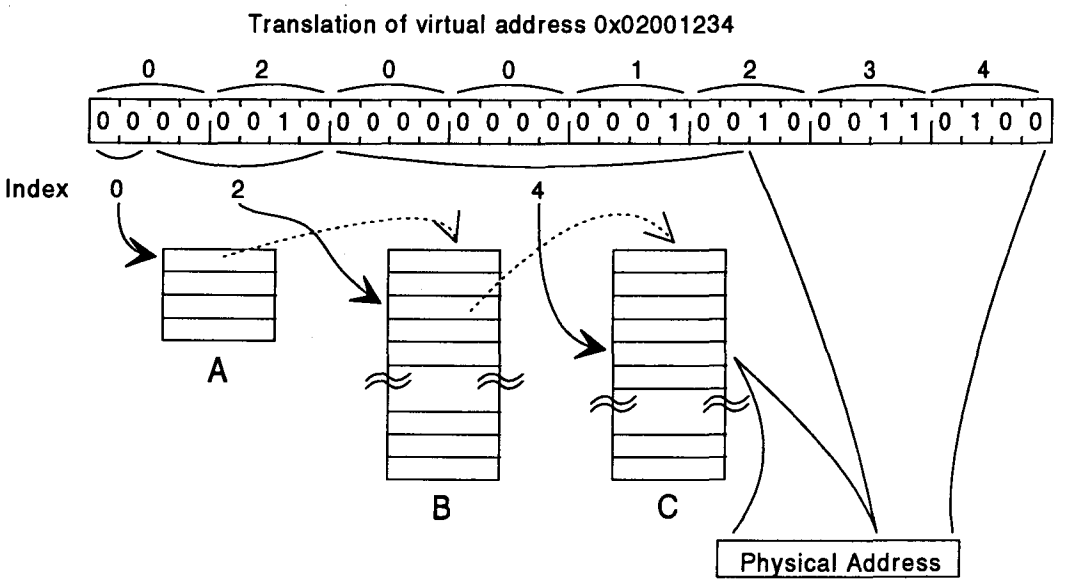

Figure 3.4: *A 3-level translation tree is used for address translation. In the first and second level, the table entries each point to a new table. In the last level, the table entries point to the physical page address.*

selected. The descriptor in this table entry contains the start address of the next table. Note that 4 of such tables can be selected. For the index of the selected second table (B), 6 bits (bit 24-29 of the virtual address) are used in this example. This bit field contains the number 2, so the start address of the table of the third level is obtained from entry 2. In principle 4×64 of such tables can be selected. The index of the selected table of this (last) level (C) of the translation tree is obtained from 14 address bits (bit 10-23 of the virtual address), which contain a 4. This table contains the physical page addresses, so in this example the address of the page is found in the table at position 4. The remaining address bits (bit 0-9) are the subaddress within the page, and are not used in the address translation. These address bits are the same for the virtual and physical address.

The start address of the first table in the tree is stored in an MMU register. The virtual memory map that is used, can be changed by making this MMU register point to another translation tree. This happens in the kernel when a context switch occurs, so each process has its own virtual memory map.

In the example of figure 3.4, bit fields of sizes 2, 6, and 14 were used to obtain an index of a table of level A, B, and C, respectively. The number of bits that are used for each index can be programmed when the MMU is initialized. When the MMU is initialized by the kernel, the sizes of the bit fields comply with the example.

The address translation process outlined above requires 3 accesses to tables in the translation tree for every address translation. If this tree search would be performed for each memory access, a lot of overhead would be caused by reading the tables. Therefore, the MMU contains an Address Translation Cache (ATC), which keeps the results of the last 22 address translations. When a memory address is used for which the translation is present in the ATC, no tree search is needed.

The descriptors in a translation table not only supply either the address of the next level table or the physical page address, but also contain other information items that are used in the translations process. The items that are used by the kernel are described next.

3.3.1 Table index limits

In section 3.2, it was mentioned that the table used for translation of virtual addresses into physical addresses could become unacceptably large. The fact that a tree of tables is used for address translation does not solve this problem: if the whole tree with all its branches would be present, the total number of table entries in the last level of the tree would be equal to the number of pages.

To reduce the size of each table in the tree, a limit can be placed on the index that is used for accessing the tree. Imposing a limit on the size of a table in the first or the second level of the tree means that the number of tables in the next level is also reduced. For this purpose, a *limit* field is present in every descriptor in the translation table. The limit field contains the extreme value that will be accepted as an index of the next level table in the tree. An index that falls outside the range of allowed values causes an exception which can be handled by the kernel.

The limit that is imposed on the index, can be either a lower limit or an upper limit. If it is a lower limit, the index of the next table level must be greater than or equal to the value in the limit field. If it is an upper limit, the index value must be less than or equal to the limit. Whether the limit is a lower or an upper limit, is determined by the L/U bit in the descriptor.

3.3.2 Descriptor type

From the description of the address translation process, it is clear that not all descriptors are the same: the descriptors in the first two levels of the translation tree contain the start address of the next table, whereas in the last level, the descriptors contain the physical page address which results from the translation. In the kernel, 3 types of descriptors are used:

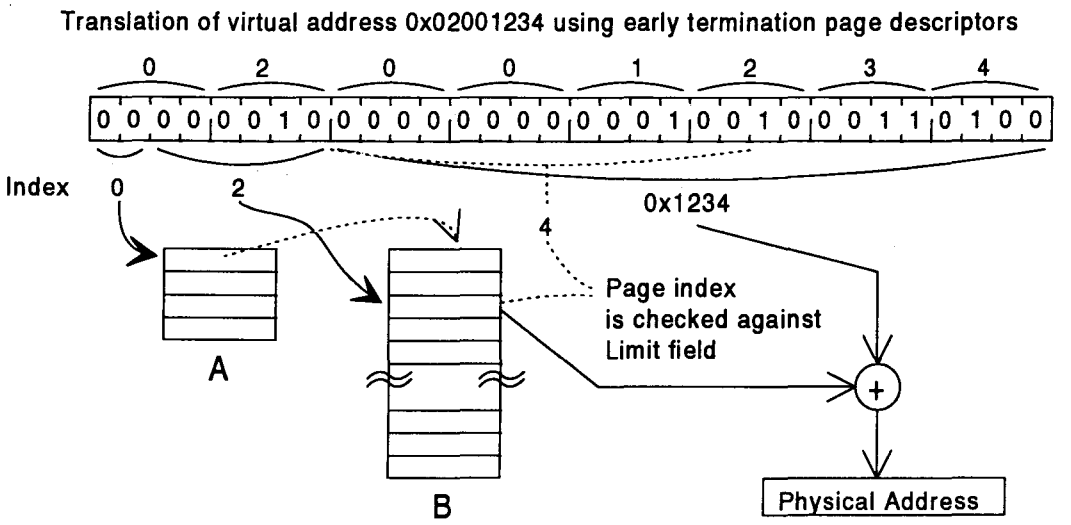

Figure 3.5: *An example of a translation tree which contains early termination page decriptors. The last level of the tree is omitted.*

1. INVALID DESCRIPTOR. When an invalid descriptor is encountered in the translation tree, an exception occurs. Invalid descriptors are used if the virtual memory map contains gaps for which no translation is available.

2. TABLE DESCRIPTOR. In the first two levels of the translation tree, the descriptors normally contain the address of another table. In that case, the descriptors are called table descriptors.

3. PAGE DESCRIPTOR. The last level of the tree contains descriptors that corresponds to the page address which results from the address translation and are called page descriptors.

When a page descriptor is encountered, the physical page address is obtained from the descriptor. The use of the page descriptor type is not restricted to the last level of the translation tree. It can also be used in the first two levels. If a descriptor in the first two levels of the translation tree is marked as a page descriptor, it is called an *early termination page descriptor*, because the tree search is terminated when this descriptor is found.

Early termination page descriptors are used for the translation of a contiguous range of pages. The limit field of an early termination page descriptor is used to check the index of the page, similar to table descriptors. In normal page descriptors, the limit field is ignored. An example of a translation tree in which an early termination page descriptor is used, is shown in figure 3.5.

The use of early termination page descriptors reduces the total number of descriptors in the translation tree, because such a descriptor replaces multiple normal page descriptors. Early termination page descriptors can only be used for a range of pages that are physically contiguous. Normal page descriptors provide a separate translation for each page, so that pages that are contiguous in the virtual memory map, do not have to be physically contiguous.

3.3.3 Supervisor only/Read only protection

The MMU offers protection of memory areas against illegal accesses. Memory areas can be protected against access in the user mode of the processor by setting the S bit in the descriptor, and against overwriting by setting the WP bit in a descriptor. If during the table search a descriptor is encountered in which the S bit or the WP bit does not comply with the access mode, an exception occurs.

3.4 The old memory management implementation

One of the reasons for using the MMU is to allow application programs to use absolute addressing. Therefore, the MMU address translation must be created in such a way that the virtual memory map of a process is always the same, independent of its physical memory location.

As described in the previous section, the address space of an application process basically consists of five areas: the code area, the static data area, the heap, the user stack and the supervisor stack. Each of these areas has its own MMU tables for address translation.

In the original version of the EMPS kernel, the MMU translation tables are stored in static array variables, and as a result they have the same size for every process. The arrays contain 64 entries for the code, 32 for data, 16 for heap, and 64 for both stack areas. Since the pagesize is 1 kbyte, the maximum size of each area in kbytes is equal to the number of entries. (Note that the number of entries in a table defines the *maximum* size of an area, but that in general a process will occupy less memory. The remaining entries are marked as *INVALID* descriptors by the kernel when a program is loaded.)

When the EMPS kernel is used as a basis for data acquisition in physics experiments, the limitation of the size of memory areas caused by the static size of MMU tables is unacceptable. In particular the heap, which is used to store the data acquired from the experiment, may grow to a size of several megabytes.

Increasing the size of the MMU tables may seem to be a solution. However, the amount of memory required for storing the tables becomes prob-

lematic when the table size is increased. In the present version of the EMPS kernel, the maximum number of processes is chosen as 50. Because the MMU translation tables are allocated statically, the total number of MMU descriptors in the last level of the translation tree is $50 * (64 + 32 + 16 + 64 + 64) = 12000$. Each descriptor occupies 8 bytes, so that 96000 bytes are used. If the number of page descriptors would be increased such that megabytes of memory can be accessed, the translation tables would become excessively large.

Another problem with the original implementation is that each process has its own MMU descriptors for code, data, and heap, whereas the memory areas for code, data, and heap are shared between all processes in one task. The descriptors of all these processes must therefore be kept consistent: they all contain a copy of the same information. Every time a new process is created in a task, all descriptors for code, data and heap must be copied, and every time the heap area is expanded, the heap descriptors of all processes in the task must be updated. Apart from the overhead that is created, this implementation also involves a large number of MMU tables.

3.5 The new memory management implementation

To avoid the problems of statically declared MMU tables, a new implementation of the memory management routines has been made. In this new implementation, MMU tables are allocated dynamically, and the translation tables of the task address space (the code, data, and heap sections) are shared between all processes in the task. Also, early termination page descriptors are used for address translation of contiguous memory areas.

The translation tree that is built by the new implementation, is presented in figure 3.6.

3.5.1 Shared translation tables for processes in one task

As already mentioned above, application processes that belong to the same task share memory areas for program code, static data, and heap. Therefore, the part of the translation tree that is used for these areas can also be shared.

In some cases, memory areas of an application will be accessed by external devices using DMA. In this case, memory accesses do not occur via the MMU, and the physical address of the memory area is used. Because the data that is transferred using DMA will often cross page boundaries, the memory area must be physically contiguous. Therefore, a contiguous range of pages is

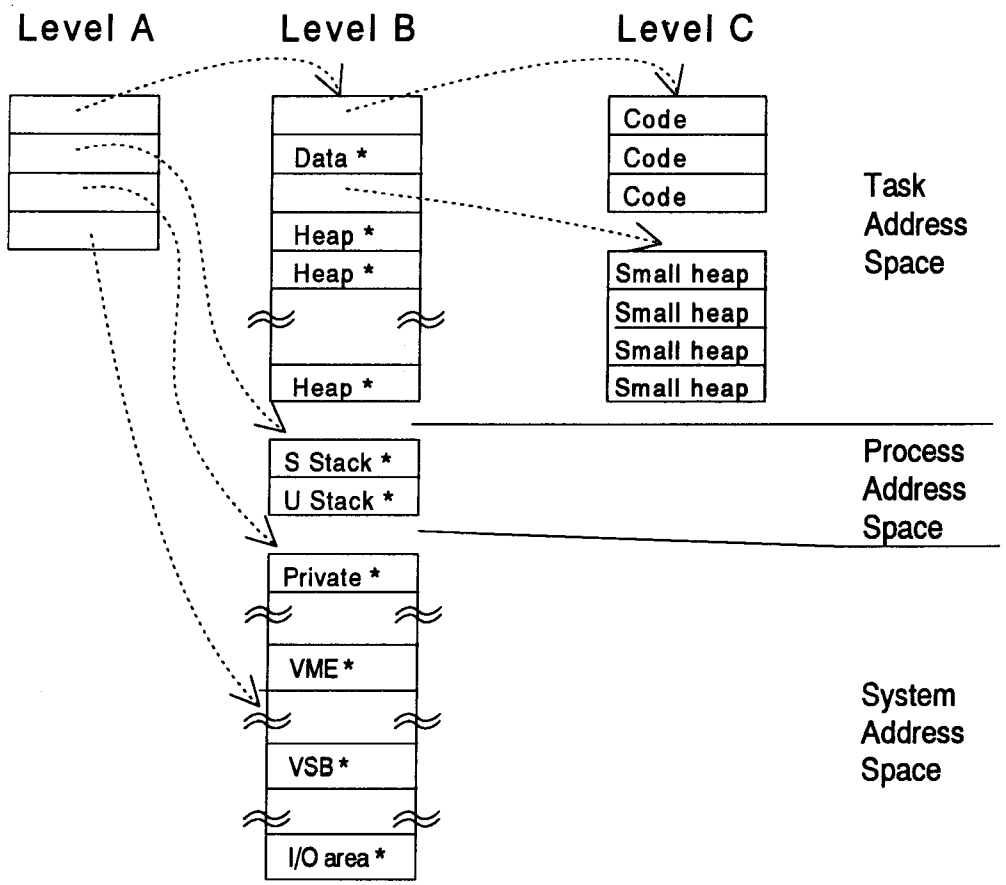

Figure 3.6: *In the address translation tree created by the new memory management routines, early termination page descriptors are used, and the descriptors for code, data, and heap are shared between all processes in a task. Early termination page descriptors are marked with a *.*

reserved in all cases where DMA is possible. These are the data, heap, and stack sections.

Contiguous memory areas are also needed when data structures of an application process are accessed by a kernel process. The kernel process is not part of the application, and cannot access the data structure at the same virtual address as the application process, because these processes have a different address translation tree. Therefore, the kernel process translates virtual addresses of the application process to virtual addresses in the system address space, through which all memory locations can be reached. Because the data structure may cross page boundaries, the pages of this structure must be physically contiguous.

3.5.2 Dynamic allocation of memory for translation tables

The memory that is used to store the task related part of the address translation tree is allocated dynamically when a new task is created on the system. Similarly, the process related part of the tree is allocated when a new process is created in a task. This way, the size of the translation tables can be adjusted to the requirements of the task and the process.

A new task can be created by loading an application program via the LAN. The size of the translation tables is derived from the file header, which contains the size of the code area and the size of the data area. A detailed description of the file header can be found in appendix H.

3.5.3 Early termination page descriptors

Most memory areas of an application program are required to be physically contiguous, as explained above. For these areas, the use of early termination page descriptors is advantageous, because the last level of the translation tree is omitted. Early termination page descriptors are used for the memory areas of data, heap, and stack.

The early termination page descriptors are located in the second level of the translation tree. In this level of the tree, the translation tables contain 64 descriptors (see section 3.3). The code area and data area each use 1 descriptor, so that 62 descriptors remain for the heap area. The heap area contains memory that is allocated dynamically when the application program is executing, and therefore the descriptors for the heap area must be written into the table when the application program allocates memory. If only early termination page descriptors would be used for the heap area, this would limit the number of memory blocks that could be allocated to 62 for each task, because only 62 descriptors are available for the heap area. This can be a problem for programs that allocate many small blocks of memory. To bypass this limitation, the implementation of the memory allocation functions also supports use of normal page descriptors. Normal page descriptors are located in the last level of the tree, in which the translation tables contain 16384 descriptors. The number of normal page descriptors that are required for a task, is a parameter of the kernel function that creates the task related part of the MMU tree.

3.5.4 Protection of memory areas

Restricting access to certain memory areas in application programs can protect programs from the consequences of errors in other programs. The MMU offers read-only and supervisor-only access restrictions for memory areas.

Combining these two options gives protection against write access in supervisor mode. It is not possible to protect a memory area against write access in user mode, while allowing write access in supervisor mode, except by prohibiting access of that memory area in user mode completely.

Because all memory locations can be accessed in the system address space, protection of this area against write access by application programs is desired. Application programs execute in the user mode of the processor, and therefore the system address space should be protected against write access in user mode. As explained above, the MMU doesn't offer this protection, but it can prohibit access to the system address space in user mode completely. However, in the current version of the kernel, it is assumed that application programs can read some parts of the system address space (e.g. when a message is sent via a mailbox). Completely prohibiting access to this area for application programs is therefore only possible by making substantial changes to the kernel. Because those parts of the system area which must be accessible to applications are all located in common memory, protection of the private memory part of the system address space is possible.

In the current kernel version, access restrictions have been implemented for the code area of application programs, which has read-only protection, and for the private memory part of the system area, which has supervisor-only protection.

3.6 Use of the MMU to divert VME memory access to the VSB bus

Communication between processors within one node uses common memory, as explained in chapter 1. In the EMPS system, processors are connected to common memory with two separate hardware buses, namely the VME bus and the VSB bus. All processors within one node can access common memory via the VME bus, which has 20 slots. Access via the VSB bus is only possible if the processor and the memory module belong to the same cluster, which is a group of 5 slots. Which bus is used, depends on the address that is used to access a location in common memory. Some memory locations in shared memory can thus be accessed at two different addresses, one for access via the VME bus, and the other for access via the VSB bus, as shown in figure 3.7.

The bus system of a multi-processor system often forms a communication bottleneck. The availability of two separate buses for access to common memory can help to avoid this. Access to shared memory should occur via the VSB bus if the processor module and the memory module belong to the same cluster, so that the VME bus is available for communication between

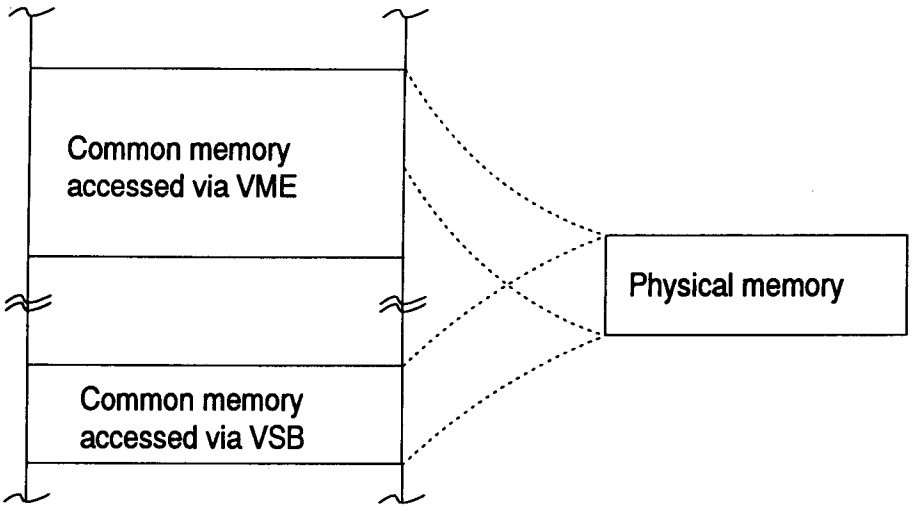

Figure 3.7: *Memory that is shared between processors within one cluster can be accessed at two different addresses, corresponding to the VSB and the VME bus, respectively.*

modules that do not belong to the same cluster. Even when the EMPS system is equipped with only one processor module (which is a configuration that is likely to be used for the control of many physics experiments), using the VSB bus for memory access by the processor can be advantageous when the VME bus is used for DMA data transfer.

To exploit the access hierarchy formed by the VME/VSB buses in software, additional actions would be required. Each time shared memory is accessed, a program would have to check whether the memory module belongs to the same cluster as the processor module. In that case, it would have to calculate the address at which the memory location can be reached via the VSB bus. This would increase the complexity of the software, and add significant overhead.

Using the MMU, communication via the VSB bus can be made completely transparent. The MMU translation tables can be filled in such a way that when a memory location is accessed with an address that corresponds to the VME bus, but refers to a memory module that belongs to the same cluster, the translation of that address causes the VSB bus to be used instead. Any access to shared memory can therefore be done using the virtual address that belongs to the VME memory area.

Creating MMU translation tables in such a way that a virtual address which normally corresponds to the VME bus translates to a physical address that causes access via the VSB bus is straightforward. At system start up, on every processor a table is generated that contains the VME address areas

of memory modules which can also be reached via the VSB bus. When an entry in the MMU translation table is filled, it is checked whether the physical destination address corresponds to a VME area that can also be reached through the VSB bus. If this is so, the corresponding VSB address is substituted.

Because for some applications the automatic use of the VSB bus may be undesirable, it has been made optional. At compile time of the kernel, generation of the extra code for transparent access via the VSB bus can be enabled with a `#define TRANSPARENT_VSB` statement.

Chapter 4

Performance tests

Since the EMPS system will be used for real-time experiment control and data acquisition, knowledge of the amount of time that various kernel functions take is essential when writing application programs. Another reason for determining the performance of kernel functions is to gain some insight in the overhead that is caused by processes running within the kernel. In this chapter, the performance of several kernel functions is discussed on the basis of test results. The tests that have been used give the average time for executing each kernel function. Worst case behavior is not determined for two reasons:

1. Various data acquisition interfaces that are used within the PhyDAS/ EMPS system contain a large internal buffer for the data that is measured, so the soft real-time constraints for controlling these interfaces are more important than worst case behavior.

2. It is very difficult to measure or calculate the worst case time, because this depends on many aspects, e.g. the number of wait states for accessing memory locations, the state of the MMU translation cache, and the instruction that is being executed at the time an interrupt occurs.

As mentioned in chapter 2, in the EMPS kernel interrupts are handled by Interrupt Service Processes. The overhead caused by using an Interrupt Service Process instead of using an Interrupt Service Routine is determined by comparing the two (section 4.2). This overhead can cause significant performance loss when interrupt activity is high, e.g. when writing output to the terminal (section 4.3).

The tests results in this chapter have been obtained with an MPS030 computer module with a clock frequency of 33 MHz.

4.1 Context switch time

Context switching can happen as a result of executing a kernel function, and hence the amount of time used by a kernel function depends on the context

switch time. In the EMPS kernel, the context switch time is influenced by the presence of a floating point coprocessor, because the registers of the coprocessor must be stored in memory at each context switch. It is therefore useful to measure the context switch time, and especially, to measure the extra amount of time that is needed when a coprocessor is used. When a coprocessor is present, at each context switch it is checked whether the coprocessor has been used by the process that is being deactivated and by the process that is activated. This information is obtained from the state register of the coprocessor. Only if the coprocessor was used, the register if the coprocessor are saved resp. restored.

For this test, the regular kernel functions in which a context switch is performed cannot be used, because an unknown amount of overhead is caused by executing instructions apart from the context switch. Therefore, in this test only the kernel function *cswitch*, which is the lowest level context switching function, is used. Application programs cannot call the function *cswitch* directly, so the test program for measuring the context switch time involved temporarily altering the kernel. In the program that was used for this test, a large number of context switches are executed between two processes that are created inside the kernel especially for this purpose, and the time used is derived from the number of clock interrupts.

The context switch time measured with this test amounts to $39\mu s$ if no floating point coprocessor is present. If both processes involved in the context switch use the coprocessor, the context switch time amounts to $72\mu s$, an increase of $33\mu s$ with respect to the test where no coprocessor is present. The MC68882 coprocessor that is used in the EMPS system has eight 80 bit floating point data registers, a 16 bit control register, a 32 bit status register, and a 32 bit instruction address register. At each context switch, these registers must be stored in memory for the process that is deactivated, and must be restored from memory for the process that is activated. This accounts for the extra time used.

In subsequent sections, measurements have only been made without the presence of a coprocessor. If a coprocessor is used, the results determined in these sections can be adapted by counting the number of context switches that occur, and adding $33\mu s$ for each context switch.

4.2 Overhead from Interrupt Service Processes

The use of Interrupt Service Processes instead of Interrupt Service Routines for interrupt handling clearly will have some effect on the time needed to service an interrupt. Figure 4.1 illustrates the timing of the various stages in

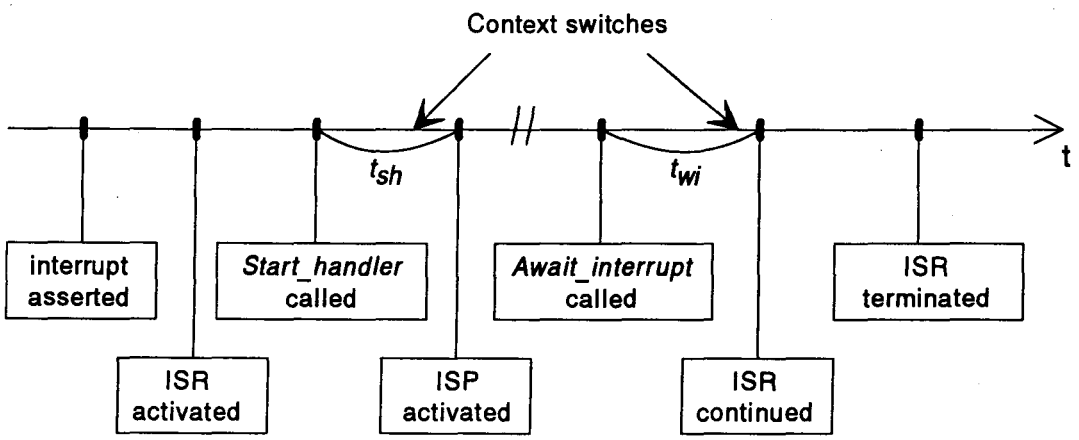

Figure 4.1: *The various steps involved when an interrupt is handled by an ISP.*

which an interrupt is handled.

The extra time used by handling interrupts in an ISP instead of an ISR is $t_{sh} + t_{wi}$. To measure the extra time, some temporary alterations have been made to the clock interrupt handler. Normally, the clock interrupt starts an ISR which increments the system time and then starts the clock ISP. The clock ISP checks whether a device timeout has occurred and handles processes in the *DELAY* state. For the measurements made in this test, the clock ISP has been altered such that no action is performed.

Two tests are used to find the extra amount of time used for the ISP. In both tests, an application program containing an empty waiting loop is used. The loop will take a fixed number of clock cycles. By counting the clock interrupts that occur while a certain number of loops is executing, the time is measured. In the first test, the ISP is altered in the way described above. In this case, the time measured is equal to the time which the processor uses to go through the empty loops, plus the total time for handling the clock interrupts. In the second test, calling the ISP is omitted by changing the ISR. Now, the time measured does not include the overhead for activating the ISP.

The two tests are performed with the clock interrupts occurring at different rates, so that the total time for handling the clock interrupt can be calculated for each test. This time is called t_{ISP} for the test where the interrupt is handled by an ISP, and in the test where calling of the ISP is omitted it is called t_{ISR}. Comparing these times for the two tests gives the amount of time used for calling the clock interrupt process:

$t_{sh} + t_{wi} = t_{ISP} - t_{ISR}$

t_{ISP} can be calculated from the test results with the clock interrupts

occurring at different rates. If the time between subsequent clock interrupts is called t_{clock1} resp. t_{clock2} and the number of clock interrupts occurring during the executions of the wait loops is n_{clock1} resp. n_{clock2}, the time elapsed during the test is $t_{clock1} \cdot n_{clock1}$ resp. $t_{clock2} \cdot n_{clock2}$. The difference between these two times, $t_{clock1} \cdot n_{clock1} - t_{clock2} \cdot n_{clock2}$, is the time used for handling for handling $n_{clock1} - n_{clock2}$ clock interrupts. The time needed to handle one clock interrupt is:

$$t_{ISP} = \frac{t_{clock1} \cdot n_{clock1} - t_{clock2} \cdot n_{clock2}}{n_{clock1} - n_{clock2}}$$

t_{ISR} is calculated similarly.

From these tests, t_{ISP} is found to be $275\mu s$ and t_{ISR} is found to be $11\mu s$. Use of ISPs instead of ISRs for interrupt handling causes an increase of $264\mu s$ in the time needed to handle an interrupt. Which consequences this has for the performance, depends on how frequently interrupts occur. For the interrupts associated with the clock and the terminal, the consequences are discussed in section 4.3. For interrupts from the PhyBUS, the interrupt rate depends on the types of interfaces that are used and on the experiment for which the interfaces are used, and therefore no general conclusions are possible about the performance loss. In considering whether to use an ISR instead of an ISP for handling certain interrupts, the advantage of location independent communication that is possible with ISPs is an important factor.

In the handling of an interrupt by an ISP, two context switches occur. The time needed for two context switches ($78\mu s$) is much less than the extra time needed to handle an interrupt with an ISP ($264\mu s$). This suggest that a significant amount of overhead is present in other routines.

4.2.1 Effects of the MMU

When a context switch occurs, the contents of the MMU Address Translation Cache are erased and must be refreshed each time a new process is started. In application programs that depend heavily on the use of the ATC, the context switch that happens after an ISP is responsible for ATC misses in the application program that is reactivated. The time needed to refill the ATC is therefore attributed to the ISP.

To find out how much effect intensive use of the ATC has on the amount of time used by the context switch that accompanies an ISP, an application program consisting of two test procedures is used. The test procedures only differ in the number of memory pages that are accessed. In both procedures, a loop is executed in which an array is accessed. In the first procedure, all array elements are located within the same page, whereas in the second loop, the array elements are located in 21 different pages. Together with the page from which program code is read, the first test procedure accesses 2 pages,

whereas the second test procedure accesses all 22 pages for which an address translation can be stored in the ATC.

Measuring the amount of time used by the ISP is done in exactly the same way as the determination of the overhead from ISPs outlined above. From the results of these tests, is found that the time used by the ISP is 275μs if only 2 pages are accessed by the test program, and 322μs if 22 pages are accessed. The 20 extra ATC misses that occur after each clock interrupt are responsible for the extra 47μs. In most programs it is unlikely that all 22 ATC entries will remain constant during execution of the whole, hence the effect of ATC misses due to an ISP will be less than 47μs.

4.3 Performance loss from ISPs

In this section, the performance loss from the ISPs that are active in the kernel is analyzed. The performance loss caused by two ISPs is measured:

1. The clock ISP; this process is activated at regular intervals and hence the performance loss is always present

2. The terminal output ISP; this is the process that is associated with the most frequently used device interrupt. It has been selected because the terminal output has a high interrupt rate.

4.3.1 The clock ISP

Normally, the only ISP in the kernel that is activated at regular intervals is the clock process. The clock process is used to re-activate processes in the *DELAY* state, and to start device timeout routines. To determine how much performance is lost due to this interrupt, only the situation that no other process is activated by the clock process is of importance.

The performance loss caused by the clock process is measured similarly to the method that was used to find the time for handling an ISP, as discussed in the previous section. Again, an application program consisting of a wait loop is used, and the time to execute a certain number of such loops is measured at different clock interrupt rates. The time used for service of the clock interrupt by the clock process is computed from the results.

From this test, the time used for the clock ISP is found to be 396μs. The clock rate that is normally used in the EMPS kernel corresponds to time intervals of 20ms between subsequent clock interrupts, and hence the performance loss due to the clock ISP is about 2%.

These results suggest that it is unlikely that the performance loss due to the clock ISP will play an important role for most applications.

4.3.2 The terminal output ISP

The terminal output ISP is activated when the serial output line, controlled by a MC68681 DUART [mot 85], is ready to transmit a character. This serial output line is connected to a terminal and operates at 9600 bits per second (which corresponds to nearly 1000 interrupts per second) in the current version of the kernel. In the near future, the speed of the terminal output will be increased to 19200 bits per second.

The DUART contains an output buffer for 2 characters, and generates an interrupt if the buffer is not full. The interrupt remains active until the buffer is full, and must be disabled when there is no data to be transmitted. In the original implementation of the kernel, the interrupt activates an ISP for each character to be transmitted. When text is being sent to the terminal, the terminal output interrupt is activated at a much higher frequency than the clock interrupt of which the performance loss was described in the previous section. It can be expected that the performance degradation during output to the terminal is also much higher.

To measure the performance loss when output is sent to the terminal, an application program containing two processes is used. One process contains an empty waiting loop which is executed twice: the first time while no output occurs, and a second time while output is generated by the other process. This second process continuously prints lines of 80 characters on the terminal after it has been activated, and has the highest priority of the two processes. The printing process is suspended from execution when it must wait until the terminal output is ready. The time that is used to execute the loop in the first process is measured both without printing and with printing, and the performance loss is computed from the result. From this test, it appears that the execution time of the program increases by a factor of 1.84 when output is being sent to the terminal. This corresponds to a performance loss of 46%. If the speed of the terminal output would be doubled, the performance loss would also roughly double. In that case, the performance loss during output to the terminal would become very severe.

The performance loss can be reduced by lowering the number of times the ISP is activated. To accomplish this, handling of terminal output of characters is moved from the ISP to the ISR. A buffer is used to store a whole line of characters, and the ISP is only activated when the buffer is empty. Because a line of output characters usually contains a number of characters, the ISP is activated less often. The terminal ISP is used to handle what can be called a logical interrupt, that indicates that a whole line of characters has been transmitted. In figure 4.2 the relation between the ISP and the ISR is shown.

By moving the handling of terminal output from the ISP to the ISR, the latter will be expanded. This expansion is only small, because reading

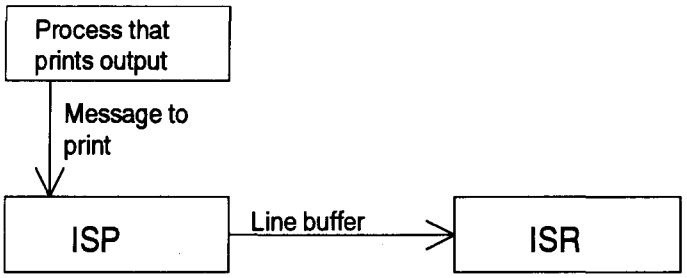

Figure 4.2: *Output for the terminal is sent in a message to the terminal output ISP. The ISP transfers the message to the ISR, and wait until the transmission is completed.*

characters from a buffer and sending them to the DUART is a very simple action. By altering the terminal output in this way, all of the advantages of using ISPs are still valid.

The performance test described above is used to test the results of the changes. The result indicates that with the new implementation, the execution time of the test program increases only by a factor of about 1.03 times as long when output is sent to the terminal, and hence the performance loss is reduced to 3%. Of course, the present method that is used to reduce performance loss from the terminal output can also be applied to other devices which allow a number of I/O operations to be buffered.

4.4 Semaphores

In the EPEP interpreter, semaphores can be used for shielding of resources that are shared between processes. The EPEP semaphores will somehow be mapped to semaphores as provided by the EMPS kernel. The semaphores from the kernel will also be used to activate EPEP processes that handle a PhyBUS interrupt. Therefore, the time used for signal-wait operations on semaphores in the kernel directly effects the time used by the corresponding EPEP routines.

Two types of semaphores are available in the kernel: local semaphores, which can only be used by processes running on the same processor, and common semaphores, which can be accessed from all processors in a node. Of both semaphore types, the time used for a pair of signal-wait operations has been measured. Two situations are distinguished: those in which the signal-wait operations cause a context switch, and those in which the signal wait operations do not cause a context switch. The test results are shown in table 4.1.

	Local semaphore	Common semaphore
with context switch	217μs	675 μs
without context switch	56 μs	78μs

Table 4.1: *The time used by a pair of signal-wait operations on local- and common semaphores.*

The results of the semaphore tests show that the local semaphores are more than 3 times faster than the common semaphores if the wait-signal operations cause a context switch. In the implementation of the EPEP interpreter, semaphores are used to signal that a PhyBUS interrupt has occurred. Using common semaphores for this purpose would allow handling of a PhyBUS interrupt by any processor in a node, but has the disadvantage of increased response time. Because in practical use a single processor system is a likely configuration of the EMPS when it is used for experiment control, local semaphores are used to signal PhyBUS interrupts. Even if more than one processor is available in the EMPS, handling all the PhyBUS interrupts by a single processor is the most practical solution.

Chapter 5

Conclusions and suggestions

In the EMPS kernel, multi-processor services consist of mailboxes, which offer location transparent communication, and process migration, which can be used to balance the processing load over all processors in the system. These services are related in the sense that location transparency of mailboxes is exploited when processes are migrated. All necessary reconfiguration of the communication is done inside the kernel. A restriction to process migration is that the migrated process should not use heap memory. The migration facility might be made more useful by removing this restriction.

Interrupts are handled by dedicated processes, instead of interrupt service routines. This method of interrupt handling offers location independent access to I/O devices. One of the advantages of using processes to handle interrupts is that stack memory required for interrupts is minimized. Another way to reduce the stack memory that is needed, is using the interrupt stack pointer provided by the MC68030. Whether use of this stack pointer is advantageous should be examined in more detail.

The memory management in the EMPS kernel uses a Memory Management Unit to create a virtual address space for each process in the system. In the original kernel, hard limits are placed on the amount of memory that can be used by each application program, because MMU translation tables of a fixed size are used. This restriction prohibits the use of the kernel for applications that involve large data structures. Therefore, a new implementation of the memory management was made, in which the tables are allocated dynamically when a new task is created. In this new implementation, the early termination page descriptors provided by the MMU are exploited to reduce the size of the tables. As a result of these improvements, the amount of memory that can be used by application programs is -in practical terms- only limited by the physically available memory.

For memory allocation, an array of free- and occupied pages is maintained. This array is shielded against simultaneous access by different processes, by disabling interrupts during access to the array. A consequence of this approach is that task switching is temporarily disabled. This situation can be improved by using semaphores for shielding. When semaphores are used, context switching is not disabled. It can also be advantageous to replace the

current allocation routine by another routine, which is based on a linked list of occupied memory areas. Linked list allocation routines are generally faster then routines that use an array of pages.

Performance tests show that the extra amount of time needed to handle an interrupt by an Interrupt Service Process instead of by an Interrupt Service Routine is more than 260μs. This means that a severe performance loss can be caused by interrupts that occur at a high rate. For one interrupt source, the terminal output device, an improvement is implemented by using a buffer in the ISR, and activating the ISP only when the buffer needs updating. This method can also be useful for some other interrupt sources.

Two projects, which will use the EMPS kernel as a basis, are the development of the EPEP interpreter for control of physics experiments, and the development of a dependable distributed operating system. Most of the improvements that were made with respect to the original version of the kernel are useful in both areas. The EPEP interpreter is application of the kernel that is of interest in the Department of Physics. At this moment, the development of a C-language version of the EPEP interpreter is nearing its completion. In preliminary tests, the EMPS has successfully been used as a basis for EPEP.

Not all improvements and extensions that were made to the original version of the kernel have been discussed in this report. A summary of these changes is given in appendix I.

Bibliography

[mot 90] Motorola, MC68030 Enhanced 32 bit Microprocessor User's Manual, rev.2, 1990.

[mot 92] Motorola, Programmers Reference Manual, rev.1, 1992.

[dij 93] G.J.W. van Dijk, The Design of the EMPS Multi-processor Executive for Distributed Computing, 1993.

[oas 93] Oasys/Green Hills 68K, Cross Development Guide, version 1.8.6, 1993.

[tan 87] A.S. Tanenbaum, Operating Systems, Design and Implementation,1987.

[mot 85] Motorola, MC68681 Dual Asynchronous Receiver/Transmitter, 1985.

Appendix A

Command interpreter

The command interpreter is a process within the kernel that executes commands entered by a terminal connected to one of the processor modules. With the command *SET_PROC* the processor module that executes the command interpreter can be changed. The command *SET_OUTPUT* changes the processor module to which the terminal output is send.

The commands recognized by the command interpreter are:.

- *BATCH filename:* execute batch file *filename*. The batch file may contain any valid command.
- *BROADCAST filename /q:* load Motorola "S" or "Q" file *filename* at all processors. Qualifier *q* is the type of file to be loaded, "S" for a Motorola "S" file (default), or "Q" for a "Q" file. This command is used **only** at system startup, to load the kernel software at all processors in the system.
- *DEV devicenr:* display information about device *devicenr*: the device name, the first process in the device queue, the device count (this is the number of interrupts that the device has generated), timeout flag, timeout time, the address of the routine that is called when a timeout occurs, and a list of device registers.
- *DIR name:* display the contents of directory *name*.
- *DOWNLOAD filename AT procnr/q:* load Motorola "S" or "Q" file *filename* at processor *procnr*. Qualifier *q* is the type of file to be loaded, "S" for a Motorola "S" file (default), or "Q" for a "Q" file. This command is the same as the *BROADCAST* command, except that the file is only loaded at the specified processor.
- *DUMP start end:* display memory contents from *start* to *end*.
- *HELP:* display all valid commands.
- *KILL pid:* terminate process *pid* and release the process memory.

- *LOAD filename [AT procnr]:* load a new program with name *filename* from the file-server. The process is loaded at processor *procnr* (default current processor).
- *MIGRATE pid procnr:* migrate process *pid* to processor module *procnr.*
- *PD pid:* display information about process *pid.* This command shows the process name, process state (ready, blocked, ...), additional information depending on the process state, the queue in which the process is, the memory used by the process, and the mailbox connections of the process.
- *REGS pid:* show the CPU register contents of process *pid*
- *RESTART procnr:* restart processor *procnr* after a hardware system reset.
- *SET_OUTPUT procnr:* redirect all terminal output to processor module *procnr.*
- *SET_PROC procnr:* change processor that executes the commands that are entered via the keyboard to *procnr.*
- *SHOW_ALL:* display the process table. Of each process (on the current processor), the pid, process name, priority, and process state are displayed.
- *SHOW_CHANNELS:* display the status of file-server channels.
- *SHOW_CLOCKQ:* display the processes and their wake-up time in the clock queue of the current processor.
- *SHOW_CSW:* display the number of context switches that have occurred on the current processor.
- *SHOW_DEV:* display the device table. Of each device, the device number, the device name, the waiting process, and the number of times the device has been used is displayed.
- *SHOW_ERRORS:* display the number of VME bus errors.
- *SHOW_MAILBOXES:* display mailbox information. Of each mailbox, the pid of senders, the pid of receivers, and the mailbox queue is showed.
- *SHOW_MEM:* Display a list of free memory pages and the number of free pages.

- *SHOW_MSG:* display the number of messages that have been sent from the current processor.
- *SHOW_READY:* Display all processes that are in the READY state. The processes are listed in the order in which they will be executed. Of each process, pid, priority, and name are displayed.
- *SHOW_SEM:* display the semaphore table. Of each local semaphore, the name, count, and the queue of waiting processes is shown. Of each common semaphore, the count, head, tail, a variable (listfull), name, and a list of waiting processes is shown.
- *START pid:* put process *pid* in the *READY* queue.
- *TIME:* display the time since system start-up.
- *TRANSLATE pid viradr:* translate the virtual address *viradr* of process *pid* into a physical address.

Additionally, any command that is not recognized is interpreted as a name of an EMPS program file. If the program file can be loaded, execution of the first process declared in the file is started immediately and input from the terminal is redirected to the newly created process. The <control> Z key can be used to connect the terminal input to the command interpreter again.

Appendix B

Memory maps

Two memory maps are of importance in the kernel:

1. The physical memory map, which lists the addresses at which memory locations can be reached physically. The addresses listed here appear on the address bus of the system, and are used by all hardware devices that access memory directly.

2. The virtual memory map, which list the addresses that are used by program instructions. These addresses are translated by the MMU to a physical address.

The physical memory map is represented in figure B.1.

The virtual memory map that is created by programming the MMU, consists of a system area which is only used within the kernel, and other memory areas which are used for application programs and process stacks, as described in chapter 3. The system area occupies virtual addresses 0x80000000 and higher, whereas the address range below 0x80000000 is used for the other areas. In the system area, all physical memory- and I/O addresses can be reached. A simple relation exists between physical memory addresses and virtual memory addresses in this area: they differ only by an offset. The offset has a different value for private memory and common memory. Two different offsets are necessary because all physical memory addresses must fit in the system area. The system area is shown in figure B.2.

Common memory can be accessed via the VSB and the VME bus. Which addresses are used to access common memory locations, depends on the slot of the memory module. Table B.1 lists the addresses that corresponds to each slot.

The addresses of the VME communication register of a processor module also depends on the slot of the module, and are listed in the table. From this table it follows that an area of 32 Mbyte is available for each memory module. In principle, the memory modules can contain 64 Mbyte of memory, and in order to address the second 32 Mbyte, a feature of the VME bus is used. The VME bus contains 6 so called Address Modifier (AM) lines. These lines are used to signal special VME bus accesses, such as access to modules

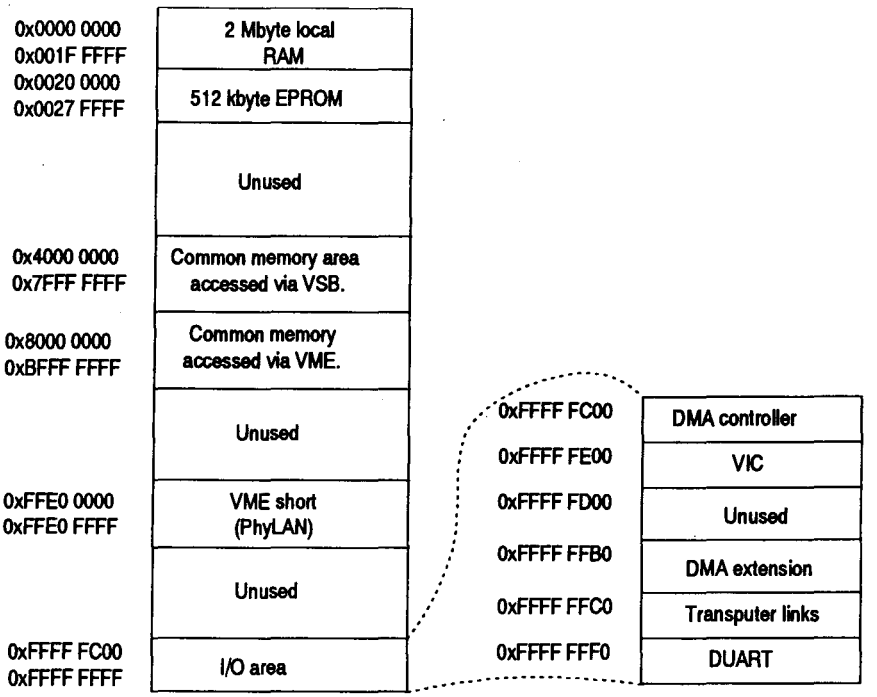

Figure B.1: *The physical memory map.*

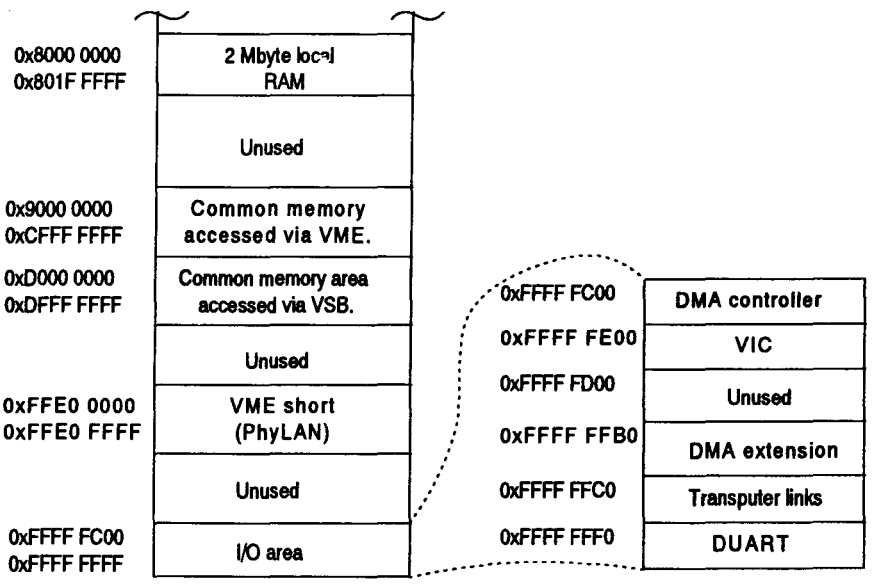

Figure B.2: *The virtual memory map of the system area.*

Slot	VME address	VSB address	Com. Reg. address
0	0x8000 0000	0x4000 0000	0xFFFE F07C
1	0x8200 0000	0x4200 0000	0xFFFE F0FC
2	0x8400 0000	0x4400 0000	0xFFFE F17C
3	0x8600 0000	0x4600 0000	0xFFFE F1FC
4	0x8800 0000	0x4800 0000	0xFFFE F27C
5	0x9000 0000	0x4000 0000	0xFFFE F47C
6	0x9200 0000	0x4200 0000	0xFFFE F4FC
7	0x9400 0000	0x4400 0000	0xFFFE F57C
8	0x9600 0000	0x4600 0000	0xFFFE F5FC
9	0x9800 0000	0x4800 0000	0xFFFE F67C
10	0xA000 0000	0x4000 0000	0xFFFE F87C
11	0xA200 0000	0x4200 0000	0xFFFE F8FC
12	0xA400 0000	0x4400 0000	0xFFFE F97C
13	0xA600 0000	0x4600 0000	0xFFFE F9FC
14	0xA800 0000	0x4800 0000	0xFFFE FA7C
15	0xB000 0000	0x4000 0000	0xFFFE FC7C
16	0xB200 0000	0x4200 0000	0xFFFE FCFC
17	0xB400 0000	0x4400 0000	0xFFFE FD7C
18	0xB600 0000	0x4600 0000	0xFFFE FDFC
19	0xB800 0000	0x4800 0000	0xFFFE FE7C

Table B.1: *Physical addresses of common memory areas and communication registers for each slot number.*

that only use 16 address lines (VME short AM). One of the AM codes is used to access the second half of the memory in 64 Mbyte modules. Generating AM signals to access this part of memory is not implemented in the kernel. It is doubtful whether a transparent access method can be created using the AM signals.

Appendix C

Initialization

During initialization of the kernel, actions taken by the processor module in slot 0 (the system controller) are different from those taken by the other processor modules. In particular, data structures in common memory which are shared by all processors are allocated and initialized by the processor in slot 0. Therefore it is necessary that initializations performed by processor 0 are finished before the kernel is started at other processors. The PhyLAN interface is also controlled exclusively by the system controller. When the system is booted, the kernel is first loaded via PhyLAN at the processor module in slot 0. The other processors in the system can then be started by issuing the command "broadcast mps:emps.q".

In the startup sequence described above, the kernel starts executing at processor module 0 before it can be started at other processors. The initializations that must be performed by processor module 0 are therefore always completed before other processors are used. In future versions of the kernel, another startup method will be used, in which the kernel is loaded at all processors at once. In that case, it must be arranged explicitly that initializations are performed by the system controller before other processors are activated (i.e. by sending a message to all processors using the communication registers).

Initialization can be split in two parts: the initialization that is performed before the kernel is loaded, and the initialization performed in the kernel. Before the kernel is loaded, the MMU is programmed such that private memory is mapped to address 0x80000000 and higher. This is necessary because the kernel code is compiled for execution in this memory area. At processor 0, the mapping is created by running the program "init" whereas at other processors, it is automatically done at start-up by the boot EPROM (slave monitor). After programming the MMU, the init routine running at processor 0 loads the kernel and starts execution of the kernel at address 0x80010000.

The rest of the initialization phase occurs inside the kernel. The following actions are performed:

1. The assembly routine in *load.s* is executed:

- The static variables of the kernel are filled with 0.
- The Supervisor Stack Pointer (SSP) is filled with virtual address 0x8000FFA8.
- The Vector Base Register (VBR) is filled with virtual address 0x80000000.
- The vector for TRAP #13 is initialized to point to the trap13 handler (in *empscall.s*). This trap (software interrupt) is used when an application program calls a kernel routine.
- The vectors for TRAP #14 and TRAP #15 are initialized to point to the trap14 and trap15 handlers (in *enable.s*). Trap 15 is used to switch to supervisor mode, trap 14 tests whether the processor is in user mode or in supervisor mode.
- The User Stack Pointer (USP) is filled with 0x8000F000.
- The Status Register (SR) is filled with 0x2000, which means that execution is continued in supervisor mode at processor priority 0.
- Initialization is continued by calling the routine *main* in *initmps.c*.

2. *initmps.c*

- The version of the EMPS kernel is printed.
- The number of the processor within the node is read from memory location 0x4 (this number is written here by *init.c* for processor 0 or by *slavemon.c* for the other processors).
- If the processor number is 0, the node number is read from the variable PhyLANFlags.ownsta (PhyLANFlags is initialized by the monitor).
- The variables *pid_NULL*, *currpid* and *currprio* are initialized.
- *SYSTABLEPTR* (at fixed memory location 0x80000004) is initialized. *SYSTABLEPTR is* used when a kernel function is called from an application.
- Memory management is initialized. See the chapter 3 for a full description.
- Data structures are initialized (Semaphores, message buffers, process descriptors, etc.).
- A number of system processes are initialized and started.

Appendix D

Kernel services for application programs

D.1 Calling of kernel functions by applications

Using the EMPS kernel, application programs normally run in user mode, whereas the kernel always works in supervisor mode. Therefore, a switch from user mode to supervisor mode is necessary when an application calls a kernel service. This switch is achieved by executing a TRAP instruction, which generates a software interrupt. The software interrupt is handled inside the kernel.

Only 16 different TRAP interrupts can be generated by the MC68030 CPU, which is not sufficient for all kernel functions. Therefore, each kernel function has been given a number that is put in register D0 when an application issues a kernel call, and one software interrupt is used (TRAP #13). The interrupt handler of this TRAP instruction uses the value in register D0 to determine which function must be called. The value in D0 is used as an index for a table that contains the start addresses of all the kernel functions that are accessible for applications.

Arguments for the kernel routines are written to the user stack by the application program. Because the kernel executes in supervisor mode, the kernel routines expect the arguments on the supervisor stack. The arguments must therefore be copied from the user stack to the supervisor stack, which is done by the TRAP interrupt handler. The TRAP interrupt handler always copies 10 arguments, irrespective of the number of arguments that the function actually expects. This method can be used because all functions have less than 10 arguments (with the possible exception of functions that have a variable number of arguments, such as *printf*).

Access to the kernel functions is provided through a table that contains the start addresses of each function. This table is located in private memory, and starts at the address pointed to by the pointer *SYSTABLEPTR*. The pointer *SYSTABLEPTR* is located at a fixed memory location (at virtual

memory address 0x80000004).

The functions of the kernel that can be called by an application program, are accessible by linking *SDTEMPS.OBJ* with the application program. This file contains the code that puts the function number in register D0, copies the stack pointer in A0, and executes the TRAP instruction. The program code in this file must start at address 0. The reason for this is that when a process has finished, it jumps to address 0, where code should be present that leads to leads to the correct termination of the process. *STDEMPS.OBJ* starts with code that terminates the current process, and therefore it must be the first file in the link command.

Not all functions provided by including *STDEMPS.OBJ* are appropriate for use in application programs. Some low level kernel functions, that should only be used in system software, can also be called. These functions are only available for testing of extensions or changes of the kernel, and in a future version of the kernel they will possibly be removed from *STDEMPS.OBJ.*

D.2 ANSI C functions

Only the most essential functions from the standard libraries are provided by the kernel. Other ANSI C functions are available from the Oasys libraries.

Definition *SLONG open(SBYTE *filename, SLONG mode)*

Description The standard C *open* function.

Definition *SLONG creat(SBYTE *filename, SLONG prot)*

Description The standard C *creat* function. Parameter *prot* is not used.

Definition *SLONG read(SLONG fno, SBYTE *buf, SLONG n)*

Description The standard C *read* function. Due to limitations of the PhyLAN file-server, the return value (number of bytes read) is always a multiple of the PhyLAN block size (512 bytes) when the end of file is reached during the read operation. Also, if the file size is not a multiple of 512 bytes, the file is expanded to fill an integer number of blocks.

Definition *SLONG write(SLONG fno, SBYTE *buf, SLONG size)*

Description The standard C *write* function. Due to limitations of the PhyLAN file-server, the size of the file that is written is always a multiple of the PhyLAN block size (512 bytes).

Definition *SLONG close(SLONG fno)*

Description The standard C *close* function.

Definition *SLONG lseek(SLONG fno, SLONG offset, SLONG end)*

Description The standard C *lseek* function. Only moving the file position pointer to *offset* (*end* = 0) and increasing the file position pointer by *offset* (*end* = 1) are supported.

Definition *void *malloc(SLONG size)*

Description Described in appendix E.

Definition *void free(void *ptr)*

Description Described in appendix E.

Definition *void realloc(void *ptr, SLONG newsize)*

Description Described in appendix E.

Definition *void calloc(SLONG number, SLONG size_each)*

Description Described in appendix E.

Definition *void malloc_type(SLONG size)*

Description Described in appendix E.

Definition *void putchar(SBYTE c)*

Description Write character *c* to the terminal. This function differs from the ANSI definition because no return value is given, and because output is sent directly to the terminal, and not to *STDOUT*. Unlike other terminal output functions, this function is not interrupt controlled. Conflicts may occur if putchar is used together with other terminal output functions (such as *printf*).

Definition *void printf(SBYTE str, ...)*

Description Send formatted text to *STDOUT* (the terminal). Because at present, only 10 arguments can be passed to a kernel routine, the number of items that can be printed is limited. Another difference from the ANSI standard is that no return value is generated. Printing of floating point numbers is not yet supported.

D.3 Mailbox functions

Definition *STATUS CreateMailBox(SBYTE *key, MEMTYPE memtype, SLONG procnr)*

Description Create a mailbox data structure at processor *procnr*. Mailboxes are used for location transparent communication between processes and are described in chapter 2.

Definition *STATUS RemoveMailBox(SBYTE *key)*

Description Remove the mailbox with identifier *key*.

Definition *STATUS Connect(PORT *port, SBYTE *Key)*

Description Before a process can exchange information via a mailbox, a port must be connected to the mailbox. The *Connect* routine establishes a connection with the mailbox that is identified by *Key*. The port data structure is used to access the mailbox in the routines *SendToPort* and *ReceiveFromPort*. Three different port types exist:

1. Sender port. A sender port type provides a non blocking send-receive service. This port type establishes a communication path in which the *SendToPort* and *ReceiveFromPort* routines are similar to the (non location transparent) *Send* and *Receive routines* (which are described in section D.8).
2. Client port. For implementing a (blocking) Remote Procedure Call (RPC), the client- and server port types can be used. When a message is sent to a client port with the *SendToPort* routine, the current process blocks, and the message is transferred to a server process that is connected to the mailbox via a server port. After the server process completes the RPC, it returns a message, and the client process unblocks. Because a client port can only be used to invoke a service, calling *ReceiveFromPort* on a client port is invalid. For a client process, the *SendToPort* routine is similar to the (non location transparent) *SendRequest* routine.
3. Server port. A server port is connected to a server process which services a RPC. The server process receives its parameters via a call to *ReceiveFromPort*. When the server is finished, it sends a reply message with *SendToPort*. For a server process, the *ReceiveFromPort* and *SendToPort* routines are similar to the (non location transparent) *ReceiveRequest* and *SendReply* routines.

Before *Connect* is called, the *PortType* field of *port* must be filled, to indicate the port type that is required:

port.PortType = *SENDER_PORT*, *port.PortType* = *CLIENT_PORT* or *port.PortType* = *SERVER_PORT*.

All other fields in the port data structure are initialized by the *Connect* routine.

Definition *STATUS DisConnect(PORT *port)*

Description *DisConnect* removes the connection between a port and a mailbox (e.g. before a process terminates).

Definition *STATUS SendToPort(PORT *port, MSG *message)*

Description *SendToPort* sends a message to the mailbox that is connected to *port*. The port type (see *Connect*) determines what message is sent:

- Sender port. For a sender port, the information in the message buffer of the sending process (*proctable[currpid]->CMmsg)* is sent, and not the information pointed to by the *message* parameter. The *message* parameter is ignored for this port type.
- Client port. For a client port, the information pointed to by the *message* parameter is sent to the mailbox. After transmission, the sending process blocks until a reply is received.
- Server port. For a server port, the information pointed to by the *message* parameter is sent to the mailbox, which routes it to the client process. The client process then unblocks.

Definition *STATUS ReceiveFromPort(PORT *port, MSG **message)*

Description By *ReceiveFromPort* the current process receives information from the mailbox connected to *port*. The port type (see *Connect*) determines where the received information is placed:

- Sender port. For a sender port, the information received with *ReceiveFromPort* is placed in the message buffer of the receiving process (*proctable[currpid]->CMmsg)*. Actually, the buffer pointers of the sending and receiving process are exchanged, so that the sending process receives the buffer of the receiving process.
- Server port. For a server port, the pointer **message* is filled with the address of the message that is received.

For a client port, *ReceiveFromPort* is not a valid operation.

D.4 Location independent process management routines

The routines described here can be used to create new processes within a task, start processes, and terminate processes. Location independent process management routines can act on any process running on any processor in the EMPS system. The functions are executed by the process-server process which gets its commands through a mailbox. When one of these routines is called, a mailbox message is sent to the server process, which performs the requested action.

Definition *STATUS CreateProcess(SLONG *pid, void (*func)(), SBYTE pname[], SLONG prio, LONG usz, LONG ssz, MEMTYPE ust, MEMTYPE sst, SLONG ProcNr)*

Description *CreateProcess* initializes a new process by sending a message to the process server.

The process identifier is returned in **pid.*

func is the start address of the function that is called when the process is started.

pname is the name of the process.

The process starts execution at priority *prio.*

usz and *ssz* are the user- and supervisor stack sizes. *ust* and *sst* are the user- and supervisor stack memory types, either *PRIVATE*, *COMMON_VME* or *COMMON_VSB*. For application processes, *ust* and *sst* should always be *PRIVATE.*

ProcNr is the processor number where the process is created.

Definition *STATUS StartProcess(SLONG pid)*

Description *StartProcess* starts execution of process *pid* by sending a message to the process server. If the priority of process *pid* is higher than the current process, then process *pid* starts executing immediately. Otherwise, process *pid* is put in the ready queue.

Definition *STATUS KillProcess(SLONG pid)*

Description *KillProcess* terminates process *pid* by sending a message to the process server. The process is dequeued from the queue it is in, and the memory used by the process is released.

Definition *STATUS GetProcessIdentifier(SLONG *pid, SBYTE *pname, SWORD pn)*

Description *GetProcessIdentifier* finds the process identifier *pid* of the process with name *pname.* The function looks for the process on processor *pn.* The process identifier is returned in *pid.*

If a process with name *pname* does not exist, the value *NoSuchProcess* is returned.

D.5 Other process related functions

Definition *SYSCALL delay_process(SLONG n)*

Description The routine *delay_process* delays the execution of the current process for *n* clock ticks. To this end, the process is inserted in the clock queue, the state of the process is changed into *DELAY*, and the process blocks.

Definition *SLONG LoadFile(SBYTE *file_name, WORD proc_nr)*

Description Load the Motorola S program file with name *file_name* at processor *proc_nr*. If the file could be loaded, the pid of the first process defined in the file is returned. Otherwise *INVALID_PID* is returned.

Definition *STATUS Migrate(SLONG pid, WORD proc_nr)*

Description Migrate process *pid* to processor *proc_nr*. This routine can be invoked by any process, except by the process that is migrated.

D.6 Semaphore routines

Two types of semaphores are available in the EMPS kernel: local semaphores, which can only be used within one processor module, and common semaphores, which can be used by all processors within one node. Common semaphores are slower than local semaphores, but allow distributed application programs to use the same semaphore. A distributed version on the EPEP interpreter will therefore require common semaphores.

Definition *SYSCALL create_csemaphore(CommonSemaphore **sem, SLONG init_count, SBYTE csname[])*

Description The routine *create_csemaphore* creates a semaphore. If a semaphore could not be created the function returns the value *ERROR*. Otherwise, a structure of type *CommonSemaphore* is reserved and initialized. The structure is located in common memory so the semaphore can be accessed by all processors in a node. The routine returns a pointer to the structure of type *CommonSemaphore* in *sem*, which is used by *cwait* and *csignal* to identify the semaphore. *init_count* is the initial semaphore counter. *csname* is the semaphore name.

Definition *SYSCALL cwait(CommonSemaphore *sem)*

Description The routine *cwait* executes a P operation on a semaphore: If the semaphore counter of the semaphore is greater than or equal to 1, the counter is decremented. Otherwise, the current process is inserted into the semaphore queue, and it changes its state into *WAIT_FOR_COMMONSEMAPHORE* and blocks. *sem* is a pointer to a structure of type *CommonSemaphore* on which the operation is to be performed.

Definition *SYSCALL csignal(CommonSemaphore *sem)*

Description The routine *csignal* executes a V operation on a semaphore: if the semaphore queue for this semaphore is not empty, the first process in the queue is dequeued and added to the ready list of processes; if the semaphore queue is empty, the semaphore counter is incremented. Parameter *sem* is a pointer to a structure of type *CommonSemaphore* on which the operation is to be performed .

Definition *SYSCALL create_semaphore(semaphore **sem, SLONG init_count, SBYTE sname[])*

Description The routine *create_semaphore* creates a semaphore. The semaphore that is created, is a local semaphore: it can only be accessed by processes on the current processor. If a semaphore could not be created the function returns the value *ERROR*. Otherwise, a structure of type semaphore is reserved and initialized. The routine returns a pointer to the structure of type *semaphore* in *sem*, which is used by *wait* and *signal* to identify the semaphore. *init_count* is the initial semaphore counter. *sname* is the semaphore name.

Definition *SYSCALL wait(semaphore *sem)*

Description The routine *wait* executes a P operation on a semaphore: If the semaphore counter of the semaphore is greater than or equal to 1, the counter is decremented. Otherwise, the current process is inserted into the semaphore queue, it changes its state into *WAIT_FOR_SEMAPHORE* and blocks. Parameter *sem* is a pointer to a structure of type *semaphore* on which the operation is to be performed.

Definition *SYSCALL signal(semaphore *sem)*

Description The routine *signal* executes a V operation on a semaphore: If the semaphore queue for this semaphore is not empty, the first process in the queue is dequeued and added to the ready list of processes; If the semaphore queue is empty, the semaphore counter is incremented. Parameter *sem* is a pointer to a structure of type *semaphore* on which the operation is to be performed.

D.7 Inquiry routines

For the purpose of obtaining information about the system, the following routines are available:

Definition *SYSCALL GetTime(LONG *Time)*

Description *GetTime* puts the system time (the time in ms since startup) in *Time*.

Definition *SYSCALL GetCurrentPD(PD **p)*

Description The routine *GetCurrentPD* puts a pointer to the process descriptor (of type *PD*) of the current process in *p*.

Definition *SYSCALL GetPD(PD **p, SLONG pid)*

Description The routine *GetPD* puts a pointer to the process descriptor (of type *PD*) of process *pid* in *p*.

Definition *STATUS GetPID(SLONG *pid, SBYTE *pname, WORD pn)*

Description *GetPID* returns the process identifier *pid* of the process with name *pname*. The process must be running on the same processor that make the function call, which must be the processor with number *pn*. The process identifier of a process that runs on another processor can be found with the function *GetProcessIdentifier*. If the function could not be found, is returns the value *NoSuchProcess*.

Definition *SLONG getpid(void)*

Description *getpid* returns the process identifier of the current process. Notice the subtle spelling difference with the previous function.

Definition *SYSCALL GetSEM(semaphore **sem, SBYTE *sname)*

Description *GetSEM* returns a pointer to the local semaphore with name *sname* in *sem*.

Definition *SYSCALL GetCSEM(CommonSemaphore **csem, SBYTE *csname)*

Description *GetCSEM* returns a pointer to the common semaphore with name *csname* in *csem*.

Definition *SYSCALL GetDevice(DEVICE **dev, SBYTE *dname)*

Description *GetDevice* returns a pointer to the device with name *dname* in *dev*.

Definition *PROCESSORS *GetProcessors(void)*

Description Return a pointer to an array with the processor numbers of each processor in this node.

Definition *SLONG GetNrOfProcessors(void)*

Description Return the number of processors in this node.

D.8 Send/Receive routines

The send and receive routines described in this section offer communication between processes in a way that is not location transparent. They are used inside the kernel for communication between system processes. When process migration of a specific application program is desired, that application program should use the location transparent mailbox communication services instead of the routines described in this section.

Definition *SYSCALL Send(SLONG pid)*

Description *Send* and *Receive* are routines that are used within the kernel for passing information between processes. For application processes, it may be better to use mailbox functions instead (this provides location transparent addressing). The routine Send sends the information in the message buffer of the current process (*proctable[currpid]->msg)* to another process. The state of the process is changed into *BLOCKED_ON_SEND*, and the process is inserted in the message queue of the receiving process. If the state of the other process is *RECEIVING*, the receiving process is added to the list of ready processes.

pid is the process identifier of the process to which the message is sent.

Definition *SLONG Receive(void)*

Description *Send* and *Receive* are routines that are used within the kernel for passing information between processes. *Receive* makes the current process receive a message. If no sending processes are pending in the message queue, the state of the current process is changed into *RECEIVING*, and the process blocks. If a process is pending in the message queue, the pointer to the message buffer of the sending process is exchanged with the pointer to the message buffer of the receiving process. The sending process is de-queued from the message queue, and added to the list of ready processes.

The value *ERROR* is returned if something went wrong, otherwise the pid of the sending process is returned.

Definition *STATUS SendRequest(SLONG pid, MSG *message)*

Description *SendRequest*, *ReceiveRequest* and *SendReply* are routines used inside the kernel for Remote Procedure Calls (RPC). For application

processes, it may be better to use mailbox functions instead (this provides location transparent addressing). *SendRequest* sends the information in *message* to process *pid.*, and blocks the current process until a reply message is received.

Definition *SLONG ReceiveRequest(MSG **message)*

Description *ReceiveRequest* makes the current process receive a request for a RPC. The parameters for the RPC are received through *message.* After the request is handled, the current process responds by a message via *SendReply.* The process identifier of the sending process is returned.

Definition *STATUS SendReply(SLONG pid, MSG *message)*

Description *SendReply* sends the result of a RPC which was initiated by *SendRequest.* The result is returned in *message* to process *pid.* The process that initiated the RPC is added to the list of ready processes.

D.9 Interrupt related routines

Interrupts are handled by ISPs (chapter 2). The ISP waits for an interrupt by issuing a call to *Await_interrupt*, and is activated from an ISR via a call to *Start_handler.* A Device Control Block forms the connection between an ISP and an ISR.

In the design of kernel, it is assumed that both the ISR and the ISP are system routines operating within the kernel. Therefore, these routines should not be used by application programs. The description of the interrupt related routines can be useful when new interrupt based devices (such as an ETHERNET controller) are added to the EMPS.

Definition *SYSCALL create_device(DEVICE **dev, SBYTE dname[], void *Regs, SLONG MaxInterrupts)*

Description The routine *create_device* creates a Device Control Block (DCB). First a free index in the device table is searched for. If no free entry is found, the routine returns the value *ERROR.* Otherwise, a structure of type *DEVICE* (the DCB) is allocated and initialized.

The routine returns a pointer to the new allocated structure of type *DEVICE* in **dev. dname* is the name of the device.

Paramerer *MaxInterrupts* is the size of the buffer that is associated with the device. If *MaxInterrupts* = 1, no buffering is applied. If *MaxInterrupts* > 1, *Regs* points to a buffer for the device registers.

Definition *SYSCALL Await_interrupt(DEVICE *dev)*

Description The routine *Await_interrupt* waits for an interrupt to occur. The process identifier of the waiting process is written in the device control block pointed to by *dev*. This indicates that an interrupt service process is expecting the interrupt. The process blocks until the interrupt service routine calls the routine *Start_handler*.

Definition *SYSCALL Start_handler(DEVICE *dev)*

Description *Start_handler* activates the process awaiting the interrupt from device *dev*. The routine *Start_handler* is called by interrupt service routines (ISR). The routine obtains the process identifier of the process waiting for the interrupt from the device control block.

D.10 Low level process management routines

The low level process management routines form the nucleus of the kernel. These are single processor routines, and are used within the kernel as a basis for multi-tasking services. Although the low level functions are made available to application programs in the current version of *STDEMPS.OBJ*, application programs should not use these routines. The description of these routines can be useful when updates are made to the kernel.

Definition *STATUS create_proc(SLONG *pid , void (*func)(), SBYTE pname[], SLONG prio, LONG usz, LONG ssz, MEMTYPE ust, MEMTYPE sst)*

Description The routine *create_proc* initializes a new process on the current node, and sets the state of the new process to BLOCKED.

The process identifier is returned in **pid*.

func is the function in the process that is called when the process is started.

pname is the name of the process.

prio is the process priority.

usz and *ssz* are the user- and supervisor stack sizes.

ust and *sst* are the user- and supervisor stack memory types, either *PRIVATE*, *COMMON_VME* or *COMMON_VSB*. For application processes, *ust* and *sst* should always be *PRIVATE*.

Definition *SYSCALL Block(void)*

Description The routine *Block* blocks the current process: The current process is de-queued from the ready queue, the highest priority ready process is looked up, and a context switch occurs in order to execute the latter process.

Definition *STATUS Add_ready(SLONG pid)*

Description Add process *pid* to the ready list and activate this process immediately if the priority is higher than the priority of the current process.

Definition *SYSCALL Block_and_add_ready(SLONG pid)*

Description For time-efficiency, *Block_and_add_ready* combines the functions *Add_ready* and *Block*. The routine *Block_and_add_ready* inserts the process with process identifier *pid* in the ready queue and blocks the current process, thus scheduling the highest priority process. If the priority of the process *pid* is higher than the priority of the current process, the process can be executed immediately, without having to search through the ready list.

Definition *SYSCALL wake_up(SLONG curtime)*

Description The routine *wake_up* wakes up all processes in the clock queue with wake up time < *curtime*: All these processes are inserted into the ready queue and dequeued from the clock queue. The highest priority ready process is then searched for. If that process is the current process, no context switch occurs.

Definition *SLONG Disable(void)*

Description *Disable* disables immediate handling of device interrupts. When a device interrupt has occurred, and no call to *Disable* was made, the process waiting for the device is placed in the ready queue and a context switch is made to activate the process immediately. Otherwise, the process is only placed in the ready queue.

Return value The return value (Status) indicates whether the immediate interrupt handling already was disabled. The return value must be used as the *Status* parameter when *Enable* is called. This way, multiple nested calls to Disable are allowed.

A more detailed description of the use of the *Disable* and *Enable* routines can be found in appendix G.

Definition *void Enable(SLONG Status)*

Description *Enable* enables handling of device interrupts by an ISP. Interrupts are buffered while disabled by *Disable*, and will be handled immediately when *Enable* is called. *Status* is the value returned by the corresponding call to *Disable*.

D.11 Other routines

The following routines are probably less interesting for application programs. These routines are only used by the Oasys development system, or are obsolete.

- **Low level routines required for the Oasys compiler:** *sbrk, getpid, unlink, isatty, _exit*
- **Supervisor/user mode switch:** *ChangeModeToSuperVisor, ChangeModeToUser, CheckMode*
- **Low level PhyLAN routines:** *LookupFile, EnterFile, DeleteFile, ReadBlock, WriteBlock, CloseFile, OpenChannel, CloseChannel, EchoMessage*
- **Low level communication register routines:** *ExchangeMessage, InitializeComRegMessage*

Appendix E

Memory related routines

In chapter 3 the memory management in the EMPS kernel was discussed. In the present section, some more practical information is given about the kernel functions that are related to the memory routines.

For allocating and releasing memory, the standard ANSI C routines *malloc*, *free*, *calloc* and *realloc* are provided. An extra routine, called *malloc_type*, can be used to allocate private memory.

- *void *malloc(SLONG size)*

  Allocate *size* bytes of memory, and return a pointer to the first byte. If no memory could be allocated, return *NULL*. This function allocates memory of type *COMMON_VME*, which can be accessed by all modules connected to the VME bus. *malloc* calls the routine *malloc_type*, which has the same functionality, but can also allocate *PRIVATE* and *COMMON_VSB* memory. For the exact behavior of malloc, see *malloc_type*.

- *void *malloc_type(SLONG size, MEMTYPE type)*

  Allocate *size* bytes of memory, and return a pointer to the first byte. *type* is the type of memory allocated. If no memory could be allocated, return *NULL*.

  When *malloc_type* is called from an application program, it behaves different from when it is called from within the kernel. Memory allocated in the kernel is accessed via the system address space, which is the area with addresses 0x80000000 and higher. For this memory area, a MMU mapping is created when the kernel is initialized, and no additional mapping is necessary. In this case, *malloc_type* just reserves the requested amount of memory.

  If an application process calls malloc, just reserving memory is not sufficient, because application processes should not access the system address space. Therefore, it is necessary to create an additional MMU mapping, so that the allocated memory can be accessed via the address space that is reserved for application programs. How this mapping is

actually created, depends on the amount of memory requested. If the number of memory pages requested is less than *MAX_SMALL_PAGES* (defined in the file *memory.c* as 16), normal page descriptors are used for the mapping. If there are insufficient page descriptors available, or if the number of memory pages requested is more than *MAX_SMALL_PAGES*, the mapping is made with early termination page descriptors.

Memory allocated with malloc_type is both physically and virtually contiguous. This is obvious for memory allocated by system processes, because for the system address space, physical addresses and virtual addresses only differ by a constant offset. In application processes, this is achieved of by allocating the requested memory contiguously, and by using contiguous page descriptors to create a mapping.

malloc_type returns *NULL* if the requested amount of memory could not be allocated. For application programs, this can have two reasons: either not enough contiguous free memory is available, or not enough contiguous page descriptors are available. For system processes, it can only mean that not enough contiguous free memory is available.

- *void free(void *ptr)*

  Release memory that was allocated with *malloc*, *calloc*, *realloc* or *malloc_type*, with start address *ptr*. If *ptr* does not point to a memory area that was allocated with one of the above routines, the behavior is undefined.

  When *free* is called by a system process, it just releases the memory. When *free* is called by an application process, it releases memory and also releases the MMU descriptors used for the mapping of the memory area.

- *void *calloc(SLONG nr, SLONG size)*

  calloc allocates memory for nr items of size bytes each, and initializes all items to the value 0. If memory could be allocated, a pointer to the first byte is returned, else *NULL* is returned. See the description of *malloc_type* for details of memory allocation.

- *void *realloc(void *ptr, SLONG size)*

  realloc changes the number of memory bytes allocated by *malloc*, *calloc*, *realloc* or *malloc_type* to *size*, copying the contents of the old memory area. *ptr* points to the old memory area, which is freed. The return value is the address of the new memory area. See the description of *malloc_type* and *free* for details of allocating and releasing memory.

Translation of addresses is often necessary in system software, e.g. when a hardware device is programmed to perform I/O via DMA, physical memory addresses are needed. Also, if an application process sends a message to another process, the message is located in the virtual address space of the sending process. The receiving process has a different virtual address space, and cannot access the message at the same address as the sending process. Therefore, the message address must be translated to an address in the system address space (which can always be reached).

- *STATUS VirtualToSystemAddress(LONG *virtual_address, LONG *system_address, SLONG pid)*

  The routine *VirtualToSystemAddress* translates *virtual_address* (located in the address space of process *pid*) into *system_address*, which is a virtual address located in the system address space. *virtual_address* can be located anywhere in the address space that can be reached by process *pid*, i.e. the code, data, heap, and stack areas, as well as the system address space.

  VirtualToSystemAddress is used when a process sends information to another process, which does not use the same virtual address space. To perform the translation, the routine uses the same translation tree as the MMU.

- *STATUS TranslateVirtualAddress(LONG *virtual_address, LONG *physical_address, SLONG pid)*

  The routine *TranslateVirtualAddress* translates *virtual_address* of process *pid* into *physical_address*, which is the physical address used by the hardware. *TranslateVirtualAddress* is used when a hardware device performs I/O via DMA. The device must then be programmed with the physical address of the memory area used in the I/O operation. To perform the translation, the routine uses the same translation tree as the MMU.

- *void *Virt(void *physical_address)*

  Translate *physical_address* into a virtual address which can be reached via the system address space. The return value of this routine is the virtual address. Translation from a physical address into a system address is only possible because the relation between the two addresses is simple: the differ by a constant offset. Private memory, and common memory each have a different offset.

- *void *Phys(void *virtual_address)*

  Phys translates the system address *virtual_address* into a physical address. The return value of this routine is the physical address. *Phys*

assumes that *virtual_address* is located in the system address space, and cannot be used to translate virtual addresses from application programs.

E.1 Other memory allocation routines

In the original kernel, some memory related routines were defined that are not available in the current kernel version. Also, some routines perform slightly different actions in the new implementation. Although these routines are no longer needed, because the ANSI routines can now be used, they are still available for consistency. The memory related routines of the original kernel, and the difference in behavior between the old- and new implementations are listed next. In both implementations, the behavior is undefined if a parameter contains an illegal value.

- *STATUS AllocateContiguousMemory(MALLOC *m, SLONG size, MEMTYPE type)*

  The routine *AllocateContiguousMemory* allocates *size* bytes of memory in the system address space. The start address is returned in *m.StartAddress* and the number of memory pages in *m.NrOfPages*. *type* is the memory type to be allocated: *PRIVATE*, *COMMON_VME* or *COMMON_VSB*.

- *STATUS ReleaseContiguousMemory(MALLOC *m)*

  The routine *ReleaseContiguousMemory* releases memory allocated with *AllocateContiguousMemory*. In the new implementation, when the memory area at *m.StartAddress* is released, the number of pages that is released is the same as was allocated with *AllocateContiguousMemory*. As mentioned in section 3.1, the number of pages is derived from the table in figure 3.1. The value in *m.NrOfPages* is not used. In the old implementation, *m.NrOfPages* is the number of pages released.

- *STATUS AllocateMemory(LONG *memory_address, SLONG size, MEMTYPE type)*

  The routine *AllocateMemory* allocates *size* bytes of memory in the system address space. The start address is returned in *memory_address*. *type* is the memory type to be allocated: *PRIVATE*, *COMMON_VME* or *COMMON_VSB*.

- *void ReleasePage(LONG *page_address)*

  The routine *ReleasePage* releases a memory area at address *page_address*. In the new implementation, a memory range is released

of the same size as was previously allocated at address *page_address*. In the old implementation, only one memory page was released. Because no use was made of the fact that only one page was released in the old implementation, the inconsistency has no effects.

Appendix F

Process queues

In the EMPS system, process queues are used in the scheduling of processes. The queue in which a process is, depends on its state and its priority (appendix H). When a process is in the *READY* state, the priority of the process determines in which queue it is. For each process priority, a different queue exists.

A process queue is organized as a linked list as shown in figure F.1. The object that is related to the process state, e.g. a semaphore, contains a pointer to the head of the queue and a pointer to the tail of the queue. If the queue is empty, the head pointer contains NIL. Each process in the queue contains a pointer to the next queue element, or NIL if there is no next queue element. Processes are added at the tail of the queue, which can be done very efficiently via the tail pointer.

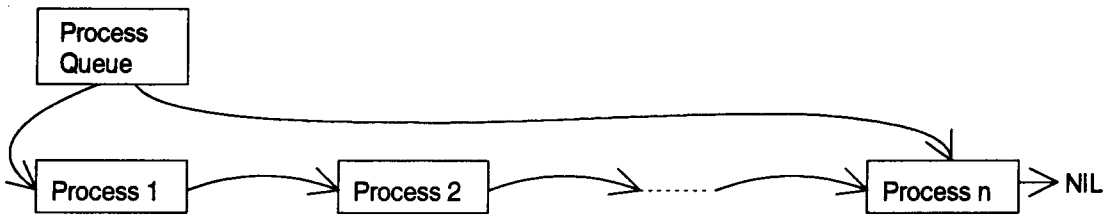

Figure F.1: *The process queues are organized as linked lists.*

Because of the uniformity of the process queues, inserting and deleting processes from a queue can be done with the same routines for all objects that contain a process queue. In the current implementation, however, a separate process queue routine exists for each object.

Appendix G

Interrupt Service Processes

In the EMPS kernel, interrupts are handled by dedicated Interrupt Service Processes. Two steps lead to the activation of an ISP:

1. The interrupt causes the CPU to execute an Interrupt Service Routine (ISR). This is the routine to which the interrupt vector points.
2. The ISR unblocks the Interrupt Service Process (ISP). All further actions required to handle the interrupt are performed by the ISP.

In the EMPS kernel, an ISP is associated with a device, which is the logical source of the interrupt. One physical device may be the source of a number of different interrupts, each of which is associated with a logical device and thus with an ISP. An example of a device that can cause more that one interrupt is the DUART (clock, serial input lines, serial output lines).

The following kernel routines are involved in interrupt handling:

- *Start_handler:* called by the ISR in order to unblock the ISP.
- *Await_interrupt:* called by the ISP when it is ready to handle an interrupt. If no interrupts are pending in the device queue, the ISP is put in the state *AWAIT_INTERRUPT*. Otherwise, execution of the ISP is continued.
- *Disable:* Disable the activation of ISPs by *Start_handler*. During some actions that involve common resources (e.g. updating of process descriptors, updating of the list of free memory pages, handling of an interrupt by an ISP) task switching must temporarily be disabled. If an interrupt occurs while activation of the ISP is disabled, the interrupt is registered in the device queue.
- *Enable:* Undo the effects of disable: if an ISP was put in *READY* state, start its execution immediately.

In the implementation of these routines, the variable *InterruptScheduler* plays an important role: this variable is contains the value *IS_ENABLE* if

activation of ISPs is enabled or *IS_DISABLE* if it is disabled. **The variable *InterruptScheduler* is saved on stack by the routine *cswitch* whenever a context switch takes place.** This means that the value put in *InterruptScheduler* (and therefore calling of Enable and Disable) only has an effect on the current process, although it is declared as a global variable

G.1 Process priorities

The relative urgency of a process is indicated by its priority. In the EMPS kernel, two different

types of priorities can be distinguished:

- the **hardware** priority, which is the processor priority or interrupt level. In the EMPS kernel, only Interrupt Service Routines execute at a hardware priority different from 0.
- the **software** priority, used by the Interrupt Scheduler for scheduling of processes.

In the EMPS kernel, all processes execute at hardware priority 0. Therefore, scheduling of processes is only determined by the software priority. There are 16 software priorities available for processes, of which the 8 highest priorities are reserved exclusively for Interrupt Service Processes. Priority 0 is reserved for the NULL process, so 7 process priorities can be used in application programs.

For each process priority, a separate *READY* queue exists in the kernel. The process priority only affects processes that are in one of the *READY* queues. Other process queues in the kernel, such as the semaphore queues, are not affected by the priority of processes.

Processes with priority ≥ 8 differ from other processes in that they can become READY asynchronously with respect to the active process, because they are inserted in the ready queue as soon as the corresponding interrupt occurs.

Appendix H

Data structures

H.1 Process descriptor

The process descriptor of each process stores all information or pointers to the information that belongs to a process. For each process in the system a process descriptor data structure of type PD is created. PD is defined as:

```
typedef struct PD {
  struct PD *next;      /* Pointer to next process in ready queue */
  SLONG     pstate;     /* Process state */
  SLONG     prio;       /* Process priority */
  SLONG     pid;        /* Process identifier */
  SBYTE     pname[ MAXPNAME ];/* Process name */
  struct PD *nextsq;    /* Pointer to next process in semaphore queue */
  semaphore *sema;      /* Pointer to semaphore */
  SLONG     psp;        /* Process stack pointer */
  LONGTABLEDESCRIPTOR srp;/* SuperVisor Root Pointer */
  LONGTABLEDESCRIPTOR crp;/* CPU Root Pointer */
  MSG       *msg;       /* Pointer to message buffer for this process */
  SLONG     nr_msg;     /* Number of processes in message queue */
  struct PD *next_clockq;
                        /* Pointer to next entry in clock queue */
  CommonSemaphore *csema;
                        /* Pointer to common semaphore */
  SLONG     time;       /* Wake up time */
  SLONG     msgQhead;   /* Index in msgQ (see below)  */
                        /* of first process in message queue */
  SLONG     msgQtail;   /* Index in msgQ (see below)  */
                        /* of last process in message queue */
  SLONG     msgQfull;
  LONG      msgQ[ MAXMSGQUEUE ];
  void      *msgQmsg[ MAXMSGQUEUE ];  /* Array of pointers to */
                        /* message data (for IN communication only) */
  MSG       *CMmsg;     /* Pointer to the message in the common memory */
```

```
  SLONG     BlockedOnSendPid;
  CommonEvent *cevt;
  MB_LOCATION MBLocation;
            /* If pstate==RECEIVING then MBLocation valid */
  PORT      *Ports[ MAXPORTS ];
  void      *Packet;
  PORT      *BlockedOnMailBox;
  MSG       *MailBoxMessage;
  MSG       *RequestMessage;
  MSG       *ThreadMessage;
  TASKDESCRIPTOR *td; /* Task to which the process belongs */
  SLONG     supervisor_stack_pages;
                    /* Number of supervisor stack pages */
  SLONG     user_stack_pages;
                    /* Number of user stack pages */
} PD;
```

The most important fields of PD are described below:

next: if the process is in the ready queue, **next** contains a pointer to the next process in the ready queue, or NIL if there is no next process.

pstate: defines the process state and can have the following values:

- *CURRENT* (0): the state of the process that is currently being executed. On each computer module, only one process can be in this state. Normally this is the highest priority process that has been in the ready queue for the longest time. The only exception is when activation of ISPs has temporarily been disabled by a call to Disable. In that case, when a ISP is un-blocked, it is not immediately activated even though it has a higher priority than the current process.
- *READY* (1): the process is ready to be executed.
- *BLOCKED* (2): the process is blocked. This is the state of a process that has just been created.
- *TERMINATED* (3): the process is terminated. When a process terminates, it is removed from all queues, and all references to the process descriptor are removed. Therefore, this process state can never occur.
- *RECEIVING* (4): the process is waiting until it receives a message.
- *WAIT_FOR_EVENT* (5): the process is waiting for an event. Events are not supported in the current kernel version.
- *WAIT_FOR_SEMAPHORE* (6): the process is waiting for a semaphore.

- *BLOCKED_ON_SEND* (7): the process has been sending information and is waiting for a reply.
- *DELAY* (8): the process is delayed for a specified amount of time, and will be restarted by the clock process.
- *WAIT_FOR_COMMONSEMAPHORE* (9): the process is waiting for a common semaphore.
- *DEBUG_TRAPPED* (10): the process is trapped for debugging. In the current kernel version, debugging is not supported.
- *BLOCKED_ON_COMREG* (11): the process is waiting until a message is received via the communication register. This is used for communication between processors.
- *WAIT_FOR_COMMONEVENT* (12): the process is waiting for a common event. Common events are supported, but are not used in the current kernel version.
- *AWAIT_INTERRUPT* (13): the (interrupt service) process is waiting for an interrupt.
- *SEND_TO_MBX* (14): the process has sent a message to a mailbox, and is waiting for a reply.
- *RECEIVE_FROM_MBX* (15): the process is waiting until it receives information from a mailbox.

`prio`: The process priority (0-15). Priority 0 is reserved for the NULL process. Priorities 8-15 are reserved exclusively for ISPs.

`pid`: The process identifier, which is a 32 bit number associated with each process in the system. Each process has a number that is system wide unique. The 16 most significant bits are determined by the geographical location of the process: the first byte contains the number of the node, and the second byte contains the number of the processor within a node. The 16 least significant bits are a unique number within the processor module.

`psp`: When a context switch occurs, all processor registers of the current process are put on the supervisor stack. The supervisor stack pointer is then copied to `psp`, and is used when the process is re-activated.

`srp`, `crp`: The MMU root pointers for supervisor- and user mode respectively. These define the start address of the MMU translation table for each process. Within the process descriptor, the position of `srp` and `crp` relative to `psp` must not be changed, because this relative position is used in the context switch routine *cswitch*.

H.2 Task descriptor

The task descriptor contains information about the MMU tables of a task. The memory used for a task descriptor is allocated together with the memory used for storing the MMU tables when a new task is created.

```
typedef struct {
  SLONG                  nr_pages;
  SLONG                  nr_of_processes;
  LONGTABLEDESCRIPTOR    *level_b0_descriptors;
  LONGPAGEDESCRIPTOR     *user_code_area;
  LONGPAGEDESCRIPTOR     *user_data_area;
  LONGPAGEDESCRIPTOR     *user_heap_area;
  SLONG                  nr_of_code_pages;
  SLONG                  nr_of_data_pages;
  SLONG                  nr_of_heap_pages;
} TASKDESCRIPTOR;
```

`nr_pages`: The number of memory pages used by the task descriptor plus the MMU tables.

`nr_of_processes`: The number of processes is the task. When a process is created, `nr_of_processes` is incremented, and when a process is terminated, it is decremented. If a process is terminated and `nr_of_processes` reaches 0, the memory used by the task, as well as the memory used by the task descriptor is released.

The other fields in the task descriptor contain the start addresses of MMU descriptor tables and the sizes of these tables.

H.3 Common memory vector table

The common memory vector table contains information that can be accessed by all processors in a node. It is initialized by the system controller, and is located at the first available memory locations in common memory. The address of the first free memory location is determined by each processor when the system is started and is stored in the pointer variable `CommonMemoryVectorTable`. It points to a structure of type `COMMONMEMORYVECTORTABLE`, which is defined as:

```
typedef struct {
  LONG        MemorySize;
  LONG        *CommonSemasAddress;
  LONG        *CommonMemoryBuffersAddress;
  LONG        *PHYLANBuffersAddress;
```

```
  LONG          *ComRegBuffersAddress;
  LONG          *ProcessNameTables;
  LONG          *FSChannelsAddress;
  LONG          *CommonEventsAddress;
  LONG          NrOfProcessors;
  LONG          *CommonMailBoxes;
  LONG          *Processors;
  LONG          *PacketBuffersAddress;
  LONG          NodeNumber;
  LONG          *XmitBuffer;
  LONG          *RcvBuffer;
  LONG          *XmitBuffers;
  LONG          *RcvBuffers;
  BYTE          *vme_mem_adr[MAX_VME_SLOTS];
  LONG          vme_mem_size[MAX_VME_SLOTS];
} COMMONMEMORYVECTORTABLE;
```

H.4 Program file header

The program file header must be present at the start of each application program file. It consists of three parts: EMPS program file identifier plus an EMPS file header version number, information about the program (task) that is to be loaded, and information about the processes in the task which are created when the task is loaded.

The program file identifier plus header version are defined in `EMPS_VERSION`:

```
typedef struct {
  SBYTE          emps_id[4];
  SBYTE          version[4];
} EMPS_VERSION;
```

`emps_id`: This field must contain the characters "EMPS" to identify the file as an EMPS program file.

`version`: The EMPS file header version number. In the current implementation, only EMPS file header version "0000" is used.

The task related part of the file header of version "0000" is defined in the type `EMPS_0000`:

```
typedef struct {
  LONG           nr_of_processes;
  MEMTYPE        memtype;
  LONG           code_size;
  LONG           data_size;
```

```
  LONG          max_heap;
} EMPS_0000;
```

nr_of_processes: The number of processes created when the task is loaded. For each process, a structure of type PROCESSIDENTIFICATION is present in the file header.

mem_type: The memory type that is allocated for program code and data. **mem_type** can be either PRIVATE, COMMON_VME, of COMMON_VSB.

code_size: The number of bytes in the code area of the program.

data_size: The number of bytes in the data area of the program.

max_heap: As explained in chapter 3, programs can allocate 61 blocks of memory using early termination page descriptors. For programs that allocate many small blocks of memory, normal page descriptors can be reserved by setting **max_heap** to the required memory size for normal page descriptors of 1 page each. Because most programs don't allocate more than 61 blocks of memory, it is usually sufficient to set **max_heap** to 0.

The task related part of the file header is followed by one or more process related items. For each process that is created when the task is loaded, a structure of type PROCESSIDENTIFICATION is present in the file header.

```
typedef struct {
  void          (*func)();
  SBYTE         pname[ 20 ];
  SLONG         prio;
  LONG          UserStackSize;
  LONG          SuperVisorStackSize;
  MEMTYPE       UserStackMemType;
  MEMTYPE       SuperVisorStackMemType;
  SLONG         ProcNr;
} PROCESSIDENTIFICATION;
```

func: The function that is called when the process is started.

pname: The process name.

prio: The process priority.

UserStackSize: The size of the user stack in bytes. The user stack is used for the local variables and return addresses of the process.

SuperVisorStackSize: The size of the supervisor stack in bytes. Although application processes run in user mode, and therefore do not use the supervisor stack directly, each time a kernel function is invoked, a switch to supervisor mode is made. The kernel function then uses the supervisor stack. When an interrupt occurs, the supervisor stack is also used.

UserStackMemType: The type of memory that is allocated for the user stack, either PRIVATE, COMMON_VME, of COMMON_VSB.

`SuperVisorStackMemType`: The type of memory that is allocated for the user stack, either PRIVATE, COMMON_VME, of COMMON_VSB.

`procnr`: The number of the processor at which the process is created.

Appendix I

Improvements and extensions of the original kernel

The following list contains a summary of improvements and extensions that have been made to the original kernel.

- **Adaptation of the C source code to the ANSI standard.** The original kernel contained a mixture of pre-ANSI C and ANSI C program code. A consistent use of ANSI C improves readability and allows type checking of parameters by the compiler.
- **Implementation of the XON/XOFF protocol for flow control of output to the terminal.** The XON/XOFF protocol prevents characters to be lost when output is sent to the terminal at a higher rate than the terminal can handle. It also allows the use of the *hold* key on the terminal to suspend output.
- **Implementation of time slicing.**
- **Redesign of memory management services.** The new memory management in the kernel is described in chapter 3. Apart from the issues discussed in chapter 3, the changes in memory management caused significant alterations of other parts of the kernel, e.g. process migration and the loading of application programs.
- **Revision of the *init* program and slavemonitor.** The *init* program, which performs the part of the initialization of the MMU that must be executed before the kernel can be loaded at the system controller processor module, has been completely rewritten. The slavemonitor performs a similar function at other processor modules and is present in EPROM memory. It has been updated to comply with the requirements of the new memory management implementation.
- **Implementation of a new file header for application programs.** In the new kernel version, MMU tables used for application programs are allocated dynamically. Therefore, the file header must contain the