AN EMBEDDED SYSTEMS KERNEL

Lars Munch Christensen

IMM-THESIS-2001-47

IMM

Trykt af IMM, DTU

Foreword

The present report is the result of a master thesis entitled “An Embedded Systems Kernel”. The project was done from mid February until the end of October 2001.

I would like to use the opportunity to thank all the parties who have contributed to this project. A special thank you goes to my wife Eva, who has used valuable time finding spelling and grammar errors in the report. I would also like to thank MIPS for sponsoring hardware and thank you to the people at the linux-mips mailing list for valuable MIPS information.

October 26th, 2001.

Lars Munch Christensen

Abstract

The process of composing a development system environment, suitable for embedded system development in a Free Software environment, is discussed. The theory of protection and sharing of memory in a single space operating system is presented. A design for a small embedded systems kernel is presented and the actual implementation of the kernel is described. A generalized bootstrap is proposed. The actual implementation of the kernel is included in the appendix.

Keywords

Embedded systems kernel development and implementation, single address space operating systems, generalized bootstrapping.

Contents

List of Figures

List of Tables

Chapter 1

Preface

1.1 Executive summary

The present report is the result of a master thesis entitled "An Embedded Systems Kernel". The process of composing a development system environment, suitable for embedded system development in a Free Software environment, is discussed. The theory of protection and sharing of memory in a single space operating system is presented. A design for a small embedded systems kernel is presented, the actual implementation of the kernel is described and a generalized bootstrap is proposed. The actual implementation of the kernel is included in the appendix.

The kernel developed is released under the GNU General Public License. The reason for this decision is that I want to allow people to use it freely, modify it as they wish and then give their ideas and modifications back to the community.

1.2 Prerequisites

The prerequisites for reading this report is a common knowledge of operating system kernels and operating systems in general. Terms such as, remote procedure calls and virtual memory should be familiar to the reader.

A basic knowledge of C programming, MIPS assembler and the use of the GNU development tools is preferable. Finally, some basic understanding of standard PC hardware will come in handy.

1.3 Typographical conventions

The following typographical conventions are used throughout the report:

Italic

  is used for the introduction of new terms.

`Constant width`

  is used for names of files, functions, programs, methods and routines.

Chapter 2

Introduction

This chapter contains an introduction to embedded systems and to the project itself. The chapter finishes with a section describing the motivation for this project.

2.1 Introduction to the embedded systems

An embedded system is a combination of computer hardware, software and and perhaps additional mechanical parts, designed to perform a specific function. A good example is the microwave oven. Millions of people use one every day, but very few realize that a processor and software are involved in preparation of their dinner.

The embedded system is in direct contrast to the personal computer, since it is not designed to perform a specific function but to do many different things. The term general-purpose computer may be more suitable to make that distinction clear.

Often, an embedded system is a component within a larger system. For example, modern cars contain many embedded systems; one controls the brakes, another controls the emission and a third controls the dashboard. An embedded system is, therefore, designed to run on its own without human intervention, and may also be required to respond to events in real-time, for example, the brakes has to work immediately.

2.2 Introduction to the project

An important concern, in the development of kernels for operating systems or embedded systems in general, is portability across different hardware platforms. Most kernel subsystems, including the ones that are machine dependent, are written in high level languages such as C or C++. As a result, very little machine dependent assembly code needs to be rewritten for each new port. But, writing a kernel in a high level language is not enough for a kernel to be easy portable. If all the machine independent code is mixed together with the machine dependent, you still have to touch most of the kernel code in the porting process.

More recently, the notion of nanokernels[11] has been introduced representing the virtual hardware support for the rest of the machine independent kernel. This project strives to create a small nanokernel and a few subsystems for use in embedded systems. The kernel subsystems will therefore have a clean interface to the nanokernel.

The problems concerning coldboot will be analysed with the goal of reducing dependencies to the hardware to as little as possible.

If coldboot is neglected the embedded system can be considered as one program with more activities. There will only be one activity, when the program starts, and this activity will be executed without restrictions in privileges. The creation of activities should be expressed by means of the nanokernel's routines, and both voluntary and forced process switch should be supported.

The concrete goal for the project is to implement a nanokernel and some subsystems, exercising it so far that an embedded system is able to coldboot and use a simple external device. The project should also provide a useful basis for further work.

2.3 Motivation for the project

There are several motivations for the project both personal and educational.

My personal motivation for the project is a long time interest in kernel development and operating systems. To get the opportunity and time to build a kernel is absolutely the best way to learn practical embedded systems implementation.

The educational motivation was to try and create a very small kernel, providing only the necessary features for use in an embedded system with parallel processes.

Perhaps the most important motivation was to start up a kernel development project, on which several different kernel related projects could be based. This project is the first project in a, hopefully, long series of projects concerning the construction of nanokernels for embedded systems.

2.4 Organization

The report contains 12 chapters, two appendixes and an annotated bibliography. The 12 chapters are divided into four parts. The first part that consists of chapters 1 through 6, contains introductory contents. Chapter 7 presents single space operating systems. Chapters 8 and 9 contains the design of the kernel and the boot process. Chapter 10 contains a description of the kernel implementation and chapter 11 describes the current status of the kernel. The report finishes in chapter 12 with a conclusion.

Chapter 2 you are reading it.

Chapter 3 describes the properties that the kernel were given before choosing hardware and before going into a detailed kernel design.

Chapter 4 describes the process of choosing the right hardware for the development of the kernel. The different hardware, which where considered, will be described.

Chapter 5 contains a description of the hardware used in this project. This includes a description of the main board, the CPU and the test bed used for development.

Chapter 6 contains a description of the software used in the implementation of the kernel. This includes the compiler toolchain, the debugger and the considerations done when choosing development tools.

Chapter 7 describes Single Address Space Operating Systems (SASOS). It begins by introducing single address space operating systems with comparison to the traditional multiple address space operating systems. After this introduction three different single address space operating systems are discussed.

Chapter 8 describes the kernel design. All major components of the kernel are described, that includes the timer, the synchronization mechanisms, the interrupt handling and scheduling.

Chapter 9 describes bootstrapping in general and then gives an introduction to boot loaders. This is followed by a description of what happens, the moment after the Malta system has been powered on. The chapter finishes with a description of, how bootstrapping a kernel is done in practice on the Malta system.

Chapter 10 describes the kernel implementation. The main focus will be on, how to interface with the hardware, since this subject has been the most time consuming part of the kernel implementation.

Chapter 11 first gives a short overview of kernel status, as of this writing. After this the future development of the kernel is described.

Chapter 12 contains the conclusion.

Throughout the report, I have eliminated minor details to make it more readable, but in some cases small details may have taken significant time to figure out or solve, these will then be described thoroughly. This will, hopefully, save future project-students a lot of hair- pulling. The report is also written in a way that enables future students to make a jump start to continuing work on the kernel project.

Chapter 3

Kernel properties

This chapter describes the properties that the kernel were given before choosing hardware and before going into a detailed kernel design.

3.1 Introduction

Before going into a detailed kernel design some general kernel properties have to be given. Some of these properties are made from personal preferences while others are made for pure educational purposes.

The idea of these kernel properties are to narrow down the huge number of possibilities, one is faced with when designing a kernel for an embedded system.

3.2 Kernel properties

All embedded systems contain a processor and software, but they also have other features in common. In order to have software, there must be a place to store the executable code and a storage for runtime data manipulation. This storage will take the form of RAM and maybe also ROM. All embedded systems also contain a kind of input and output system. Figure 3.1 shows a generic embedded system.

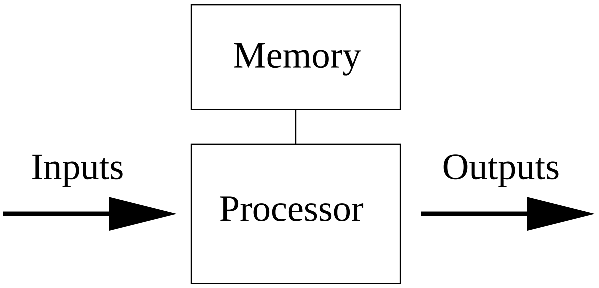

Figure 3.1: Generic embedded system

The kernel developed in this project will take form of a generic embedded system and will strive to be the smallest common kernel for embedded systems.

When choosing a language, in which the kernel should be implemented, there are several choices. It could be implemented in ADA, Java, C++ and several others. I choose to implement it in C and assembler. The motivation for implementing the kernel in C is that C, more or less, has become the standard language in the embedded world, and free C compilers exists for almost all platforms.

The following list describes the properties, the kernel strives to follow:

Micro kernel structure The kernel can be considered as one program with more activities. This is almost the same as saying that the kernel has a micro kernel structure, in the sense, that a micro kernel also has several activities running as separate processes. The Minix[17] kernel is divided into I/O tasks and server processes. In this kernel there will be no real difference in these processes besides their priority, so to be able to differentiate between these processes, a process controlling a device will be called a driver, and a process doing a non-device related task, will just be called a task. If the term process is used, it includes both drivers and tasks.

Stack based context switch When changing from one process to another the context should be saved and restored by manipulating the stack. Each process will have its own stack and use this to save and restore the context. This will be discussed further in the "Kernel Design" (chapter 8). The kernel will only run in one address space, so after a context switch we will still be in the same address space but in a different process. This type of context switching is very similar to the principles used in coroutines.

Message passing To communicate between two processes the concept of message passing should be introduced and a simple *send* and *receive* mechanism will be used to implement this. The semantics of these will be very similar to the ones used in the Minix kernel.

Semaphores Since the kernel has several processes running, it is feasible to introduce the concept of shared memory between the processes. A common way, to get mutual exclusion to shared memory, is by introducing semaphores.

Scheduling The scheduler should be simple and the interface to the scheduler should be generic. This will enable one to write a completely different scheduler, without dealing with architecture-specific issues and without changing the nanokernel. The scheduler itself should be kept as simple as possible and is not considered as the important part of this project.

Modularized design The kernel itself will not maintain the protection between processes. Instead protection will be introduced by using an modularized design in the kernel. Different solutions to the problem will be discussed and one will be implemented.

Global exception handling Using exceptions in an embedded system, to handle failures in a modular manner, could be of great advantage in bug-finding and system recovery. Different methods for doing exceptions in C will be analysed.

Portability Portability is also an important property of the kernel. Implementing the kernel as a nanokernel is definitely a huge step in the right direction. But other things such as the size of pointers and the addressing should be paid attention. The use of assembler should be kept at a minimum.

C Compiler requirements The kernel will be licensed under the GPL license, which is the license of the GNU project. Releasing code under the GPL and using a non-free compiler could lead to licensing problems. A requirement will therefore be that the compiler is also under a free software license. The obvious choice could be the GNU compiler collection (GCC), but other compilers under GPL compatible licenses could also do. This choice creates some restrictions in possible hardware choices, since not all platforms are well supported by a GPL compatible compiler.

3.3 Summary

This chapter has listed several properties to the kernel, the tools used in the development, and to what should be of concern in the analysis and design phase of the kernel. Some relation exists among these kernel properties and some may argue against each other, but this is unavoidable. The chapter has also defined a basis for the kernel to the extent that feasible choices of hardware and software used for the implementation can be made.

Chapter 4

Choosing hardware

This chapter describes the process of choosing the right hardware for the development of the kernel. The different hardware, which has been considered, will be described.

4.1 Introduction

With the previous defined kernel properties in mind, it is now possible to choose hardware for the project. The different requirements to the hardware can be summed up to:

The price It is a personal wish that the price of the development hardware for the embedded system is low. The motivation for this is that everyone interested in using the kernel should be able to get the hardware without being ruined. Having cheap development equipment motivates using it in all kinds of devices, such as home build MP3 players.

Single board computer The development hardware has to be in the category of single board computers. A single board computer is a small motherboard with a processor, some memory and input/output devices. Many single board computers also contains network adapters, USB and other peripherals.

Fast stack operations Since the kernel is going to have a microkernel structure, it is crucial that the stack operations on the single board computer runs at a decent speed. If not, the kernel will run too slow and be unusable. Fast stack operations are often a matter of good access speed to memory.

Free tools available Development tools for the given hardware have to come with a free software license, which is compatible with the GPL license, the kernel is released under.

In the following the four different single board computers, which have been investigated, are described.

4.2 Intel 8051

Despite its relatively old age, the 8051 is one of the most popular microcontrollers in use today. Many of the derivative microcontrollers that have been developed since, are based on and compatible with the 8051. The 8051 is used in everything from DVD-drives to smartcards.

The 8051 is an 8 bit microcontroller originally developed by Intel in 1980. Now it is made by many independent manufacturers. A typical 8051 contains a CPU with boolean processor, 5 or 6 interrupts, 2 or 3 16-bit timer/-counters, a programmable full-duplex serial port and 32 I/O lines. Some models also include RAM or ROM/EPROM.

Single board computers with an 8051 integrated come in many shapes and normally cost at most 100$.

Since, it is a widely used microcontroller, there are also a lot of development-tools for this microcontroller. Of the free tools available, the SDCC, Small Device C Compiler project[27], looks the most promising.

After talking to a long time 8051-developer, the conclusion was that it is not suitable for developing a small microkernel, which is heavily based on stack usage. This is due to the fact that the 8051 compiler does not use the stack to save parameters to functions, as we know it from e.g Intel's i386 systems. If we did use the stack anyway, the result would be slow and not usable.

4.3 Atmel AVR 8-Bit RISC

Atmel has a series of AVR microcontrollers that have an 8 bit RISC core running single cycle instructions and a well-defined I/O structure that limits the need for external components. Internal oscillators, timers, UART, analog comparator and watchdog timers are some of the features, that are found in AVR devices.

The AVR instructions are tuned to decrease the size of the program, whether the code is written in C or Assembly does not matter. It has on-chip in-system programmable Flash and EEPROM, which makes it possible to upgrade the embedded software, even after the microcontroller has been implemented in a larger system.

To do development on the AVR, a viable choice would be to buy the STK500 development kit [2], which costs around 100$. This development kit includes the AT90S8515 microcontroller, which has 8Kb of flash memory but only .5Kb RAM.

The development kit comes with all necessary tools for developing software for the microcontroller, but GCC also have very good support for all the different AVR microcontrollers.

The price and the development tools fulfill the requirements, but the AVR is too limited in FLASH and RAM. The RAM can be extended but only with SRAM, and SRAM is very difficult to find, since it has been replaced with newer types of RAM.

4.4 Atmel AT91 ARM Thumb

The Atmel AT91 microcontrollers are targeted at low-power, real-time control applications. They have already been successfully designed into MP3 players, Data Acquisition products, Pagers, Medical equipment, GPS and Networking systems.

Atmel's AT91 ARM Thumb microcontrollers provide the 32-bit performance every 8-bit microcontroller user is dreaming of, while staying within a tight system budget. The AT91EB40 Evaluation Kit[3] costs around 200$ and includes the AT91R40807 microcontroller. This microcontroller has a 16 bit instruction set, 136Kb of on-chip SRAM, 1Mb of flash, 32 programmable I/O lines, 2 UART's, 16 bit timers, watchdog timers and many other features.

The GNU Compiler Collection also have a port of their tools for this microcontroller. Red Hat has even ported their real-time kernel eCos [28] to this microcontroller, so the community support for this microcontroller is good.

This microcontroller definitely fulfills all the requirements given to the hardware. It is cheap, it has the right tools, it has enough memory to do a lot of stack operations, and it has a wide community support.

4.5 MIPS Malta development board

The MIPS processors are widely used in the industry and comes in many shapes. MIPS has several development boards, where the MIPS Malta development board is the most comfortable system to develop embedded kernels on.

The Malta board, which is used in this project, contains the 64 bits 5Kc MIPS CPU with 16x64Kb cache. This may be a more powerful system, than originally intended for this project. The CPU is so powerful that Infineon Technologies chose to use it in their specialized local area network switching applications. The MIPS Malta development board will be described further in the next chapter.

MIPS supports the free software community very well, and it is even possible to get a Linux kernel running on the Malta board. The GCC is also ported to both MIPS32 and MIPS64.

This system does not fulfill the price requirement of being a low budget system, since the price is approximately 3000$, but it is definitely a nice system to develop on. It has all the right tools for development and as the AT91, it has a wide community support.

4.6 Summary

This chapter has discussed the different single board computers, which have been investigated thoroughly for this project. The choice in hardware fell on the MIPS Malta development board with 64 bit CPU. It was chosen, even though the system did not fulfill the price requirement of being a low budget system. But, who can say no to a free 64 bits MIPS system?

Chapter 5

Hardware

To be able to explain the specific implementation of the kernel in the following chapters, an overview of the hardware is given. The level of detail in the hardware description is just enough to understand some hardware specific implementation issues. This hardware description includes the main board, the CPU and the test bed used for development.

5.1 The Malta system

The Malta system is designed to provide a platform for software development with MIPS32 4Kc- and MIPS64 5Kc-based processors. A Malta system is composed of two parts: The Malta motherboard holds the CPU-independent parts of the circuitry, and the daughter card holds the processor core, system controller and fast SDRAM memory. The daughter card can easily be swapped to allow a system to be evaluated with a range of MIPS-based processors. It can be used stand-alone or in a suitable ATX rack system. The daughter card used in this project is the CoreLV card and it is described below.

Malta is designed around a standard PC chipset, giving all the advantages of easy-to-obtain software drivers. It is supplied with the YAMON ("Yet Another MONitor") ROM monitor in the on-board flash memory, which, if required, can be reprogrammed from a PC or workstation via the parallel port. YAMON contains a lot of nice features like RAM configuration, PCI configuration, debug interface and simple networking support. The YAMON ROM monitor will be described further in chapter 9.

The feature set of the Malta system extends from low-level debugging aids, such as DIP switches, LED displays and logic analyzer connectors, to sophisticated EJTAG debugger connectivity, limited audio support, IDE and flash disks and Ethernet. Four PCI slots on the board give the user a high degree of flexibility enabling the user to extend the functionality of the system.

5.1.1 The CoreLV

As mentioned above the daughter card is a MIPS CoreLV[6]. The card contains several components, and how they interact is roughly shown in the block diagram on figure 5.1. The two main components are the Galileo System Controller[4] and the MIPS64 5Kc CPU.

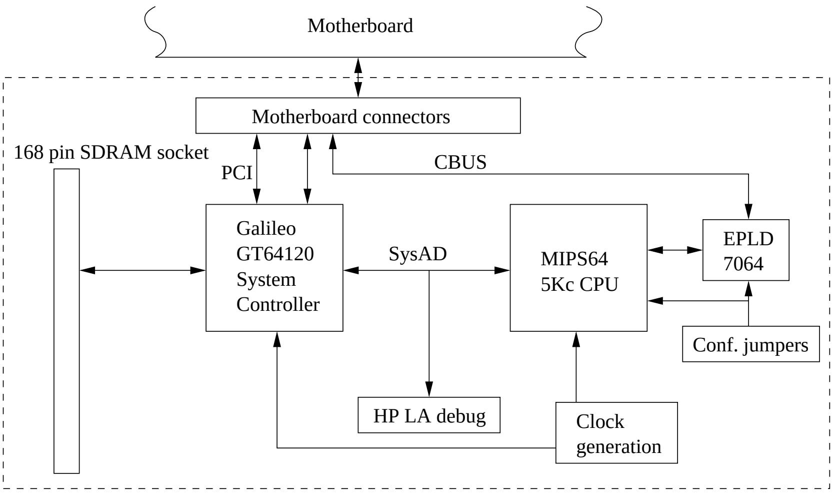

Figure 5.1: Overview of the CoreLV card

The Galileo is an integrated system controller with three different interfaces and is especially designed for MIPS CPUs, including 64bit MIPS CPUs. Galileo's main functions in the CoreLV device includes:

- Host to PCI bridge functionality.
- Synchronous DRAM controller and host to SDRAM interface. The SDRAM controller support an address space of 512Mb, but only 64Mb is installed in the test equipment. The SDRAM type has to be PC100 RAM.
- Device bus interface. The device bus from the Galileo is modified in the EPLD component on the Core card to provide the CBUS, which is used for access to Boot Flash, Flash memory and peripheral devices as LED's and switches places on the motherboard.

The Galileo is connected to the CPU bus (SysAD), which allows the CPU to access the PCI and memory buses.

It should be noted already here that due to a bug in the Galileo chip, all register contents are effectively byte-swapped in big-endian mode, which should be taken into account.

The CPU mounted on the CoreLV card is a MIPS64 5Kc[7] CPU, which is a 64-bit MIPS RISC microprocessor core that is designed for high-performance, low-cost and low-power embedded systems. The CPU executes the MIPS64^{TM} instruction set architecture but also provides 32-bit compatibility mode, in which code compiled for MIPS32^{TM} processors can run unaltered.

Features of the 5Kc CPU include:

- Two pipelines. One six-stage integer pipeline and a separate execution pipeline for multiply and divide operations. The two pipelines operate in parallel.
- System Controller Coprocessor (CP0). This is responsible for virtual-to-physical address translation and cache protocols, the exception control system and the operating modes: Kernel, Supervisor, User and Debug.
- Cache Controller. The cache controller supports several different cache protocols, write around, write through and write back. Write around is the same as disabling the cache.

The Memory Management Unit (MMU) in the 5Kc CPU provides a 64-bit virtual address space, subdivided into four segments. Two for the Kernel mode, one for Supervisor mode, and one for User mode. To provide compatibility for MIPS32 programs a 2^{32}-byte compatibility address space is defined. For further information on the MMU refer to 5Kc Processor Core Datasheet[8].

5.1.2 The motherboard

The motherboard contains several components, and how they interact are roughly shown in the block diagram on figure 5.2. From the CoreLV card there are three interfaces to the motherboard, of which only the PCI and CBUS interface are shown on the figure. The third interface is a I^2C bus, which is not used in this project.

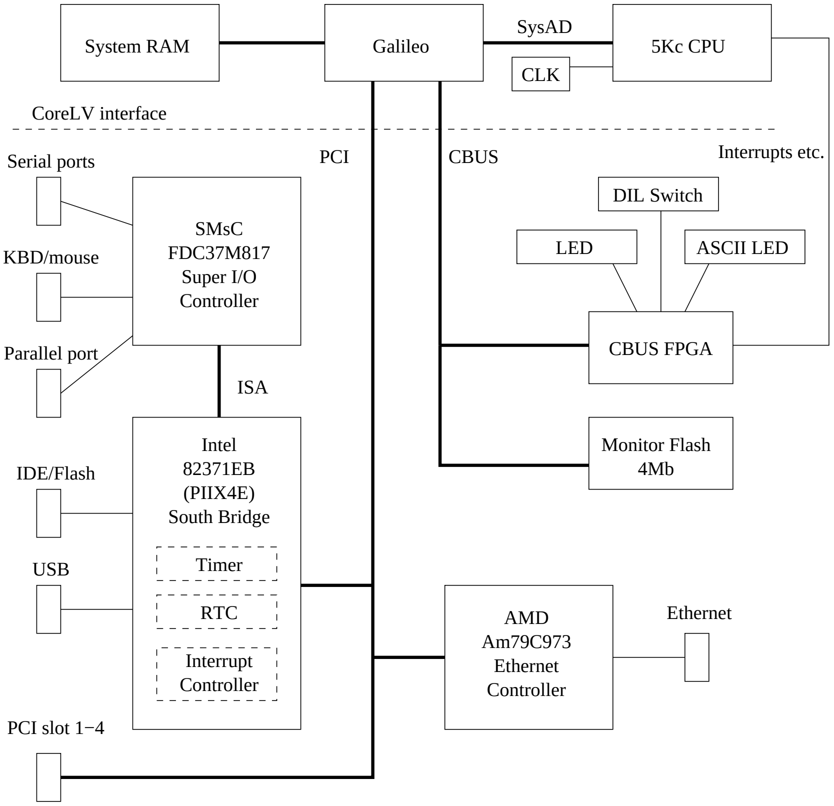

Figure 5.2: Overview of the motherboard

The PCI bus is connected to a PIIX4[5] multi-function PCI device, an on-board ethernet device, and of course to the four PCI slots. The PIIX4 is a standard Intel chipset, found on many modern PC motherboards. It implements PCI-to-ISA bridge function, a PCI IDE function, and a Universal Serial Bus host/hub function. If a Compact Flash is installed, this chip is also able to control this device through the IDE interface.

To the ISA bridge of the PIIX4 a Super I/O Controller from SMsC[1] is connected. This I/O controller contains functionality to control input devices, such as keyboard and mouse, as well as standard serial and parallel ports.

The CBUS exists to allow the CPU to access peripherals, which have to be available before the CPU bus is configured, for instance, the flash memory YAMON is booting from. The CBUS is also used for those peripherals that require simple, low-latency access, e.g. the ASCII display.

The largest difference from using peripherals on the MIPS Malta and on a standard PC is that all devices are memory mapped. This really eases the task of controlling hardware tremendously. The physical memory mapping is shown on table 5.1. In some memory areas the mapping depends on the implementation of the CoreLV card and of the software configuration of these areas, but the table shows a typical configuration.

Base address	**Size**	**Function**
0000.0000	128Mb	Typically SDRAM
0800.0000	256Mb	Typically PCI
1800.0000	62Mb	Typically PCI
1BE0.0000	2Mb	Typically system controllers internal registers
1C00.0000	32Mb	Typically not used
1E00.0000	4Mb	Monitor flash
1E40.0000	12Mb	reserved
1F00.0000	12Mb	Switches, LEDs, ACSII display, soft reset, FPGA revision number, CBUS UART (tty2), General purpose I/O, I^2C controller
1F10.0000	11Mb	Typically system controller specific
1FC0.0000	4Mb	Maps to monitor flash
1FD0.0000	3Mb	Typically system controller specific

Table 5.1: Malta physical memory map

5.2 Test bed

Figure 5.3 shows the development test bed used for kernel development. The workstation is connected to a LAN and has a TFTP server installed, on which the kernel is placed. From the workstation to the Malta system is a serial line used for remote debugging facilities included in YAMON. The Malta system is also connected to the LAN, and is, with help from YAMON, able to download and run the kernel served on the TFTP server. Finally, there is also a serial line connecting the Malta system with an old vt220 terminal. This terminal is used as console output, and to interface and control the YAMON monitor. The serial line connected to the old terminal could just as well be connected to the workstation, but due to the lack of a second serial port in the workstation the good old terminal came in handy again.

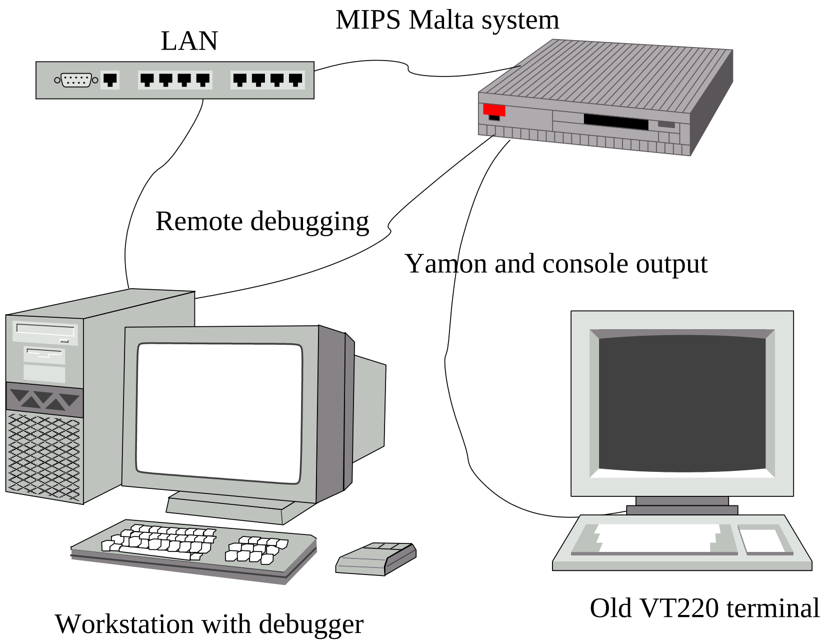

Figure 5.3: Development test bed

5.3 Summary

This chapter has given a short description of the hardware, which should be sufficient to understand the kernel implementation. The main focus has been on how the different components interfaces, and where devices are mapped in memory. The chapter also described the test bed used for kernel development.

Chapter 6

Software

This chapter contains a description of the software used in the implementation of the kernel. This includes the compiler toolchain, the debugger and the considerations done, when choosing development tools.

6.1 Introduction

As mentioned earlier, the obvious choice for a C compiler is to use the C compiler included in GCC (GNU Compiler Collection). This may sound easy, but as it turns out, it is very difficult to find a good version of the compiler for the MIPS architecture. The problem is that, there are so many different versions, and every developer is using his own patched version of the toolchain. There is no central place, where patches are gathered, so it is a difficult job to collect information about creating a good working toolchain.

Another problem is that, when a new version of the GCC is released, it does not have MIPS as it primary target, and it will, most likely, not compile for this architecture without patching. So, the option to select the latest and greatest release, could lead to problems.

6.2 The different toolchains

In the following some of the most important toolchains will be described. A toolchain includes a cross-compiler, linker, assembler and sometimes even a C library.

Hard Hat Linux Monta Vista[22] is a company, which specializes in embedded Linux distributions and development kits. They have a version of their Hard Hat Linux distribution that runs on the MALTA board with a MIPS 32 processor.
Monta Vista supplies cross-development toolchains with their product for MIPS 32 and for both little- and big-endian architecture. All of Monta Vista's cross-development packages come in forms of RPM packages.

Linux VR project Linux VR project[23] is a project that brings the Linux operating system to NEC VRSeries devices, most of which were originally designed to run Windows CE. The NEC VRSeries devices all contain MIPS processors.
The project developers have created a set of RPM packages that even includes the C library. The difference compared to all the other toolchains is that this toolchain uses soft floating point. More about this below.

SGI MIPS project SGI MIPS project[25] is SGI's project to create a Linux distribution for their MIPS based workstations, like the Indy. The SGI MIPS project has more or less become the centerpoint for all Linux-MIPS development, and a lot of valuable information can be received by joining their mailing list.
SGI MIPS project has created a nice collection of RPM's for doing cross-development to both MIPS32 and MIPS64. The toolchains are based on a rather old version of the C compiler, namely the EGCS compiler, which is now merged with GCC. Because it is old, it is well tested and easy to install, and all relevant patches are included in the RPM as well.

RedHat GNUpro This is RedHat's[24] commercial version of the GNU toolkits. Even though it is not free, it is worth mentioning this toolkit. The toolkit includes support for a lot of different platforms, including MIPS 32/64. One really nice feature of the compiler toolchain is that you can choose between little-endian and big-endian, and between MIPS 32 and MIPS 64 ABI (Application Binary Interface) as a compile option. In the normal GCC toolchain you will have to have a different toolchain for each architecture. Another great thing is the graphical debugger interface to gdb, see 6.4 chapter. Besides a lot of great features, you will also get support, if you buy this product.

Instead of getting a pre-compiled cross-development toolchain, you can build the toolchain yourself, as mentioned earlier, this could very well lead to problems, but it is possible. The information on actually doing this, is very sparse, and the official cross-compiler HOWTO has not been updated for several years.

If the latest toolchain, for some reason, is needed for this kernel project, the trick is then to first build binutils (ld, gasm etc.) and then only enable the C language when building GCC. There is then no need for the C library, which is not used for this kernel development anyway.

For this project I chose to use the pre-compiled RPM from the SGI Linux project. There are several reasons for this; first of all, they are well tested, so most problems are known, secondly, they are also build for MIPS64, and I would really like for the kernel to run in 64 bit mode, and thirdly, it is easy to get support for compiler problems. It should be noted already here that the MIPS64 linker is very broken, but that there are solutions for this.

6.3 Floating point

The Malta board does not contain a floating point processor, and this could potentially lead to problems, if floating points are used. There are three solutions to this, of which the two first are the most common:

1. Create floating point emulation in the kernel. Every time a process uses a floating point instruction, the system traps to the emulator in the kernel. This has become the most common way to solve the problem in the Linux world.
2. Use the emulated floating point in the C library. This is the option called `-msoft-float`. This does require the C library to be especially build with soft floating point. Using the C library is not a good idea for kernel development, since the C library is huge and therefore not recommended to compile into a kernel for small embedded systems.

3. Use the emulated floating point from the small C library newlib. Newlib is a small C library created especially for embedded systems, this library can be build to emulate floating point and is small enough to include in a kernel. More about newlib below.

I have solved the problem simply by not using any floating point operations at all. If floating point, for some reason, is needed for this kernel, I would recommend using newlib, since it is much easier to integrate than a real kernel floating point emulator, and you get the benefit of the rest of newlib as well, i.e memory copying functions, string comparing functions etc.

6.4 Remote debugging

Since the MALTA board supports remote debugging, one might as well take advantage of this. A debugger is not a part of the SGI Linux project cross-development toolchain, so this should be retrived elsewhere.

One option is to use the nice debugger from the GNUpro package, if one has already invested in the GNUPro package, see figure 6.1. It has a graphical interface for viewing registers, stacks, memory and source code. The graphical interface is build on top of the GNU debugger and is very usable.

Another option is to use standard GNU debugger gdb, which is free. It may not have a nice graphical user interface, but it works just as well. There exists free graphical frontends for gdb, but these have not been investigated. The only downside to gdb is that, you have to build it yourself, but compared to building GCC, this is an easy job.

Using a debugger for kernel development does not come without costs. There must be some kernel support for the debugger, otherwise, you will only be able to execute the kernel through the debugger and nothing else. See “Kernel implementation” (chapter 10) for more information about remote debugging.

6.5 Newlib

As mentioned above, newlib[26] is a C library intended for use in embedded systems. It is a collection of several library parts, all under the GPL license.

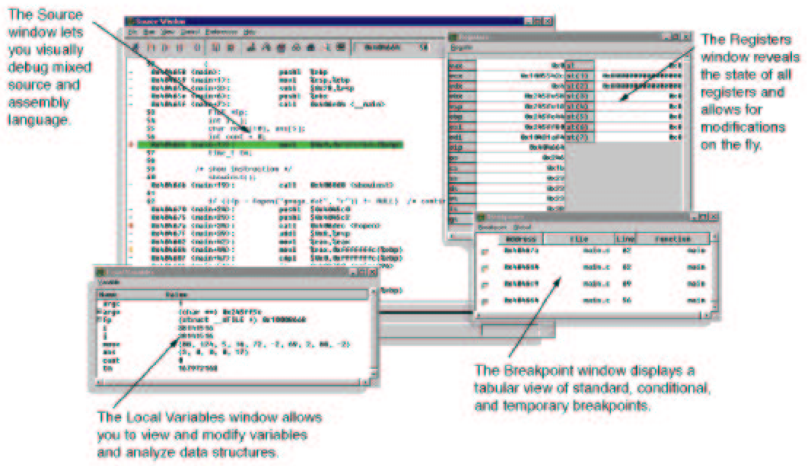
The Source window lets you visually debug mixed source and assembly language.
The Registers window reveals the state of all registers and allows for modifications on the fly.
The Breakpoint window displays a tabular view of standard, conditional, and temporary breakpoints.
The Local Variables window allows you to view and modify variables and analyze data structures.

Figure 6.1: GNUPro debugger

In being a C library, it contains usefull functions for kernel development, especially the string functions `memset` and `strcpy`, which most likely will be required in the kernel.

As a part of newlib, there is a library called libgloss. Libgloss contains code to bootstrap kernels and applications for different architectures including MIPS.

In this kernel project only small code snippets of the newlib have been used. In future work newlib would be a good thing to include, especially if the kernel is going to by ported to another architecture, since most of the functions in newlib has been tested on a variety of different platforms. Also libgloss could save you from writing the bootstrap code all over again.

6.6 Summary

This chapter has described the different tools for doing MIPS kernel development and argued which tools to use. It also gave a small description of the very usefull library, which is used to some extend in this project. Now it is time for some real work.

Chapter 7

SASOS

This chapter describes Single Address Space Operating Systems (SASOS). It begins by introducing single address space operating systems with comparison to the traditional multiple address space operating systems. After this introduction three different single address space operating systems are discussed, namely Angel, Opal and Mungi. The focus will be on the sharing and protection of memory between processes in the single address space operating system. The three single address space operating systems are very similar in the mechanisms they use for sharing and protection of memory. Therefore, the first system described, which is Opal, will be used as a reference model when discussing the last two single address space operating systems.

7.1 Introduction

As described in "Kernel Properties" (chapter 3) the context switch between two processes will be a stack based context switch. That is, when changing from one process to another, the context switch should be done by manipulating the stack as described in chapter 3. The address space is, therefore, the same before and after a context switch, hence, the kernel will only run in one address space.

Running several processes in the same address space could result in strange behavior or system crashes in an embedded system, if there is nothing to prevent a misbehaving process from writing in another process' memory. It would be even worse in a multiuser operating system, if there were no protection between processes, because it would be impossible to give different privileges to different users of the system. Another issue is finding bugs and recovering from a process failure. If a process writes data in some place, where is was not supposed to, there will be no warning from the system and the bug would be very hard to find. It would also be impossible to recover from this situation, since the system will give no warning, when the process begins to misbehave.

Because these problems with single address space operating systems are also valid in this kernel project, it was natural to research solutions to protecting processes from each other. There have been several attempts to create Single Address Space Operating Systems (SASOS) and three of these will be described in the following.

Before examining the concepts of a single address space operating system, it is useful to review the multiple address space approach[12], where every process has its own private address space. The major advantage of private address spaces are:

1. They increase the amount of address space available to all programs.
2. They provide hard memory protection boundaries.
3. They permit easy cleanup when a program exits.

The disadvantage of this approach is that the mechanism for memory protection, which is isolating a program within a private virtual address space, is an obstacle for efficient communication between two protected processes. Especially pointers have no meaning outside a process memory protection boundary and the primary communication mechanisms rely on copying data between private virtual memories. The address translation between two private virtual memories can be calculated fast, but the copying is expensive.

The common communication choices between processes are to exchange data through pipes, files or messages, and neither choice is adequate for programs requiring high performance. Most modern operating systems have introduced facilities for shared memory, for example in Linux there are two methods for sharing memory, namely System V IPC and BSD mmap. However, the mix of shared and private memory regions does introduce several problems; private data pointers are difficult to handle in a shared memory region, and private code pointers cannot be shared.

Single address space operating systems avoid these problems by treating a single virtual address as a global resource controlled by the operating system, just as the disc space or the physical memory is a global resource controlled by the system. With the appearance of 64-bit address space architectures the need to re-use addresses, which is required on 32-bit architectures, is eliminated. A 32-bit address space may be enough for a single address space embedded system not requiring that many resources, but for general purpose systems, 32-bit is no longer sufficient as a single global virtual address space.

The main goal of single address space systems is to enhance sharing and to improve performance of co-operation programs. The problems with a mix of shared and private memory regions in multiple address systems can, in fact, be avoided in single address space operating systems without sacrificing the previously mentioned advantages of multiple address space systems. That is, a SASOS will still be able to:

1. provide sufficient address space without multiple address spaces due to the use of 64-bit architectures.
2. provide the same protecetion level as the multiple address space's system.
3. cleanup after a process without adding complexity to this action

There are, of course, also several tradeoffs in a single address space system. For example, the virtual address space is managed as a global system resource which has to be used fairly and this requires accounting and quotas. Another example is that a process' memory region may not be continuous in the address space. There are a lot of pros and cons for both single and multiple address space systems, but these will not be discussed futher. In the following the main focus will be on, how the single address space operating systems implements the sharing and protection of memory between processes.

7.2 Opal

Opal[12] is an experimental operating system developed at the University of Washington, Seattle. The purpose of Opal is to explore the strengths and weaknesses of the single address space approach. Opal is built on top of the Mach 3.0 microkernel.

The fundamental mechanisms used for management of the single address space are described in the following.

In Opal, a unit of protected allocated storage is called a *segment*. A segment is, in essence, a contiguous set of virtual pages and the virtual address is permanently set by the system at allocation time. The smallest possible segment is one page, but segments are allocated in bigger chunks by the system, to allow continuous growth of the data contained in the segment.

In Opal, all processes are called *threads*, and a *protection domain* is an execution context for threads, which restricts their access to a specific set of segments at a particular instant in time. Many threads may execute in the same protection domain, see figure 7.1. The Opal protection domain is very similar to a process on the Linux platform, except that protection domains are not a private virtual address space.

The resources, protection domains and segments, are named by *capabilities*. A capability is a reference that grant permission to operate on the resource in a specific way. Given a segment capability an execution thread can explicitly *attach* that segment to its protection domain, and thereby permitting the thread to access the segment directly. The opposite is also possible, a thread can *detach* a segment from a protection domain, and thereby deny access to the segment. The attach request can specify a particular access directly to a segment, for example read-only access. The attach request can only request the rights that are permitted by the capabilities at a given segment.

The attach request is very similar to Linux's BSD mmap system-call for mapping files into a process, except that in Opal, the system, rather than an application, always chooses the mapped address. Another difference from mmap is that in Opal all segments are potentially attachable, given the right capabilities, so no data is inherently private to a particular thread.

To enable communication from one protection domain to another, a *portal* is used. A portal is an entry point to a protected domain and can be

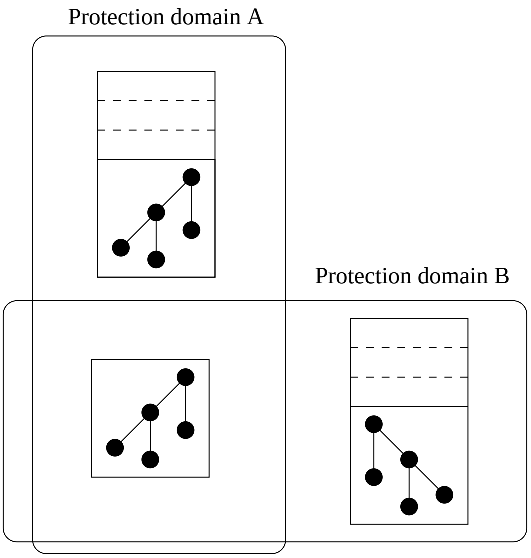

Figure 7.1: Opal threads can be placed in overlapping protection domains and more than one thread is able to run in each protection domain.

used to implement servers or protected objects. Any thread that knows the existence of a given portal, can make a system-call that transfers the control into the protected domain associated with the portal. The name space for portals is global in Opal and allows the exchange of data during uses of a portal through shared memory. The result is that there is no copying of data in communication between protection domains.

The key point in the Opal's handling of protection and sharing of memory is the use of protection domains, where a group of threads in a protection domain, can communicate in a protected and controlled manner by attaching and detaching segments. If communication has to be done with threads in another protection domain, portals are used. The portals are essentially the same as a remote procedure call, where the data is passed along through the use of shared memory segment between the two protection domains, as shown in figure 7.1, where a thread is running in a temporarily overlapping protection domain.

7.3 Angel

Angel[13] is a single address space operating system developed at the City University of London. Angel was developed after a study on how to address some of the problems with the two microkernels Topsy and Meshix:

- The Meshix operating system exhibited poor performance, especially in the message passing system.
- It was difficult to extend the base system to provide more complex services.
- The UNIX environment proved too restrictive as a research platform.

Adaption of the Meshix platform could not address these problems and a radically different operating system structure was required. The result was a single address space microkernel named Angel.

Angel is in many ways similar to Opal and many of the design ideas are also a direct derivation of Opal's design. Angel has a similar concept of protection domains, as the one previously described in the Opal system, which is that a protection domain is an execution context for threads, see figure 7.2. For some reason Angel groups protection domains together and calls this for a process. This grouping serves no real purpose and is somewhat misleading, since a protection domain is very similar to a normal UNIX process.

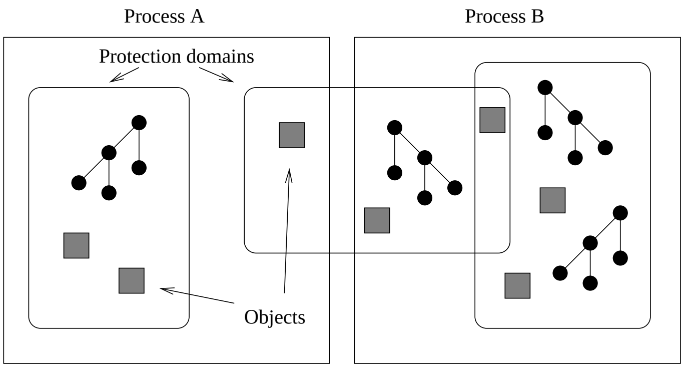

Figure 7.2: Protection domains in Angel

The protection in Angel is provided on *objects*, which consist of one or more pages of virtual memory. Objects cannot overlap, nor must they be contained within other objects. As with Opal, the system manages the objects and not the applications themselves. The semantic of an object differs from segments in Opal. An object in Angel is an instance of C++ class, whereas a segment in Opal was merely a chunk of memory which could be used by a thread in a protected manor.

The consequence of using objects instead of segments, is that, every time a new instance of an object is created, it is assigned with capabilities and explicitly protected by the system, as the segments are in Opal. This may seem like a nice and dynamic solution compared to Opal, but the result is a lot of unnecessary management of objects that are not shared. Another issue is that if an object is an instance of a data structure, which is able to expand, it would not be expanded continuously in the virtual memory.

Even though this fine grained management of object does reduce the performance, it does provides the ability to create very advanced management of the objects. Angel takes advantage of this, by allowing the possibility of creating dependencies between the capabilities of object. For example, expressing that one object is not accessible, before another is also accessible.

The communication between the protected domains are in essence the same as in Opal, but instead they are called light-weight remote procedure calls.

The key point in the Angel's handling of protection and sharing of memory is the use of objects with associated capatilities in protection domains and the protection domains are controlled by the system instead of by the processes themselves.

7.4 Mungi

The final system to be discussed is Mungi[13]. Mungi is the first real native implementation of a SASOS on standard 64-bit hardware. The previously discussed systems, Opal and Angel, are both proof of concept implementations and have not been able to fully demonstrate the potential of a SASOS. Mungi is built on top of the L4 microkernel and is developed at The University of New South Wales' Department of Computer Systems.

Mungi is very similar to Opal, even the type of capabilities, it uses, are the same. The only thing that is different, in the design of protection and sharing, is that objects are used instead of segments. Due to the great similarity to Opal, Mungi's design of protection and sharing will not be covered in detail.

It should be noted though that the actual management of objects by the system is somewhat simplified compared to the management used in Angel. This is definitely a good decision since, what is gained by having a single address space should not be lost in a complex and time consuming object management.

Another thing, which should be noted, even though it is off topic in this chapter, is that Mungi has been performance tested very thoroughly and the result has shown a vast improvement in performance compared to traditional multiple address space systems. The most significant improvement was with database operations.

7.5 Summary

Even though this kernel project, as of this writing, does not have any mechanisms to protect one process from another, it is interesting to see how other kernel projects have solved this problem in a single address space operating system. As it will be described briefly in "Kernel design" (chapter 8), there are other options than using protection domains for creating protection and sharing of memory between processes, though some of the other options will not provide the same level off protection as the operating systems described in this chapter.

The solution to protection and sharing of memory in the discussed systems has been to use protection domains and a mechanism similar to remote procudure calls to communicate between threads in different protection domains. This is done with a heavy use of the virtual memory mechanisms provided by the hardware. This indicates that this is the best known method to do protection and sharing in a SASOS, without sacrificing the level of protection.

The major difference in the three systems lie in, how they actual manage the protected domain. This management has not been discussed in detail since, it was not the primary focus of this chapter. Whether one version of the protection domain management is better than the other is very difficult to conclude. Personally, I liked Mungi the best, due to its very clean and simple way to manage objects in its protection domains. Mungi also seems to have combined the best from Angel and Opal into one system.

Personally, I feel that there is a need for research on mechanisms for protection and sharing of memory in a SASOS without using virtual memory. Even though the main motivation for designing a SASOS was the huge virtual address space, I am sure that small real-time systems, running on limited hardware, could benefit from this research.

Chapter 8

Kernel design

This chapter describes the kernel design. All major components of the kernel are described, that includes the timer, the synchronization mechanisms, the interrupt handling and scheduling. The chapter finishes with a brief analysis of exceptions in C, but first an overview of the kernel is given.

8.1 Kernel overview

The kernel is not going to be designed to solve specific tasks, instead the design aims to make the kernel general within the previous mentioned kernel properties in chapter 3. General means that the kernel is going to include the common features of an embedded systems kernel. These features can then be tuned for specific purposes in future use of the kernel.

As described in the kernel properties chapter, the kernel should have a micro-kernel-like structure that is, a small kernel with several kernel subsystems running as separate processes, and where processes are able to communicate with each other and with the kernel. Besides having a micro-kernel structure the design also strives to fulfill the following areas:

- Separate the process management and scheduling completely from the hardware dependent code. This serves two important purposes: first, you do not have to touch the process management and scheduling code, if you want to port the kernel to a different architecture, and secondly, you can easily change the scheduler without having to modify strange assembly routines.

- The processes in the kernel could range from drivers controlling the ethernet, subsystems implementing an IP stack and processes, which would normally be running in userspace with lower priority. The last is very unusual from normal micro-kernels but also very powerful in embedded systems, for example, if some calculation is more important to get done in time, it may have to have a higher priority than a driver. This is not be possible in a system like Minix without modifying the kernel.

- Build the processes around a nano-kernel. This has become a common way for constructing modern micro-kernels[15]. More on this below.

- Build the kernel as a single address space kernel without using the memory management unit. The advantages of this is, as described earlier, that the message passing can be done very fast. Another important issue is that many micro-controllers, like the previous mentioned AT91, do not have a memory management unit at all, so the kernel has to seek other methods for protecting the different processes from each other.

The definition of a nano-kernel is not unambiguous, thus there is no list of components, which are allowed in the nano-kernel and what hardware that has to be abstracted in the nano-kernel.

Common components of the nano-kernel[15] is:

Boot component responsible for booting and initializing the system.
Interrupt handler responsible for handling interrupts and activation of the scheduler.
Scheduler responsible for doing scheduling decisions.
Boot console responsible for console output at boot time.
Debugger component responsible for debugger hooks in the kernel.
Interface component responsible for providing a single interface for accessing the hardware.

The problem is where to draw the line between the nano-kernel and the processes and what hardware to create an abstraction layer for in the nano-kernel. For example, it makes no sense to abstract a PCI bus with a general bus interface, since the PCI bus is used the same way whether implemented on a PowerPC, MIPS or Intel platform. On the other hand, it makes perfect sense to abstract I/O to devices in the nano-kernel, since I/O to devices is not the same on the Intel platform and the MIPS.

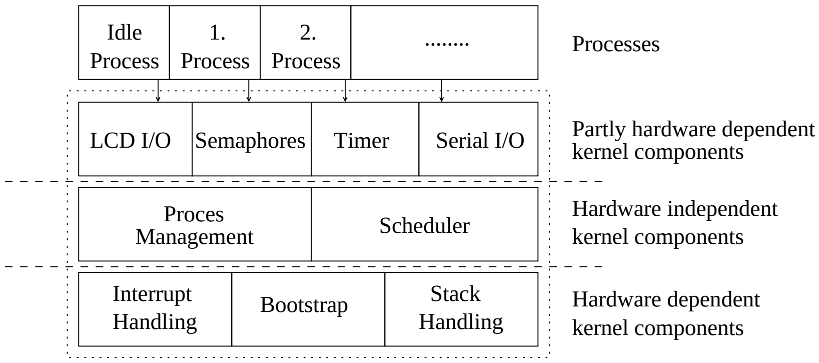

Figure 8.1: Overview of the kernel

On figure 8.1 an overview of the kernel is shown. The dotted line delimits the nano-kernel and the small arrows denotes function-calls from the processes to the nano-kernel. As shown on the figure a process only interfaces the kernel through the I/O interfaces and the services provided by the Timer and Semaphores components.

The nano-kernel components are divided into three different groups:

Hardware independent kernel components These components are written in C and should be portable without changing the code.

Partly hardware dependent kernel components These are the components written in C but they still depend somewhat on the hardware. If implemented carefully the components could be portable between platforms.

Hardware dependent kernel components These are the components that have to be implemented in assembly code.

It could be argued that the Serial I/O, as well as the Timer component, should not be in the nano-kernel. Serial I/O is included for simplicity, because the boot console is part of that component. If this component eventually becomes a full featured serial driver, it should be moved out of the nano-kernel into its own process. The Timer components have been kept in the nano-kernel for performance issues, because when a timer interrupt occurs, it should be handled as fast as possible. A closer look at the Minix kernel revealed that it requires several hacks to circumvent the fact that the timer was placed in its own driver in Minix.

All the processes has a unique priority associated and its own stack. The nano-kernel does not have its own stack, it uses the stack of the current running process when handling interrupts. All processes are started up at kernel boot time, and all processes have to run forever. When a process is initialized, a predefined stack size is allocated for the task. If the kernel runs out of stack it will panic during the initialization.

Even though the kernel is highly modularized, it does not prevent a process from writing in other processes' data area. It will therefore require some coding discipline to use the kernel as it is. The modularization could be taken one step further by using the GNU C extension of nested functions. Each process could be wrapped into one function and thereby creating an environment for this process only. For other processes to access the nested function would require explicit authorization by giving the function pointer to another process.

If the kernel were restructured using the GNU C nested functions extention, it might have an influence the interpretation of what should be called a nano-kernel. This is because the boundary between the nano-kernel and the kernel processes will become more blurred.

The subject of encapsulating the processes using nested functions is out of scope for this project, but as of this writing, an initiative to do this is already in progress by another student at DTU.

8.2 Scheduling

As mentioned previously, the scheduling should be kept simple and easy to replace. The scheduling is based on the process priority and follow the rule: at any given time only the process with highest priority should be running.

As shown on figure 8.2 a process can be in three different states, *waiting*, *ready* and *running*. Only one process can be in the *running* state at a time and all processes in the *waiting* state are waiting on a semaphore to be released. More about this below.

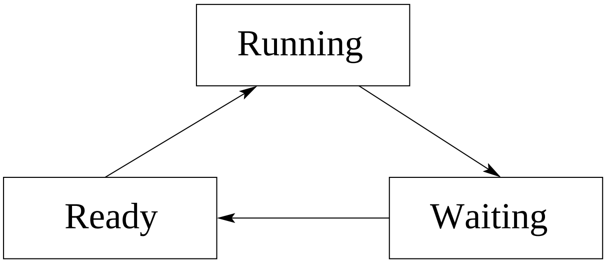

Figure 8.2: The different process states

Preemption of a process can happen while the process is doing a routine-call to the nano-kernel. Being able to preempt a process while it is running a routine-call in the nano-kernel gives a more responsive system, but it also introduces some problems. To avoid problems, some parts of the nano-kernel should run without interruption, and all functions provided by the nano-kernel to the processes should be re-entrant. One of the obvious places, where the nano-kernel must have a critical section to avoid interruption is during scheduling.

The scheduling decisions will happen, when a timer has expired resulting in a process being ready again and during process synchronization using semaphores. Timers and process synchronization will be described further below.

8.3 Timer

In embedded systems some types of jobs must run once after a given time and other types of jobs must run cyclic with a fixed period and this requires the use of a timer. I have decided to have two different types of timers:

One shot timer will, when started, wait for a specified time and when the time is up, the process waiting for the timer, will be put in the ready queue.

Cyclic timer will, when started, wait for a specified time and when the time is up, the process will be put in the ready queue. If a process is not waiting, it has probably missed its deadline, so to avoid kernel panic, the timer will be reset, and the process will try to catch the next deadline. After this the timer will be reset and start over again.

Every timer can have one, and only one process waiting.

Initialized timers can be in three states, *idle*, *active* and *done*. *Idle* state is when the timer is initialized but not started, and *active* is, when the timer has been started. The *done* state indicates that the timer is not used anymore and should be removed from the timer list.

As mentioned previously, the only time a process is in the *waiting* state, is when it waits for a semaphore to be released. This also applies to at process waiting for a timer. When a timer is started, a semaphore associated to the timer is locked, when the timer is fired, the semaphore is released and the process waiting for the timer can continue. The idea of using the semaphores for the timer comes from the Adeos[18] kernel and simplifies timer implementation.

The speciel case, where there is no process waiting to be activated, has to be handled gracefully. There are two reasons why there can be no processes waiting, first the process could have missed its deadline and secondly if the process does not cancel the timer, it has created. In both cases the cyclic timer simply ignores that there are no processes waiting and continues a new cycle. In hard real-time systems it would be a disaster to miss a deadline, but in this kernel it is ignored and the process missing a deadline will simply try and catch the next one.

In most operating systems the timer hardware is programmed to interrupt at a rate within the magnitude of 50Hz-200Hz. There are two problems with this when creating a timer; first, the timer is not very precise due to the low frequency and secondly the overhead of handling the timer interrupt is unnecessary, if the timer is not used for anything when the timer hardware interrupts.

To overcome this problem, the amount of time until the next timer interrupt should occur is calculated, and the timer hardware is adjusted accordingly.

The next timer interrupt should occur when the nearest timer should be activated. By using this method all unnecessary timer interrupts are eliminated unless a process creates a cyclic timer with a period greater than the value which the timer hardware could be programmed.

The maximum amount of time the timer-interrupt can be postponed, is dependent on the timer hardware used and the CPU speed. On the MIPS64 hardware used in this project, the timer interrupt can be postponed around 200 seconds, that is, it has to tick with a rate, which is at least $5e-03$Hz. The precision of the timer is in the magnitude of 0.5 micro seconds. Compared to the traditional timer implementations this is a huge step in the right direction.

8.4 Synchronization

To introduce synchronization between processes, I have decided to use a binary semaphore, with a queue of suspended processes, which is sorted by priority.

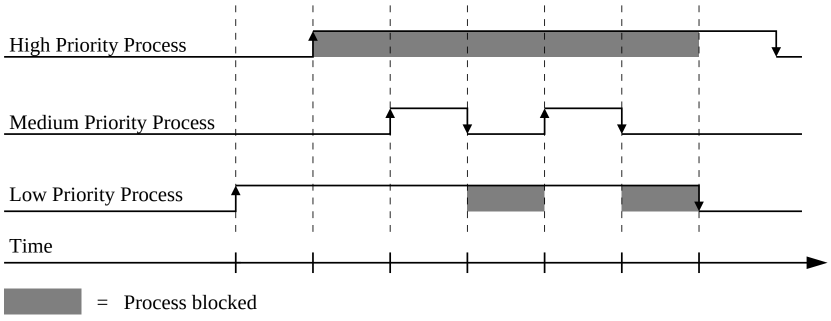

Figure 8.3: An example of priority inversion

The introduction of semaphores is not without cost. Consider the following example[18] on figure 8.3. Here there are three processes: high priority, medium priority and low priority. Low becomes ready first, indicated by the rising edge, and shortly thereafter it takes a semaphore, which is also used by the high priority process. Now, when high becomes ready it must block on the semaphore, until the low priority process releases it. The problem then arises, when the medium process becomes ready, then it is able to preempt the low priority process and thereby delay the high priority process. This phenomena is called priority inversion.

There are several solutions to the priority inversion problem. I have decided to use the Basic Priority Inheritance Protocol (PIP). In short the protocol works like this:

> When a process blocks one or more higher priority processes, it ignores its original priority assignment and executes its critical section at the highest priority level of all the jobs it blocks.

This protocol only deals with priority inversion and does not prevent deadlock. If, for example, a process locks S1, and then tries to lock S2, but S2 gets locked by a higher priority process, which now tries to lock S1 and a deadlock occurs. Instead of using PIP, the Priority Ceiling Protocol or Highest Locker could be used. This would prevent deadlock as well as priority inversion.

8.4.1 Message passing

As mentioned in chapter 3 message passing between processes should be introduced by a simple *send* and *receive* mechanism as known from Minix[17]. This, however, does not have to be a part of the nano-kernel, since this can be solved by implementing the producer-consumer problem[19] using the binary semaphore provided by the nano-kernel. This feature is then up to the user of the kernel to implement, and therefore not included as a nano-kernel functionality.

It is no requirement that message is introduced at all to use the kernel. One could just as well choose to have a monolithic kernel structure with shared memory between the processes.

8.5 Interrupt handling

When designing interrupt handling, several design issues have to be taken into account:

1. Decide how the interrupts priority should be.

2. Decide whether interrupt handler should be nested or not.

The MIPS CPU has a simpleminded approach to interrupt priority, in that all interrupts are equal. This leaves it completely up to the programmer to decide, how the interrupts should be prioritized. The MIPS CPU has two software and six hardware interrupts, see table 8.1. In this kernel there is no need for software interrupts, so these will be ignored. All interrupts for the Malta board end up in a combined hardware interrupt (MIPS IRQ 2), which is asserted, when devices such as the serial port interrupts. When receiving this type of interrupt the external interrupt controller has to be checked to actually see, which device asserted the interrupt. The last interrupt of interest is the timer interrupt asserted by the CPU itself, the rest are ignored in this kernel. This leaves the choice of creating a priority scheme, between the timer interrupt and the combined interrupt. I have decided to give the timer the highest priority, and the combined hardware interrupt the lowest priority, that is if the two types of interrupts is asserted at the same time, the timer interrupt should be handled first.

MIPS IRQ	Source
0	Software (ignored)
1	Software (ignored)
2	Combined hardware interrupt
3	Hardware (ignored)
4	Hardware (ignored)
5	Hardware (ignored)
6	Hardware (ignored)
7	Timer interrupt

Table 8.1: Used MIPS interrupts

The nesting of interrupts is closely connected to the priority of interrupt. For example, nesting should not be allowed when handling the highest priority interrupt, on the other hand, it is preferable to be able to handle the timer interrupt while handling an interrupt from the serial port. My solution sums up to:

1. Disable all interrupts when handling the timer interrupt.
2. Disable all but the timer interrupt when handling the combined hardware interrupt.

Disabling all but the timer interrupt when handling the combined hardware interrupt, may be a brutal decision and one could argue that combined hardware interrupts with a higher priority should be allowed to interrupt another combined hardware interrupts with lower priority. I have decided to keep things simple and handle the combined hardware interrupt with the highest priority first and without interrupts.

Another important issue when designing an interrupt handler is the performance of the interrupt handling. If the interrupt handling takes a long period of time, the system will then not be responsive to other events during this time. The two topics which deserves special attention are:

- The interrupt latency time should be minimized
- The interrupt handling time should be minimized

The time that passes between the interrupt and the execution of the interrupt handler is called the *interrupt latency*. The interrupt handling time is the time passed between the first intruction in the interrupt handler is executed to the last instruction in the interrupt handler is executed.

In the above the interrupt latency for the timer interrupt has been minimized by allowing nesting of interrupts.

The interrupt handling time can be reduced in several ways. First, most of the interrupt handler could be written in assembly code reducing unnecessary code generated by the compiler. Secondly, and this is often the issue which takes the most time, not calling the scheduler at every interrupt. When handling an interrupt the handler is often aware of whether a scheduling is needed or not. In this kernel all waiting processes are waiting for a semaphore to be released. If the interrupt handler releases a semaphore, it knows that a scheduling decision has to be made and marks this by raising a flag. This design removes all unnecessarily scheduling decisions.

The interrupt handler is summarized in the following:

1. Save the current state
2. Increment the nesting level
3. If timer interrupt, then call its handler and go to 5
4. If combined hardware interrupt, then enable the timer interrupt and call the combined hardware interrupt handler
5. If the flag, indicating a scheduling decision has to be made, is raised then call the scheduler

6. Decrement the nesting level
7. Restore to the previous state or a new state and return

This interrupt handler could easily be generalized to handle more than two priority levels. But on the other hand, additional priority levels also means a worse performance. Sitting in a loop and moving across all the pending interrupt bits is not the answer, the common case is one pending interrupt so it is optimized in that direction.

8.6 Context switch

Context switch, the switch between two processes, can happen in two different ways; when a process releases or locks a semaphore, and during interrupt handling. The context switch is done by switching the stacks. The stack contains the state to which it should restore to after the context switch.

When changing from one process to another by switching the stacks, special attention has to be paid to the problem, as to whether it is an interrupt, who triggered the context switch, or a process using a semaphore. In systems where processes uses system-calls to the kernel, it is customary to trigger a software interrupt and then by means of this, switch to the kernel. In these types of systems, only the interrupt handler is used to save and restore a given process state.

In systems, where the context switch can be done by means of a routine call to the kernel or by means of an interrupt handler doing the context switch, these two methods has to coorporate. This leads to four special cases of context switches:

1. Changing from a process preempted by an interrupt to a process preempted by a using a semaphore.
2. Changing from a process preempted by a using a semaphore to a process preempted by an interrupt.
3. Changing from a process preempted by an interrupt to a process preempted by an interrupt.
4. Changing from a process preempted by a using a semaphore to a process preempted by a using a semaphore.

It is point 1. and 2., which make things difficult because the two different types of context switch have to coorporate. How this is solved in practice is described in detail in chapter 10.

8.7 Global exception handling

As stated in “Kernel properties” (chapter 3) the use of global exception handling in an embedded system, to handle failures in a modular manner, could be of great advantage in bug-finding and system recovery, thus methods for implementing exceptions in C should be analysed. This issue will in the following only be described briefly, since it was not used in the kernel and the reasoning for not using exceptions will be stated.

Exception handling provides a way of transferring control and information from a point in the execution of a program to an *exception handler* associated with a point previously passed by the execution. A handler will be invoked only by a *throw-expression* invoked in code executed in the handler's *try-block* or in functions called from the handler's *try-block*. What is trying to be achieved in C is something similar to the following:

```
try {
  /* Do something and throw an exception if something goes wrong */
} catch {
  /* Handle it here if something went wrong */
}
```

Listing 8.1: Exception example in C++

There are basically two methods for implementing exceptions in C. The first method is by using the POSIX functions calls `setjmp` and `longjmp`. These functions have been implemented in the file `setjmp.S` in appendix B). The `setjmp` function saves the stack context for non-local goto and `longjmp` makes non-local jump to a saved stack context. The two functions calls can easily be wrapped into two macros *throw* and *try*, where *throw* would use `longjmp` to jump to an exception handler and *try* would use `setjmp` to save the exception environment. The result is actually a very nice implementation of exceptions in C, but it does have some problems.

The problems with the `setjmp` method is that C lack of stack-cleanup facilities which means that code written to be exception safe must include far more *try* blocks than it would in C++ or Java. Another issue is that the exception implementation has to be threadsafe, since the kernel is running several processes which are able to use the exceptions. Implementation of a threadsafe exception is not a trivial task. For these reasons this method is not used.

The second method was described in the "C/C++ Users Journal"[16] and implemented using the `goto` call. The method requires that an error status is passed on to every function and after all function-calls this value must be checked. If the error status indicates an error, it would throw an exception. After some experimentation with the code described in the article the conclusion was that, it was clumsy to use and made the code difficult to read. Besides that, I did not like the fact that an extra parameter was required to be passed on to every function call.

Since the exception handling is not used another approach to handling errors should be taken. I choose to use a very brutal method: if something goes wrong then report the error and make a kernel panic immediately. If exceptions were used, the panic would instead be in the exception handler, so when the kernel panics it would not be in the same state as when the error ocurred. Besides, there is absolutely no reason to try to continue execution if the kernel enters some unexpected state, so the best approach is to create a kernel panic and fix the bug. In real-world embedded systems the kernel panic should be combined with a failure handler which often just restarts the whole system.

8.8 Summary

This chapter covered the kernel design. All the major components of the kernel have been designed and described and the general kernel structure is now clear.

Chapter 9

Bootstrapping

This chapter first describes bootstrapping in general and then gives an introduction to boot loaders. This is followed by a description of what happens, the moment after the Malta system has been powered on. Then a comparison of the MIPS bootstrapping to other systems is made, and the chapter finishes with a description, of how bootstrapping a kernel is done in practice on the Malta system.

9.1 Bootstrapping in general

In operating systems, the term *bootstrapping* denotes the process of bringing the operating system's kernel into main memory and executing it. It is highly system dependent, how much work is actually required of the system programmer to get an operating system's kernel up and running. Often the bootstrapping process is simplified by the introduction of a BIOS, firmware, boot loaders and so forth.

When the system cold boots that is, when it starts up just after a power on, nearly all the hardware is in a random state and has to be initialized, before it is used. The classical system startup is devided into the following three steps:

1. In the first step the BIOS, firmware or similar is executed. This is the first code that is actually executed after power on. This code is

provided by hardware manufacturer and will in the following be referred to as the BIOS. Its main job is to initialize the basic hardware such as the CPU, the CPU-cache and the RAM. It is completely up to the hardware vendor, how much functionality is actually included in this part of the system startup. For instance, the previously mentioned AT91 system has no BIOS at all, and all initialization has to be written from scratch. Depending on the system configuration the BIOS transfers the CPU control to the boot loader or the operating systems kernel itself.

2. In the second step the boot loader is executed. Whether this is necessary or not depends on the BIOS, but the boot loader provides a convenient way to choose between, which different system configurations to run. This could be different operating systems, different kernel versions or different kernel configurations by passing options to the kernel.

3. The third and last step executed on system startup is the kernel-bootstrap. One could expect this part to be small, but on the contrary this part is rather big, since much of the hardware is initialized all over again. It is generally believed that the BIOS is filled with bugs, and therefore the hardware state cannot be trusted, when the kernel-bootstrap gets control of the CPU, so the hardware is initialized once again.

As one can imagine there are huge differences in hardware and the functionality provided, in the BIOS, by the hardware manufacturer, and therefore the system startup is different from system to system. However, due to the uncertainty of the BIOS functionality and the hardware state after the BIOS has been executed, an increasingly amount of the hardware initialization code is going into the modern operating system's kernel-bootstrap. As a consequence to this, the BIOS has become almost negligible in the system startup. Another consequence is that the kernel-bootstrap can be generalized, since it always has to go through a certain number of steps in a certain order, no matter what hardware the kernel is running on.

The kernel-bootstrap code suffers from the clash of two opposing but desirable goals. On one hand, it is robust to make minimal or no assumptions about the state of the hardware and then attemt to initialize and check every subsystem, before it is used. On the other hand, it is desirable to minimize the amount of tricky assembler code, but changing to a high-level language, like C, tends to require more subsystems to be operational.

The above classical system startup sequence step 1. through step 3. can be generalized to the following generic startup sequence, where 1. is the first code to be executed at power on:

1. Initialize CPU registers e.g setup addressing modes and disable interrupts.
2. Check and initialize RAM. This is often a very hard part, since RAM chips seems to differ a lot. Fortunately, modern chipsets do this tedious initialization.
3. Now establish some contact with the outside world. This could be through the parallel port, which has become the most standard way on modern PCs. This step is not a necessity, but it is convenient to see, what is actually going on during startup.
4. Initialize a stack, registers and call a C function. Now the rest of the initialization can be done from C.
5. Initialize any other devices needed to load the kernel and load the kernel or call the boot loader, which loads the kernel.
6. Transfer the control to the loaded kernel and the kernel-bootstrap is executed. The kernel bootstrap is, most likely, the same as step 1. through 4. again.

Step number 5. is the one step that differs mostly from system to system, since the kernel can be loaded from a lot of different devices, such as from a flash disc, a floppy disc, a harddisc, over an ethernet or even over the Internet. All of these different devices have to have a device driver to provide access for the boot loader or the BIOS to load the kernel. One could say that step 5 provides the glue between the initial bootstrapping and the kernel-bootstrapping, and that the glue is provided by the BIOS or a boot loader. More about boot loaders below.

In very small systems like the previously mentioned AT91, step 1 through 4 are the kernel-bootstrap and step 5 and 6. are not needed. This is because the bootstrap and the kernel, lie as one continuous piece of code in the flash RAM and the kernel can therefore be loaded without special communication with other devices.

9.2 Introduction to boot loaders

As mentioned above the boot loader provides the glue between the BIOS and the actual kernel-bootstrapping, that is, it is responsible for loading a kernel and transferring the control to the kernel. Boot loaders vary a lot from system to system, due to the differences in the BIOS provided by the hardware. That is, if the hardware manufacturer provides hardware with a small BIOS or no BIOS at all, this has to be compensated for in the boot loader.

In the following, four different free software boot loaders will be described, and some of their system-specific features will be mentioned:

LILO - LInux LOader LILO[30] has been the standard boot loader for Linux on the Intel platform for a long time. It is placed in the boot sector of a floppy disc or in the master boot record of a harddisc, and is therefore executed right after the PC BIOS is done with hardware initialization. To enable LILO to load a kernel, the location of a kernel on the disc media is hardcoded into LILO. As an example on how the boot loader compensates for the shortcomings of the PC BIOS; LILO extends the PC BIOS harddisc drivers, since older PC BIOSes cannot address a kernel, if it is not located within the first 1024 cylinders of the harddics.

GRUB - GRand Unified Bootloader GRUB[31] is a very dynamic boot loader and for that reason becoming an increasingly popular boot loader for the Intel platform. GRUB is file system aware, so that the location of the kernel on a disc media does not have to be hard-coded into boot loader, as it was the case with LILO. GRUB is also compliant to the "Multiboot Specification", which is an initiative to standardize the booting of different operating system's kernels. If the kernel developed in this project is going to be ported to the Intel platform, this would be the best boot loader to use, if the kernel is loaded from a disc media.

EtherBoot EtherBoot[32] is a boot loader, which is created for discless computers. The boot loader is placed in a flash ROM on an ethernet adapter and supports loading a kernel over the network using a TFTP server. The boot loader located in the ethernet adapters flash ROM is called right after the PC BIOS is done initializing the hardware, as it was the case with the other boot loaders mentioned above. To load the kernel over an ethernet, EtherBoot has to provide special driver support for the ethernet adapter in question and a small network stack.

RedBoot RedBoot[33] is possibly the most advanced free software boot loader available. RedBoot combines the BIOS and the boot loader. RedBoot is available for a wide range of platforms including ARM, MIPS, PowerPC and Intel platforms. The features include network booting, flash booting and remote debugging. Unfortunately, the bootloader only works with the eCos kernel and the Linux kernel.

Of all of the above described boot loaders only RedBoot runs on the MIPS platform. The reason for this is that MIPS-based hardware products often come with a boot loader especially designed for the system, and therefore there has never been a real need for other boot loaders for the MIPS platform. The MIPS Malta system comes with a combined BIOS and boot loader very similar to RedBoot. This is an open source boot loader called YAMON, and it is described futher below.

Whether or not a boot loader is really necessary on an embedded system, all depends on the BIOS features provided with the hardware and the actual location of the kernel.

9.3 Bootstrapping MIPS

Above, it has been generalized how bootstrapping is done on a general system with at least some memory and a CPU. Now, the low level details of how to cool boot the MIPS Malta system will be described.

During a power on or cold reset, the SI_ColdReset signal is asserted. The SI_ColdReset is a hard reset signal, and the assertion of the SI_ColdReset signal completely initializes the internal state machines of the 5Kc CPU, without saving any state information. When SI_ColdReset is deasserted, a reset exception is taken by the 5Kc CPU. When the 5Kc CPU takes the reset exception, it is hard coded to execute the code located at the address 0xFFFF.FFFF.BFC0.0000, and this is where the YAMON boot loader is located.

When YAMON gets the control of the CPU, it first disables the interrupts, determines the endianess of the system, initializes the status register of the CPU, creates a small stack, and then jumps to the first C function, which does the rest of the initialization. Furtunately, the chipset on the board is able to initialize the RAM, so this step can be ignored.

This should be compared with the traditional MIPS startup sequence as described in “See MIPS Run”[20] which is as follows:

1. Reset exeption entry point
2. Initialize status registers
3. Initialize and check the RAM integrity.
4. Make contact with the outside world.
5. Initialize stack and registers to call C function
6. Initialize the cache

The comparison shows that YAMON follows the traditional MIPS startup sequence almost to the point, except for the fact that contact to the outside world is postponed in YAMON, until all devices are initialized.

The C function that takes the Malta system through the rest of the initialization is, slightly simplified, as follows:

1. Initialize the PIIX4 chip
    - Initialize the ISA bus on which the Super I/O Controller is connected to the the PIIX4 chip, see figure 5.2.
    - Initialize the serial ports
    - Initialize the parallel port
    - Initialize the keyboard and mouse IRQs
    - Enable IO access to Power Management device
2. Initialize the peripherals
    - Initialize the exception handlers
    - Initialize the real time clock
    - Initialize the PCI bus memory mapping
    - Initialize the network adapter
3. Start the YAMON shell

As noted above the traditional MIPS startup sequences, and the YAMON startup sequence are very similar, but it should also be noted that these startup sequences fit very nicely into the generic startup sequence described in section 9.1. A study of bootstrapping a PowerPC also turned out to fit in the generic startup sequence as well.

9.4 MIPS vs. Intel I386

In the following a comparison, of the startup sequence on the Malta system and the Intel platform, is made.

On the standard Intel PC the initialization, after power on, is as follows:

$$\text{BIOS} \rightarrow \text{boot loader} \rightarrow \text{kernel}$$

Whereas the initialization on the Malta system, with its combined BIOS and boot loader is:

$$\text{YAMON} \rightarrow \text{kernel}$$

When creating embedded systems for real world applications the standard Intel PC boards are rarely used. This may seem strange, since it is so widely available. One of the main reasons is that the BIOS is extremely slow at initializing the hardware. From power on to the actual boot loader is running takes approximately 10 seconds, whereas on the Malta it takes approximately 2 seconds. People using embedded systems often expect the system to be up and running in no time, making the standard Intel PC unusable for these purposes.

Initiatives to solve the long boot times on the Intel platform is in the works. The most promising project, is the LinuxBIOS[29] project which is able to reduce the startup time to 3 seconds from power on, to a stripped down Linux kernel is running. The only problem with this project is that the hardware vendors are not very co-operative, when it comes to datasheets for their hardware.

Another problem, which could be noted, is that many of the features provided by the PC BIOS is unusable in modern operating systems, since they always run in protected mode whereas the BIOS can only be used in real mode. Many of the features that the PC BIOS provides are only for backwards compatibility with DOS.

9.5 Probing hardware

Above, it has been mentioned that after the CPU and RAM have been initialized, all the different hardware parts of the system are initialized or at least the ones that are actually used. But how do you know which hardware is in the system? The method for finding hardware is called “probing the hardware”.

Since not much hardware is supported in this kernel, not much hardware is actually probed. The only hardware probed is the CPU type and the CPU speed, this is implemented in `cpu.c`

To identify the MIPS CPU type the CPU contains an implementation-number and a manufacturer-defined revision-level in a register called `PRId` in the coprocessor CP0 in the CPU. However, it is best not to rely on the revision level information, since changes in the CPU are not always reflected in the revision level. The implementation number, on the other hand, characterizes the CPU, and from this it can be varified that the kernel is really running on the 5Kc CPU.

The CPU speed is needed when creating a timer. The CPU speed is often found by running a loop of known length that will take a fixed large number of CPU cycles and then compare the *count* register before and after running the loop. The *count* register is described in section 10.7.1.

I have chosen another approach to this problem. The Malta system comes with a Real Time Clock (RTC) included in the PIIX4 chip, see figure 5.2. This RTC is updated every second and at every update, a flag is set in the RTC. By using the flag and the *count* register, the CPU speed is easy to get:

1. Wait for the flag to be raised
2. Reset the *count* register
3. Wait for the flag to be raised again
4. Read the value *count* register

The *count* register now contains half the numbers of clock cycles the 5Kc CPU has in one second. It is half the numbers of clock cycles, since the *count* register is only updated at every other clock cycle. By multiplying this number by two, the result is the CPU speed in Hz. The resulting probed CPU speed is 40MHz. Running several tests verify that this method is very precise with a deviation of only ±1 Hz.

9.6 Bootstrapping the kernel using YAMON

As mentioned above, YAMON serves both as a BIOS and a boot loader for the Malta system. On power on the Malta system will initialize and end up in the YAMON shell. From this shell the command `load` can be used to load a kernel and the command `go`, to run a kernel.

The state of the CPU, when the kernel-bootstrap begins, is called the Initial Application Context and is as follows:

Register	Value
`a0`	This register is set to the argument count from the YAMON shell. The arguments are tokens seperated with white spaces.
`a1`	This register is set to the address of an array of string pointers holding the arguments, for example `argv[0]="go"` and so forth.
`a2`	This register holds a pointer to a table holding YAMON environment variables.
`a3`	This register holds the memory size of the SDRAM mounted on the core card.
`ra`	This register holds the return address that the kernel can use to return to YAMON when it is done running. This will never be used.

Table 9.1: Initial Application Context

The registers `a0` through `a3` are argument registers, hence, if they are unchanged in the kernel-bootstrap the first C function can have the following prototype, which gives easy access to the kernel parameters:

```
int
entry(
        unsigned int argc,         /* Number of tokens in argv array    */
        char **argv,               /* Array of tokens ( first is "go")  */
        t_yamon_env_var *env,      /* Array of env. variables           */
        unsigned int memsize);     /* Size of memory (byte count)       */
```

Listing 9.1: Prototype for first C entry

An example of the use of kernel parameters could tell the kernel, which serial port the serial terminal is connected to.

Besides providing the functionality of loading kernels, YAMON also provides a set of functions which can be used from the kernel. The functionality includes functions to register exeception service routines, reading and writing to the serial port. Even though these functions would be very convinient to use they are *not* used in this kernel. The reasoning behind this decision is that a kernel should control the hardware itself and no rely on system specific functions, which the YAMON provided functions are.

9.7 Kernel bootstrap

When the kernel-bootstrap is executed, it goes through the following steps, this is implemented in the file `start.S`:

1. Disable interrupts and disable the cache
2. Setup a stack, the address of the stack top is hard coded in the linker script.
3. Initialize the .bss section to zero, see figure 10.2
4. Call the first C function (`entry`), which does the rest of the initialization.

This bootstrap also fits nicely into the generic startup sequence described in section 9.1.

The first C function that is called is named `entry` and is implemented in the file `kernel.c`. This function takes the kernel through the rest of the initialization:

1. Initialize the serial terminal driver. This is done already here, because it enables kernel to write status messages to the terminal during the rest of the initialization.
2. Initialize CPU and probe CPU type and speed
3. Initialize timer driver
4. Initialize the process manager and register the processes
5. Initialize and enable interrupts
6. Call schedule and the kernel is up and running.

It should be noted here that not all peripherals that are used in the kernel are initialized during the kernel-bootstrap. This includes setting up the memory mapping of the PCI bus and initializing the PIIX4 chip. It is relied on YAMON to do this initialization properly, before the kernel is runned. Fortunately, this is the case and it saves the kernel a lot of work during the kernel-bootstrapping.

9.8 Summary

From this long discussion of bootstrapping and boot loaders the conclusion is; even though no code can be reused from one bootstrap to another, the skeleton, on which the bootstrap is build, has been the same on all systems studied in this project.

Another thing, which should be noted, is that even though hardware manufacturers provide the hardware with a BIOS, this should not be relied on to heavily, since it may be buggy and it decreases the protability of the kernel if the supplied BIOS functions are used.

Chapter 10

Kernel implementation

This chapter describes the kernel implementation. The main focus will be on, how to interface with the hardware, since this subject has been the most time-consuming part of the kernel implementation. The chapter finishes with a brief description of the actual kernel construction. All the source code for the kernel is listed in appendix B.

During the chapter, interfaces to several kernel components will be described. The prototypes for these interface functions will not be included in the interface description, since if would not contribute significantly to the understanding of the semantics of the interface.

10.1 Compiling

This section describes the compilation of the kernel. This includes the Makefile, the source code layout and the compilation parameters.

10.1.1 The Makefile

The Makefile for building the kernel is created in a very naive way. There is a perfectly good reason for this, and that is, while the source code is settling, it would require to much work to keep changing an very advanced build system, every time the source code is moved around and files are renamed.

The Makefile has been created with portability in mind, and therefore there are several user configurable options in the Makefile. These are listed below.

`TFTPDIR`	This is where the kernel is placed upon installation. The TFTP server must be configured to point here. The default is `/kernel`
`TOOLCHAIN`	This option sets the tool-chain, the kernel is compiled with. By default it is the tool-chain for generating a kernel for big endian MIPS64.
`ARCH`	This option sets the architecture for which the kernel is going to be compiled. By default this is the MIPS64 architecture.
`ENDIAN`	This option chooses the endianess. The default target is big-endian.

Table 10.1: Options in the Makefile

10.1.2 Source code layout

The source code is organized as shown on figure 10.1. As the figure shows, the root of the tree has three branches, one for include files, one for the kernel and one for the library functions.

The `include` and `kernel` branch each has architecture specific branches. The `mips` branch contains generic MIPS code, the `mips32` branch contains MIPS32 specific code and the `mips64` branch contains MIPS64 specific code.

The `arch` file is a symbolic link that links to the architecture to be compiled. This link is created at compile-time by the Makefile, and the destination point for the link depends on the user-defined options in the Makefile, described above.

10.1.3 Compilation options

When developing normal applications, the compile options are not that important, and the default options are sufficient for most purposes. When

```
│
├──include
│   ├─arch ──> mips64
│   ├─mips
│   ├─mips32
│   └─mips64
├──kernel
│   ├─arch ──> mips64
│   ├─mips
│   ├─mips32
│   └─mips64
└──lib
```

Figure 10.1: Kernel directory structure

developing kernels on a cross-development platform, the compile options can be crucial and should be carefully chosen. In table 10.2 the most important compiler options are listed.

10.2 Linking

Every link is controlled by a linker script. This script is written in a linker command language. The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file. Most linker scripts do nothing more than this. However, when necessary, the linker script can also direct the linker to perform many other operations.

In the kernel there is a need for complete control of positions, size and alignment of the different sections, so the default linker script is not sufficient, and a custom linker script has been written, see the file `link.xn` in appendix B.

The custom linker script solves four kernel specific linking issues:

1. Organizing the *text* section at the address `0x80200000` which is where YAMON loads the kernel. The rest of the sections will follow the *text* section continuously.

`-O2`	This level of optimization should be safe.
`-mcpu=r4600`	Choose the RISC 4600 cpu type.
`-mabi=64`	Choose the mips3 instruction set.
`-fno-strict-prototype`	egcs 1.1 invokes this optimization per default, but this optimization is rather untested and should therefore not be used.
`-nostdinc`	Do not search the standard system directories for header files. Only the directories that are specified with '-I' options are searched.
`-fomit-frame-pointer`	Do not keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. Remove this option if remote debugging is implemented as the latest version of gdb does not support this.
`-Wa,-32`	This is an undocumented, but very important option. It tells the assembler to generate 32 bit ELF code instead of the default 64 bit ELF. This is needed due to the previous mentioned MIPS64 linker bug.
`-G num`	Puts the global and static items less than or equal to `num` bytes into the small data or bss sections instead of the normal data or bss section. This allows the assembler to emit one word memory reference instructions based on the global pointer (gp) instead of the normal two words used. Using this type of global pointer optimization, could lead to problems and is therefore eliminated by supplying the compiler with the value `0` for `num`.
`-Tlink script`	Specifies the custom linker script

Table 10.2: Compilation options

2. Putting the file `start.o` at the very beginning of the linked kernel. This file contains the bootstrap.
3. Setting the stack size of the kernel. This is hard coded in the linker script, but could be changed into an option to the linking process.
4. Defining a symbol in the kernel which enables the kernel to get the size of the stack available. This symbol is called `_sp_end`

In ISO/ANSI C the symbols *end*, *edata*, and *etext* are elements of the space of names reserved for the user. Thus, they have to be defined by the linker script to conform to the standard.

Figure 10.2 shows the layout of the different sections of the linked kernel.

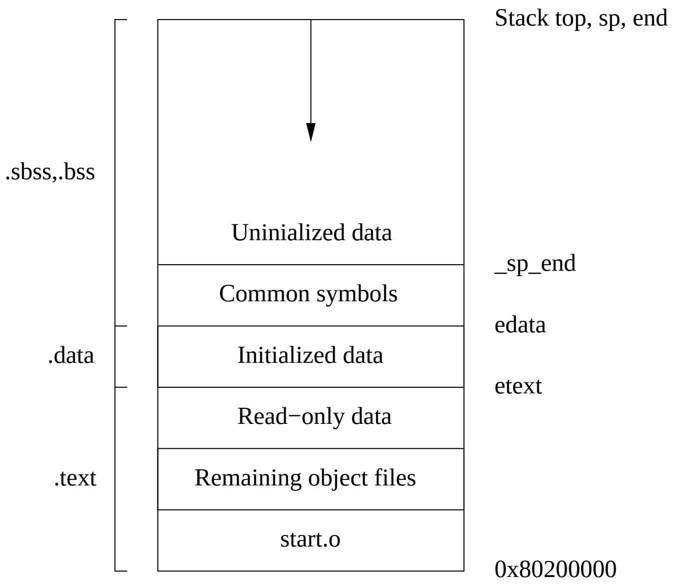

Figure 10.2: Overview of the linked kernel

After the linking of the kernel, it is still not ready for use on the MIPS64 system. As mentioned previously, the output format of the file is 32 bits elf code, but it has to be 64 bits elf code since, it is a 64 bit kernel. Besides that all the addresses have to be converted from 32 bits addresses to 64 bits addresses. Both of these conversions are easily done by the program `objcopy`.

To be able to load the kernel on the MIPS64 system, the linked kernel has to be converted into `srec` format, since this is the only file format that YAMON can handle. The `srec` format is the Motorola S-records format, which is a format that makes it easier to transfer files over low speed connection with a high error rate and using an extremely simple protocol like the TFTP protocol. This conversion is done with `objcopy` as well.

10.3 Header files

There is only one special thing, which has to be noted about the header files, and this is the use of the file `stdarg.h`. This is a header from the glibc library, which defines a set of hardware specific macros for implementing the functionality of passing an arbitrary number of arguments passed on to functions. The header file does not provide any function prototypes to glibc itself and can therefore safely be used in the kernel implementation. To be more specific, it is used in the kernel implementation of `printf`.

10.4 Handling interrupts

In the MIPS architecture, interrupts, traps, system-calls, and everything else that disrupts the normal flow of execution, are called exceptions.

The 5Kc CPU is hard coded to execute program code at one of five different locations depending on the type of exception, which has occurred. This could be a cache error exception, an interrupt exception etc. In this kernel, only interrupt exceptions are of interest, and all other types of exceptions are left up to YAMON to handle. In the following, interrupt exceptions will be called interrupts for simplicity.

The types of interrupt that can occur are a timer interrupt from the 5Kc CPU, and a combined hardware interrupt generated by the interrupt controller, that is located in the PIIX4 device on the MALTA board. The interrupt controller located in the PIIX4 device is a standard 82C95 interrupt controller.

10.4.1 Registering the interrupt handler

As mentioned above the CPU is hard coded to execute program code at a specific location when an interrupt is asserted. In the 5Kc CPU the address is `0xffffffff80000200`, and there is exactly room to put 32 bytes of code to handle the interrupt at this address .

Instead of copying a full interrupt handler to this address a jump instruction to the real interrupt handler should be copied on this address. The jump instruction is composed of the op-code for the jump instruction and the address, see figure 10.3. The address is not allowed to be more than 28 bits and has to lie on 32 bit word boundaries, since the last two bits are thrown away during the op-code generation.

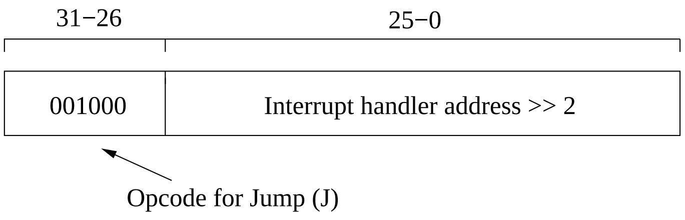

Figure 10.3: Jump op-code construction

Normally, the cache would have to be flushed after writing the jump op-code into the memory, but as previously mentioned the cache has been disabled to avoid these sort of problems. When the cache is enabled in future versions of the kernel, special attention has to be paid to all the places, where the kernel writes to memory mapped hardware, which is mapped in the cache-able memory.

10.4.2 Combined hardware interrupt

When the interrupt handler receives a combined hardware interrupt, the function `interrupt_hw` in `interrupt.c` is called. Its primary job is to figure out which device caused the interrupt and then call the appropriate interrupt handler for the device.

By requesting information from the 82C59 interrupt controller the IRQ, which triggered the combined hardware interrupt, is found. By using this

IRQ as an index in a interrupt handler table, the right handler can be found and called in constant time.

10.4.3 Interrupt interface

The interface to the interrupt component consists of only two functions, see table 10.3. These are implemented in the file `interrupt.c`.

`interrupt_init`	This function initializes the interrupt component and registers the interrupt handler, which receives MIPS interrupts.
`interrupt_register`	This function takes two arguments, namely the IRQ number and the interrupt handler. It then registers the handler in the interrupt table. If an interrupt handler for a given IRQ is already registered, the kernel will panic, since shared interrupts are not supported in this kernel. This function should only be called during kernel initialization.

Table 10.3: Interrupt component interface

10.5 Context switch

As mentioned in chapter 8, two different types of context switches have to cooperate e.g. a context switch by using a semaphore and context switch during interrupt.

The real difference between these two types of context switches lie in the way they return to a new process after restoring a new context. When returning from an interrupt, the instruction `eret` is used. When the `eret` instruction is executed, it clears a bit in the status register and then jumps to the address held in the `EPC` register. When returning from a context switch, which has been issued by using a semaphore, it uses a normal jump instruction. To make these two cases work together gracefully, the special register `k0` is used to store the address, to which a process should continue after both types of context switch. The `k0` is a special register that is reserved for use in interrupt handlers. The register is safe to use, since exceptions are disabled during both context-saving and context-restoring and this prevents other exceptions handlers changing the `k0` register, while it is in use.

The context switch in the interrupt handler is implemented in the file `mipsirq.S` and works as follow:

1. Save the `EPC` register in `k0`
2. Save the current CPU context on the stack
3. Save the stack pointer
4. Handle the interrupt
5. Get a new stack pointer for the next process to run
6. Restore to the new context
7. Restore the saved `EPC` register
8. Return from the interrupt handler

The context switch used, when using semaphores, is implemented in the file `stack.S` and works as follow:

1. Save the return address (`ra` register) in the `k0` register
2. Save the current context on the stack
3. Save the stack pointer
4. Get a new stack pointer for the next process to run
5. Restore to the new context
6. The `k0` register now contains the return address, so this is used as return address.

All the macros used for saving and restoring the states are located in the file `stackframe.h`.

10.6 Semaphores

As previously mentioned, the semaphore is a binary semaphore with a waiting queue of blocked processes sorted after priority.

The basic priority inversion protocol, described in chapter 8, has been implemented as part of the semaphore. This is basically done:

- If a process tries to take a semaphore, which is locked by a lower priority process, the priority of lower priority process is raised to

the priority of the process wanting the semaphore. After this a re-schedule is issued.
- When a process releases a semaphore, the process priority is lowered to its original priority and a re-schedule is issued.

All operations in the semaphore implementation, which has to be done as an atomic operation, has been put in a critical section by disable and enabling interrupts. The MIPS CPU has special support for atomic operations, but these have not been used, because the semaphore implementation would then be hardware specific.

10.6.1 Semaphore interface

The interface to the semaphore is similar to the one used in M. Ben-Ari[19] and is implemented in the file `semaphore.c`. The component works as follows:

`semaphore_setup`	Setup the semaphore and initialize its waiting queue.
`semaphore_wait`	Takes the semaphore if it is available or waits if it is locked.
`semaphore_signal`	Releases a semaphore

Table 10.4: Semaphore component interface

10.7 Kernel drivers

In the following, the three kernel drivers will be described. During the description of the driver interfaces the expression "from the kernel users point of view" is used. By this is meant, the functions in the interface that are of relevance for the kernel user e.g. a person implementing processes to run on the nano-kernel.

10.7.1 Timer driver

On Intel based architectures it is normal to create a timer using the 8253A chip, which is placed on almost all Intel based motherboards. In the

MIPS64 CPU there are two registers, which are very useful when creating a timer, namely the *count* and *compare* registers. These two registers will be used for the timer implementation instead of the usual timer hardware.

The *count* register acts as a timer, incrementing by one every other clock cycle, whether or not an instruction executed. The *count* register can be written for diagnostic purposes as it is during boot in this kernel. This register is 32 bits long.

The *compare* register is used in conjunction with the *count* register. The *compare* register contains a value, which does not change unless explicitly updated by software. When the value of the *count* register is equal to the value of the *compare* register, hardware interrupt 5 is asserted and the interrupt pending bit is raised in the *cause* register. Hardware interrupt 5 is asserted and continues to be asserted, until the *compare* register is written to by software. This register is also 32 bits long.

From the kernel users point of view, the interface to the timer driver consists of four functions, which are implemented in the file `timer.c`:

`timer_setup`	This function initializes a timer structure.
`timer_start`	This function starts the timer.
`timer_waitfor`	This function blocks the process until the time is up.
`timer_cancel`	This function cancels a timer. This function must be used if the timer is no longer in use.

Table 10.5: Timer interface

10.7.2 LCD driver

The LCD display is a small 1x8 characters wide LCD display mounted on the Malta board. The LCD display can be used for debugging purposes or just to show off. The driver is extremely simple, and is used for writing characters or numbers on the LCD display mounted on the MALTA board.

The LCD display works by writing to the addresses listed in table 10.6.

The interface to the driver consists of the two functions listed in table 10.7.

Name	Offset	Function
ASCIIWORD	0x000.0010	Writing a 32-bit number to this address will cause the LCD display to show the number in hex on the display
ASCIIPOS0	0x000.0018	Writing an ASCII value to this address updates position 0 on the LCD display
ASCIIPOS1	0x000.0020	Writing an ASCII value to this address updates position 1 on the LCD display
ASCIIPOS2	0x000.0028	Writing an ASCII value to this address updates position 2 on the LCD display
ASCIIPOS3	0x000.0030	Writing an ASCII value to this address updates position 3 on the LCD display
ASCIIPOS4	0x000.0038	Writing an ASCII value to this address updates position 4 on the LCD display
ASCIIPOS5	0x000.0040	Writing an ASCII value to this address updates position 5 on the LCD display
ASCIIPOS6	0x000.0048	Writing an ASCII value to this address updates position 6 on the LCD display
ASCIIPOS7	0x000.0050	Writing an ASCII value to this address updates position 7 on the LCD display

Table 10.6: LCD display addresses. Base address is 0x1f00.0400

`lcd_int`	Takes an integer as argument an prints it in hex on the LCD display.
`lcd_message`	Takes a string as argument and displays the eight first characters on the LCD display.

Table 10.7: LCD driver interface

10.7.3 Serial terminal driver

This driver controls the serial VT220 terminal connected to the Malta system's serial port.

The serial port on the MALTA board is controlled by a Super I/O Controller from SMcS and incorporates two full function UARTs. The file `serial.h` defines all the register addresses in the UART and most of the register settings.

As of this writing, input on the serial line is not supported, but it should be fairly simple to implement, as the interrupt handler is already registered to receive the data from the UART. Since, DMA is not supported by the Super I/O Controller, the data should simply be read from directly from the memory mapped serial buffer.

Initialization

The VT220 terminal is configured to run at 19.200 baud, with 8 bit data, one stop bit and no parity, so the serial port in the Super I/O Controller has to be initialized the same way.

The serial port contains a programmable Baud Rate Generator that is capable of dividing the internal UART clock by any divisor from 1 to 65535. The clock runs at 1.8462Mhz and the output from the Baud Rate Generator is 16 times the baud rate, therefore, to set the desirable baud rate, the divisor is calculated like this:

$$Baud\ rate\ divisor = \frac{UART\ clock\ speed}{16 \times bps}$$

Inserting the numbers in question the resulting baud rate divisor is:

$$\frac{1846200Hz}{16 \times 19200bps} = \ 6$$

The baud rate divisor is therefore 6, when a baud rate of 19200 is desired. This value should be written to the divisor registers. The high register with the value 0 and the low with the value 6. These registers are the same as the transmitter and receiver registers, so a special bit (DLAB) has to be set high to tell the serial port that the baud rate is going to be set.

The rest of the configuration of the serial port is done by writing to the Line Control Register.

The interrupt is not initialized, since the input is not read anyway. Initialization of this interrupt should be located in the function `serial_init` along with the rest of the serial terminal initialization.

Serial terminal driver interface

From the kernel users point of view the interface to the serial terminal driver consists of two functions, see table 10.8. Through these functions special control characters can be sent to the serial terminal to control cursor position and to print characters an the terminal. These functions are implemented in the file `serial.c`.

`serial_putchar`	Prints a character on the terminal.
`serial_print`	Prints a string on the terminal.

Table 10.8: Serial terminal interface

In future development this driver should be split into two, one for the UART itself and one for the VT220 terminal using the UART, but during the kernel development this implementation has been sufficient.

10.8 Kernel construction

In the above, the implementation has been described, but not how the kernel was actually constructed. In the following the steps I went through, to get the kernel up and running, is summarized:

- First thing to be implemented was the bootstrap. This was debugged by writing to the LCD display.
- Once the bootstrap was working an environment, for changing from MIPS assembler into C, was created.

- Next the serial terminal driver was created and nice `printf` function was included. The main reason for the inclusion of a full blown `printf` function at this early stage of development was that it would speed up the debugging process.
- Then an implementation of coroutines was made. This was done using a *yield* function, which where very similar to a combination of the `setjmp` and `longjmp` functions, which are implemented in the file `setjmp.S`. This was tried out with disabled interrupts. The *yield* function later evolved into a stack switch function.
- At this point an interrupt handler was implemented and tested using the timer interrupt.
- Once this was running a timer driver was implemented.
- The missing link before combining all the pieces was to implement semaphores and process management. These two components were implemented and tested as far as possible at this stage of development.
- Finally, all the components were combined.

The kernel status, as of this writing, is described in the next chapter.

10.9 Summary

In this chapter most of the kernel implementation has been described. The main focus has been on hardware architecture specific issues, and how the hardware is used in the kernel.

Chapter 11

Status

This chapter first gives a short overview of the kernel status, as of this writing. After this, the future development of the kernel is described. During the development of the kernel a lot of ideas for future projects came to mind, as well as some small improvements. This chapter gives a summary of some of those ideas.

11.1 Current kernel status

All the components described in "Kernel design" (chapter 8) has been implemented and tested to the extend possible at the time of the implementation. The components have not been exhaustively tested, since this would be very difficult before all the components of the kernel have been combined.

After all the kernel components were combined, the kernel were handed over to another project student. At the time of the handover, context switch, as described in section 10.5 of "Kernel implementation", was not fully working, thus the kernel was not tested, as a whole, before the handover.

11.2 Small kernel improvements

Besides fixing bugs in the context switch implementation, there are several other small improvements, which should be done:

- Every time the kernel changes to the Idle process, it does a full context switch. This is unnecessary and it increases latency.
- Currently the cache is disabled. Enabling the cache again will give a vast increase in the performance. This would require implementing cache flush functions and call these after a write to devices, which are mapped in the cache-able memory segment.
- Create a real serial driver interface. Chances are that the serial port will be used for other devices than the serial terminal and this would require a clean interface to the serial port.
- A faster scheduler implementation, especially the switch from the Idle process can improved.

11.3 Large kernel related projects

There are several kernel or operating system- related projects, which could be based on this kernel project. To name a few:

Memory protection Experimenting with memory protection domains as a way of protecting processes from each other. For example, creating the worlds smallest SASOS. Experiment with other types of memory protection, such as the one mentioned in section 8.1.

Porting Porting the kernel to other platforms such as the AT91 or the Intel I386 architecture. The kernel has been designed and implemented with portability in mind and should be fairly easy to port to other platforms.

Remote debugging Remote debugging requires special support in the kernel. Enabling remote kernel debugging would be a useful feature and an interesting project.

Scheduling Scheduling experimentation and analysis. The kernel is small and easy to edit and different scheduling algorithms could be tried out without modifying much of the kernel. For example, changing the scheduler to “earliest deadline first” would not require touching any tricky assembly code at all.

Real-time Modification of the kernel in such way that it would be suitable as a hard real-time system. This would not require much coding, but a thorough analysis of the kernel.

Kernel locking Create a finer grained locking, and maybe even new mechanisms for the locking such as spin-locks. To reduce the kernel latency, locking needs thorough analysing.

11.4 Summary

Hopefully, some of the above suggestion will inspire other students to continue the work on this kernel project.

Chapter 12

Conclusion

A development system environment suitable for embedded system development has been composed. The environment includes Free Software tools and some interesting, reasonably fast and easy to use hardware with a good community and commercial support.

The resulting kernel, which has been developed during this thesis, is very small. The source code is approximately 3500 lines of C and assembly code and this includes all comments and header files. Even though its size is very limited, it does include some important features, to mention a few:

- Effective timer implementation
- Binary Semaphores which implements the basic priority inheritance protocol
- A pre-emptive nano-kernel which reduces latency

The resulting kernel is highly modularized and the scheduler can be changed with minimum effort. With reasonably little coding effort the kernel could be used in hard real-time systems.

The kernel has been designed and implemented with portability in mind and as much code as possible has been written in C. The result is a kernel where the porting process to another architecture can be done almost by rewriting the assembly routines to fit the new architecture.

Even though there has been hard times carrying out this thesis, with linker bugs and hardware bugs that took several weeks to work around, it has been an interesting and educational project. It has given me a good understanding of embedded development, hardware bootstrapping and initialization, as well as good insight to many different aspects of operating system's theory.

Is has also been a good experience to hand over this kernel project to another student, and enable him to carry on the future kernel work.

Appendix A

Project description

Danish title: Bidrag til udvikling af nanokerne
English title: Contribution to development of a nanokernel

Participant: Lars Munch Christensen

Danish project description:

Det langsigtede mål er at konstruere biblioteksrutiner, der kan karakteriseres som en nanokerne, idet de skal kunne lænkes sammen med dels anvendelses- dels maskinspecifikke rutiner til indlejrede systemer. Ses bort fra koldstart skal et indlejret system kunne opfattes som et enkelt program med flere aktiviteter. Når programmet starter eksisterer kun en enkelt aktivitet, der afvikles uden begrænsninger i privilegier.

Oprettelse og start af aktiviteter skal kunne udtrykkes ved hjælp af nanokernens rutiner. Der skal kunne styres både frivilligt og påtvungent processkifte.

Til udnyttelse af materiel til understøttelse af begrænsninger i forskellige aktiviteters privilegier søges udarbejdet og afprøvet et sæt passende konventioner.

Koldstartsproblemer skal analyseres og behandles med henblik på at reducere afhængigheder af materiel så meget som muligt.

Det konkrete mål for projektet er at implementere en nanokerne så vidt, at et indlejret system kan koldstartes og udnytte en simpel ydre enhed. Indhentning af oplysninger om lignende systemer betragtes som en væsentlig del af projektet.

Bibliography

[1] SMcS FDC37817 Super I/O Controller datasheet.

[2] Atmel, AVR Microcontrollers, STK500 Starter Kit and Development system.

[3] Atmel, AT91EB40 Evaluation Board Users Guide.

[4] Galileo Technology, GT-64120A System Controller for RC4650/4700/5000 and RM526x/527x/7000 CPUs

[5] Intel, 82371AB PCI-TO-ISA / IDE XCELERATOR (PIIX4)

[6] MIPS Technologies, CoreLV User's Manual, Document Number: MD00007, Revision 02.06

[7] MIPS Technologies, Processor Core Family Software User's Manual, Document Number: MD00012, Revision 02.04

[8] MIPS Technologies, 5Kc Processor Core Datasheet, January 15, 2001.

[9] GNU linker ld version 2.10.91 info pages

[10] GNU Automake version 1.4 info pages

[11] S. Tan et al., An Object-Oriented Nano-Kernel for Operating System Hardware Support, Proceedings of the Fourth IWOOOS, IEEE Computer Society, Aug, 1995, Lund, Sweden.

[12] Jeffrey S. Chase, Henry M. Levy, Michale J. Feeley and Edward D. Lazowska, Sharing and Protection in a Single Address Space Operating System, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195 USA.

[13] Design and implementation of an Object-Oriented 64-bit Single Address Space Microkernel, Kevin Murray, Tim Wilkinson, Peter Osmon - SARC, City University. Ashley Saulsbury - Swedish Institute of Computer Science. Tom Stiemerling, Paul Kelly - Imperial College.

[14] Implementation and Performance of the Mungi Single Address Space Operating System. Jochen Liedtke, The University of New South Wales.

[15] S. Tan et al. An Object-Oriented Nano-Kernel for Operating System Hardware Support, Department of Computer Science, University of Illinois at Urbana-Champaign.

[16] Exception Handling in Embedded C Programs, C/C++ Users Journal, Yonatan Lehman

[17] A. S. Tannenbaum, A. S. Woolhull, Second edition, Operating Systems Design and implementation. Prentice Hall, 1997

[18] Michael Barr, Programming embedded Systems, O'Reilly and Associates, 1999

[19] M. Ben-Ari, Principles of Concurrent Programming.

[20] Dominic Sweetman, See MIPS Run. Morgan Kaufmann Publishers, Inc.

[21] GNU Project, http://gcc.gnu.org

[22] MontaVista, http://www.mvista.com

[23] Linux-VR Project, http://www.linux-vr.org

[24] Red Hat Inc., http://www.redhat.com

[25] Linux on SGI/MIPS, http://oss.sgi.com/mips/

[26] Newlib Library, http://sources.redhat.com/newlib/

[27] Small Device C Compiler project, SDCC http://sdcc.sourceforge.net/

[28] Red Hat eCos, http://www.redhat.com/embedded/technologies/ecos/

[29] The LinuxBIOS Home Page, http://www.acl.lanl.gov/linuxbios/index.html

[30] The LILO Home Page, http://brun.dyndns.org/pub/linux/lilo/

[31] The GRUB Home Page, http://www.gnu.org/software/grub/

[32] The EtherBoot Home Page, http://etherboot.sourceforge.net/

[33] The RedBoot Home Page, http://www.redhat.com/embedded/technologies/redb

[34] The Embedded PowerPC Linux Boot Project, http://ppcboot.sourceforge.net/

Appendix B

Source code

File	Page	File	Page
Makefile	96	include/stddef.h	126
kernel/mips64/link.xn	99	include/system.h	126
include/addrspace.h	100	include/timer.h	128
include/asm.h	102	include/yamon.h	129
include/byteorder.h	103		
include/cpu.h	103	kernel/cpu.c	136
include/interrupt.h	104	kernel/interrupt.c	138
include/kernel.h	105	kernel/kernel.c	142
include/lcd.h	105	kernel/lcd.c	144
include/list.h	105	kernel/list.c	145
include/mipsregs.h	107	kernel/mipsirq.S	147
include/piix4.h	111	kernel/panic.c	149
include/printf.h	113	kernel/process.c	150
include/process.h	113	kernel/sched.c	154
include/regdef.h	114	kernel/semaphore.c	156
include/regoffset.h	115	kernel/serial.c	158
include/rtc.h	117	kernel/setjmp.S	160
include/sched.h	118	kernel/stack.S	161
include/semaphore.h	119	kernel/start.S	162
include/serial.h	119	kernel/test1.c	163
include/setjmp.h	121	kernel/test2.c	163
include/stackframe.h	122	kernel/timer.c	164

: Makefile

```
# ******************************************
# Kernel installation dir
# ******************************************

TFTPDIR = /kernel

# ******************************************
# Active compilation toolchain
# ******************************************

TOOLCHAIN = mips64
#TOOLCHAIN = mips64el

# ******************************************
# Architecture
# ******************************************

ARCH = mips64

# ******************************************
# Endianness EB | EL
# ******************************************

ENDIAN = EB
#ENDIAN = EL

# ******************************************
# Name of kernel
# ******************************************

IMAGENAME = kernel

# ******************************************
# The following stuff should not be touched
# ******************************************

# ******************************************
# Directories
# ******************************************

ROOT     = .
SUBDIRS = lib kernel
SRCDIR  = $(ROOT)
VPATH   = $(SRCDIR)
BINDIR  = $(ROOT)

# ******************************************
# Image file names and map, disassembly file
# ******************************************

IMAGE_BIN = $(IMAGENAME).bin
IMAGE_REC = $(IMAGENAME).rec
IMAGE_ELF = $(IMAGENAME).elf
IMAGE_MAP = $(IMAGENAME).map
IMAGE_DIS = $(IMAGENAME).dis

# ******************************************
# Compiler toolchain
# ********************************************

ifeq ($(TOOLCHAIN),mipsel)
CC       = mipsel-linux-gcc
LD       = mipsel-linux-ld
OBJCOPY = mipsel-linux-objcopy
OBJDUMP = mipsel-linux-objdump
endif

ifeq ($(TOOLCHAIN),mips64)
CC       = mips64-linux-gcc
LD       = mips64-linux-ld
OBJCOPY = mips64-linux-objcopy
OBJDUMP = mips64-linux-objdump
endif

ifeq ($(TOOLCHAIN),mips64el)
CC       = mips64el-linux-gcc
LD       = mips64el-linux-ld
OBJCOPY = mips64el-linux-objcopy
OBJDUMP = mips64el-linux-objdump
endif

# ********************************************
# Compiler and linker options
# ********************************************

INCLUDE = -I$(ROOT)/include
W_OPTS = -Wimplicit -Wformat -Wall -Wstrict-prototypes
W_OPTS_A = -Wformat -Wall -Wstrict-prototypes

ifeq ($(ARCH),mipsel)
DEFS     =
CC_OPTS =
endif

ifeq ($(ARCH),mips64)
DEFS     =
CC_OPTS = -g -Wa,-32 -mcpu=r4600 -mabi=64 -mips3 -G0 -pipe \
  -D$(ENDIAN) -fno-strict-aliasing -g -c -O2 -nostdinc $(INCLUDE) $(DEFS)
CC_OPTS_A = $(CC_OPTS)
endif

LD_SCRIPT = $(ROOT)/kernel/$(ARCH)/link.xn
LD_OPTS = -g -G 0 -static -T $(LD_SCRIPT) -o $(IMAGE_ELF) \
          -Map $(IMAGE_MAP)

ifeq ($(TOOLCHAIN),mips64)
LD_FORMAT = elf64-bigmips
endif

ifeq ($(TOOLCHAIN),mips64el)
LD_FORMAT = elf64-littlemips
endif

# ********************************************
# Files to be compiled
# *********************************************

OBJ    = kernel/start.o       \
         kernel/mipsirq.o     \
         kernel/stack.o       \
         kernel/kernel.o      \
         kernel/serial.o      \
         kernel/lcd.o         \
         kernel/setjmp.o      \
         kernel/cpu.o         \
         kernel/list.o        \
         kernel/timer.o       \
         kernel/interrupt.o   \
         kernel/process.o     \
         kernel/sched.o       \
         kernel/semaphore.o   \
         kernel/panic.o       \
         kernel/test1.o       \
         kernel/test2.o       \
         lib/vsprintf.o

# *********************************************
# Rules
# *********************************************

%.o : %.c
	$(CC) $(W_OPTS) $(CC_OPTS) -o $@ $<

%.o : %.S
	$(CC) $(W_OPTS_A) $(CC_OPTS_A) -o $@ $<

all : prepare $(IMAGE_BIN) $(IMAGE_REC) $(IMAGE_DIS)

prepare:
	rm -f include/arch
	ln -s $(ARCH) include/arch
	rm -f kernel/arch
	ln -s $(ARCH) kernel/arch

$(IMAGE_BIN) : $(IMAGE_ELF)
	$(OBJCOPY) -O binary $(IMAGE_ELF) $(IMAGE_BIN)

$(IMAGE_REC) : $(IMAGE_ELF)
	$(OBJCOPY) -O srec $(IMAGE_ELF) $(IMAGE_REC)

$(IMAGE_DIS) : $(IMAGE_ELF)
	$(OBJDUMP) -S $(IMAGE_ELF) > $(IMAGE_DIS)

$(IMAGE_ELF) : $(OBJ)
	$(LD) $(LD_OPTS) $(OBJ)
	$(OBJCOPY) -O $(LD_FORMAT) --change-addresses=0xffffffff00000000 \
	$(IMAGE_ELF) $(IMAGE_ELF)

install  : $(IMAGE_REC)
	cp $(IMAGE_REC) $(TFTPDIR)

clean :
	rm -f $(OBJ) $(IMAGE_BIN) $(IMAGE_REC) $(IMAGE_DIS)
```

```
        rm -f $(IMAGE_ELF) $(IMAGE_MAP)

realclean : clean
        rm -f include/arch
        rm -f kernel/arch
        find . -name '*~' | xargs rm -f
        find . -name 'semantic.cache*' | xargs rm -f
```

: kernel/mips64/link.xn

```
/*
 * Linker script for the kernel. Created for the 64bit mips
 * big endian achitecture. Since the linker sucks at 64bit elf
 * we link in 32bit elf and then change it afterwards with objcopy.
 */

OUTPUT(kernel.elf)          /* Default output name                  */
OUTPUT_ARCH(mips)           /* Output arch is mips... no shit :-)   */
ENTRY(_start)               /* Entry point of kernel                */

SECTIONS
{
  /**** Code and read-only data ****/

   . = 0x80200000;          /* Here the code should be loaded so we */
                            /* set the location counter to this     */
                            /* address.                             */
   .text . : {

    _ftext = .;             /* Start of code and read-only data     */

    kernel/start.o (.text)  /* This must be the first file since    */
                            /* this has the kernel entry point      */
    *(.text)                /* The rest of the object files         */
    _ecode = .;             /* End of code                          */

    *(.rodata)

    . = ALIGN(8);
    _etext = .;             /* End of code and read-only data       */
  } = 0

  /**** Initialised data ****/

  .data :
  {
    _fdata = .;             /* Start of initialised data            */
    *(.data)

    . = ALIGN(8);

    *(.lit8)                /* Place 8-byte constants here          */
    *(.lit4)                /* Place 4-byte constants here          */
    *(.sdata)               /* Place subsequent data                */

    . = ALIGN(8);

    _edata = .;             /* End of initialised data              */
  }

  /**** Uninitialised data ****/

  _fbss = .;                  /* Start of uninitialised data          */

  .sbss :
  {
    *(.dynsbss)
    *(.sbss)
    *(.sbss.*)
    *(.scommon)               /* Place small common symbols here      */
  }

  .bss :
  {
    *(.dynbss)
    *(.bss)
    *(.bss.*)
    *(COMMON)                 /* Place common symbols here            */

    _sp_end = .;
    /* Allocate room for stack */
    .   = ALIGN(8) ;
    .   += 0x100000 ;
    _sp =  . - 16;
  }

  _end = .;                   /* End of unitialised data              */

  /**** These must appear regardless of  .   ****/
  .gptab.sdata : { *(.gptab.data) *(.gptab.sdata) }
  .gptab.sbss  : { *(.gptab.bss) *(.gptab.sbss) }

  /* Provide the symbols etext, edata and end if they are not defined
   * by the kernel. It in the ISO/ANSI C standard that these should
   * be defined.
   */
  PROVIDE(etext = _etext);
  PROVIDE(edata = .);
  PROVIDE(end = .);
}
```

: include/addrspace.h

```
/*
 * This header defines the address space stuff for the
 * Malta board e.g convertion macros and addresses.
 *
 * Some of the macros has been taken from the Linux kernel
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */
#ifndef _ADDRSPACE_H
#define _ADDRSPACE_H
```

```
/* When addressing a byte you have to byteswap the
 * address due to a bug in the Galileo chip when running
 * in big endian mode.
 */
#ifdef EB
#define swap8addr(addr) ((addr) ^ 0x0000000000000003)
#else
#define swap8addr(addr) addr
#endif


#define MALTA_GT_PORT_BASE (KSEG1ADDR(0x18000000))

/*
 * Malta RTC-device addresses
 */
#define MALTA_RTC_ADR_REG 0x70
#define MALTA_RTC_DAT_REG 0x71

/*
 * TTY addresses
 */
#define TTYS0 0x3F8
#define TTYS1 0x2F8

/*
 * Memory segments (64bit kernel mode addresses)
 */
#define KUSEG                   0x0000000000000000
#define KSEG0                   0 xffffffff80000000
#define KSEG1                   0 xffffffffa0000000
#define KSEG2                   0 xffffffffc0000000
#define KSEG3                   0 xffffffffe0000000

/*
 * Returns the kernel segment base of a given address
 */
#define KSEGX(a)                (((unsigned long)(a)) & 0xe0000000)

/*
 * Map an address to a certain kernel segment
 */
#define KSEG0ADDR(a) ((__typeof__(a)) \
                     (((unsigned long)(a) & 0x000000ffffffffffUL) | KSEG0))
#define KSEG1ADDR(a) ((__typeof__(a)) \
                     (((unsigned long)(a) & 0x000000ffffffffffUL) | KSEG1))
#define KSEG2ADDR(a) ((__typeof__(a)) \
                     (((unsigned long)(a) & 0x000000ffffffffffUL) | KSEG2))
#define KSEG3ADDR(a) ((__typeof__(a)) \
                     (((unsigned long)(a) & 0x000000ffffffffffUL) | KSEG3))

/*
 * Memory segments (64bit kernel mode addresses)
 */
#define XKUSEG                  0x0000000000000000
#define XKSSEG                  0x4000000000000000
#define XKPHYS                  0x8000000000000000
```

```
#define XKSEG               0xc000000000000000
#define CKSEG0              0 xffffffff80000000
#define CKSEG1              0 xffffffffa0000000
#define CKSSEG              0 xffffffffc0000000
#define CKSEG3              0 xffffffffe0000000

/*
 * Memory segments sizes
 */
#define KUSIZE        0x0000010000000000          /* 2^^40 */
#define KUSIZE_64     0x0000010000000000          /* 2^^40 */
#define K0SIZE        0x0000001000000000          /* 2^^36 */
#define K1SIZE        0x0000001000000000          /* 2^^36 */
#define K2SIZE        0x000000ff80000000
#define KSEGSIZE      0x000000ff80000000          /* max syssegsz */

#endif /* _ADDRSPACE_H */
```

: include/asm.h

```
/*
 * Some useful macros for MIPS assembler code
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _ASM_H
#define _ASM_H

/*
 * LEAF - declare leaf routine
 */
#define LEAF(symbol)                                \
                .globl  symbol;                     \
                .align   2;                         \
                .type   symbol,@function;           \
                .ent    symbol,0;                   \
symbol:         .frame  sp,0,ra

/*
 * NESTED - declare nested routine entry point
 */
#define NESTED(symbol, framesize, rpc)              \
                .globl  symbol;                     \
                .align   2;                         \
                .type   symbol,@function;           \
                .ent    symbol,0;                   \
symbol:         .frame  sp, framesize, rpc

/*
 * END - mark end of function
 */
#define END(function)                               \
                .end    function;                   \
                .size   function,.-function
```

```
/*
 * EXPORT − export definition of symbol
 */
#define EXPORT(symbol)                                  \
                .globl  symbol;                         \
symbol:

/*
 * Print formated string
 */
#define PROM_PRINT(string)                              \
                .set    push;                           \
                .set    reorder;                        \
                la      a0,8f;                          \
                jal      serial_print ;                 \
                .set    pop;                            \
                TEXT(string)

#define TEXT(msg)                                       \
                .data;                                  \
8:              . asciiz  msg;                          \
                .previous;

#endif /* _ASM_H */
```

: include/byteorder.h

```
/* $Id: byteorder.h,v 1.1.1.1 2001/09/23 15:00:00 lmc Exp $
 *
 * This file is subject to the terms and conditions of the GNU General Public
 * License. See the file "COPYING" in the main directory of this archive
 * for more details.
 *
 * Copyright (C) 1996, 1999 by Ralf Baechle
 */
#ifndef _ASM_BYTEORDER_H
#define _ASM_BYTEORDER_H

#include <asm/types.h>

#ifdef __GNUC__

#if !defined(__STRICT_ANSI__) || defined(__KERNEL__)
# define __BYTEORDER_HAS_U64__
#endif

#endif /* __GNUC__ */

#if defined (__MIPSEB__)
# include <linux/byteorder/big_endian.h>
#elif defined (__MIPSEL__)
# include <linux/byteorder/little_endian.h>
#else
# error "MIPS, but neither __MIPSEB__, nor __MIPSEL__???"
#endif

#endif /* _ASM_BYTEORDER_H */
```

: include/cpu.h

```
/*
 * CPU functions
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _CPU_H
#define _CPU_H

/*
 * Assigned values for the product ID register. In order to detect a
 * certain CPU type exactly eventually additional registers may need to
 * be examined.
 */
#define PRID_IMP_R2000 0x0100
#define PRID_IMP_R3000 0x0200        /* Same as R2000A */
#define PRID_IMP_R6000 0x0300        /* Same as R3000A */
#define PRID_IMP_R4000 0x0400
#define PRID_IMP_R6000A 0x0600
#define PRID_IMP_R10000 0x0900
#define PRID_IMP_R12000 0x0e00
#define PRID_IMP_R4300 0x0b00
#define PRID_IMP_R12000 0x0e00
#define PRID_IMP_R8000 0x1000
#define PRID_IMP_R4600 0x2000
#define PRID_IMP_R4700 0x2100
#define PRID_IMP_R4640 0x2200
#define PRID_IMP_R4650 0x2200        /* Same as R4640 */
#define PRID_IMP_R5000 0x2300
#define PRID_IMP_SONIC 0x2400
#define PRID_IMP_MAGIC 0x2500
#define PRID_IMP_RM7000 0x2700
#define PRID_IMP_NEVADA 0x2800
#define PRID_IMP_5KC 0x8100
#define PRID_IMP_20KC 0x8200

void cpu_init(void);

void cpu_status(void);

void cpu_probe(void);

void cpu_speed(void);

#endif /* _CPU_H */
```

: include/interrupt.h

```
/*
 * Interrupt functions
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */
```

```
#ifndef _INTERRUPT_H
#define _INTERRUPT_H

#include <regoffset.h>

extern int interrupt_nested;

typedef void (*interrupt_handler)(void);

void interrupt_register (int irq , interrupt_handler handler);

void interrupt_hw( reg_offset *regs );

//void interrupt_timer(struct reg_offset *regs );

void interrupt_init (void);

#endif /* _INTERRUPT_H */
```

: include/kernel.h

```
/*
 * Kernel header. Global kernel stuff
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _KERNEL_H
#define _KERNEL_H

void panic(char *buf);

#endif /* _KERNEL_H */
```

: include/lcd.h

```
/*
 * LCD Display driver header
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _LCD_H
#define _LCD_H

void lcd_int(unsigned int num);

void lcd_message(const char* str);

#endif /* _LCD_H */
```

: include/list.h

```
/*
 * Double non-circular linked list functions
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _LIST_H
#define _LIST_H

#include <stddef.h>

/* The StructOffset macro returns the byte offset of the field "field"
 * in the structure "st"
 */
#define StructOffset(st, field) \
        ((long) &(((st*)0)->field))

/* The StructBase returns a pointer to the structure of type "st"
 * where "ptr" is pointing to the filed "field" in that structure.
 */
#define StructBase(ptr, st, field) \
      ((st *) (((unsigned char*)(ptr)) - StructOffset(st, field)))

typedef struct s_list_element {
        struct s_list_element * pNext;
        struct s_list_element * pPrev;
} t_list_element;

typedef struct {
        int               number;
        t_list_element *  pFirst;
        t_list_element *  pLast;
} t_list_head;

#define list_empty(pHead) ((int)((pHead)->pFirst == NULL))

void  list_init ( t_list_head * pHead);

void list_put ( t_list_head *    pHead,
               t_list_element * pElement);

void  list_put_after ( t_list_head *    pHead,
                       t_list_element * pElement1,
                       t_list_element * pElement2);

void  list_put_before ( t_list_head *    pHead,
                        t_list_element * pElement1,
                        t_list_element * pElement2);

t_list_element *  list_get ( t_list_head * pHead);

void list_remove( t_list_head *    pHead,
                  t_list_element * pElement);

int  list_length ( t_list_head *    pHead);
```

```
#endif /* _LIST_H */
```

: include/mipsregs.h

```
/*
 * MIPS registers
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */
#ifndef _MIPSREGS_H
#define _MIPSREGS_H

/*
 * The following macros are especially useful for __asm__
 * inline assembler.
 */
#ifndef __STR
#define __STR(x) #x
#endif
#ifndef STR
#define STR(x) __STR(x)
#endif

/*
 * Coprocessor 0 control register names
 */
#define CP0_INDEX $0
#define CP0_RANDOM $1
#define CP0_ENTRYLO0 $2
#define CP0_ENTRYLO1 $3
#define CP0_CONTEXT $4
#define CP0_PAGEMASK $5
#define CP0_WIRED $6
#define CP0_BADVADDR $8
#define CP0_COUNT $9
#define CP0_ENTRYHI $10
#define CP0_COMPARE $11
#define CP0_STATUS $12
#define CP0_CAUSE $13
#define CP0_EPC $14
#define CP0_PRID $15
#define CP0_CONFIG $16
#define CP0_LLADDR $17
#define CP0_WATCHLO $18
#define CP0_WATCHHI $19
#define CP0_XCONTEXT $20
#define CP0_FRAMEMASK $21
#define CP0_DIAGNOSTIC $22
#define CP0_PERFORMANCE $25
#define CP0_ECC $26
#define CP0_CACHEERR $27
#define CP0_TAGLO $28
#define CP0_TAGHI $29
#define CP0_ERROREPC $30
#define CP0_DESAVE $31
```

```
/*
 * Macros to access the system control coprocessor
 */
#define read_32bit_cp0_register(source)                         \
({ int __res ;                                                  \
        __asm__ __volatile__ (                                  \
        "mfc0\t%0,"STR(source)                                  \
        : "=r" (__res ));                                       \
        __res ;})

#define write_32bit_cp0_register(register,value)                \
        __asm__ __volatile__ (                                  \
        "mtc0\t%0,"STR(register)                                \
        : : "r" (value));

/*
 * R4x00 interrupt enable / cause bits
 */
#define IE_SW0          (1<< 8)
#define IE_SW1          (1<< 9)
#define IE_IRQ0         (1<<10)
#define IE_IRQ1         (1<<11)
#define IE_IRQ2         (1<<12)
#define IE_IRQ3         (1<<13)
#define IE_IRQ4         (1<<14)
#define IE_IRQ5         (1<<15)

/*
 * R4x00 interrupt cause bits
 */
#define C_SW0           (1<< 8)
#define C_SW1           (1<< 9)
#define C_IRQ0          (1<<10)
#define C_IRQ1          (1<<11)
#define C_IRQ2          (1<<12)
#define C_IRQ3          (1<<13)
#define C_IRQ4          (1<<14)
#define C_IRQ5          (1<<15)

#ifndef _LANGUAGE_ASSEMBLY
/*
 * Manipulate the status register .
 * Mostly used to access the interrupt bits .
 */
#define __BUILD_SET_CP0(name,register)                          \
extern __inline__ unsigned int                                  \
set_cp0_##name(unsigned int change, unsigned int new)           \
{                                                               \
        unsigned int res;                                       \
                                                                \
        res = read_32bit_cp0_register (register);               \
        res &= ~change;                                         \
        res |= (new & change);                                  \
        write_32bit_cp0_register (register, res );              \
                                                                \
        return res;                                             \
}
```

```
__BUILD_SET_CP0(status,CP0_STATUS)
__BUILD_SET_CP0(cause,CP0_CAUSE)
__BUILD_SET_CP0(config,CP0_CONFIG)

#endif /* defined (_LANGUAGE_ASSEMBLY) */

/*
 * Bitfields in the R4xx0 cp0 status register
 */
#define ST0_IE                  0x00000001
#define ST0_EXL                 0x00000002
#define ST0_ERL                 0x00000004
#define ST0_KSU                 0x00000018
#  define KSU_USER              0x00000010
#  define KSU_SUPERVISOR        0x00000008
#  define KSU_KERNEL            0x00000000
#define ST0_UX                  0x00000020
#define ST0_SX                  0x00000040
#define ST0_KX                  0x00000080
#define ST0_DE                  0x00010000
#define ST0_CE                  0x00020000

/*
 * Status register bits available in all MIPS CPUs.
 */
#define ST0_IM                  0x0000ff00
#define STATUSB_IP0             8
#define STATUSF_IP0             (1   << 8)
#define STATUSB_IP1             9
#define STATUSF_IP1             (1   << 9)
#define STATUSB_IP2             10
#define STATUSF_IP2             (1   << 10)
#define STATUSB_IP3             11
#define STATUSF_IP3             (1   << 11)
#define STATUSB_IP4             12
#define STATUSF_IP4             (1   << 12)
#define STATUSB_IP5             13
#define STATUSF_IP5             (1   << 13)
#define STATUSB_IP6             14
#define STATUSF_IP6             (1   << 14)
#define STATUSB_IP7             15
#define STATUSF_IP7             (1   << 15)
#define ST0_CH                  0x00040000
#define ST0_SR                  0x00100000
#define ST0_TS                  0x00200000
#define ST0_BEV                 0x00400000
#define ST0_RE                  0x02000000
#define ST0_FR                  0x04000000
#define ST0_CU                  0xf0000000
#define ST0_CU0                 0x10000000
#define ST0_CU1                 0x20000000
#define ST0_CU2                 0x40000000
#define ST0_CU3                 0x80000000
#define ST0_XX                  0x80000000      /* MIPS IV naming */

/*
 * Bitfields and bit numbers in the coprocessor 0 cause register .
 *
 * Refer to your MIPS R4xx0 manual, chapter 5 for explanation.
 */
#define CAUSEB_EXCCODE  2
#define CAUSEF_EXCCODE  (31  << 2)
#define CAUSEB_IP       8
#define CAUSEF_IP       (255 << 8)
#define CAUSEB_IP0      8
#define CAUSEF_IP0      (1   << 8)
#define CAUSEB_IP1      9
#define CAUSEF_IP1      (1   << 9)
#define CAUSEB_IP2      10
#define CAUSEF_IP2      (1   << 10)
#define CAUSEB_IP3      11
#define CAUSEF_IP3      (1   << 11)
#define CAUSEB_IP4      12
#define CAUSEF_IP4      (1   << 12)
#define CAUSEB_IP5      13
#define CAUSEF_IP5      (1   << 13)
#define CAUSEB_IP6      14
#define CAUSEF_IP6      (1   << 14)
#define CAUSEB_IP7      15
#define CAUSEF_IP7      (1   << 15)
#define CAUSEB_IV       23
#define CAUSEF_IV       (1   << 23)
#define CAUSEB_CE       28
#define CAUSEF_CE       (3   << 28)
#define CAUSEB_BD       31
#define CAUSEF_BD       (1   << 31)

/*
 * Bits in the coprozessor 0 config register .
 */
#define CONF_CM_CACHABLE_NO_WA  0
#define CONF_CM_CACHABLE_WA     1
#define CONF_CM_UNCACHED        2
#define CONF_CM_CACHABLE_NONCOHERENT 3
#define CONF_CM_CACHABLE_CE     4
#define CONF_CM_CACHABLE_COW    5
#define CONF_CM_CACHABLE_CUW    6
#define CONF_CM_CACHABLE_ACCELERATED 7
#define CONF_CM_CMASK           7
#define CONF_DB                 (1 << 4)
#define CONF_IB                 (1 << 5)
#define CONF_SC                 (1 << 17)

/*
 * Events counted by counter #0
 */
#define CE0_CYCLES              0
#define CE0_INSN_ISSUED         1
#define CE0_LPSC_ISSUED         2
#define CE0_S_ISSUED            3
#define CE0_SC_ISSUED           4
#define CE0_SC_FAILED           5
#define CE0_BRANCH_DECODED      6
#define CE0_QW_WB_SECONDARY     7
#define CE0_CORRECTED_ECC_ERRORS 8
#define CE0_ICACHE_MISSES       9
```

```
#define CE0_SCACHE_I_MISSES        10
#define CE0_SCACHE_I_WAY_MISSPREDICTED 11
#define CE0_EXT_INTERVENTIONS_REQ 12
#define CE0_EXT_INVALIDATE_REQ  13
#define CE0_VIRTUAL_COHERENCY_COND 14
#define CE0_INSN_GRADUATED       15

/*
 * Events counted by counter #1
 */
#define CE1_CYCLES                 0
#define CE1_INSN_GRADUATED       1
#define CE1_LPSC_GRADUATED       2
#define CE1_S_GRADUATED          3
#define CE1_SC_GRADUATED         4
#define CE1_FP_INSN_GRADUATED  5
#define CE1_QW_WB_PRIMARY        6
#define CE1_TLB_REFILL           7
#define CE1_BRANCH_MISSPREDICTED 8
#define CE1_DCACHE_MISS          9
#define CE1_SCACHE_D_MISSES      10
#define CE1_SCACHE_D_WAY_MISSPREDICTED 11
#define CE1_EXT_INTERVENTION_HITS 12
#define CE1_EXT_INVALIDATE_REQ  13
#define CE1_SP_HINT_TO_CEXCL_SC_BLOCKS 14
#define CE1_SP_HINT_TO_SHARED_SC_BLOCKS 15

/*
 * These flags define in which priviledge mode the counters count events
 */
#define CEB_USER     8        /* Count events in user mode, EXL = ERL = 0 */
#define CEB_SUPERVISOR 4 /* Count events in supvervisor mode EXL = ERL = 0 */
#define CEB_KERNEL 2          /* Count events in kernel mode EXL = ERL = 0 */
#define CEB_EXL      1        /* Count events with EXL = 1, ERL = 0 */

#endif /* _MIPSREGS_H */
```

: include/piix4.h

```
/*
 * Carsten Langgaard, carstenl@mips.com
 * Copyright (C) 2000 MIPS Technologies, Inc. All rights reserved.
 *
 *  This program is free software; you can distribute it and/or modify it
 *  under the terms of the GNU General Public License (Version 2) as
 *  published by the Free Software Foundation.
 *
 *  This program is distributed in the hope it will be useful, but WITHOUT
 *  ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 *  FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
 *  for more details.
 *
 *  You should have received a copy of the GNU General Public License along
 *  with this program; if not, write to the Free Software Foundation, Inc.,
 *  59 Temple Place - Suite 330, Boston MA 02111-1307, USA.
 *
 * Register definitions for Intel PIIX4 South Bridge Device.
 *
 */

#ifndef PIIX4_H
#define PIIX4_H

/************************************************************************
 * IO register offsets
 ************************************************************************/
#define PIIX4_ICTLR1_ICW1  0x20
#define PIIX4_ICTLR1_ICW2  0x21
#define PIIX4_ICTLR1_ICW3  0x21
#define PIIX4_ICTLR1_ICW4  0x21
#define PIIX4_ICTLR2_ICW1  0xa0
#define PIIX4_ICTLR2_ICW2  0xa1
#define PIIX4_ICTLR2_ICW3  0xa1
#define PIIX4_ICTLR2_ICW4  0xa1
#define PIIX4_ICTLR1_OCW1 0x21
#define PIIX4_ICTLR1_OCW2 0x20
#define PIIX4_ICTLR1_OCW3 0x20
#define PIIX4_ICTLR1_OCW4 0x20
#define PIIX4_ICTLR2_OCW1 0xa1
#define PIIX4_ICTLR2_OCW2 0xa0
#define PIIX4_ICTLR2_OCW3 0xa0
#define PIIX4_ICTLR2_OCW4 0xa0


/************************************************************************
 * Register encodings.
 ************************************************************************/
#define PIIX4_OCW2_NSEOI  (0x1 << 5)
#define PIIX4_OCW2_SEOI   (0x3 << 5)
#define PIIX4_OCW2_RNSEOI (0x5 << 5)
#define PIIX4_OCW2_RAEOIS (0x4 << 5)
#define PIIX4_OCW2_RAEOIC (0x0 << 5)
#define PIIX4_OCW2_RSEOI  (0x7 << 5)
#define PIIX4_OCW2_SP     (0x6 << 5)
#define PIIX4_OCW2_NOP    (0x2 << 5)

#define PIIX4_OCW2_SEL    (0x0 << 3)

#define PIIX4_OCW2_ILS_0    0
#define PIIX4_OCW2_ILS_1    1
#define PIIX4_OCW2_ILS_2    2
#define PIIX4_OCW2_ILS_3    3
#define PIIX4_OCW2_ILS_4    4
#define PIIX4_OCW2_ILS_5    5
#define PIIX4_OCW2_ILS_6    6
#define PIIX4_OCW2_ILS_7    7
#define PIIX4_OCW2_ILS_8    0
#define PIIX4_OCW2_ILS_9    1
#define PIIX4_OCW2_ILS_10   2
#define PIIX4_OCW2_ILS_11   3
#define PIIX4_OCW2_ILS_12   4
#define PIIX4_OCW2_ILS_13   5
#define PIIX4_OCW2_ILS_14   6
#define PIIX4_OCW2_ILS_15   7

#define PIIX4_OCW3_SEL    (0x1 << 3)
```

```
#define PIIX4_OCW3_IRR      0x2
#define PIIX4_OCW3_ISR      0x3

#endif /* !(PIIX4_H) */
```

: include/printf.h

```
/*
 * Printf Header
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _PRINTF_H
#define _PRINTF_H

#include <arch/stdarg.h>

/*
 * Format a string and place it in a buffer
 */
int sprintf(char * buf, const char *fmt, ...);

/*
 * Send a print message to the console driver.
 * Used for debug only.
 */
void printf(char *fmt , ...);

#endif /* _PRINTF_H */
```

: include/process.h

```
/*
 * Process header file
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _PROCESS_H
#define _PROCESS_H

#include <list.h>

/* Size of the kernel/idle process stack in dwords */
#define KERNEL_STACK 2000

/* The maximum number of processes */
#define MAX_PROCESSES 20

/* Process states */
#define READY 0
#define RUNNING 1
```

```
#define WAITING 2

typedef struct {
        t_list_element         process_elem;
        int                    id;
        int                    priority;
        int                    orig_priority ; /* original priority of a process */
        int                    state;
        unsigned long*         stack_pointer;
} t_process;

#define process_base(ple) StructBase(ple, t_process, process_elem)

/* Pointer to current running process */
extern t_process* process_current;
extern t_process* process_old;

/* Sorted of ready processes */
extern t_list_head process_list ;

void process_init(void);

void process_insert( t_list_head * pHead, t_list_element * pElement);

void process_reorder( t_list_head * pHead, t_list_element * pElement);

void process_create(void (*function)(void),
                    int priority ,
                    int stack_size );

void process_list_print ( t_list_head * pHead);

#endif /* _PROCESS_H */
```

: include/regdef.h

```
/*
 * Register definitions
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */
#ifndef _REGDEF_H
#define _REGDEF_H

#define zero   $0     /* wired zero */
#define AT     $at    /* assembler temp - uppercase because of ".set at" */
#define v0     $2     /* return value - caller saved */
#define v1     $3
#define a0     $4     /* argument registers */
#define a1     $5
#define a2     $6
#define a3     $7
#define t0     $8     /* caller saved in 32 bit (arg reg 64 bit) */
#define t1     $9
#define t2     $10
#define t3     $11
```

```
#define t4     $12     /* caller saved */
#define t5     $13
#define t6     $14
#define t7     $15
#define s0     $16     /* callee saved */
#define s1     $17
#define s2     $18
#define s3     $19
#define s4     $20
#define s5     $21
#define s6     $22
#define s7     $23
#define t8     $24     /* caller saved */
#define t9     $25     /* callee address for PIC/temp */
#define k0     $26     /* kernel temporary */
#define k1     $27
#define gp     $28     /* global pointer - caller saved for PIC */
#define sp     $29     /* stack pointer */
#define fp     $30     /* frame pointer */
#define s8     $30     /* callee saved */
#define ra     $31     /* return address */

#endif /* _REGDEF_H */
```

: include/regoffset.h

```
/*
 * MIPS regs offsets
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _REGOFFSET_H
#define _REGOFFSET_H

#ifndef _LANGUAGE_ASSEMBLY

/* MIPS register offsets for use in C */

typedef struct {
        /* Saved main processor registers. */
        unsigned long zero;
        unsigned long at;
        unsigned long v0;
        unsigned long v1;
        unsigned long a0;
        unsigned long a1;
        unsigned long a2;
        unsigned long a3;
        unsigned long t0;
        unsigned long t1;
        unsigned long t2;
        unsigned long t3;
        unsigned long t4;
        unsigned long t5;
        unsigned long t6;
        unsigned long t7;
        unsigned long s0;
        unsigned long s1;
        unsigned long s2;
        unsigned long s3;
        unsigned long s4;
        unsigned long s5;
        unsigned long s6;
        unsigned long s7;
        unsigned long t8;
        unsigned long t9;
        unsigned long k0;
        unsigned long k1;
        unsigned long gp;
        unsigned long sp;
        unsigned long s8;
        unsigned long fp;
        unsigned long ra;

        /* Other saved registers. */
        unsigned long lo;
        unsigned long hi;

        /* Saved cp0 registers. */
        unsigned long cp0_epc;
        unsigned long cp0_badvaddr;
        unsigned long cp0_status;
        unsigned long cp0_cause;
} reg_offset ;

#else

/* MIPS register offsets for use in assembler */

#define R_ZERO 0
#define R_AT   8
#define R_V0   16
#define R_V1   24
#define R_A0   32
#define R_A1   40
#define R_A2   48
#define R_A3   56
#define R_T0   64
#define R_T1   72
#define R_T2   80
#define R_T3   88
#define R_T4   96
#define R_T5   104
#define R_T6   112
#define R_T7   120
#define R_S0   128
#define R_S1   136
#define R_S2   144
#define R_S3   152
#define R_S4   160
#define R_S5   168
#define R_S6   176
#define R_S7   184
```

```
#define R_T8   192
#define R_T9   200
#define R_K0   208
#define R_K1   216
#define R_GP   224
#define R_SP   232
#define R_S8   240
#define R_FP   240
#define R_RA   248
#define R_LO   256
#define R_HI   264
#define R_EPC 272
#define R_BVADDR 280
#define R_STATUS 288
#define R_CAUSE 296
#define R_SIZE 304

#endif /* _LANGUAGE_ASSEMBLY */

#endif /* _REGOFFSET_H */
```

: include/rtc.h

```
/*
 * Register  definitions  for  the Real−Time−Clock / CMOS RAM
 *
 * Modified for  use  in  this  kernel  by Lars Munch, 2001
 *
 * Copyright Torsten Duwe <duwe@informatik.uni−erlangen.de> 1993
 * derived  from  Data Sheet, Copyright Motorola 1984 (!).
 * It  was written to be part  of  the Linux operating system.
 *
 * This  file   is  subject  to the terms and conditions of the GNU General
 * Public License.   See the  file  "COPYING" in the main directory of
 * this  archive  for  more details.
 */

#ifndef _RTC_H
#define _RTC_H

/* Registers */
#define RTC_SECONDS         0
#define RTC_SECONDS_ALARM 1
#define RTC_MINUTES         2
#define RTC_MINUTES_ALARM 3
#define RTC_HOURS           4
#define RTC_HOURS_ALARM  5
#define RTC_DAY_OF_WEEK  6
#define RTC_DAY_OF_MONTH 7
#define RTC_MONTH           8
#define RTC_YEAR            9

/* Control  registers */
#define RTC_REG_A           10
#define RTC_REG_B           11
#define RTC_REG_C           12
#define RTC_REG_D           13
```

```
/***************************************************************
 * register details
 ***************************************************************/
#define RTC_FREQ_SELECT RTC_REG_A

/* update-in-progress - set to "1" 244 microsecs before RTC goes off the bus,
 * reset after update (may take 1.984ms @ 32768Hz RefClock) is complete,
 * totalling to a max high interval of 2.228 ms.
 */
# define RTC_UIP          0x80
# define RTC_DIV_CTL      0x70
   /* divider control: refclock values 4.194 / 1.049 MHz / 32.768 kHz */
# define RTC_REF_CLCK_4MHZ 0x00
# define RTC_REF_CLCK_1MHZ 0x10
# define RTC_REF_CLCK_32KHZ 0x20
   /* 2 values for divider stage reset, others for "testing purposes only" */
# define RTC_DIV_RESET1   0x60
# define RTC_DIV_RESET2   0x70
  /* Periodic intr. / Square wave rate select. 0=none, 1=32.8kHz,... 15=2Hz */
# define RTC_RATE_SELECT  0x0F


#define RTC_CONTROL RTC_REG_B
# define RTC_SET 0x80          /* disable updates for clock setting */
# define RTC_PIE 0x40          /* periodic interrupt enable */
# define RTC_AIE 0x20          /* alarm interrupt enable */
# define RTC_UIE 0x10          /* update-finished interrupt enable */
# define RTC_SQWE 0x08         /* enable square-wave output */
# define RTC_DM_BINARY 0x04 /* all time/date values are BCD if clear */
# define RTC_24H 0x02          /* 24 hour mode - else hours bit 7 means pm */
# define RTC_DST_EN 0x01       /* auto switch DST - works f. USA only */


#define RTC_INTR_FLAGS RTC_REG_C
/* caution - cleared by read */
# define RTC_IRQF 0x80         /* any of the following 3 is active */
# define RTC_PF 0x40
# define RTC_AF 0x20
# define RTC_UF 0x10

#define RTC_VALID   RTC_REG_D
# define RTC_VRT 0x80          /* valid RAM and time */

#endif /* _RTC_H */
```

: include/sched.h

```
/*
 * Sched Header
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _SCHED_H
#define _SCHED_H
```

```
extern int sched_now;

void schedule(void);

void schedule_frominterrupt(void);

#endif /* _SCHED_H */
```

: include/semaphore.h

```
/*
 * Semaphore definitions
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _SEMAPHORE_H
#define _SEMAPHORE_H

#include <list.h>
#include <process.h>

/* Semaphore state */
#define FREE 0
#define LOCKED 1

typedef struct {
        t_list_head       list_waiting ;  /* Waiting processes list */
        int               state;          /* Semaphore state */
        t_process        *owner;          /* process using the semaphore */
} t_semaphore;

void semaphore_setup(t_semaphore* pS);
void semaphore_wait(t_semaphore* pS);
void semaphore_signal(t_semaphore* pS);

#endif /* _SEMAPHORE_H */
```

: include/serial.h

```
/*
 * This header file defines all the registers and
 * settings of the FDC37M817 on the Malta board.
 * Should be compatible with the NS16C550A, the
 * 16450 ACE and the NS16C550A.
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _SERIAL_H
#define _SERIAL_H

/*
 * Addressing the Serial Port
 */
#define UART_RX      0      /* In:  Receive buffer (DLAB=0) */
#define UART_TX      0      /* Out: Transmit buffer (DLAB=0) */
#define UART_DLL     0      /* Out: Divisor Latch Low (DLAB=1) */
#define UART_IER     1      /* I/O: Interrupt Enable Register
                             * (DLAB=0) */
#define UART_DLM     1      /* Out: Divisor Latch High (DLAB=1) */
#define UART_IIR     2      /* In:  Interrupt ID Register */
#define UART_FCR     2      /* Out: FIFO Control Register */
#define UART_LCR     3      /* I/O: Line Control Register */
#define UART_MCR     4      /* I/O: Modem Control Register */
#define UART_LSR     5      /* I/O: Line Status Register */
#define UART_MSR     6      /* I/O: Modem Status Register */
#define UART_SCR     7      /* I/O: Scratchpad */

/*
 * These are the definitions for the Interrupt Enable Register (IER)
 */
#define UART_IER_RDI 0x01 /* Enable receiver data
                             * available interrupt */
#define UART_IER_THRI 0x02 /* Enable Transmitter holding
                             * empty register int. */
#define UART_IER_RLSI 0x04 /* Enable receiver line status interrupt */
#define UART_IER_MSI 0x08 /* Enable Modem status interrupt */

/*
 * These are the definitions for the FIFO Control Register (FCR)
 */
#define UART_FCR_ENABLE_FIFO 0x01 /* Enable the RCVR/XMIT FIFO */
#define UART_FCR_CLEAR_RCVR 0x02 /* Clear the RCVR FIFO */
#define UART_FCR_CLEAR_XMIT 0x04 /* Clear the XMIT FIFO */
#define UART_FCR_TRIGGER_1 0x00 /* Mask for trigger set at 1 */
#define UART_FCR_TRIGGER_4 0x40 /* Mask for trigger set at 4 */
#define UART_FCR_TRIGGER_8 0x80 /* Mask for trigger set at 8 */
#define UART_FCR_TRIGGER_14 0xC0 /* Mask for trigger set at 14 */

/*
 * These are the definitions for the Interrupt Identification Register (IIR)
 */
#define UART_IIR_NO_INT 0x01 /* No interrupts pending */
#define UART_IIR_ID 0x06     /* Mask for the interrupt ID */

#define UART_IIR_MSI 0x00   /* Modem status interrupt */
#define UART_IIR_THRI 0x02 /* Transmitter holding register empty */
#define UART_IIR_RDI 0x04   /* Receiver data interrupt */
#define UART_IIR_RLSI 0x06 /* Receiver line status interrupt */
#define UART_IIR_CTII 0x0C /* Character timeout ID interrupt */

/*
 * These are the definitions for the Line Control Register (LCR)
 *
 * Note: if the word length is 5 bits (UART_LCR_WLEN5), then setting
 * UART_LCR_STOP will select 1.5 stop bits, not 2 stop bits.
 */
#define UART_LCR_WLEN5 0x00 /* Wordlength: 5 bits */
#define UART_LCR_WLEN6 0x01 /* Wordlength: 6 bits */
#define UART_LCR_WLEN7 0x02 /* Wordlength: 7 bits */
#define UART_LCR_WLEN8 0x03 /* Wordlength: 8 bits */
```

```
#define UART_LCR_STOP 0x04 /* Stop bits: 0=1 stop bit, 1= 2 stop bits */
#define UART_LCR_PARITY 0x08 /* Parity Enable */
#define UART_LCR_EPAR 0x10 /* Even parity select */
#define UART_LCR_SPAR 0x20 /* Stick parity */
#define UART_LCR_SBC 0x40 /* Set break control */
#define UART_LCR_DLAB 0x80 /* Divisor latch access bit */

/*
 * These are the definitions for the Modem Control Register (MCR)
 */
#define UART_MCR_DTR 0x01 /* DTR complement */
#define UART_MCR_RTS 0x02 /* RTS complement */
#define UART_MCR_OUT1 0x04 /* Out1 complement */
#define UART_MCR_OUT2 0x08 /* Out2 complement */
#define UART_MCR_LOOP 0x10 /* Enable loopback test mode */

/*
 * These are the definitions for the Line Status Register (LSR)
 */
#define UART_LSR_DR 0x01 /* Receiver data ready */
#define UART_LSR_OE 0x02 /* Overrun error indicator */
#define UART_LSR_PE 0x04 /* Parity error indicator */
#define UART_LSR_FE 0x08 /* Frame error indicator */
#define UART_LSR_BI 0x10 /* Break interrupt indicator */
#define UART_LSR_THRE 0x20 /* Transmit-hold-register empty */
#define UART_LSR_TEMT 0x40 /* Transmitter empty */

/*
 * These are the definitions for the Modem Status Register (MSR)
 */
#define UART_MSR_DCTS 0x01 /* Delta CTS */
#define UART_MSR_DDSR 0x02 /* Delta DSR */
#define UART_MSR_TERI 0x04 /* Trailing edge ring indicator */
#define UART_MSR_DDCD 0x08 /* Delta DCD */
#define UART_MSR_ANY_DELTA 0x0F /* Any of the delta bits! */
#define UART_MSR_CTS 0x10 /* Clear to Send */
#define UART_MSR_DSR 0x20 /* Data Set Ready */
#define UART_MSR_RI 0x40   /* Ring Indicator */
#define UART_MSR_DCD 0x80 /* Data Carrier Detect */

int serial_putchar (char c);

void serial_print (char *buf);

void serial_init (void);

#endif /* _SERIAL_H */
```

: include/setjmp.h

```
/*
 * setjmp header.
 *
 * This file is MIPS64 specific
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _SETJMP_H
#define _SETJMP_H

#define _JBLEN 23
#define _JBTYPE long long

typedef _JBTYPE jmp_buf[_JBLEN];

int  setjmp(jmp_buf);
void longjmp(jmp_buf, int);

#endif /* _SETJMP_H */
```

: include/stackframe.h

```
/*
 * Stackframe macros.
 *
 * Based on code from the Linux kernel.
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _STACKFRAME_H
#define _STACKFRAME_H

#include <asm.h>
#include <addrspace.h>
#include <regdef.h>
#include <regoffset.h>

/*
 * Macros to save the current state
 */
        .macro  SAVE_SOME
        .set    push
        .set    reorder
        //move k1, sp
        move    k0, sp
        dsubu   sp, sp, R_SIZE
        sd      k0, R_SP(sp)
        sd      v1, R_V1(sp)
        sd      zero, R_ZERO(sp)
        dmfc0   v1, CP0_STATUS
        sd      v0, R_V0(sp)
        sd      v1, R_STATUS(sp)
        sd      a0, R_A0(sp)
        dmfc0   v1, CP0_CAUSE
        sd      a1, R_A1(sp)
        sd      v1, R_CAUSE(sp)
        sd      a2, R_A2(sp)
        dmfc0   v1, CP0_EPC
        sd      a3, R_A3(sp)
        sd      v1, R_EPC(sp)
        sd      t9, R_T9(sp)
        sd      gp, R_GP(sp)
        sd      ra, R_RA(sp)
        .set    pop
        .endm

        .macro  SAVE_AT
        .set    push
        .set    noat
        sd      AT, R_AT(sp)
        .set    pop
        .endm

        .macro  SAVE_TEMP
        mfhi    v1
        sd      t0, R_T0(sp)
        sd      t1, R_T1(sp)
        sd      v1, R_HI(sp)
        mflo    v1
        sd      t2, R_T2(sp)
        sd      t3, R_T3(sp)
        sd      v1, R_LO(sp)
        sd      t4, R_T4(sp)
        sd      t5, R_T5(sp)
        sd      t6, R_T6(sp)
        sd      t7, R_T7(sp)
        sd      t8, R_T8(sp)
        .endm

        .macro  SAVE_STATIC
        sd      s0, R_S0(sp)
        sd      s1, R_S1(sp)
        sd      s2, R_S2(sp)
        sd      s3, R_S3(sp)
        sd      s4, R_S4(sp)
        sd      s5, R_S5(sp)
        sd      s6, R_S6(sp)
        sd      s7, R_S7(sp)
        sd      s8, R_S8(sp)
        .endm

        .macro  SAVE_ALL
        SAVE_SOME
        SAVE_AT
        SAVE_TEMP
        SAVE_STATIC
        .endm

/*
 * Macros to restore to some state
 */
        .macro  RESTORE_SOME
        .set    push
/*      .set    reorder
        mfc0    t0, CP0_STATUS
        .set    pop
        ori     t0, 0x1f
        xori    t0, 0x1f
        mtc0    t0, CP0_STATUS
        li      v1, 0xff00
        and     t0, v1
        ld      v0, R_STATUS(sp)
        nor     v1, zero, v1
        and     v0, v1
        or      v0, t0
*/
        .set push
        .set    reorder
        ld      v0, R_STATUS(sp)
        .set pop
        dmtc0   v0, CP0_STATUS
        ld      v1, R_EPC(sp)
        dmtc0   v1, CP0_EPC
        ld      ra, R_RA(sp)
        ld      gp, R_GP(sp)
        ld      t9, R_T9(sp)
        ld      a3, R_A3(sp)
        ld      a2, R_A2(sp)
        ld      a1, R_A1(sp)
        ld      a0, R_A0(sp)
        ld      v1, R_V1(sp)
        ld      v0, R_V0(sp)
        .endm

        .macro  RESTORE_AT
        .set    push
        .set    noat
        ld      AT, R_AT(sp)
        .set    pop
        .endm

        .macro  RESTORE_TEMP
        ld      t8, R_LO(sp)
        ld      t0, R_T0(sp)
        ld      t1, R_T1(sp)
        mtlo    t8
        ld      t8, R_HI(sp)
        ld      t2, R_T2(sp)
        ld      t3, R_T3(sp)
        mthi    t8
        ld      t4, R_T4(sp)
        ld      t5, R_T5(sp)
        ld      t6, R_T6(sp)
        ld      t7, R_T7(sp)
        ld      t8, R_T8(sp)
        .endm

        .macro  RESTORE_STATIC
        ld      s0, R_S0(sp)
        ld      s1, R_S1(sp)
        ld      s2, R_S2(sp)
        ld      s3, R_S3(sp)
        ld      s4, R_S4(sp)
        ld      s5, R_S5(sp)
        ld      s6, R_S6(sp)
        ld      s7, R_S7(sp)
        ld      s8, R_S8(sp)
        .endm

        .macro  RESTORE_SP
        ld      sp, R_SP(sp)
        .endm

        .macro  RESTORE_ALL
        RESTORE_SOME
        RESTORE_AT
        RESTORE_TEMP
        RESTORE_STATIC
        RESTORE_SP
        .endm

/*
 * Disable interrupts.
 */
        .macro  CLI
        .set    push
        .set    reorder
        mfc0    t0, CP0_STATUS
        .set    pop
        li      t1, ST0_ERL | ST0_EXL | ST0_IE
        or      t0, t1
        xori    t0, ST0_ERL | ST0_EXL | ST0_IE
        mtc0    t0, CP0_STATUS
        .endm

/*
 * Enable interrupts.
 */
        .macro  STI
        .set    push
        .set    reorder
        mfc0    t0, CP0_STATUS
        .set    pop
        li      t1, ST0_ERL | ST0_EXL | ST0_IE
        or      t0, t1
        xori    t0, ST0_ERL | ST0_EXL
        mtc0    t0, CP0_STATUS
        .endm

/*
 * Just move to kernel mode and leave interrupts as they are.
 */
        .macro  KMODE
        .set    push
        .set    reorder
        mfc0    t0, CP0_STATUS
        .set    pop
        li      t1, KSU_USER | KSU_SUPERVISOR | ST0_ERL | ST0_EXL
        or      t0, t1
        xori    t0, KSU_USER | KSU_SUPERVISOR | ST0_ERL | ST0_EXL
        mtc0    t0, CP0_STATUS
        .endm

#endif /* _STACKFRAME_H */
```

: include/stddef.h

```
/*
 * Standard definitions header
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _STDDEF_H
#define _STDDEF_H

#define NULL ((void *)0)

#endif /* _STDDEF_H */
```

: include/system.h

```
/*
 * C functions for setting and clearing interrupt flags.
 *
 * Functions are taken from the Linux kernel
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */
#ifndef _SYSTEM_H
#define _SYSTEM_H

extern __inline__ void
__sti (void)
{
        __asm__ __volatile__ (
                ".set\tnoreorder\n\t"
                ".set\tnoat\n\t"
                "mfc0\t$1,$12\n\t"
                "ori\t$1,0x1f\n\t"
                "xori\t$1,0x1e\n\t"
                "mtc0\t$1,$12\n\t"
                ".set\tat\n\t"
                ".set\treorder"
                : /* no outputs */
                : /* no inputs */
                : "$1", "memory");
}

/*
 * For cli() we have to insert nops to make shure that the new value
 * has actually arrived in the status register before the end of this
 * macro.
 * R4000/R4400 need three nops, the R4600 two nops and the R10000 needs
 * no nops at all.
 */
extern __inline__ void
__cli (void)
{
        __asm__ __volatile__ (
                ".set\tnoreorder\n\t"
                ".set\tnoat\n\t"
                "mfc0\t$1,$12\n\t"
                "ori\t$1,1\n\t"
                "xori\t$1,1\n\t"
                "mtc0\t$1,$12\n\t"
                "nop\n\t"
                "nop\n\t"
                "nop\n\t"
                ".set\tat\n\t"
                ".set\treorder"
                : /* no outputs */
                : /* no inputs */
                : "$1", "memory");
}

#define __save_flags(x)                                         \
__asm__ __volatile__ (                                          \
        ".set\tnoreorder\n\t"                                   \
        "mfc0\t%0,$12\n\t"                                      \
        ".set\treorder"                                         \
        : "=r" (x))

#define __save_and_cli(x)                                       \
__asm__ __volatile__ (                                          \
        ".set\tnoreorder\n\t"                                   \
        ".set\tnoat\n\t"                                        \
        "mfc0\t%0,$12\n\t"                                      \
        "ori\t$1,%0,1\n\t"                                      \
        "xori\t$1,1\n\t"                                        \
        "mtc0\t$1,$12\n\t"                                      \
        "nop\n\t"                                               \
        "nop\n\t"                                               \
        "nop\n\t"                                               \
        ".set\tat\n\t"                                          \
        ".set\treorder"                                         \
        : "=r" (x)                                              \
        : /* no inputs */                                       \
        : "$1", "memory")

#define __restore_flags(flags)                                  \
do {                                                            \
        unsigned long __tmp1;                                   \
                                                                \
        __asm__ __volatile__ (                                  \
                ".set\tnoreorder\t\t\t# __restore_flags\n\t"    \
                ".set\tnoat\n\t"                                \
                "mfc0\t$1, $12\n\t"                             \
                "andi\t%0, 1\n\t"                               \
                "ori\t$1, 1\n\t"                                \
                "xori\t$1, 1\n\t"                               \
                "or\t%0, $1\n\t"                                \
                "mtc0\t%0, $12\n\t"                             \
                "nop\n\t"                                       \
                "nop\n\t"                                       \
                "nop\n\t"                                       \
                ".set\tat\n\t"                                  \
                ".set\treorder"                                 \
                    : "=r" (__tmp1)                                        \
                    : "0" (flags)                                          \
                    : "$1", "memory");                                     \
} while(0)

#define cli()  __cli ()
#define sti()  __sti ()
#define save_flags(x)  __save_flags (x)
#define restore_flags(x)  __restore_flags (x)
#define save_and_cli(x)  __save_and_cli (x)

#endif /* _SYSTEM_H */
```

: include/timer.h

```
/*
 * Timer Header
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#ifndef _TIMER_H
#define _TIMER_H

#include <regoffset.h>
#include <semaphore.h>
#include <list.h>

/* Amount to increment compare reg each time */
extern unsigned int timer_offset;

/* Timer state */
#define IDLE    0
#define ACTIVE 1
#define DONE   2

/* Timer type */
#define ONCE   0
#define PERIODIC 1

typedef struct {
        t_list_element   timer_elem;
        t_semaphore      semaphore;
        int              state;
        int              type;
        unsigned int     length;
        unsigned int     count;
} t_timer;

#define timer_base(ple) StructBase(ple, t_timer, timer_elem)

void timer_interrupt(void);

void timer_init(void);

void timer_setup(t_timer*);
```

```
int  timer_start (t_timer* pT, unsigned int msec,
                 int  timerType);

int  timer_waitfor (t_timer *);

void timer_cancel(t_timer*);

#endif /* _TIMER_H */
```

: include/yamon.h

```
#ifndef _YAMON_H
#define _YAMON_H

/* Basic types */
typedef unsigned int        t_yamon_uint32;
typedef signed int          t_yamon_int32;
typedef unsigned char       t_yamon_bool;

/* YAMON Environment variable */
typedef struct
{
    char *name;
    char *val;
}
t_yamon_env_var;

#endif /* _YAMON_H */
```

: lib/vsprintf.c

```
/*
 * This  file  contains a nice  vsprintf  function . This code is
 * heavily  based on stuff  by Lars Wirzenius & Linus Torvalds
 * from the linus  kernel
 *
 * This  file  is  subject  to the terms and conditions of the GNU General
 * Public License .  See the  file  "COPYING" in the main directory of
 * this  archive  for  more details .
 */

#include <arch/stdarg.h>
#include <serial.h>

#define ZEROPAD 1           /* pad with zero */
#define SIGN  2             /* unsigned/signed long */
#define PLUS 4              /* show plus */
#define SPACE 8             /* space  if  plus */
#define LEFT 16             /*  left   justified  */
#define SPECIAL 32          /* 0x */
#define LARGE 64            /* use 'ABCDEF' instead of 'abcdef' */

/*
 * NOTE! This ctype does not handle EOF like the standard C
 * library  is  required  to.
 */
```

```
#define _U      0x01    /* upper */
#define _L      0x02    /* lower */
#define _D      0x04    /* digit */
#define _C      0x08    /* cntrl */
#define _P      0x10    /* punct */
#define _S      0x20    /* white space (space/lf/tab) */
#define _X      0x40    /* hex digit */
#define _SP     0x80    /* hard space (0x20) */

unsigned char _ctype[] = {
_C,_C,_C,_C,_C,_C,_C,_C,                           /* 0-7 */
_C,_C|_S,_C|_S,_C|_S,_C|_S,_C|_S,_C,_C,            /* 8-15 */
_C,_C,_C,_C,_C,_C,_C,_C,                           /* 16-23 */
_C,_C,_C,_C,_C,_C,_C,_C,                           /* 24-31 */
_S|_SP,_P,_P,_P,_P,_P,_P,_P,                       /* 32-39 */
_P,_P,_P,_P,_P,_P,_P,_P,                           /* 40-47 */
_D,_D,_D,_D,_D,_D,_D,_D,                           /* 48-55 */
_D,_D,_P,_P,_P,_P,_P,_P,                           /* 56-63 */
_P,_U|_X,_U|_X,_U|_X,_U|_X,_U|_X,_U|_X,_U,         /* 64-71 */
_U,_U,_U,_U,_U,_U,_U,_U,                           /* 72-79 */
_U,_U,_U,_U,_U,_U,_U,_U,                           /* 80-87 */
_U,_U,_U,_P,_P,_P,_P,_P,                           /* 88-95 */
_P,_L|_X,_L|_X,_L|_X,_L|_X,_L|_X,_L|_X,_L,         /* 96-103 */
_L,_L,_L,_L,_L,_L,_L,_L,                           /* 104-111 */
_L,_L,_L,_L,_L,_L,_L,_L,                           /* 112-119 */
_L,_L,_L,_P,_P,_P,_P,_C,                           /* 120-127 */
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                 /* 128-143 */
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                 /* 144-159 */
_S|_SP,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,   /* 160-175 */
_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,_P,       /* 176-191 */
_U,_U,_U,_U,_U,_U,_U,_U,_U,_U,_U,_U,_U,_U,_U,_U,       /* 192-207 */
_U,_U,_U,_U,_U,_U,_U,_P,_U,_U,_U,_U,_U,_U,_U,_L,       /* 208-223 */
_L,_L,_L,_L,_L,_L,_L,_L,_L,_L,_L,_L,_L,_L,_L,_L,       /* 224-239 */
_L,_L,_L,_L,_L,_L,_L,_P,_L,_L,_L,_L,_L,_L,_L,_L};      /* 240-255 */

#define __ismask(x) (_ctype[(int)(unsigned char)(x)])

#define isalnum(c)      ((__ismask(c)&(_U|_L|_D)) != 0)
#define isalpha(c)      ((__ismask(c)&(_U|_L)) != 0)
#define iscntrl(c)      ((__ismask(c)&(_C)) != 0)
#define isdigit(c)      ((__ismask(c)&(_D)) != 0)
#define isgraph(c)      ((__ismask(c)&(_P|_U|_L|_D)) != 0)
#define islower(c)      ((__ismask(c)&(_L)) != 0)
#define isprint(c)      ((__ismask(c)&(_P|_U|_L|_D|_SP)) != 0)
#define ispunct(c)      ((__ismask(c)&(_P)) != 0)
#define isspace(c)      ((__ismask(c)&(_S)) != 0)
#define isupper(c)      ((__ismask(c)&(_U)) != 0)
#define isxdigit(c)     ((__ismask(c)&(_D|_X)) != 0)

#define isascii(c ) ((( unsigned char)(c))<=0x7f)
#define toascii(c) (((unsigned char)(c))&0x7f)

static  inline  unsigned char __tolower(unsigned char c)
{
        if (isupper(c))
                c -= 'A'-'a';
        return c;
}
```

```
static inline unsigned char __toupper(unsigned char c)
{
        if (islower(c))
                c -= 'a'-'A';
        return c;
}

#define tolower(c) __tolower(c)
#define toupper(c) __toupper(c)

/*
 * Hey, we're already 64-bit, no need to play games.
 * Replace this if you are going to port the kernel.
 */
#define do_div(n,base) ({ \
        int __res; \
        __res = ((unsigned long) n) % (unsigned) base; \
        n = ((unsigned long) n) / (unsigned) base; \
        __res; })

static int skip_atoi(const char **s)
{
        int i=0;

        while (isdigit(**s))
                i = i*10 + *((*s)++) - '0';
        return i;
}

void * memset(void * s,int c, unsigned long count)
{
        char *xs = (char *) s;

        while (count--)
                *xs++ = c;

        return s;
}

static int strnlen(const char * s, int count)
{
        const char *sc;

        for (sc = s; count-- && *sc != '\0'; ++sc)
                /* nothing */;
        return sc - s;
}

static char * number(char * str, long long num, int base, int size, int precision, int type)
{
        char c,sign,tmp[66];
        const char *digits="0123456789abcdefghijklmnopqrstuvwxyz";
        int i;

        if (type & LARGE)
                digits = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        if (type & LEFT)
```

```
                type &= ~ZEROPAD;
        if (base < 2 || base > 36)
                return 0;
        c = (type & ZEROPAD) ? '0' : ' ';
        sign = 0;
        if (type & SIGN) {
                if (num < 0) {
                        sign = '-';
                        num = -num;
                        size--;
                } else if (type & PLUS) {
                        sign = '+';
                        size--;
                } else if (type & SPACE) {
                        sign = ' ';
                        size--;
                }
        }
        if (type & SPECIAL) {
                if (base == 16)
                        size -= 2;
                else if (base == 8)
                        size--;
        }
        i = 0;
        if (num == 0)
                tmp[i++]='0';
        else while (num != 0)
                tmp[i++] = digits[do_div(num,base)];
        if (i > precision)
                precision = i;
        size -= precision;
        if (!(type&(ZEROPAD+LEFT)))
                while(size-->0)
                        *str++ = ' ';
        if (sign)
                *str++ = sign;
        if (type & SPECIAL) {
                if (base==8)
                        *str++ = '0';
                else if (base==16) {
                        *str++ = '0';
                        *str++ = digits[33];
                }
        }
        if (!(type & LEFT))
                while (size-- > 0)
                        *str++ = c;
        while (i < precision--)
                *str++ = '0';
        while (i-- > 0)
                *str++ = tmp[i];
        while (size-- > 0)
                *str++ = ' ';
        return str;
}

/**
```

```
 * vsprintf - Format a string and place it in a buffer
 * @buf: The buffer to place the result into
 * @fmt: The format string to use
 * @args: Arguments for the format string
 *
 * Call this function if you are already dealing with a va_list .
 * You probably want sprintf instead.
 */
int vsprintf(char *buf, const char *fmt, va_list args)
{
        int len;
        unsigned long long num;
        int i , base;
        char * str;
        const char *s;

        int flags;              /* flags to number() */

        int field_width;        /* width of output field */
        int precision;          /* min. # of digits for integers; max
                                   number of chars for from string */
        int qualifier;          /* 'h', 'l', or 'L' for integer fields */
                                /* 'z' support added 23/7/1999 S.H.   */
                                /* 'z' changed to 'Z' --davidm 1/25/99 */

        for (str=buf ; *fmt ; ++fmt) {
                if (*fmt != '%') {
                        *str++ = *fmt;
                        continue;
                }

                /* process flags */
                flags = 0;
                repeat:
                        ++fmt;          /* this also skips first '%' */
                        switch (*fmt) {
                                case '-': flags |= LEFT; goto repeat;
                                case '+': flags |= PLUS; goto repeat;
                                case ' ': flags |= SPACE; goto repeat;
                                case '#': flags |= SPECIAL; goto repeat;
                                case '0': flags |= ZEROPAD; goto repeat;
                                }

                /* get field width */
                field_width = -1;
                if ( isdigit (*fmt))
                        field_width = skip_atoi(&fmt);
                else if (*fmt == '*') {
                        ++fmt;
                        /* it's the next argument */
                        field_width = va_arg(args, int);
                        if ( field_width < 0) {
                                field_width = -field_width;
                                flags |= LEFT;
                        }
                }

                /* get the precision */
```

```
precision = -1;
if (*fmt == '.') {
        ++fmt;
        if ( isdigit (*fmt))
                precision = skip_atoi(&fmt);
        else if (*fmt == '*') {
                ++fmt;
                /* it's the next argument */
                precision = va_arg(args, int);
        }
        if ( precision < 0)
                precision = 0;
}

/* get the conversion qualifier */
 qualifier = -1;
if (*fmt == 'h' || *fmt == 'l' || * fmt == 'L' || *fmt =='Z') {
         qualifier = *fmt;
        ++fmt;
}

/* default base */
base = 10;

switch (*fmt) {
case 'c':
        if (!( flags & LEFT))
                while (--field_width > 0)
                        *str++ = ' ';
        *str++ = (unsigned char) va_arg(args, int);
        while (--field_width > 0)
                *str++ = ' ';
        continue;

case 's':
        s = va_arg(args, char *);
        if (!s)
                s = "<NULL>";

        len = strnlen(s, precision );

        if (!( flags & LEFT))
                while (len < field_width--)
                        *str++ = ' ';
        for ( i = 0; i < len; ++i)
                *str++ = *s++;
        while (len < field_width--)
                *str++ = ' ';
        continue;

case 'p':
        if ( field_width == -1) {
                field_width = 2*sizeof(void *);
                flags |= ZEROPAD;
        }
        str = number(str,
                (unsigned long) va_arg(args, void *), 16,
                field_width , precision , flags );
        continue;


case 'n':
        if ( qualifier == 'l') {
                long * ip = va_arg(args, long *);
                *ip = (str - buf);
        } else if ( qualifier == 'Z') {
                unsigned long * ip = va_arg(args, unsigned long *);
                *ip = (str - buf);
        } else {
                int * ip = va_arg(args, int *);
                *ip = (str - buf);
        }
        continue;

case '%':
        *str++ = '%';
        continue;

/* integer number formats - set up the flags and "break" */
case 'o':
        base = 8;
        break;

case 'X':
        flags |= LARGE;
case 'x':
        base = 16;
        break;

case 'd':
case 'i':
        flags |= SIGN;
case 'u':
        break;

default:
        *str++ = '%';
        if (*fmt)
                *str++ = *fmt;
        else
                --fmt;
        continue;
}
if ( qualifier == 'L')
        num = va_arg(args, long long);
else if ( qualifier == 'l') {
        num = va_arg(args, unsigned long);
        if ( flags & SIGN)
                num = (signed long) num;
} else if ( qualifier == 'Z') {
        num = va_arg(args, unsigned long);
} else if ( qualifier == 'h') {
        num = (unsigned short) va_arg(args, int);
        if ( flags & SIGN)
                num = (signed short) num;
} else {
```

```
                        num = va_arg(args, unsigned int);
                        if ( flags & SIGN)
                                num = (signed int) num;
                }
                str = number(str, num, base, field_width, precision , flags );
        }
        *str = '\0';
        return str-buf;
}

/**
 * sprintf - Format a string and place it in a buffer
 * @buf: The buffer to place the result into
 * @fmt: The format string to use
 * @args: Arguments for the format string
 */
int sprintf(char * buf, const char *fmt, ...)
{
        va_list args;
        int i;

        va_start(args, fmt);
        i=vsprintf(buf,fmt,args);
        va_end(args);
        return i;
}


/*
 * Send a print message to the console driver
 */
void printf(char *fmt, ...)
{
        char buf[1024];
        va_list args;
        int i;

        /* Format the string */
        va_start(args, fmt);
        i = vsprintf(buf,fmt,args);
        va_end(args);

        serial_print(buf);
}
```

: kernel/cpu.c

```
/*
 * CPU functions
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <addrspace.h>
#include <regoffset.h>
#include <mipsregs.h>
```

```
#include <system.h>
#include <rtc.h>
#include <kernel.h>
#include <cpu.h>
#include <printf.h>
#include <timer.h>

static unsigned char rtc_read(unsigned long addr) {
        volatile unsigned char *rtc_add =
                (unsigned char *) swap8addr(MALTA_GT_PORT_BASE +
                                              MALTA_RTC_ADR_REG);

        volatile unsigned char *rtc_dat =
                (unsigned char *) swap8addr(MALTA_GT_PORT_BASE +
                                              MALTA_RTC_DAT_REG);

        *rtc_add = addr;
        return *rtc_dat;
}

static void rtc_write(unsigned char data, unsigned long addr) {
        volatile unsigned char *rtc_add =
                (unsigned char *) swap8addr(MALTA_GT_PORT_BASE +
                                              MALTA_RTC_ADR_REG);

        volatile unsigned char *rtc_dat =
                (unsigned char *) swap8addr(MALTA_GT_PORT_BASE +
                                              MALTA_RTC_DAT_REG);

        *rtc_add = addr;
        *rtc_dat = data;
}

/*
 * Probe for cpu type
 */
void cpu_probe(void) {
        unsigned long type;

        type = read_32bit_cp0_register(CP0_PRID);
        switch (type & 0xff00) {
        case PRID_IMP_5KC:
                printf("The CPU type is 5KC\n");
                break;
        default:
                panic("PANIC: Unsupported CPU\n");
        }
}

/*
 * Find the cpu speed
 */
void cpu_speed(void) {

        unsigned int cpu_freq, bus_freq;

        /* Set Data mode - binary. */
        rtc_write(rtc_read(RTC_CONTROL) | RTC_DM_BINARY, RTC_CONTROL);
        printf("calculating cpu speed...\n");

        /* Start counter exactly on falling edge of update flag */
        while (rtc_read(RTC_REG_A) & RTC_UIP);
        while (!(rtc_read(RTC_REG_A) & RTC_UIP));

        /* Start r4k counter */
        write_32bit_cp0_register (CP0_COUNT, 0);

        /* Read counter exactly on falling edge of update flag */
        while (rtc_read(RTC_REG_A) & RTC_UIP);
        while (!(rtc_read(RTC_REG_A) & RTC_UIP));

        /* Read the r4k counter and calculate the offset.
         * The value in timer_offset is needed in timer driver init. */
        timer_offset = read_32bit_cp0_register(CP0_COUNT);

        bus_freq = timer_offset + 5000; /* Round off */
        bus_freq -= bus_freq%10000;

        /* CPU freq = 2 * bus(timer) freq */
        cpu_freq = bus_freq + bus_freq;

        printf("CPU/Bus frequency %d.%02d/%d.%02d MHz\n",
                cpu_freq/1000000, (cpu_freq%1000000)*100/1000000,
                bus_freq/1000000, (bus_freq%1000000)*100/1000000);

        /* The correct compare value should be set in the timer driver */
        write_32bit_cp0_register (CP0_COMPARE, 0);
}

void cpu_init(void)
{
        unsigned int bits;

        /* Init the CPU Config register
         * Disable cache K0 (this has to be done in start.S)
         *
         */

        cpu_probe();
        cpu_speed();

        bits = ST0_KX | ST0_SX | ST0_UX;
        set_cp0_status(bits, bits);
}
```

: kernel/interrupt.c

```
/*
 * Interrupt handling code
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <stddef.h>
```

```
#include <kernel.h>
#include <addrspace.h>
#include <regoffset.h>
#include <mipsregs.h>
#include <system.h>
#include <sched.h>
#include <printf.h>
#include <piix4.h>
#include <interrupt.h>

extern void mipsIRQ(void);

/* Counter for the nesting level */
int interrupt_nested;

#define MAX_INT 16

/* Array of interrupt handlers */
static interrupt_handler interrupt_action[MAX_INT] = {
        NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, NULL
};


/* 82C59 interrupt controllers specific functions */

/* Make come addresses for fast access */
static volatile unsigned char *ctrl11 = (unsigned char *)
        swap8addr(MALTA_GT_PORT_BASE + PIIX4_ICTLR1_OCW1);

static volatile unsigned char *ctrl21 = (unsigned char *)
        swap8addr(MALTA_GT_PORT_BASE + PIIX4_ICTLR2_OCW1);

static volatile unsigned char *ctrl12 = (unsigned char *)
        swap8addr(MALTA_GT_PORT_BASE + PIIX4_ICTLR1_OCW2);

static volatile unsigned char *ctrl22 = (unsigned char *)
        swap8addr(MALTA_GT_PORT_BASE + PIIX4_ICTLR2_OCW2);

static volatile unsigned char *ctrl13 = (unsigned char *)
        swap8addr(MALTA_GT_PORT_BASE + PIIX4_ICTLR1_OCW3);

static volatile unsigned char *ctrl23 = (unsigned char *)
        swap8addr(MALTA_GT_PORT_BASE + PIIX4_ICTLR2_OCW3);

/*
 * This contains the interrupt mask for both 82C59 interrupt controllers.
 */
static unsigned int cached_int_mask = 0xffff;

/* Disable irq in the 82C59 interrupt controller */
static void disable_irq(unsigned int irq_nr)
{
        unsigned long flags;

        if(irq_nr >= MAX_INT) {
                printf("whee, invalid irq_nr %d\n", irq_nr);
                panic("IRQ, you lose..." );
        }

        save_and_cli ( flags );
        cached_int_mask |= (1 << irq_nr);
        if ( irq_nr & 8) {
                *ctrl21 = (cached_int_mask >> 8) & 0xff;
        } else {
                *ctrl11 = cached_int_mask & 0xff;
        }
        restore_flags ( flags );
}

/* Enable irq in the 82C59 interrupt controller */
static void enable_irq(unsigned int irq_nr)
{
        unsigned long flags;

        if(irq_nr >= MAX_INT) {
                printf("whee, invalid irq_nr %d\n", irq_nr);
                panic("IRQ, you lose..." );
        }

        save_and_cli ( flags );
        cached_int_mask &= ~(1 << irq_nr);
        if ( irq_nr & 8) {
                *ctrl21 = (cached_int_mask >> 8) & 0xff;

                /* Enable irq 2 (cascade interrupt ). */
                cached_int_mask &= ~(1 << 2);
                *ctrl11 = cached_int_mask & 0xff;
        } else {
                *ctrl11 = cached_int_mask & 0xff;
        }
        restore_flags ( flags );
}

/* Acknowledge interrupt */
static void ack_int(int irq)
{
        if (irq & 8) {
                /* Specific EOI to cascade */
                *ctrl12 = PIIX4_OCW2_SEL | PIIX4_OCW2_NSEOI |
                          PIIX4_OCW2_ILS_2;

                /* Non specific EOI to cascade */
                *ctrl22 = PIIX4_OCW2_SEL | PIIX4_OCW2_NSEOI;
        } else {
                /* Non specific EOI to cascade */
                *ctrl12 = PIIX4_OCW2_SEL | PIIX4_OCW2_NSEOI;
        }
}

/* Get the interrupt */
static int get_irq(int *irq)
{
        /*
```

```
         * Determine highest priority pending interrupt by performing
         * a PCI Interrupt Acknowledge cycle.
         */

        /* Interrupt acknowledge offset */
        #define GT_PCI0_IACK_OFS          0xc34
        #define MIPS_GT_BASE (KSEG1ADDR(0x1be00000))

        /* Interrupt acknowledge register is read only.
         * Read acces to this register forces an interrupt
         * acknowledge cycle on PCI0 */
        volatile unsigned int *gt_irq =
                (void *) MIPS_GT_BASE + GT_PCI0_IACK_OFS;

        /* Store irq number in *irq */
        *irq = *gt_irq;
        *irq &= 0xFF;

        /*
         * IRQ7 is used to detect spurious interrupts.
         * The interrupt acknowledge cycle returns IRQ7, if no
         * interrupts is requested.
         * We can differentiate between this situation and a
         * "Normal" IRQ7 by reading the ISR.
         */
        if (*irq == 7)
        {
                *ctrl13 = PIIX4_OCW3_SEL | PIIX4_OCW3_ISR;
                if (!(*ctrl13 & (1 << 7)))
                        return -1;    /* Spurious interrupt. */
        }

        return 0;
}

/********************************************************/

/*
 * As a side effect of the way this is implemented we're limited
 * to interrupt handlers in the address range from
 * KSEG0 <= x < KSEG0 + 256mb.
 */
void interrupt_setvector(void *addr) {
        /* Normaly we would have to flush the cache to ensure
         * that the interrupt handler actually get registered
         * right away, but just disable cache to avoid strange
         * problems.
         */

        unsigned long handler = (unsigned long) addr;
        *(volatile unsigned int *)(KSEG0+0x200) =
                0x08000000 |(0x03ffffff & (handler >> 2));
}

/* Handle the combined interrupt */
void interrupt_hw(reg_offset *regs) {

        interrupt_handler action;
        int irq=0;

        if ( get_irq(&irq))
                return; /* interrupt has already been cleared */

        disable_irq(irq);
        ack_int(irq);
        action = interrupt_action[irq];

        /* Handler registered ? */
        if (action != NULL)
                /* call handler */
                action();
        else
                printf("IRQ %d but no handler", irq);

        enable_irq(irq);
}

/* Register an combined interrupt */
void interrupt_register(int irq, interrupt_handler handler) {

        /* No irq sharing */
        if ( interrupt_action[irq] != NULL) {
                printf("IRQ %d is already in use", irq);
                panic("PANIC: No irq sharing");
        }

        interrupt_action[irq] = handler;
}

/* Initialize the interrupt handler */
void interrupt_init() {

        /* reset nesting level */
        interrupt_nested = 0;

        /* Set the interrupt vector */
        interrupt_setvector(mipsIRQ);

        /* Until now the interrupt has been disabled, so now
         * we start everything by enabling the interrupts and
         * setting the interrupt mask.
         */
        set_cp0_status(ST0_IM, IE_IRQ0 | IE_IRQ1 | IE_IRQ2 |
                        IE_IRQ3 | IE_IRQ4 | IE_IRQ5);
        sti();

}
```

: kernel/kernel.c

```
/*
 * Highlevel kernel entry
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <yamon.h>
#include <cpu.h>
#include <serial.h>
#include <timer.h>
#include <process.h>
#include <interrupt.h>
#include <printf.h>
#include <setjmp.h>
#include <sched.h>

#include <mipsregs.h>
#include <system.h>

/* Declare processes here */
extern void process_1(void);
extern void process_2(void);

jmp_buf buf;

static void foo(void) {
        printf("In foo\n");
        longjmp(buf, 1);
        printf("This should never be seen\n");
}

int
entry(
        unsigned int argc,          /* Number of tokens in argv array    */
        char **argv,                /* Array of tokens ( first  is "go") */
        t_yamon_env_var *env,       /* Array of env. variables           */
        unsigned int memsize)       /* Size  of  memory (byte count)     */
{
        int i;
        //t_timer t;

        /* Init  serial  driver */
         serial_init ();

        /* Init  the CPU */
        cpu_init ();

        /* Init  timer driver */
        timer_init ();

        /* Create some processes */
        process_init ();
//      process_create(process_2 , 5, 5000);
        process_create(process_1 , 4, 5000);

        /* Pause to show init  status */
        for (i=0; i<2000000; i++)
                /* do nothing */ ;

        /* Test setjmp */
        printf("here 1\n");
        setjmp(buf);
        printf("here 2\n");

        if(setjmp(buf))
                printf("Back in main\n");
        else {
                printf("First time\n");
                foo();
        }

        printf("_kernel call:\n");
        process_list_print(&process_list);

        /* Init interrupts, this starts the kernel */
        interrupt_init();

        /* Schedule the highest priority process */
        schedule();

        /* Test timer */
        //timer_setup(&t);
        //timer_start(&t, 3000, PERIODIC);

        /* This is idle loop */
        while(1) {
                //printf("Idle process");
                printf("idle_%010u,", read_32bit_cp0_register(CP0_COUNT));
                for (i=0; i<1000000; i++)
                        /* do nothing */ ;
                //set_cp0_status(ST0_IM, IE_IRQ0 | IE_IRQ1 | IE_IRQ2 |
                //        IE_IRQ3 | IE_IRQ4 | IE_IRQ5);
                //sti();
        }


        return 0;
}
```

: kernel/lcd.c

```
/*
 * LCD Display driver
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <addrspace.h>

/*
 * Display register base.
 */
#define LCD_DISPLAY_WORD_BASE (KSEG1ADDR(0x1f000410))
#define LCD_DISPLAY_POS_BASE (KSEG1ADDR(0x1f000418))
#define MALTA_PORT_BASE   (KSEG1ADDR(0x18000000))

void lcd_int(unsigned int num)
{
        volatile unsigned int *display = (void *) LCD_DISPLAY_WORD_BASE;

        *display = num;
}

void lcd_message(const char* str)
{
        volatile unsigned int* display = (void*) LCD_DISPLAY_POS_BASE;
        int i;

        for (i = 0; i <= 14; i=i+2) {
                if (*str)
                        display[i] = *str++;
                else
                        display[i] = ' ';
        }
}
```

: kernel/list.c

```
/*
 * Double non-circular linked list functions
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <list.h>

/*
 * Initialize list
 */
void list_init(t_list_head * pHead)
{
        pHead->number = 0;
        pHead->pFirst = NULL;
        pHead->pLast = NULL;
}

/*
 * Put element at end of list
 */
void list_put(t_list_head *     pHead,
              t_list_element * pElement)
{
        pElement->pNext = NULL;
        if (pHead->pLast == NULL) {
                pHead->pFirst = pElement;
                pElement->pPrev = NULL;
        } else {
                pElement->pPrev = pHead->pLast;
                pHead->pLast->pNext = pElement;
        }
        pHead->pLast = pElement;
        pHead->number++;
}
```

```
/*
 * Put Element2 after Element1
 */
void list_put_after ( t_list_head *     pHead,
                      t_list_element * pElement1,
                      t_list_element * pElement2)
{
        if (pElement1 == NULL || pElement1 == pHead->pLast) {
                list_put (pHead, pElement2);
                return;
        }
        pElement2->pNext = pElement1->pNext;
        pElement1->pNext = pElement2;

        pElement2->pPrev = pElement1;
        pElement2->pNext->pPrev = pElement2;
        pHead->number++;
}

/*
 * Put Element2 before Element1
 */
void list_put_before ( t_list_head *     pHead,
                       t_list_element * pElement1,
                       t_list_element * pElement2)
{
        if (pElement1 == pHead->pFirst) {
                pElement2->pPrev = NULL;
                pElement2->pNext = pElement1;
                pElement1->pPrev = pElement2;
                pHead->pFirst = pElement2;
        } else {
                pElement1->pPrev->pNext = pElement2;
                pElement2->pNext = pElement1;

                pElement2->pPrev = pElement1->pPrev;
                pElement1->pPrev = pElement2;
        }
        pHead->number++;
}

/*
 * Get first element from list
 */
t_list_element * list_get ( t_list_head * pHead)
{
        t_list_element * pRet;

        if ((pRet = pHead->pFirst) == NULL) {
                return NULL;
        }
        if (pRet == pHead->pLast) {
                pHead->pFirst = pHead->pLast = NULL;
        } else {
                pHead->pFirst = pRet->pNext;
                pHead->pFirst->pPrev = NULL;
        }
        pRet->pNext = NULL;
        pRet−>pPrev = NULL;
        pHead−>number−−;
        return pRet;
}

/*
 * Remove specific Element
 */
void list_remove( t_list_head *     pHead,
                  t_list_element * pElement)
{
        if (pElement−>pPrev == NULL) {
                if (pHead−>pFirst != pElement)
                        printf("Inconsistent list!\n");
                pHead−>pFirst = pElement−>pNext;
        } else
                pElement−>pPrev−>pNext = pElement−>pNext;
        if (pElement−>pNext == NULL) {
                if (pHead−>pLast != pElement)
                        printf("Inconsistent list!\n");
                pHead−>pLast = pElement−>pPrev;
        } else
                pElement−>pNext−>pPrev = pElement−>pPrev;

        pElement−>pNext = NULL;
        pElement−>pPrev = NULL;

        pHead−>number−−;
}

int list_length ( t_list_head * pHead) {
        return pHead−>number;
}
```

: kernel/mipsirq.S

```
/*
 * Interrupt exception dispatch code.
 *
 * The idea of how to handle interrupts comes from the Linux
 * exception dispatch code.
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <asm.h>
#include <regdef.h>
#include <regoffset.h>
#include <mipsregs.h>
#include <stackframe.h>

        .text
LEAF(mipsIRQ)
        .set    noreorder
        CLI
        SAVE_ALL
        /* process_current->stack_pointer = sp, stack_pointer
         * has offset 32.
         */
        ld      a0, process_current
        sd      sp, 32(a0)

        /* interrupt_nested++ */
        lw      a0,interrupt_nested
        nop
        addu    a0,a0,1
        sw      a0,interrupt_nested

        /* get irq mask */
        PROM_PRINT("_INT\n")
        mfc0    s0, CP0_CAUSE
        nop

        /* First we check for r4k timer interrupt */
        andi    a0, s0, CAUSEF_IP7
        beq     a0, zero, not_timer

        /* delay slot , check hw0 interrupt */
        andi    a0, s0, CAUSEF_IP2

        /* We got a timer interrupt. */
        move    a0, sp
        jal     timer_interrupt
        nop                             # delay slot

        j       return
        nop                             # delay slot

not_timer:
        beq     a0, zero, not_hardware
        nop                             # delay slot

        /* We got a combined hardware level zero interrupt. */

        /* Update the interrupt mask for nested interrupts */
        /* TO BE DONE */

        /* Now just enable interrupts again */
        //STI
        move    a0, sp
        jal     interrupt_hw
        nop                             # delay slot

        j       return
        nop                             # delay slot

not_hardware:
        /*
         * Here by mistake? This is possible , what can happen is that by the
         * time we take the exception the IRQ pin goes low, so just leave if
         * this is the case. Another option is that the interrupt masks are
         * fucked up.
         */
        PROM_PRINT("Missed an interrupt\n")
        j       return
        nop                             # delay slot
END(mipsIRQ)


LEAF(return)
        .set    noat
        .set    reorder
        /* Disable interrupts */
        CLI

        /* interrupt_nested--; */
        lw      a0,interrupt_nested
        nop
        addu    a0,a0,-1
        sw      a0,interrupt_nested

        /* if (interrupt_nested == 0 && sched_now) {
         *      sched_now = 0;
         *      schedule_frominterrupt();
         * }
         */
        lw      a0, interrupt_nested
        bne     a0, zero, dont_schedule
        lw      a0, sched_now
        beq     a0, zero, dont_schedule
        nop

        /* sched_now = 0; */
        sw      zero, sched_now

        /* Update current process */
        jal     schedule_frominterrupt

dont_schedule:

        /* sp = process_current->stack_pointer, stack_pointer
         * has offset 32.
         */
        ld      a0, process_current
        ld      sp, 32(a0) /* <---- This makes the stuff halt */

        /* At this point we have handled the exception, maybe even
         * context switched. So we now load the registers from
         * whatever stack * we have and return
         */
        .set    noreorder
        RESTORE_ALL
        eret
        .set    at
END(return)
```

: kernel/panic.c

```
/*
 * The panic function
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <serial.h>

void panic(char *buf)
{
        char *p;

        for (p = buf; *p; p++) {
                if(*p == '\n') serial_putchar('\r');
                serial_putchar(*p);
        }

        while(1) /* now panic */ ;

}
```

: kernel/process.c

```
/*
 * Process management
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <regoffset.h>
#include <mipsregs.h>
#include <process.h>
#include <kernel.h>
#include <system.h>
#include <printf.h>

/* Pointer to current running process */
t_process*  process_current;
t_process*  process_old;

/* Sorted of ready processes */
 t_list_head   process_list ;

/* Static allocation of the maximum allowed number of processes */
static t_process  process_allocated[MAX_PROCESSES];

/* Incremental process ID */
static int nextId;

/* Allocate a stack, stack_size should be in dwords */
static unsigned long process_stackalloc(int stack_size) {
        /* These are the hardcoded values from the linker script */
        extern unsigned long _sp_end;
        extern unsigned long _sp;

        /* Calculate the first free stack top */
        static long stack_top = (unsigned long) &_sp − 8*KERNEL_STACK;
        if ((stack_top -= 8*stack_size) < (unsigned long) &_sp_end)
                panic("PANIC: Not enough stack\n");

        return stack_top + 8*stack_size;
}

/*
 * Prints and verifies a process list
 */
void process_list_print ( t_list_head * pHead) {

        t_process* p;
        t_list_element * pNext;
        t_list_element * pPrev = NULL;
        int length = 0;

        /* test the empty list */
        if (pHead->number == 0) {
                if (pHead->pFirst != NULL)
                        printf("First element is not empty and length is 0\n");
                if (pHead->pLast != NULL)
                        printf("Last element is not empty and length is 0\n");
                return;
        }

        /* test first element */
        if (process_base(pHead->pFirst)->process_elem.pPrev != NULL) {
                printf("First element previous is not NULL\n");
        }

        /* test last element */
        if(process_base(pHead->pLast)->process_elem.pNext != NULL) {
                printf("Last element next is not NULL\n");
                return;
        }

        /* print contents, damn this loop is ugly */
        for (pNext = pHead->pFirst; pNext != NULL;
             pNext = process_base(pNext)->process_elem.pNext) {

                p = process_base(pNext);
                printf("ID %d, Priority %d, State ", p->id, p->priority);
                switch (p->state) {
                case READY: printf("READY\n"); break;
                case RUNNING: printf("RUNNING\n"); break;
                case WAITING: printf("WAITING\n"); break;
                }

                /* Test prevoius pointer */
                if (p->process_elem.pPrev != pPrev)
                        printf("Wrong previous pointer\n");

                pPrev = pNext;
                length++;
        }

        /* test length */
        if (length != pHead->number)
                printf("Wrong list length. Counted %d, Expected %d\n",
                        length, pHead->number);

        /* test last actually is last */
        if (pPrev != pHead->pLast)
                printf("Wrong last element in list\n");
}

/*
 * Insert a process into an ordered process list. Highest priority in front.
 */
void process_insert( t_list_head * pHead, t_list_element* pElement)
{
        /* Get priority */
        int priority = process_base(pElement)->priority;

        /* Get first element of process */
        t_list_element * pLower = pHead->pFirst;

        /* Handle the case of an empty list */
        if (pHead->number == 0) {
                list_put (pHead, pElement);
                return;
        }

        /* Walk down the ordered list until a lower priority process is found */
        while (pLower->pNext != NULL) {

                /* If the priority is lower, the we found the right place
                 * for our new process. Congrats.
                 */
                if (process_base(pLower)->priority < priority)
                        break;

                /* Next element */
                pLower = pLower->pNext;
        }

        /* Insert the new process into the list here */
        list_put_before (pHead, pLower, pElement);
}

/*
 * Reorders a process in a process list
 */
void process_reorder( t_list_head * pHead, t_list_element* pElement) {

        /* done by removing and inserting */
        list_remove(pHead, pElement);
        process_insert (pHead, pElement);
}

/*
 * Create a new task and initialize its state.
 */
void process_create(void (*function)(void), int priority , int stack_size )
{
        reg_offset  *stack_regs;

        /* Get a free process */
        t_process* P = &process_allocated[nextId];

        /*  Initialize  the task-specific  data */
        P->id            = nextId++;
        P->state         = READY;
        P->priority      = priority;
        P->orig_priority = priority;

        /* Allocate a stack */
        stack_regs = ( reg_offset  *)  process_stackalloc ( stack_size );

        /* Build a state  corresponding to an interrupted  or
         * or  stack  switched process */

        /* R_* macros are bytes, but stack ops are  word size,
         * so  divide  by  8.   (ok,  a  hack..) */

        /* Reserve stack space. Area is  allready  zero'ed
         * from  initialization  ,  so  that  is  not  nessecary
         */

        if  (sizeof( reg_offset ) != 304)
                panic(" reg_offset  problem");

        stack_regs  -= sizeof(reg_offset );

        /* Return address for both ra and epc register*/
        stack_regs->ra = (unsigned long) function;
        stack_regs->cp0_epc = (unsigned long) function;

        /* These two also needs special  attention */
        stack_regs->sp = (unsigned long) stack_regs;
        stack_regs->fp =
                (unsigned long) stack_regs + sizeof(reg_offset);

        /* Save status  register */
        save_flags (stack_regs->cp0_status);

        /* Update stack_regs->cp0_status .. enable interrupts */
        stack_regs->cp0_status |= ST0_IE |
                IE_IRQ0 | IE_IRQ1 | IE_IRQ2 | IE_IRQ3 | IE_IRQ4 | IE_IRQ5;
#if 0
        stack -= R_SIZE/8;

        /* Return address for both ra and epc register*/
        stack[R_RA/8] = (unsigned long) function;
        stack[R_EPC/8] = (unsigned long) function;

        /* These two also needs special  attention */
        stack[R_SP/8] = (unsigned long) stack;          /* WRONG!!! */
        stack[R_S8/8] = (unsigned long) stack + R_SIZE; /* WRONG!!! */

        /* Save status  register */
        save_flags (stack[R_STATUS/8]);
        /* Update stack[R_STATUS] .. enable interrupts */
#endif

        /* Update the process structure */
        P->stack_pointer = (unsigned long *) stack_regs;

        printf("Process id: %u, stack: %x\n", P->id, P->stack_pointer);

        /* Insert the process into the ready list */
        process_insert(&process_list, &P->process_elem);
}

/*
 * Create a new linked list of tasks.
 */
void process_init(void) {

        t_process* P;

        nextId = 0;
        list_init (&process_list);

        /* Create idle process. The idle process uses the kernel stack,
         * well actually the idle is the kernel.
         */
        P           = &process_allocated[nextId];
        P->id       = nextId++;
        P->state    = READY;
        P->priority = 0;

        /* Insert the idle process */
        process_insert(&process_list, &P->process_elem);

        /* The current process is the idle process */
        process_current = P;
}
```

: kernel/sched.c

```
/*
 * Scheduler code
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <stddef.h>
#include <process.h>
#include <interrupt.h>

int sched_now = 1;

t_process*  process_old;
t_process*  process_new;
t_process*  process_highest;

/* A valid declaration ?? */
```

```
extern void stack_switch(void);

/*
 * Select a new process to be run
 */
void schedule(void)
{
        /* Dont schedule if we are nesting, but we raise a flag
         * to indicate that schedule was intended
         */
        if (interrupt_nested != 0) {
                sched_now = 1;
                return;
        }

        /* Get the process with highest priority. This one
         * is alway the first in the list.
         */
        process_highest = process_base(process_list.pFirst);

        printf("_schedule: cur_id=%u, high_id=%u\n",
               process_current->id,
               process_highest->id);

        /* If there is a higher-priority ready task, switch to it */
        if (process_current->id != process_highest->id)
        {
                process_old         = process_current;
                process_new         = process_highest;

                process_new->state = RUNNING;
                process_current     = process_new;

                /* Mark old process ready, if it was running */
                if (process_old->state == RUNNING)
                        process_old->state = READY;

                printf("_schedule: old_sp=%x, cur_sp=%x\n",
                       process_old->stack_pointer,
                       process_current->stack_pointer);
                /* Switch the stacks */
                stack_switch();
                /* New process will be running here */
                printf("_schedule: cur_id=%u, high_id=%u\n",
                       process_current->id,
                       process_highest->id);
                printf("_schedule: old_sp=%x, cur_sp=%x\n",
                       process_old->stack_pointer,
                       process_current->stack_pointer);
        }
}

/*
 * This is used when using calling scheduling from interrupt
 */
void schedule_frominterrupt(void)
{
        //t_process*  process_old;
        //t_process*  process_new;
        //t_process*  process_highest;

        /* Get the process with highest priority . This one
         * is alway the first in the list .
         */
        process_highest = process_base( process_list .pFirst );

        /* If there is a higher-priority ready task, switch to it */
        if (process_current->id != process_highest->id)
        {
                process_old        = process_current;
                process_new        = process_highest;

                process_new->state = RUNNING;
                process_current    = process_new;

                process_old->state = READY;
        }
}
```

: kernel/semaphore.c

```
/*
 * Semaphore implementation
 *
 * This semaphore is implemented as a binary semaphore with a
 * queue sorted by priority
 *
 * The idea for this semaphore implementation comes from ADEOS.
 * ADEOS do not have PIP protocol.
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <system.h>
#include <process.h>
#include <semaphore.h>
#include <kernel.h>
#include <sched.h>

void semaphore_setup(t_semaphore* pS)
{
        long flags;

        save_and_cli ( flags );

        pS->state = FREE;
         list_init (&pS->list_waiting);

         restore_flags ( flags );
}

/* Wait for a semaphore */
void semaphore_wait(t_semaphore* pS)
{
        long flags;

        t_process*   process_calling ;
        t_process*  process_semlocker;

        save_and_cli ( flags );

        printf(" _sem_wait_");

        if (pS->state == FREE) {
                printf("free\n");
                /* The semaphore is available, take it */
                pS->state = LOCKED;

                /* Update who owns the semaphore */
                pS->owner = process_current;
        } else {
                printf("taken\n");
                /* The semaphore is taken, test to see if the process
                 * using the semaphore has lower priority
                 */
                process_semlocker = pS->owner;
                if (process_semlocker->priority < process_current->priority) {

                        if (process_semlocker->state != READY) {
                                panic("PANIC: Nested semaphores are not allowed\n");
                        }

                        /* Bump up the process priority of the semaphore owner */
                        process_semlocker->priority = process_current->priority;

                        /* Reorder the process list */
                        process_reorder(& process_list ,
                                        &process_semlocker->process_elem);

                }
                /* Add the calling task to the waiting list */
                process_calling          = process_current;
                process_calling ->state = WAITING;
                list_remove(& process_list , & process_calling ->process_elem);
                process_insert (&pS->list_waiting, &process_calling->process_elem);
                printf("_sem call:\n");
                process_list_print (& process_list );

                /* Now lets get a new process */
                schedule();

                /* When the semaphore is released, the
                 * caller begins executing here.
                 */
        }

        restore_flags ( flags );
}

/* Signal for a semaphore */
void semaphore_signal(t_semaphore* pS)
{
        long flags;

        t_process*  process_waiting;

        save_and_cli(flags);

        printf("_sem_signal");

        if (pS->state == LOCKED) {

                /* Get first waiting process */
                process_waiting = process_base(pS->list_waiting.pFirst);

                if (process_waiting != NULL) {

                        /* Wake the first process on the waiting list
                         * and insert into our ready list
                         */
                        list_remove(&pS->list_waiting,
                                    &process_waiting->process_elem);
                        process_waiting->state = READY;
                        process_insert(&process_list,
                                       &process_waiting->process_elem);

                        printf(": wakeup id %u\n",
                               &process_waiting->process_elem);

                        /* Bump down our own priority and reorder if necessarily */
                        if(process_current->priority != process_current->orig_priority)
                        {
                                process_current->priority =
                                        process_current->orig_priority;

                                process_reorder(&process_list,
                                                &process_current->process_elem);
                        }

                        /* Now lets call schedule, releasing a semaphore
                         * might
                         */
                        schedule();

                        /* When the semaphore is released, the
                         * caller begins executing here.
                         */

                }
                else {
                        pS->state = FREE;
                }
        }

        restore_flags(flags);
}
```

: kernel/serial.c

```
/*
```

```
 * Putting things on the screen/serial line.
 * No setup of UART - just assume YAMON left in sane state.
 * Bit of a hack but it works.
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <arch/stdarg.h>
#include <addrspace.h>
#include <serial.h>
#include <system.h>

int serial_putchar(char c) {

        volatile unsigned char *uart_sr = (unsigned char *)
                swap8addr(MALTA_GT_PORT_BASE + TTYS1 + UART_LSR);
        volatile unsigned char *uart_data = (unsigned char *)
                swap8addr(MALTA_GT_PORT_BASE + TTYS1 + UART_TX);

        /* Wait for transmit-hold-register empty */
        while ((*uart_sr & UART_LSR_THRE) == 0)
                /* nothing */;

        /* Now write the data directly */
        *uart_data = c;

        return 1;
}

/*
 * Print a string to the serial console
 */
void serial_print(char *buf)
{
        char *p;
        long flags;

        save_and_cli(flags);

        for (p = buf; *p; p++) {
                if(*p == '\n') serial_putchar('\r');
                serial_putchar(*p);
        }

        restore_flags(flags);
}

/* handler for the serial interrupt */
void serial_interrupt(void) {

        /* */
}

void serial_init(void) {

        long flags;
        /* Status for ine Control Register (LCR) */
        volatile unsigned char *uart_lcr = (unsigned char *)
                swap8addr(MALTA_GT_PORT_BASE + TTYS1 + UART_LCR);

        /* Divisor latch */
        volatile unsigned char *uart_dll = (unsigned char *)
                swap8addr(MALTA_GT_PORT_BASE + TTYS1 + UART_DLL);
        volatile unsigned char *uart_dlm = (unsigned char *)
                swap8addr(MALTA_GT_PORT_BASE + TTYS1 + UART_DLM);

        /* Modem control register */
        volatile unsigned char *uart_mcr = (unsigned char *)
                swap8addr(MALTA_GT_PORT_BASE + TTYS1 + UART_MCR);

        /* Clean interrupts while configuring serial port
         * They should actually be off at this point, so
         * this is just paranoid.
         */
        save_and_cli(flags);

        /* Set 1 stop bit, no parity, 8 data bits, no break
         * an raise Divisor latch access bit.
         */
        *uart_lcr = UART_LCR_WLEN8 | UART_LCR_DLAB;

        /* Set 19200 baud */
        *uart_dll = 0x06;
        *uart_dlm = 0x00;

        /* Lower Divisor latch access bit */
        *uart_lcr = *uart_lcr & ~UART_LCR_DLAB;

        /* Initialize interrupts */
        // interrupt_register (irq, serial_interrupt);

        /* Restore interrupts */
        restore_flags(flags);

        /* VT220 term clear/home - escape sequence is ESC[2J ESC[0;0H */
        serial_print("\033\1332J\033\1330;0H");
}
```

: kernel/setjmp.S

```
/*
 * Implementation of setjmp and longjmp for MIPS64
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <asm.h>
#include <regdef.h>

/* int setjmp (jmp_buf); */
LEAF(setjmp)
```

```
        sd      s0,   0(a0)
        sd      s1,   8(a0)
        sd      s2,  16(a0)
        sd      s3,  24(a0)
        sd      s4,  32(a0)
        sd      s5,  40(a0)
        sd      s6,  48(a0)
        sd      s7,  56(a0)
        sd      sp, 160(a0)
        sd      fp, 168(a0)
        sd      ra, 176(a0)

        move    v0,zero
        j       ra
END(setjmp)

/* void longjmp (jmp_buf, int);   */
LEAF(longjmp)
        ld      s0,   0(a0)
        ld      s1,   8(a0)
        ld      s2,  16(a0)
        ld      s3,  24(a0)
        ld      s4,  32(a0)
        ld      s5,  40(a0)
        ld      s6,  48(a0)
        ld      s7,  56(a0)
        ld      sp, 160(a0)
        ld      fp, 168(a0)
        ld      ra, 176(a0)

        bne     a1,zero,1f
         li     a1,1
1:
        move    v0,a1
        j       ra
END(longjmp)
```

: kernel/stack.S

```
/*
 * Switch the stacks
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <asm.h>
#include <regdef.h>
#include <regoffset.h>
#include <mipsregs.h>
#include <stackframe.h>

        .text
LEAF(stack_switch)
        .set    noreorder

        /* Save current state */
        SAVE_ALL

        /* process_old->stack_pointer = sp, stack_pointer
         * has offset 32.
         */

        ld      a0, process_old
        sd      sp, 32(a0)

        /* sp = process_current->stack_pointer, stack_pointer
         * has offset 32.
         */

        .set    reorder
        PROM_PRINT("_SWITCH\n");
        ld      a0, process_current
        ld      sp, 32(a0)
        .set    noreorder

        /* Restore new state and return */
        RESTORE_ALL
        j       ra
        nop                             # delay slot (crucial .. :-)
END(stack_switch)
```

: kernel/start.S

```
/*
 * Starting point for everything
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <asm.h>
#include <regdef.h>
#include <regoffset.h>
#include <mipsregs.h>
#include <stackframe.h>


        .text
LEAF(_start):
        .set    noreorder

        /* Disable interrupts */
        CLI

        /* Disable kseg0 caching as soon as possible */
        mfc0    t0, CP0_CONFIG
        and     t0, ~CONF_CM_CMASK
        or      t0, CONF_CM_UNCACHED
        mtc0    t0, CP0_CONFIG
        nop                     /* Some nops to let the dust settle */
        nop
        nop
        /* Setup stack pointer */
        la      sp, _sp

        /* Clear bss */
        la      t0, _fbss       /* First address */
        la      t1, _end        /* Last  address */
bbs_zero:
        sw      zero, 0(t0)
        bne     t0, t1, bbs_zero
        addiu   t0, 4

        /* Get ready to jump to main */
        move    s0, ra
        la      t0, entry

        /* Jump to main */
        jal     t0
        nop                     /* Delay slot */

        /* We should never en up here */
        PROM_PRINT("Kernel terminated!?!?\n")

        .set reorder
END(_start)
```

: kernel/test1.c

```
/*
 * Test process 1
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License.  See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <printf.h>
#include <timer.h>

void process_1(void) {
        t_timer t;
        //int i;

        timer_setup(&t);
        timer_start(&t, 10000, PERIODIC);

        while (1) {
                printf("Test process 1\n");

                printf("_process_1 call:\n");
                process_list_print(&process_list);
                timer_waitfor(&t);

                //for (i=0; i<1000000; i++)
                //      /* do nothing */ ;
        }
}
```

: kernel/test2.c

```
/*
 * Test process 2
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <lcd.h>
#include <timer.h>

static char display_string[] = "          AN EMBEDDED SYSTEMS KERNEL ";
#define MAX_DISPLAY_COUNT (sizeof(display_string) - 8)

void process_2(void) {

        t_timer t;
        int display_count = 0;

        timer_setup(&t);
        while (1) {

                timer_start(&t, 5000, PERIODIC);
                timer_waitfor(&t);

                /* Display message */
                lcd_message(&display_string[display_count++]);
                if (display_count == MAX_DISPLAY_COUNT)
                        display_count = 0;
        }
}
```

: kernel/timer.c

```
/*
 * Timer driver
 *
 * This file is subject to the terms and conditions of the GNU General
 * Public License. See the file "COPYING" in the main directory of
 * this archive for more details.
 */

#include <regoffset.h>
#include <system.h>
#include <timer.h>
#include <mipsregs.h>

/* Amount to increment compare reg */
unsigned int timer_offset;

/* Self explain able :-) */
unsigned int timer_tick_per_ms;

/* What counter should be at next timer irq */
static unsigned int timer_cur;

/* List of timers */
t_list_head timer_list;
```

```
/*
 * Setup a new timer structure
 */
void timer_setup(t_timer* pT) {

        pT->state = IDLE;
        pT->type = ONCE;
        pT->length = 0;
        pT->count = 0;

        semaphore_setup(&pT->semaphore);
}

/*
 * Start a timer
 */
int timer_start(t_timer* pT,
                unsigned int msec,
                int timerType)
{
        long flags;
        unsigned int tmp;
        t_list_element * pLE;

        /* Do not start it more than one time */
        if (pT->state != IDLE) return -1;

        /* Take the semaphore. It will be released when the timer
         * expires. This should return immediately as the semaphore
         * is supposed to be free.
         */
        semaphore_wait(&pT->semaphore);

        /* Initialize the timer */
        pT->type = timerType;
        pT->length = msec * timer_tick_per_ms;
        pT->count = msec * timer_tick_per_ms;
        pT->state = ACTIVE;

        /* Uninterruptable timer list operations */
        save_and_cli(flags);

        if (timer_list.pFirst == NULL) {
                /* This is the only active timer. Update the
                 * compare register. */
                timer_offset = pT->length;
                timer_cur = (read_32bit_cp0_register(CP0_COUNT) + timer_offset);
                write_32bit_cp0_register(CP0_COMPARE, timer_cur);
        } else {
                /* Other active timers exist */
                tmp = read_32bit_cp0_register(CP0_COUNT);
                if (pT->length < timer_cur - tmp) {
                        /* This timer expires before the current
                         * closest deadline. Not quite straightforward.
                         * Decrement other timers counters until now. */
                        for (pLE = timer_list.pFirst;
                             pLE != NULL; pLE = pLE->pNext)
```

```
                                        timer_base(pLE)->count -=
                                                tmp - (timer_cur - timer_offset);
                                /* Update the compare register with new deadline */
                                timer_offset = pT->length;
                                timer_cur = tmp + timer_offset;
                                write_32bit_cp0_register (CP0_COMPARE, timer_cur);
                        }
        }

        /* Add the timer to the timer list */
        list_put (&timer_list, &pT->timer_elem);

        printf(" _timer_start : exp @ %010u\n", timer_cur);

        restore_flags ( flags );

        return (0);
}

int timer_waitfor(t_timer* pT)
{
        /* Dont wait if its not active */
        if (pT->state != ACTIVE)
                return -1;

        /* Wait for the timer to expire */
        semaphore_wait(&pT->semaphore);

        return (0);
}

void timer_cancel(t_timer* pT)
{
        /* Remove the timer from the timer list */
        if (pT->state == ACTIVE)
                list_remove(&timer_list, &pT->timer_elem);

        /* Reset the timer's state */
        pT->state = IDLE;

        /* Release the semaphore */
        semaphore_signal(&pT->semaphore);
}

void timer_interrupt()
{
        t_timer* pT;
        t_list_element * pLE;
        unsigned int offset_tmp = timer_offset;

        /* Run through the timer list and decrement the
         * counter. Mark all of the expired timers done, remove them and
         * signal there semaphore.
         */
        for (pLE = timer_list.pFirst; pLE != NULL; pLE = pLE->pNext) {
                pT = timer_base(pLE);
                pT->count -= timer_offset;
                if (pT->count == 0) {
                                /* if pT->state != WAITING */
                                /* This counter has expired */
                                semaphore_signal(&pT->semaphore);

                                /* Restart or idle the timer, depending on its type */
                                if (pT->type == PERIODIC) {
                                        pT->state = ACTIVE;
                                        pT->count = pT->length;

                                } else {
                                        list_remove(&timer_list, &pT->timer_elem);
                                        pT->state = IDLE;
                                }
                } else if (pT->count < timer_offset)
                                /* Get the closest deadline */
                                offset_tmp = pT->count;
        }
        timer_offset = offset_tmp;

        /* Acknowledge interrupt */
        printf("_timer_int: %010u_%010u->",
                read_32bit_cp0_register(CP0_COUNT),
                read_32bit_cp0_register(CP0_COMPARE));
        timer_cur = (read_32bit_cp0_register(CP0_COMPARE) + timer_offset);
        write_32bit_cp0_register(CP0_COMPARE, timer_cur);
        printf("%010u\n", timer_cur);
}

void timer_init(void)
{
        /* Init the timer list */
        list_init(&timer_list);

        /* CPU should be initialized before Timer */
        timer_tick_per_ms = timer_offset/1000;
}
```