Department of Computer Science

Johan Jorgensen

Embeddable kernel architectures

Master's Thesis

Masters Thesis 99:xx

Embeddable kernel architectures

Johan Jorgensen

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Johan Jorgensen

Approved, Date of defense

Opponent: NN

Advisor: Donald F. Ross, Ph.D.

Examiner: NN

Abstract

Put the text of your abstract here

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 2

Background

2.1 Introduction

Operating systems are computer programs that control the basic functionality of a computer. Operating systems have been used in computers for at least the last 31 years[1]. Today operating systems are not only found in traditional computers but in systems where would be unexpected, namely systems in which the computer is used only for control. This type of computers are usually referred to as *embedded systems*. Embedded systems are becoming increasingly common. A few examples of such systems are:

- The Hewlett Packard Deskjet 610C
- The Minolta DImage 1500EX zoom camera.
- The mars pathfinder (NASA Jet Propulsion Laboratory (JPL)).

All of the above systems use an operating system called VxWorks as the basic control element of their computers. There is a number of reasons to this development. The most important ones are:

[1] One of the first monolithic systems was the *"THE"* multiprogramming system (see [Dijk 68])

- Economic aspects, especially factors such as:
  - Time-to-market
  - Development costs
- Systems are becoming increasingly more complex, (re)implementing entire network stacks etc. would be too time consuming
- Standardized components or subsystems delivered by 3rd party manufacturers. In the case with the HP printer the CPU is an MC683xx CPU (Off-the-shelf CPU available from Motorola) and the OS is Wind River Systems VxWorks[2].

Many of todays common systems utilize micro controllers and advanced software in order to interact be able to interact with the outside world in an intelligent manner. The advent of networking capabilities and even cheaper CPUs will accelerate this process while interconnecting systems. The use of a high-performance embeddable OS that can easily be customized to a specific, generic, computing platform will decrease the development time. Careful design and selection of the hardware/OS combination makes it possible to reuse the specific combination, thus only the application needs to be modified.

2.1.1 Classification of computer systems

In this text computer systems are divided into two different groups. Depending on how the computers are used.

Embedded systems A computing platform built into a larger system and used to control the operation/behavior of that system. These systems ranges from cellular phones to ICBMs.

[2]See www.wrs.com

Personal Computers Modern PCs/Workstations, Network servers etc. In general any computing device used for business, leisure or scientific applications. From now on referred to simply as "PCs"

The two groups have differences both in architecture and software design. From a hardware point-of-view the main differences between the two categories are:

- PCs usually have some sort of secondary storage.
- PCs generally have better graphical capabilities than embedded systems
- PCs use dynamic memory (S)DRAM

The real difference however is found on the software side. PCs are usually able to run more than one program, and to run different programs at different times. For instance it is possible to play a game or write a letter on the standard PC whereas an aircraft navigation-system seldom are used for anything else except navigation. Embedded systems run the same code and that code rarely changes throughout the lifetime of the system.

From a hardware point-of-view the electronic devices are similar as are the technologies. There is however a great difference on the PC side when it comes to peripherals such as graphics adaptors etc. This is due to the great variety of systems available on the market. For instance there are a vast number of companies that supplies different graphics adaptors, which are more or less incompatible with each other and thus require special device drivers in order to work with the OS.

Embedded systems

An embedded system (ES) is a part of a larger system in which a computing platform is used to carry out one or more tasks that are crucial to the operation of the system[3]. Such

[3]In this text the term "system" refers to the entire system

a system might be a car, an aircraft or a CAT-scanner [4]. Examples range from microwave ovens to modern cars which typically employ 50 or more processors to monitor or perform vital functions.

The general trend today is towards more and more complex systems with networking capabilities. Secondary storage: such as hard-drives[5], and multiple CPUs are also utilized in some systems depending on complexity.

Embedded systems can be small and simple or large and complex. In any case there are some fundamental similarities regardless of complexity.

- The Hardware/Software configuration is identical among all instances of a particular version of a product
- The same software is always executed and is rarely or never updated
- Production volumes can be very high

The next two paragraphs illustrate the differences in complexity that can be found in embedded systems. One of the most complex systems found on the market to day are Tag Heuer's engine-control system used in Formula-1 cars.

Tag Heuer's on-board computer (OBC) system for F1 engines is a classical example of an embedded system. This system contains 7 Motorola DSP processors and Ford/Intel processors. This massive multiprocessor system enables the F1-engines to run at some 18000-20000 RPM which is essentially just on the edge of self destruction, by controlling the exact time of ignition and the amount of fuel that is to be distributed to each cylinder. The system must also compensate for such factors as weight, ambient-air-pressure, humidity, and temperature. A microwave link is used to transfer data to and from the

[4] *C*omputer-*A*ided *T*omography
[5] For instance in high-speed laser printers

crew in the pit allowing them to analyze the engine's performance in real-time. It is also possible to reprogram the OBC to, for instance, over-compensate for weather conditions.

Another, not so complex, example of an embedded system is the microwave oven. Most microwave ovens today have some sort of "sixth sense capabilities". These ovens are, in contrast to the Tag Heuer's computers, manufactured by the thousand. This implies that the computing platform used to control the microwave oven must be simple and cheap since it is aimed at private consumers. Making it possible to reuse parts of the software, or maybe even the entire computing platform will enable manufacturers to get new products out the door faster, thus decreasing the development costs.

Personal computers

There have been several attempts made to use the traditional PC as an embedded controller. Although the software is relatively easy to port, it does not turn the PC into a secure embedded or real-time platform for a number of reasons.

- PCs always use dynamic memory. The refresh operation is non-deterministic and thus makes it impossible to the PC as a hard real-time system.
- PCs are designed using commercial-grade components. This has several negative side-effects that cannot be ignored:
  - Commercial-grade components are more sensitive to variations in temperature than their industrial and military counterparts.
  - Commercial-grade PCBs are not vibration and shock tested as military and industrial systems are.
  - The edge-connectors are particularly sensitive to vibration and humidity

2.2 Operating systems

There are several different types of operating systems available on the market today. When most people think about OSes they envision something like Windows, DOS, or maybe a unix-like system. However there is a class of operating systems that are used in embedded systems. Any operating system, whether it is aimed at embedded systems or not can be placed in one or more of the following sub-classes, depending on its design:

- Single-task OS: A single-task operating system only run one task at a time. The most well-known system is probably DOS
- Multi-tasking OS: A multitasking operating system is capable of dividing the CPU-time between different tasks. Most modern desktop OSes are multitasking

Embedded systems have additional requirements because they almost always have interfaces to the physical surroundings (the real world).

- Real-time: Real-time capabilities are commonly found in embedded systems while it is relatively little used in PCs (sound /video) it is common in embedded systems and primarily involves the issue of time-deterministic behavior
- Reentancy: Desktop systems such as Linux usually cannot be interrupted once it has started processing a system call i.e. once it has entered the call inside the kernel it will not return
- Well defined interfaces whose correctness have been proved using a formal method
- Critical error handling. Safety critical systems such as nuclear plants et.c.

Formally verified interfaces, internal as well as at the system-call level are still uncommon. The idea behind this is to prove that all calls behave according to a formal specification in

order to minimize errors. As with all formal methods this will only work if the specification is correct i.e. it must at least be physically realizable. Correct behavior is crucial in embedded systems that control vital systems such as nuclear power-plants, life sustaining equipment used health care, aircraft etc.

2.2.1 The purpose of the OS

The purpose of the operating system is to provide abstractions to the physical system that makes up the computing platform. The abstraction is in itself desirable because it provides a "generic CPU". Apart from providing an abstraction to the CPU, the OS also provides the following functionality:

- *Resource-sharing and -allocation* - the OS enables applications to share resources among each other
- *Protection* - the OS provides some level of protection against malicious operations
- *Abstraction* - At the highest level, provides the application programmer with a set of services

Protection mechanisms

The CPU based protection mechanisms limit access to various resources such as memory and I/O ports. Almost every modern CPU provides at least two *privilege levels*, user and supervisor level. Code executing with user-privileges are not able to access memory segments that have not been mapped as usable by code running in supervisor mode. This information is generally stored in registers inside the CPU. Generally the following information must be stored in order to provide proper protection:

Base address The address where the memory segment begins

Segment size The size of the segment in bytes

Segment type The type of segment. Generally there are three different types of segments

- Read-only segments. The segment can only be read
- Read-write segments. The segment can be read and written
- Executable. The segment contains code that can be executed

If a process/task tries to perform an illegal operation, such as writing to a read-only segment, an exception is generated. This exception will switch the CPU to supervisor level and process the fault, for example by suspending the run-away process and returning all the reserved resources to the OS-kernel.

Building abstractions around the basic hardware-protection mechanisms provided by the CPU makes it possible to provide protection against the following types of violations:

- The different tasks are isolated from each other because they cannot access memory areas used by other tasks
- The kernel data-structures and the kernel code are protected from faulty or malicious tasks
- The integrity of the I/O subsystems are protected from all tasks. This is important in order to be able to share resources among tasks

It should be noted however that the tasks running in user-mode are totally unprotected from any code running in supervisor mode. This means that a faulty kernel is able to destroy all or some tasks thus stopping the entire system. This makes the correctness of the OS kernel the most important issue seen from a reliability perspective. The main reason for this is that code running in supervisor mode (the OS) is able to change the base- and segment size registers. If the OS were unable to perform this change it would not be

possible to build multitasking systems since it would be impossible to perform context switches (switching from one process to another).

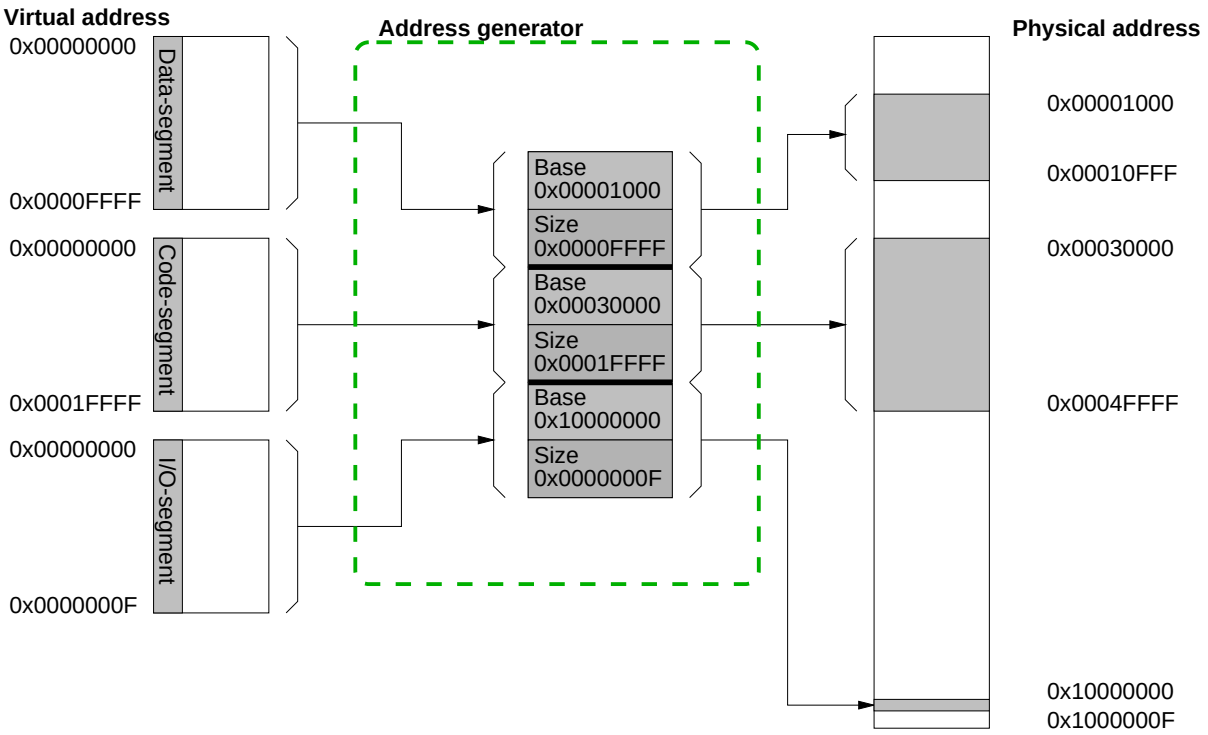

Figure 2.1: The general principle of memory protection. Each segment is has a base address and a size. The base address maps the virtual address to the physical address space.

Abstracting away from hardware

By providing high-level programming primitives the operating system hides the details of the hardware from the programmer. The OS generally does this through a, for the OS coherent application programming interface (API), the sys-call-API. The sys-call-API provides the programmer with a virtual machine that is totally independent of the underlying hardware or software layers. The level of abstraction does not say anything about *how* the functionality provided by the sys-call-API is implemented, neither is it of any great importance because the application "sees' the same sys-call-API regardless of the OS-implementation.

There are several reasons to do this:

- It enables resources to be shared among different tasks running in user-mode
- It hides the details of the hardware from the application thus providing a virtual machine allowing any application compiled for the system to run

The second item in the list above is really just an issue in conventional computers since embedded systems often have identical hardware (For instance: Two microwave ovens of the same brand and model). The abstractions provided will enable applications to be ported from one system to another and thus significantly shortens the development cycle.

Managing resources

Protection and *Abstraction* are vital requirements needed by any operating system in order to manage resources. The protection mechanisms and abstractions provided by the lower layers is used by the kernel to maintain the integrity of I/O subsystems. The high-level system calls provided by the operating system enables the kernel to serialize data to and from the physical device. Memory is managed much the same way: The OS provides a single entity.

2.3 Abstraction models

Basically there are two different versions of abstraction models. The monolithic full-service kernel that provides a very high level of service and the micro-kernel, or "kernelized structure", which hardly provides any service at all. The fundamental difference between the two different models are that the monolithic kernel is very centralized and has a strong notion of kernel-space. The micro-kernel or, kernelized structure, offer little more than a generic interface to the basic operations provided by the CPU. The classic way of increasing the abstraction level is to use logical layers stacked on top of each other.

2.3.1 Layers in OSes

One of the first layered OSes ever described is the *"THE"* system described by Dijkstra in [Dijk 68]. The model is well-known: Layers are placed on top of each other. Communicating between the layers are done through system calls. The typical layers found in a monolithic OS is shown in figure 2.2.

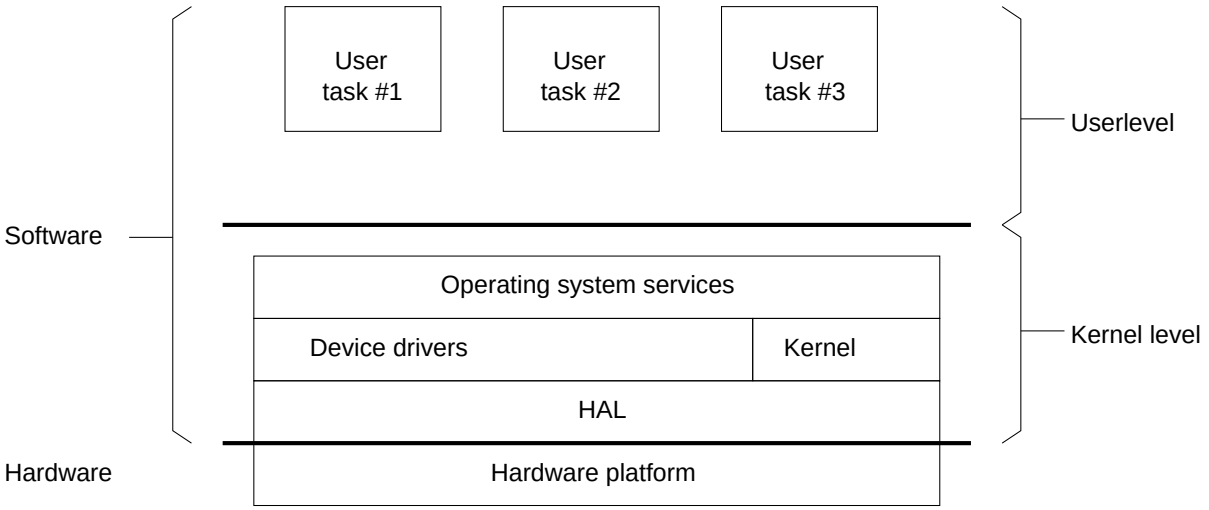

Figure 2.2: Typical layers of abstraction in a monolithic OS

The lowest layer in this model is the hardware platform. The hardware is not covered

in this text.

The generic CPU - the HAL

The HAL(*H*ardware *A*bstraction *L*ayer) implements a virtual CPU. The functionality placed in this layer are low level operations such as the following:

- Low level I/O operations i.e. functions to read and write one or more bytes from I/O devices
- Functions to test, set and clear bits in I/O devices
- Low level interrupt and exception management
- Stack management including stack-frames for exception and interrupt processing
- Functions to save and restore the CPU context
- Multiprocessor communication interface (MPCI)
- Synchronization mechanisms such as semaphores, mutexes, and events

This, the lowest level of abstraction, in the software domain will always be CPU- and platform dependent. The reasons are that different processor architectures are slightly different. For instance the Motorola MC68xxx architecture does not provide a specific I/O instruction whereas the i386 related architectures do.

The platform-to-platform differences are dependent on the architecture of the platform and not the CPU. The reason for this is that the synchronization primitives must be implemented slightly different on multiprocessor platforms than on von-Neumann machines (the . Dependent on whether the platform is a heterogenous or homogeneous multiprocessor platform, changes can be required to the MPCI functions as well.

The kernel and device driver layer

This layer performs two different functions. The part labeled "kernel" provides basic OS functionality such as:

- task/process management and scheduling
- Interrupt dispatching
- Device-driver interfaces (I/O manager)

Managing and scheduling threads are a complex issue. First and foremost a suitable scheduling policy must be decided upon. In embedded real-time systems the most frequently used algorithms are earliest deadline first (EDF) and the rate-monotonic scheduling (RMS).
The scheduler uses primitives in the HAL to load and save the CPU context. The scheduler provides functionality to create,delete, stop and restart tasks. Depending on scheduling policies a task can be blocked or waiting. Figure 2.3(A) shows the states that an RTEMS task can be and 2.3(B) shows the states of a Linux-process.

The real difference between the two is that Linux will suspend a task when it enters the kernel. An RTEMS task on the other hand will execute until its allocated time-quanta is used up or until a device driver blocks in which case it is marked as blocked.

Interrupt dispatching does not necessarily need to be placed in this layer. It could also be managed by the HAL. The hardware support for interrupt processing is fairly easy to abstract in the HAL. It is a little more difficult to implement interrupt priority levels (IPLs) just above the hardware. The number of hardware IPLs (HIPLs) and the logical IPLs (LIPLs) as seen by the OS may cause problems across platforms. As long as the number of LIPLs required are less-than-or-equal to the number of HIPLs provided by the processor the LIPLs can just be mapped to the HIPLs. Should the number of LIPLs be greater than the HIPLs the mapping becomes more difficult to implement because the

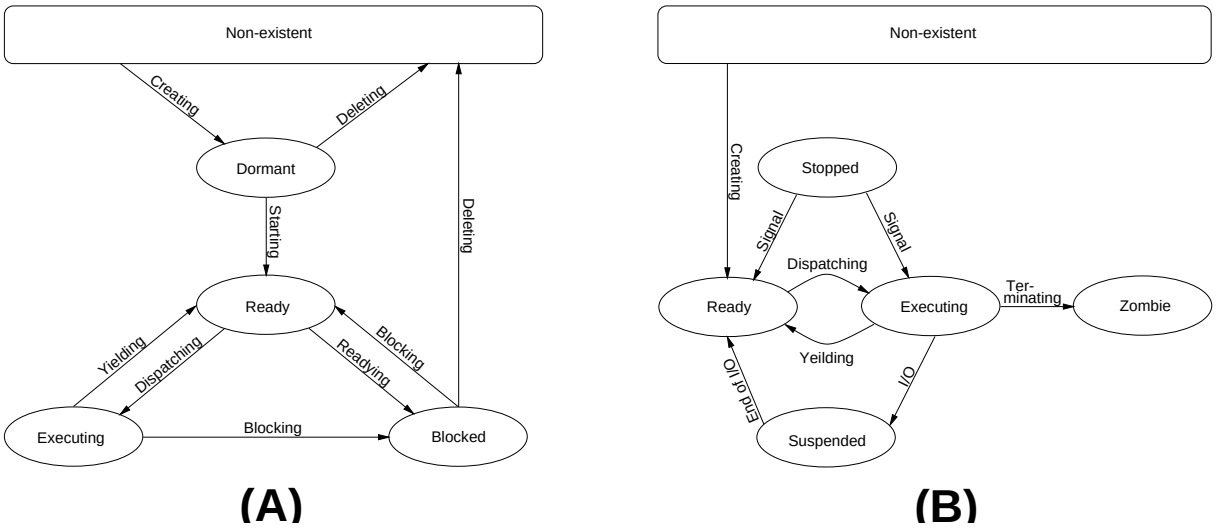

Figure 2.3: (A) The possible states of an RTEMS-task. (B) The possible states of a Linux process

HIPLs are discrete and atomic i.e. integer values, usually ranging from 0-7.

The conclusion would be that the interrupt dispatcher would be placed in the kernel-layer for portability reasons, but if the optimization goal is performance (low interrupt latency) the interrupt management should be placed in the HAL. The ideal solution is to be able to pick either or both at compile-time and thus let device drivers register an ISR according to their own preferences. In order to make this design clean, the interrupt dispatcher would have to register itself with the HAL.

Device-drivers abstracts the operations of hardware devices. The reason that this layer is not placed above and not at the same level as the HAL is that a device does not affect the Processors ability to read and write data to and from I/O ports and memory positions[6].

[6]In fact the HAL could be seen as the device driver in control of the CPU. The author of this text prefers to see the CPU as a synchronized data-operation device (SDOD) i.e. a device that is able to perform operations on data

Device-drivers must at least export the at least the following:

- An interrupt service routine (ISR)
- A device-initialization routine
- Routines for reading and writing data
- Routines for opening and closing the device
- I/O control routines

Operating systems sometimes divide device drivers into block and character devices. A block device is generally a device that supports a file system. Block devices usually exports the same set of routines as character devices but they also export a set of routines used to read and write file blocks. A third set of function can be used to control the behavior of the device. To recapitulate the following three groups of functions can be exported from a device driver:

- Character oriented I/O routines
- Block oriented I/O
- Routines used to configure and control the device

These three groups of functions are usually stored in a specific structure that holds the entry-points to the various functions in the device driver. The structures are in turn stored in a device address table, a table that stores information about all device drivers in the OS. This table is used by the I/O manager (part of kernel) to access the devices on behalf of the service layer. The principle is shown in figure 2.4

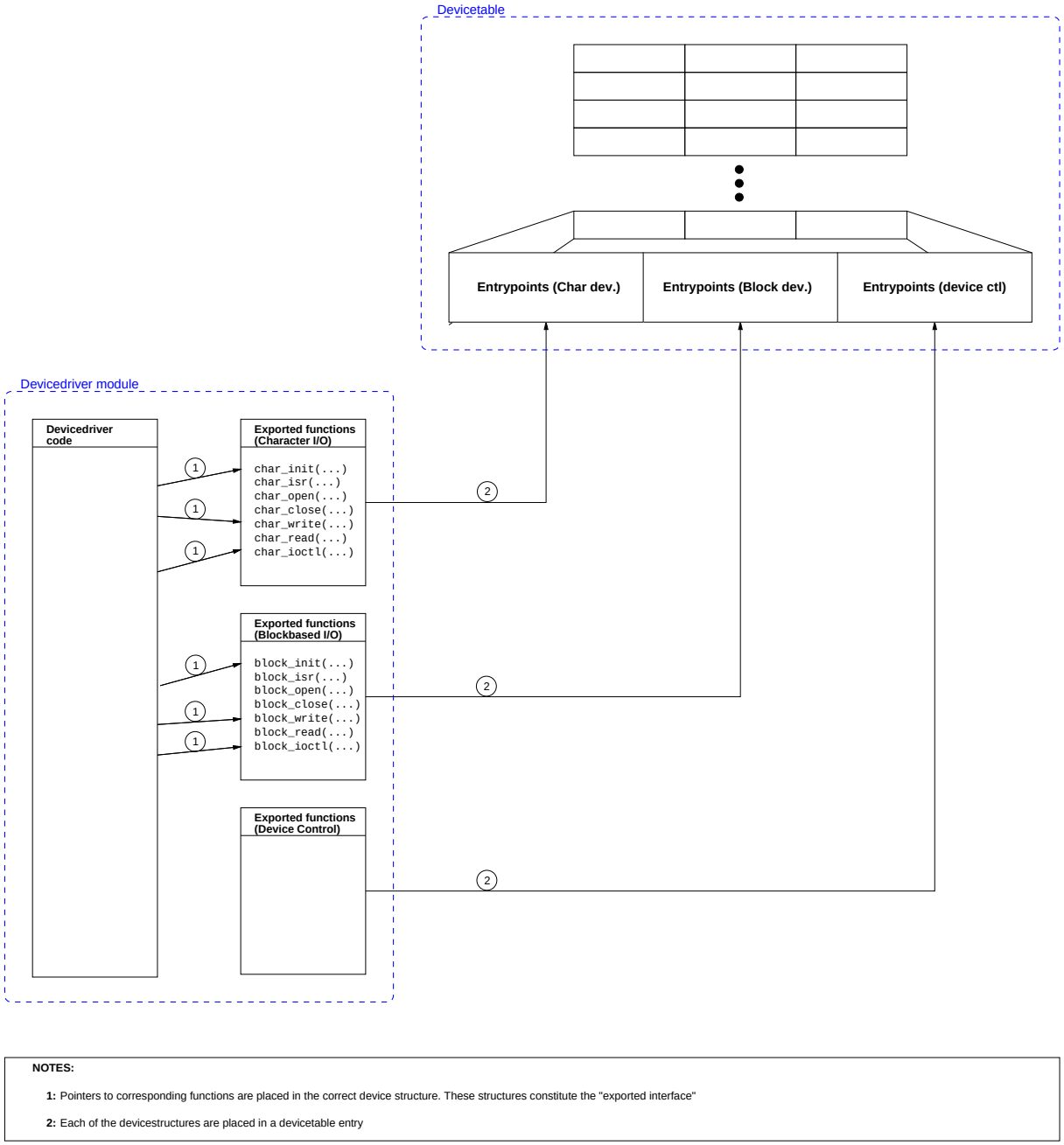

Figure 2.4: Schematic figure of the relations between the data structures that are placed in the device table and a device driver

The OS service layer

The highest level of abstraction in figure 2.2 is the OS service layer. This layer could be divided into two different parts:

System-call interface constitutes the application programming interface (sys-call API). This is the virtual machine that the OS provides to the applications running in user space

High-level services services provided by the OS. These services are used by applications and accessed through the sys-call API

The sys-call API is divided into two parts according to the following:

- A collection of functions linked with the applications. This code is executing in user-space
- An entry point in the kernel, usually an interrupt (trap) handler

The library code is partially CPU-dependent and uses either the HAL or HAL-macros in order to abstract away from the CPU-dependent trap mechanisms. Apart from providing a way to enter the kernel the library code must also make sure that the necessary parameters are passed to the kernel as well as a function-call code[7]. Multitasking OSes also require the library to be reentrant i.e. a process must be preemptable while it is executing library-code.

The trap-handler is responsible for finding the correct system call in the kernel. The mapping can be done through table lookup. The principle of the system call demultiplexing is shown in figure 2.5. This monolithic kernel structure has several benefits and a few drawbacks too. The most obvious ones are:

[7]POSIX uses a standardized encoding of all function calls. Calls are encoded as integers and passed in a register or on the stack

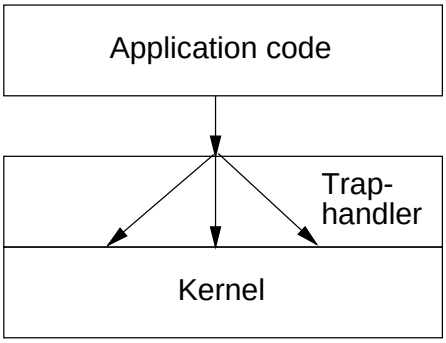

Figure 2.5: Mapping the system-call-number to the internal function.

+ A monolithic kernel is easy to implement since all the functionality of the operating system is placed in one instance executing at the same privilege level

+ Integrity is easy to maintain since all OS primitives are implemented inside the kernel

+ The application programmer is insulated from hardware/software. This makes it easy to reach a high level of abstraction

- Context switch overhead is significant.

- If the kernel fails the system fails and are not easily restarted

- The overhead induced by I/O operations is significant. This means that applications that are I/O bound execute slower

High-level services are provided in two different ways depending on the implementation model. In a monolithic OS all services are part of the kernel. The micro-kernel provides services through tasks running in user-space. The two different modes are shown in figure 2.6

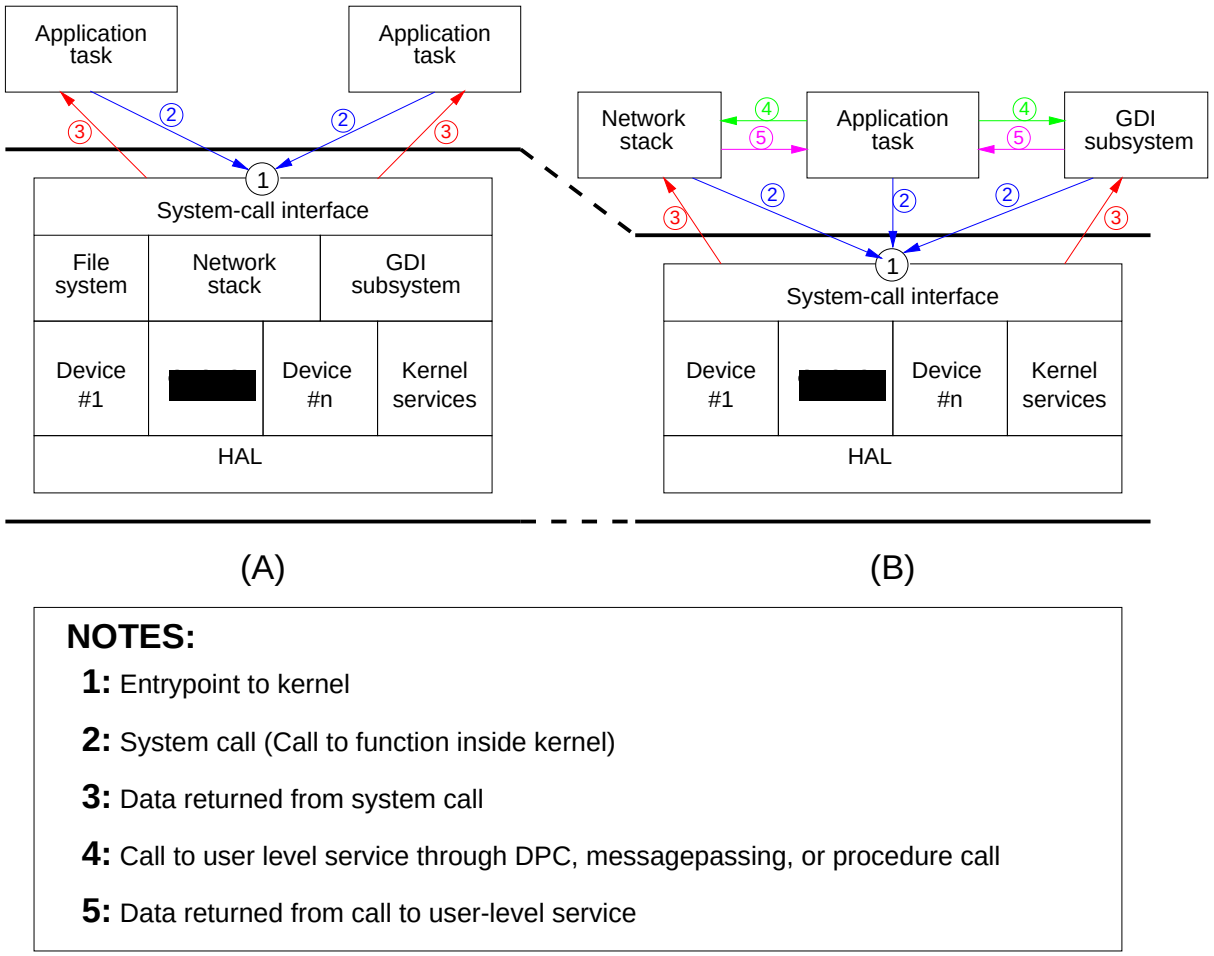

Figure 2.6: Two different ways to provide high-level services: a monolithic kernel (A) and in a micro-kernel (B)

The kernelless OS

In recent years research has been turning to that of kernel-less OS:es. These operating systems usually have some sort of "kernelized structure" or a nano/micro-kernel that are used for initialization and task switching. All other operations such as memory management etc. are done by user level processes. Due to the low level of service provided by kernel itself, communication primitives are needed to provide service to applications since it is not possible to access services in the kernel. These communication primitives can be divided into a number of categories. The most interesting ones are:

- Message passing. Messages are passed to, and-from, tasks. This method has some really intriguing features:
  - It enables event-based processing
  - It supports distributed computing
  - Isolates the private data of shared memory area from the tasks
- DPC - Deferred procedure calls. I.e. the call returns immediately but a synchronization point is placed further "down-stream" in the calling thread.
- Shared memory. Tasks that wish to communicate with each other read and write special memory areas. There is always a certain risk of data-inconsistency with this approach. This risk should not be ignored as it could have severe ramifications.

SPACE is another example of providing IPC. This approach has the following features:

- It uses a generalized exception mechanism X(E,V) where E is the exception number and V is the associated vector
- A minimal "Portal" is provided together with X(E,V). This portal is used to check validity as well as authorization.

The effect reminds a bit of tunneling[8]. Experiments have shown that substantial performance gains (5-7 times) can be reached with this method. Figure 2.7 shows the general principle of space.

Figure 2.7: The tunneling effect of SPACE

2.4 OS-performance issues

The critical goal in operating systems is performance. It can be summarized in a simple paradox. It is impossible to obtain a high level of abstraction without degrading the performance. The reason is that more code is introduced and executed every time a new abstraction layer is added. This means that operating systems are generally undesirable with regard to performance because an application executing on bare hardware will always perform better than it would if it were assisted by an operating system. Other important issues such as portability, correctness, complexity and economic efficiency is not associated with assembly-language programming.

As operating systems have become more and more complex and thus able to provide a higher level of abstraction their performance have decayed. This fact have largely been ignored due to the rapid development of the CPU:s that have become more and more powerful, thus effectively hiding the decline in OS performance that has taken place during the last five-to-ten years.

[8]Quantum physics electron tunneling, that is

2.4.1 Context-switch overhead

There is a certain overhead involved in performing a context switch. This overhead is induced by the following:

- Execution of the trap or software interrupt forces the program counter and some other registers to be pushed to the stack
- Changing the privilege level of the CPU usually takes a few microseconds. The exact number is dependent on the CPU architecture
- Saving the CPU state involves saving all registers access by the code which performed the system call
- Execution of the demultiplex code in the trap-handler.

It is, in most cases, not possible to perform a direct call to the desired function in the kernel mainly because too little is known at the higher levels of a system. A resolver or *demultiplexer* is needed to map system calls to the correct device. This overhead can be substantial. An example provided by [Prop 95.1] regarding a read operation from a UNIX file-system in System 5 Release 4 was found manage the following operations:

trap, sys-call, vnode, ufs-read, inode, uio, trap, page-fault, address-space-fault, vnode-segfault, ufs-pagein, buffer-cache-read, raw-device-switch, and device driver

The above example suggests that seemingly simple operations can have a very high overhead associated with it. This will naturally be smaller in simpler system but the demands on these systems are more stringent.

Another problem that arises when estimating the time spent to perform a context switch is architecture. Different processors provide different mechanisms and may perform certain operations by themselves such as register saving etc. Also the number of registers

varies between different families of CPU:s. RISC CPUs usually have a large number of registers compared with their CISC counterparts. This makes the context-switch in a RISC CPU a more expensive operation because more registers are to be saved in main memory. External buses always run a lot slower (1/5th of the CPU-core) than the CPU-core making the CPU sit idle for some 80% of time required by the context switch.

Another performance problem issue is the processor's cache memory. A level-1 cache running at the same speed as the CPU-core suffers the same performance loss when it is flushed to memory. If

2.4.2 Interrupt latency

Interrupts in a modern processor can usually be divided into two different categories: synchronous and asynchronous. The synchronous interrupts are software interrupts or exceptions and the asynchronous interrupt is usually generated by hardware.

The interrupt latency is the response-time for the particular interrupt. In short it can be said to be the amount of time that passes from the interrupt is generated until the correct service routine begins to execute. This

Synchronous interrupts The synchronous interrupt or exception usually does not have a high overhead associated with them. The latency basically consists of the time it takes to save the program counter and any registers used by the interrupt-handler. If the processor's privilege level remain unchanged the overhead is further lowered. This means that the minimal state to be saved in a context switch is smaller which in turn increases the performance of the context switch.

Asynchronous interrupts Asynchronous interrupts are generated by hardware to signal a need for service. Upon receipt of an interrupt the processor must save minimal state and jump to the interrupt manager. The effect is the same as that seen during a context switch. Long waiting times may result in loss of data or missed deadlines in real-time systems.

Delayed I/O operations

The basic raw I/O operations are sometimes delayed for several different reasons.
I/O operations are sometimes scheduled. This is the case in for instance Windows NT. All I/O operations in NT are delivered to a device driver as an I/O-request packet (IORP). This packet is buffered by the device driver and the operation can thus be serviced at a later point in time.

Another, similar problem, is the overhead associated with I/O operations. Because all operations are associated with the execution of instructions the

Maximizing the performance

As indicated earlier it is very easy to maximize the performance in an embedded system. The general rule is:

Do not use *any* abstractions at all. Execute on the bare hardware.

Following the above rule makes the application very sensitive to changes in hardware. If the base address of a hardware device is moved the application will have to be changed. Major changes to hardware devices such as the introduction of interrupts or DMA operations for data transfers cannot easily be achieved because such changes have a serious affect on application software design. All solutions must be a tradeoff between performance, complexity and ease of implementation. The general rules for maximizing the performance in an embedded system is:

- Use a small decentralized kernel or "kernelized" structures
- Use a simple mid-level language such as C
- Do not rely on dynamic information such function calls et.c.
- Minimize the number of instructions required to read and write I/O (Maximizes I/O bandwidth). Make sure the I/O code is written in assembler and is in-lined by the compiler, thus omitting the need for unnecessary call or jump instructions.
- Minimize the number of registers used and the amount of data that is to be saved during interrupt processing. If possible process as many interrupts as possible at the same time to further lower the overhead
- Minimize the number of changes in privilege-level

Using the optimizing features of the compiler further improves the performance of the code. However the optimizing compiler is useless if part of the code is written in assembly-language. Generally the code generated by a good, optimizing, C-compiler will match the code produced by an average assembler programmer. There are a few exceptions: Architectures with special instruction sets. The most notable example are the SIMD instructions found in DSP processors (Intels MMX is example of SIMD instructions). Many of the DSP-algorithms can benefit from the use of these instructions. It is however not trivial to get a compiler to recognize a certain algorithm (for instance the implementation of a FIR-filter) and thus to generate optimal code for that algorithm.

Further more, the efficiency of the generated code is higly language dependent. The fastest "high" level language is C and there is little doubt that the slowest is Ada. The reason for this has to do with Ada's strong typing, extensive run-time constraint checking, and exception processing just to mention a few reasons.

Chapter 3

Experiment

The purpose of this experiment is to measure the amount of time used to perform various housekeeping chores inside a typical operating system.

3.1 Computing platform

The platform required to execute the experiment should be based on a MC683XX processor. In the absence of such a platform a simulator called BSVC will be used instead. The simulator has the following properties:

- It simulates the a CPU32+ core i.e. a MC68030 CPU. The peripheral devices found in the MC683XX are not simulated.
- UARTs (*U*niversal *A*synchronous *R*eceiver *T*ransmitter) are simulated through a virtual MC68681 DUART (dual UART)
- Timers are simulated through a MC68230 PIT (*P*rogrammable *I*nterval *T*imer) device
- Memory is supported through a RAM-device

BSVC allows various "virtual platforms" to be built. A virtual platform may consist of any number of the simulated devices (SD). Each SD is mapped into the 32-bit memory-space provided by the CPU32+ architecture. The computing platform used to execute the experiment is depicted in figure 3.1

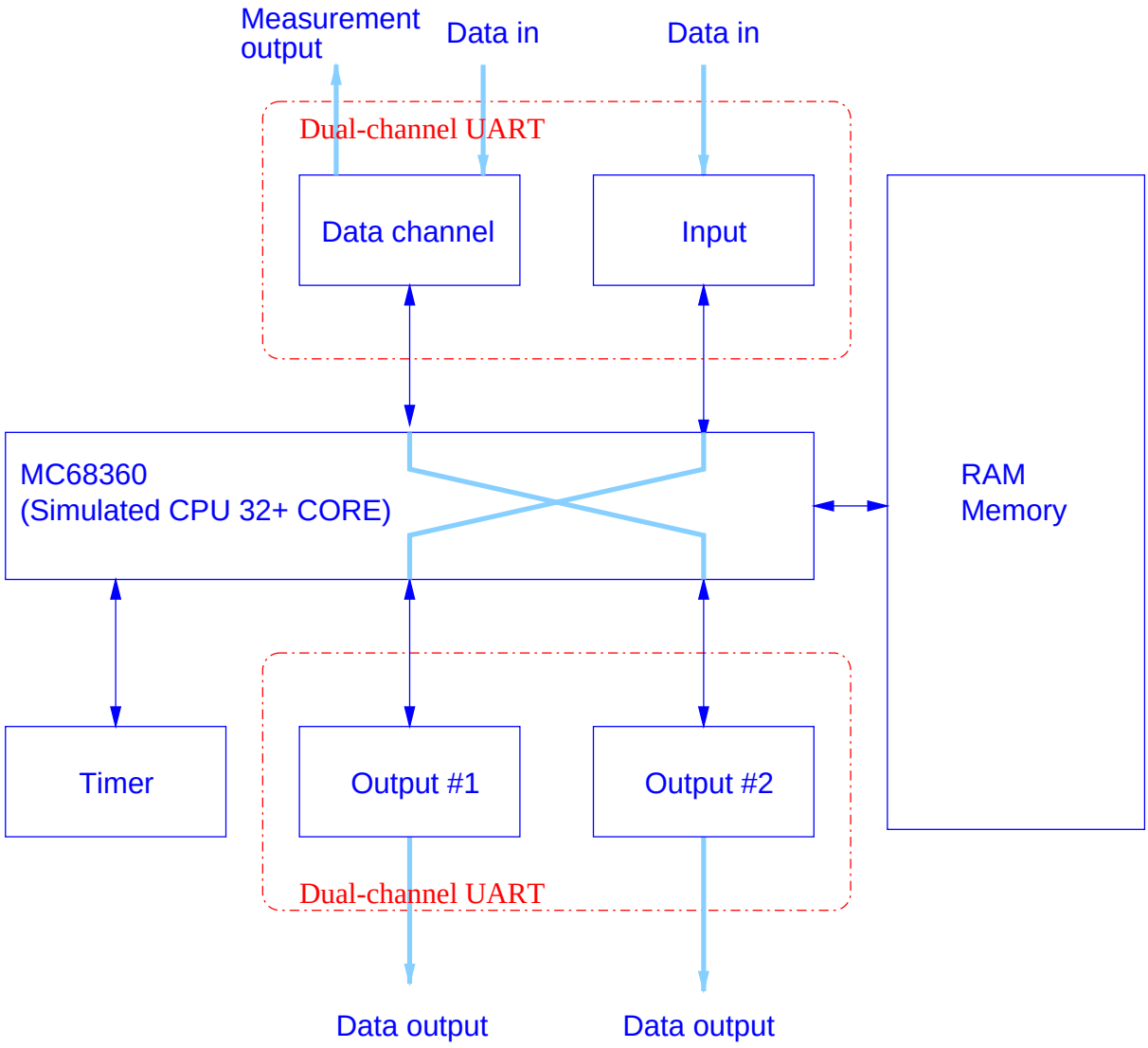

Figure 3.1: Schematic figure of the BSVC-based platform

3.1.1 Inaccuracies

The simulator introduces a few inaccuracies. These are:

- The hardware is ideal. There are no interference or other sources of disturbance.
- Bus errors and double bus-faults are not simulated
- The simulator does not run the software in real time. It keeps its own internal time

The most serious problems are the timing of instructions. This problem is solved by making sure that the load of the machine running the simulator is constant. The exact clock frequency of the CPU is measured by executing a small delay loop and reporting the number of times the loop is executed during a one-second time period. Another serious problem is the fact that bus errors are not simulated correctly. However this is easily solved by not using any of the MC68360 CPUs additional devices such as built in UARTs etc. since these devices are not simulated and are unused there cannot be any bus-errors.

3.2 Experiment requirements

The implementation of the experiment must meet the following requirements:

Req. 1 Generate asynchronous interrupts i.e hardware interrupts

Req. 2 Be I/O bound (measures I/O delays)

Req. 3 Use synchronous interrupts to switch from user-mode to supervisor-mode

Req. 4 It should have some real-time requirements although they should not be too stringent

These requirements are necessary in order to be able to measure the synchronous- and asynchronous interrupt latencies as well as the I/O delays introduced by the HAL. The motivation for using multiple threads are that most systems today are multiprocessing systems and measuring on a single-process system would not be representative because it would not take the overhead caused by the scheduler into consideration simply because a scheduler would not be needed.
It should be noted that multiprocessing is not a requirement in this case. The reason is that the overhead added by multiprocessing would be placed in kernel space in both cases and would thus just increase CPU-load, i.e. load it down. Not implementing a scheduler and multiprocessing capabilities simplifies the design and can be considered a reasonable tradeoff due to the "kernel-time-only implications".

3.3 The experiment

The test-application was chosen so that it would meet all of the requirements set forth in section 3.2.

The task used for the experiment is that of a small router-like system. Data packets enter the system through one UART and are placed in output queues for transmission on channel 1 or channel 2 of the other UART.

- A-kernel based system i.e it uses an operating system (Figure 3.2B)
- A kernel-less system i.e the application runs directly on top of the hardware (Figure 3.2A)

A schematic figure of the two different implementations are shown in figure 3.2. The principal difference is that there are that the system-call interface and the device-driver

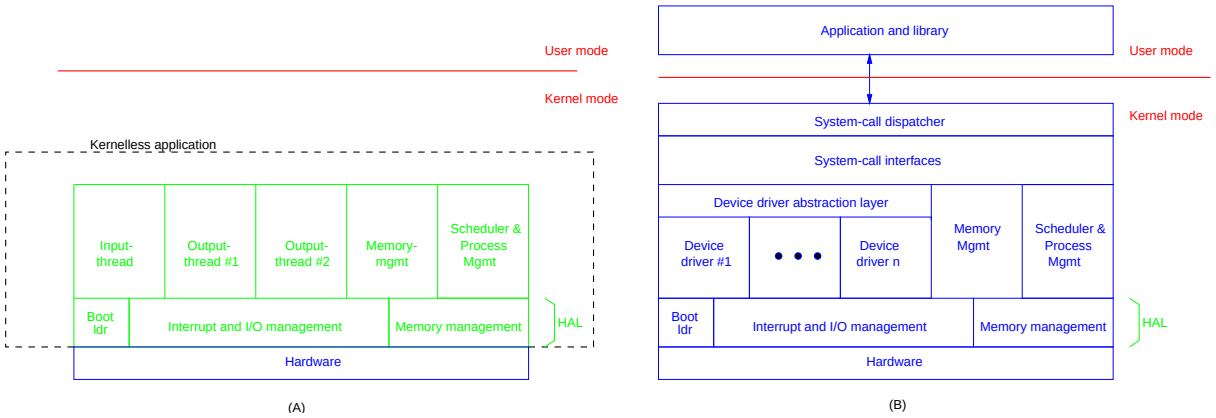

Figure 3.2: The two different implementation models used to evaluate system performance

abstraction layer is only present in figure 3.2B. The goal of the experiment is to measure the amount of time spent executing the code in the two modules.

3.3.1 Software model

Figure 3.2 shows two possible ways of designing a system. The software needed to control the behavior of the test platform fulfills the five requirements described earlier according to the following:

The UARTs generate interrupts when the receive buffer is filled. This interrupt is generated by a hardware device and is thus asynchronous. A high interrupt latency will cause data to be lost because the receive buffer will overflow (**Req. 1 & Req. 4**).

Transmission and reception of data requires the software to access I/O devices. Two Universal Asynchronous Receiver/Transmitters (UARTs) are used for receiving and transmitting data. Receipt and transmission are essentially just a primitive I/O-operation(**Req. 2**).

Data from the input channel should be placed in the input queues for the two transmitters (**Req. 3**).

3.4 Gathering data

Data is returned from the simulator using a the output of serial channel 1. The data returned by this port is parsed by a Linux application and can then be graphed. These graphs are used to evaluate the overhead in the system. There are three different time-quantities to be measured:

- The time spent in user mode. This measurement is initiated every time code running in user mode is executed (denoted t_u)
- The time spent in I/O functions or device-drivers. This is the time that passes from entry to exit of a device read and write function (denoted t_{io})
- The time spent in the operating system. This is the elapsed spent in supervisor mode (denoted t_k)

The kernel-less application will not be able to measure the time spent in user-mode simply because there is no user mode code to execute.

The collected data is transmitted to the evaluation process in chunks. This transmission is triggered by a certain software trap from the application. All timers are reset before a new cycle is initiated.

3.4.1 Measurement accuracy

There are a few inaccuracies in the collected data. These inaccuracies have two different sources:

- There is a small quantization error introduced by the counter. The smallest amount of time that can be measured are $\approx 1\ \mu Sec$
- Preparation for reading is not identical in all cases. This is due to the compiler-generated prolog- and epilogue code.
- Inaccuracies in the measurement system

In order to compensate for the third entry in the system the measurement system will have to add or subtract a certain amount of time from each sample in order to compensate for the time required to obtain the timing values. This is done in the following steps:

- A processor calibration routine is run at initialization. The results are transmitted and used to calculate the equivalent clock-frequency of the CPU.
- The number of cycles required to start and stop timers are calculated based on the assembly-code that constitutes the timing mechanisms.
- A global compensation value is used to compensate for time spent in the interrupt manager.

This protocol easily be implemented on a single-task system. When a timer is started the compensation value is preset to value representing the number of clock-cycles spent to setup the timer. Every timer-interrupt updates the compensation value by adding the number of CPU-cycles required to *completely return* to the main thread. Finally the stop routine adds its own compensation value. The compensation value is recorded and transmitted to the analyzing task which can then calculate a fixed compensation value based on the frequency found by the CPU-calibration.

3.4.2 Measurement interpretation

The entire goal of this experiment is to measure the overhead introduced by an operating system. The data is interpreted according to the following rules: (Accumulated time for a

chunk is represented by a capital "T")

Rule 1: The total execution time for a thread is $t = t_u + t_k$ where $t_u = 0$ if there is no kernel

Rule 2: The overhead of the kernel based system is $t_{oh} = t_k - t_{io}$

Rule 3: The equivalent of t_u in the kernel-less system is $t_{eq} = t_k - t_{io} \neq t_{oh}$

Rule 4: Total execution time for the kernel based system is $T = \sum_{i=0}^{n} t_i$ where n is the number of chunks of returned data

Rule 5: Total execution time for the kernel-less system is $T = \sum_{i=0}^{n} t_{k_i}$

Rule 6: Performance gain for a kernel-less system is given by $G = \frac{T}{T}$

3.4.3 Timing constants

As mentioned earlier the constants to be used to compensate for the overhead of the measurement system must be calculated by hand once the code is written. This is easily done by simply adding up the number of CPU-cycles required to execute each instruction in corresponding functions. The following three constants are calculated:

- The overhead required to execute a *timer_setup()* function (denoted c_{start})
- The overhead required to execute a *timer_stop()* function (denoted c_{stop})
- The overhead required to execute the timer-ISR routine (denoted c_isr)

The timers are essentially just global variables stored in the timing modules. These values are updated according to the following rules:

Setup: $T_{now} =< TIMER > +c_{setup}$ where <TIMER> is the value read from the hardware timer and c_{setup} the estimated time it will take to return to the next function in the calling process

Stop: $T_{acc} =< TIMER > -T_{now} - c_{stop}$ where c_{stop} is the estimated time to it took to call and calculate the current value

Chapter 4

Results

Chapter 5

Conclusion

References

[And 91] Anderson, Thomas E. et al: *The Interaction of architecture and operating system design*, ASPLOS IV, pages 108-120, April 1991

[Bara 96] Barabanow, Michael & Yodaiken, Victor: *Real-time Linux*, New Mexico institute of technology, 1996

[Bers 94] Bershad, Brian N et. al: *SPIN - An Extensible Microkernel for application-specific Operating System Services*, Dept. of Computer science and Engineering, University of Washington, Seattle, Feb 1994

[Bers 96] Bershad, Brian N. et. al: *Dynamic Binding for an Extensible System*, Dept. of computer science and engineering, University of Washington, Seattle, 1996?

[Card 98] Card, Remy et. al. *The linux kernel book*. Chichester, England: John Wiley & Sons Ltd 1998

[Dijk 68] Dijkstra, Edsger W: *The structure of the "THE"-multiprogramming system*, Communications of the ACM, Vol 11, pages 341-346, May 1968

[Eppl 98] Epplin, Jerry: *Linux as an embedded operating system*, Embedded systems Programming, Miller Freeman Inc., October 1997

[Humm 92] Hummel, Robert L. *The Processor and coprocessor*. Emeryville, California: Ziff-Davis Press 1992

[Moto 96] Motorola Semiconductors, *CPU 32 reference manual*, Motorola Inc 1996

[OAR 98:1] OAR Corporation: *RTEMS C user's guide Edition 4.0.0* , On-line Application Reasearch Corporation 1998

[Prop 95.1] Propert, Dave et. al: *Building fundamentally extensible application specific operating systems in SPACE*, University of California, Samta Barbara, Ca 93106, 1995

[Prop 95.2] Propert, Dave et. al: *Impementing Operating systems without kernels*, University of California, Santa Barbara, Ca 93106, 1995

[Silb 9x] Silberschatz, avi & Galvin, Peter: *Operating system Concepts*, 5th edition , Addison wesley 199x

[Solo 98] Solomon, David A: *Inside Windows NT*, 2nd edition, Microsoft Press 1998

[Thek 94] Thekkath, Chandramohan A.& Levy, Henry M: *Hardware and software support for efficient exception handling*, University of Washington Seattle, 1994

Appendix A

Experiment documentation and comments

TODO: Go through all the code (≈ 220 files) and write this section.

Appendix B

EM340 platform documentation and design notes

This is a design proposal for an embedded platform suitable to run VCOS or any other embedded OS that can be ported to Motorola's CPU32 architecture.

B.1 Features

The proposed hardware platform is based on the Motorola 68340 CPU. It is an extensible design. The main features are:

- 1 MC68340 CPU
- 1M word flash ROM
- 512K word static ram
- 2 on-board serial interfaces (EIA-232)
- 2 8-bit ports for keyboard interfaces
- 1 Standard DMC-LCD[1] suitable for displays based on the standard Hitachi chipset and interface
- 1 extension I/O port
- 1 BDM[2] port
- 1 JTAG[3] port

The basic platform provides the necessary hardware capabilities to run advanced embedded applications. A schematic figure of the platform is shown in figure B.1

B.1.1 Changes needed to VCOS

In order to run VCOS on the EM340 platform the following changes to VCOS are needed:

- New initialization and boot code is needed to boot and initialize the MC68340 CPU.

[1]DMC/LCD=Dot Matrix Character Liquid Crystal Display
[2]Background Debugging Module
[3]Joint Test Action Group. A hardware specification used to test and program Programmable logic that has already been soldered to a circuit board

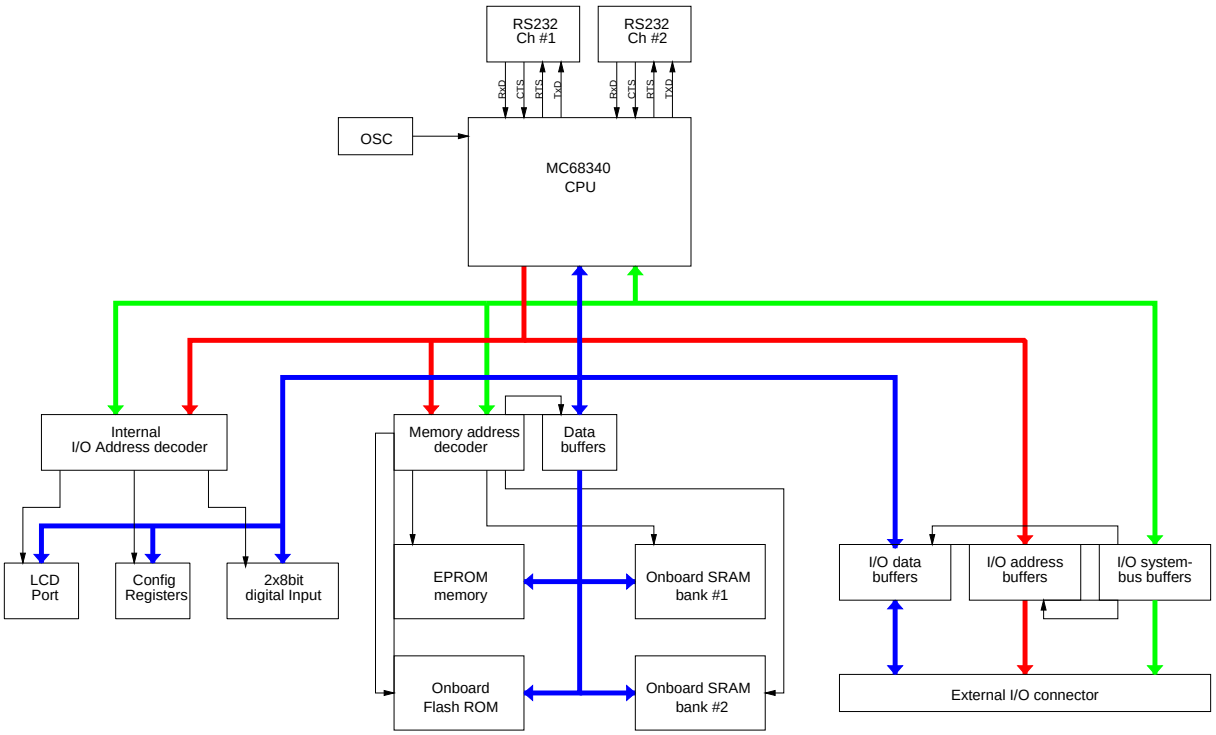

Figure B.1: Schematic figure of the EM340 platform

- The RS232 device-driver needs to be changed in order to support the MC68340s built-in UART.
- Addition of exception handlers for Bus-errors and the double-bus faults.

Apart from the changes to VCOS a few changes are needed to the linker scripts:

- The base address and the size of the ROM-region must be specified
- The base address and the size of the RAM-region must be changed

B.2 Schematics

The rest of this appendix shows the schematics for the EM340 system. A brief description of each page is given below: (Referenced by drawing #)

EM340-9901-01 This is the MC68340 CPU, the oscillator and the core CPU subsystems

EM340-9901-02 Buffers for the memory and I/O databuses

EM340-9901-03 Flash ROM bank #0

EM340-9901-04 Flash ROM bank #1

EM340-9901-05 Address decoding for the memory system

EM340-9901-06 RS232 interface (signal conversion)

EM340-9901-07 SRAM bank #0 (Todo)

EM340-9901-08 SRAM bank #1 (todo)

EM340-9901-09 SRAM bank #2 (todo)

EM340-9901-10 SRAM bank #4 (todo)

B.3 PAL code documentation

This section contains the code needed to program the two PAL[4] ICs placed on the EM340 testboard.

Code note included yet, will be soon

[4] *P*rogrammable *A*rray *L*ogic

B.4 PCB layout masks

This section contains the PCB-layouts for the 4-layer printed-circuit board on which the EM340 system is assembled *≈30 hours work when once the platform design is finished!*

B.5 Design notes: EM340 platform

This section contains the design notes for the EM340 platform. These notes are needed in order to be able to write any code at all.

TODO: Write the rest of this document. It should include documentation of all bits in hardware registers, addresses of the on-board I/O devices etc.

Appendix C

Implementation proposal for an embeddable microkernel