MinixPPC

A port of the MINIX OS to the PowerPC platform

Creating a programming model for architecture independency

Master Thesis Computer Science

Ingmar A. Alting

September 15, 2006

vrije Universiteit amsterdam

First reader and supervisor:

Andrew S. Tanenbaum
Dept. of Computer Science
Faculty of Sciences
Vrije Universiteit
De Boelelaan 1081A
1081 HV Amsterdam, the Netherlands

e-mail: ast@cs.vu.nl

Second reader:

Herbert Bos
Dept. of Computer Science
Faculty of Sciences
Vrije Universiteit Amsterdam
De Boelelaan 1081A
1081 HV Amsterdam, the Netherlands

e-mail: herbertb@cs.vu.nl

Thesis by:

Ingmar A. Alting
Weth. W. de Boerstraat 18
1788AT Den Helder, the Netherlands

email: iaalting@gmail.com

Abstract

The main goal of this project is to indicate what it means to port an operating system from one architecture to another, and provide a programming paradigm that would make future ports easy and fast.

The “natively” supported architecture of MINIX is the IBM PC compatible, that's built around the Intel architecture. This is a CISC architecture with hardware support for easy stack usage. The choice for the POWER architecture could not have been further away as this is a RISC architecture, and completely” different in many ways.

This thesis will focuses on the model created for creating portable system code. Not to be confused with portable programs using a “standard” API. It will describe the changes made and problems faced porting the MINIX code base. The places where changes are made can be viewed as hotspots. For every new architecture compatibility problems are to be expected there. Some hotspots are used as example and the solution taken for MinixPPC is presented to the reader.

A number of problems were found at the start of the project. The MINIX OS is (still) using the “old” a.out format for it's executables. There is no (recent) public compiler kit that is able to generate a.out format executables for the PowerPC. A utility program had to be written to convert a “minimum section count” Elf32 executable to a.out format. This way the installed compiler kit from host OS can be used, which is a recent version of the GNU/C compiler. Getting the kernel to load and executed by the “boot software” of the PowerPC architecture was the next challenge. With the aid of “Open Source” software a preliminary scheme is created until MinixPPC is able to compile itself. This introduces a new project of getting a recent version of the GCC compiler kit ported to MinixPPC.

The following reasoning defines the choice for the driver model used in the creation of MinixPPC.

1) System dependencies are located inside devices drivers. Defining a method to create and develop device drivers that have isolated system dependencies contribute to the portability of the driver (this could isolate the whole driver).
2) The CPU is (just) a device.
3) Creating a device driver for the CPU isolates the CPU “functionality” from the rest of the system.
4) With every device hidden behind its driver, architecture dependent and independent code are separated.

In principle the CPU could have multiple devices inside, for example MMU and timers. For MinixPPC a logical separation is made by creating two drivers for the CPU, the Memory and System driver.

At the moment MinixPPC is able to boot, access a MINIX v3 file system and run programs, but there are still problems. Not all system calls are debugged and the system must be thoroughly tested. The difficult part of the MinixPPC project is done, but there is more work ahead. There is working code and to some degree only “hard” work is needed, most of the figuring-out, and “trial-and-error” is done.

Preface

I wrote this thesis to close my study "informatica" at the "v*rije* **Universiteit**" in Amsterdam, the Netherlands. It describes the port of the MINIX operating system from the IBM PC compatible designed around the Intel x86 architecture to a IBM PowerPC architecture designed around the POWER CPU. I got the possibility to get in touch with Andrew S. Tanenbaum who has made this project possible by providing me a Apple iBook G4. It is a very nice system build of high quality components. It passed the test of booting more than 30 times a day for about 5 months. As base for the port the MINIX v3.1.3rc1 code base, targeted to the Intel Architecture (IA) is used.

At the time I was living in Amsterdam while my parents home is in Den Helder. As hard as it is, it was clear that I needed to focus all my energy into the project. The decision was made to return to Den Helder. I thank my parents who made that possible. If I didn't had the possibility to get to a nice and quiet place to do my "thing" I am sure nothing solid had come out of my fingers.

I consider this project as a "crown" on my programming capabilities and computer architecture understanding, for now. Its a first step on the "path" I like to take. There where a couple of occasions when I had the feeling "O it works like this, hey it works!" In the end you can always re-read the workings in the documentation and find out it stood there already, but you only "really" understand some problems when you have programmed the implementation.

I am lucky that there are several open source kernels for the Power PC architecture before the project begun,

- The Linux kernel
- Several BSD flavours, NetBSD, FreeBSD and
- XNU, the Darwin kernel used by Mac OS X

Some of the people involved in these projects where still on forms and/or mailing lists to give tips and solutions faced in the early phases of the project.

I learned that when doing a project like this one of the biggest qualities you must possess is "patience." Some bugs or programming errors appear under special circumstances and are not easily found. One day you update code in memory management, and after a day or two the changes become accepted. Then the problem occurs, to trace it back to the changes at the memory management code could take some time. Also driver development can take some time, especially when documentation is short or out of date. The development then becomes more of a "trail-and-error" approach to reverse engineer hardware.

The environment where you are working on the project can help a lot. To have people around who understand programming problems in general can help. The interpretation of a piece of text about the workings of a part of the CPU could be interpreted in many ways but only one is correct. Just to be able to talk about it or about other problems your having could give extra views that would bring you closer to the actual workings or solution. As a bonus talking with other experts eases the mind and will build confidence.

Content

1 **1**
Introduction... 1
1.1 About MINIX... 1
1.2 MINIX v3... 1
2 **4**
Porting an OS... 4
2.1 A complete system... 4
2.2 Creating portable code... 5
2.2.1 Creating portable code from existing files... 10
2.3 Driver programming model... 11
2.3.1 Clock system example... 12
3 **15**
Knowing your architectures... 15
3.1 Start... 15
3.2 IBM PC compatible... 16
3.3 PowerPC... 17
3.3.1 Booting... 17
3.3.2 PowerPC CPU details... 19
3.3.3 Memory management... 21
3.3.3.1 Block Address Translation... 23
3.3.3.2 Page Address Translation... 24
3.3.4 I/O... 27
3.3.5 Interrupts and exceptions... 28
3.3.5.1 Interrupts... 29
3.3.5.2 Exceptions... 30
3.3.5.3 System call... 31
3.3.5.4 Exception and interrupt return... 31
3.4 Software... 31
4 **35**
Development environment... 35
4.1 Why... 35
4.2 Two computer setup... 35
5 **37**
MinixPPC... 37
5.1 Libraries... 38
5.2 Boot monitor... 39
5.2.1 Image format... 40
5.2.2 Loading and executing the kernel... 41
5.3 Kernel organisation... 43
5.3.1 Kernel driver model... 46
5.4 Memory management... 50
5.4.1 Mapping a new process... 53
5.4.2 Remote segments... 54
5.4.3 MMU library functions... 55
5.4.4 I/O... 57
5.5 Exceptions and context switching... 59
5.6 Signals... 63
5.7 New drivers and changes... 66

5.7.1 MacIO........ 66
5.7.2 PCI manager........ 69
5.9 Utilities........ 70
5.9.1 elf2aout........ 70
5.9.2 mkimage........ 73
5.9.3 mkffs........ 74

6 **77**

Compiling MinixPPC........ 77
6.1 How to........ 77
6.2 Link scripts........ 78
6.3 Debugging........ 79

7 **81**

Aftermath........ 81
7.1 Other examples........ 81
7.2 Conclusion........ 81
7.3 Known issues........ 81

Appendixes

A

Code Listings........ A-1
A.1 Exception phase1........ A-1
A.2 Exception phase2........ A-2
A.3 Definition PCI device........ A-3

B

Bibliography........ A-4

C

Enhancements........ A-6

D

Library Notes........ A-7

E

Kernel Files........ A-8
E.1 Missing symbols........ A-8
E.2 PPC architecture files........ A-10

F

File System Prototype File........ A-11

G

Elf32 Section Listings........ A-17
G.1 Elf32 section listing of the elf2out program........ A-17
G.2 With compile-time options........ A-18
G.3 Linker script for the monitor........ A-19
G.4 With compile-time options and linker script........ A-21

H

Kernel Symbol Listing........ A-22

I

Kernel Interfaces........ A-40
I.1 interface.h........ A-40
I.2 System........ A-43

I.3 Memory.......... A-47
I.4 Interrupt.......... A-54
I.5 Clock.......... A-57
J
Open Firmware.......... A-59
J.1 Welcome screen.......... A-59
J.2 Some useful commands.......... A-59

Figures

Figure 2.1: Kernel tree (simplified).. 5
Figure 2.2: Kernel Hardware Interface (KHI) location.. 6
Figure 2.3: Ways to create portable code.. 6
Figure 2.4: Multiple files providing a body for function "f1()."............................ 7
Figure 2.5: Including header files with the same name...................................... 8
Figure 3.1: Open Firmware, user, client and device interface............................ 18
Figure 3.2: PowerPC architecture levels.. 19
Figure 3.3: PowerPC instruction set, OEA { VEA { UISA } }............................. 20
Figure 3.4: Simple MMU overview... 22
Figure 3.5: BAT translation.. 23
Figure 3.6: Page table layout... 24
Figure 3.7: PAT translation, from effective to physical address.......................... 25
Figure 3.8: Conceptual view of segment (identifier) to physical memory............. 26
Figure 3.9: Video memory map for iBook, used as bitmap................................ 27
Figure 3.10: PowerPC, from interrupt source to device driver path..................... 29
Figure 5.1: MinixPPC development source tree... 37
Figure 5.2: PowerPC library tree.. 39
Figure 5.3: Process headers in image of two processes.................................... 40
Figure 5.4: Physical memory layout at end of phase one................................. 41
Figure 5.5: The first MB's of the physical memory.. 41
Figure 5.6: Old (left) and new(right) kernel tree organization........................... 44
Figure 5.7: Disabling external interrupts and allocating a memory map.............. 47
Figure 5.8: Kernel hardware interface (MinixPPC)... 48
Figure 5.9: Virtual to physical memory map... 52
Figure 5.10: Mapping a remote segment, index 0, start 0x3000_0000............... 54
Figure 5.11: Control flow of user process requesting data for a file on disk.......... 58
Figure 5.12: The first bytes in memory, <minix.o> object loaded at 0x0........... 59
Figure 5.13: Context switch P1 to clock task, and (re)starting other process....... 63
Figure 5.14: Stack phases of signalled process.. 65
Figure 5.15: Logical view of MacIO driver.. 67
Figure 5.16: MacIO and TTY driver relation... 68
Figure 5.17: The PCI manager, route to returning information of the PCI device. . 69
Figure 5.18: Usage "elf2out" program... 71
Figure 5.19: Converting ash from elf32 to a.out using a stack size of 100 KB...... 72
Figure 5.20: Snippet of the file system prototype file....................................... 75
Figure 5.21: Current iBook partition table, using the "mac-fdisk" program.......... 76
Figure 7.1: Known issues.. 83

Listings

Listing 2.1: Code fragment illustrating the "#ifdef" directive............................... 9
Listing 2.2: Clock hardware access.. 11
Listing 2.3: Clock system files.. 12
Listing 2.4: Possible interface file for a clock driver, in <interface.h>................. 12
Listing 2.5: Implementation of MDC.. 13
Listing 2.6: Calling a "driver function", to initialize the clock hardware............... 14
Listing 3.1: Differences between assembler code for CISC and RISC.................. 16
Listing 3.2: PowerPC register names.. 21

Listing 3.3: Example assembler I/O for the PowerPC (A) and x86 (B)................ 28
Listing 3.4: C source file <empty_file.c>, only two functions FA( ) and FB( )....... 32
Listing 3.5: The assembler from the source code in listing 3.4.......................... 33
Listing 5.1: Kernel segment register ID calculation.. 42
Listing 5.2: The MinixPPC kernel entry definition and entry point assignment...... 42
Listing 5.3: Entering the kernel just like any other function call........................ 43
Listing 5.4: Setting the i8259 controller interrupt mask register....................... 45
Listing 5.5: Kernel interfaces.. 46
Listing 5.6: The C type definition of the clock interface.................................. 47
Listing 5.7: Kernel hardware interface access points...................................... 47
Listing 5.8: Original lines in part A and replacement lines in part B.................. 49
Listing 5.9: How the architectural layer is initialized....................................... 49
Listing 5.10: Calculation of segment ID with PID... 50
Listing 5.11: Segment allocation interface... 51
Listing 5.12: Data structure used in any memory map, for every section............ 51
Listing 5.13: Page table initialization.. 55
Listing 5.14: Updating a PTE... 56
Listing 5.15: Mapping the kernel text and data segment (done by the monitor)... 56
Listing 5.16: Assembling code to a exception vector (ev)................................ 59
Listing 5.17: Context switch steps.. 60
Listing 5.18: Signal handling phases... 64
Listing 5.19: Signal handling phases for process U and S................................ 64
Listing 5.20: Typical output from the “mkimage” program............................... 74
Listing 6.1: Steps for building the MinixPPC system.. 77
Listing 6.2: Linker scripts used by the MinixPPC system.................................. 78
Listing 6.3: Using the debug and warning macro's... 80

Tables

Table 3.1: IBM PC compatible vs. PowerPC CHRP.. 15
Table 3.2: PowerPC exception vectors.. 30

Commands

Command 5.1: Create a 10 MB file system in the <10MB.img> file.................... 75
Command 5.2: Create a 1 MB or 10 MB file containing only “zero's.”................. 76
Command 5.3: Installing the MINIX file system... 76

Overview

I Document Layout

For people unfamiliar with MINIX this document will first introduce MINIX as OS and its goals. It will give a small introduction to the workings of MINIX v2 and v3. People who are only familiar with MINIX v2 are encouraged to read at least this introduction; it's better to read [7] which describes the transformation of MINIX v2 to v3. These are very brief introductions and don't describe how operating systems works in general.
The guidelines for porting an operating system and some pitfalls will be discussed next. It will go deeper into the problems the author faced when making the port to the POWER architecture and the methods devised and used. Both hardware architectures are discussed in the third chapter. It will point out where the IBM PC compatible and POWER architectures differ most and what impact this has on programming.

For people new to kernel development or development in general, the set up of the development environment is discussed next. It will give a brief overview for ideas and possibilities on how to create your own development environment.

The implementation of MinixPPC is discussed next, what choices were made and why. Many portability problems were found at the start by reading source files and using "common sense." Most problems will be discussed with examples found in the first code base and the eventual code for MinixPPC. The chapter includes (new) drivers needed (only) for MinixPPC and a number of utilities to compile and convert programs.

To continue the project, chapter six is written on how to compile MinixPPC after a change to the kernel, server or driver. Where files are located and which directories contain what. Chapter seven will give a overview on know issues with the current version of MinixPPC, and a conclusion about the project.

II Open Source Software

To understand the workings of certain peripherals in the development machine it was needed to view driver code used in Open Source software. Especially for the MacIO ASIC driver. There is no "real" documentation to find freely on the internet or provided by people for it. There are general guides about the workings but one wants to have register names, locations and purposes.

The drivers located in the Open Source software where built by people who have reversed engineered code developed by Apple itself, and with a lot of trial and error. The driver used by the NetBSD kernel has been in development for some time and has been a source of information for the MinixPPC MacIO driver. It is easy to underestimate the importance of drivers, most of the time in the first half of the project went into understanding the peripherals.

This project would still been possible if there was no open source software available, but would have taken a "lot" longer to come to it's current state.

III Public

People who would like to know what it takes to port a operating system should read this thesis. But you must have some background in computer science. The author started without any knowledge about the target system, but with a “decent” education in computer science. The reader should be familiar with languages like C and needs some skills in understanding assembler. Assembler is being kept to a minimum and where possible C is used to explain the problem.

Writing and compiling programs should not be a issue. The reader should understand what differ Makefiles, source and header files, there content and goals. He or she knows what object and library files are.

Terms like “booting”, “loading” or “main memory” should be familiar. The best way to guarantee you know most of the general terms in this document, you could read the book “Modern Operating Systems” or “Operating Systems design and implementation” [8]. If you could chose, better to take the latter, it tells in details about the MINIX operating system.

IV Documentation

This is in no way a replacement of the documentation written for the PowerPC architecture. If you need to program assembler for the PowerPC on a user level make sure you read at least Book I for the PowerPC [10]. Book II [11] covers extra instructions users could use to access certain system registers on special cases. To program at system level read Book III [12], this covers the additional supervisor instructions and functionality.

This document only touches the architecture, it does not provide “the solution.” Only the parts of the architecture involved to the port of MINIX are described. Details like which bit sets NO-EXEC for a memory segment are omitted, only the “goal” of operations are described. The PowerPC CPU used supports various power modes and virtual memory support but these features are not studied. To fully understand the possibilities of the used PowerPC architecture one should read at least [9] and the specific processor type documentation. All these documents are included on the project CD-ROM and downloadable from IBM for free.

Much specific documentation about hardware needs to be ordered (if possible at all), especially for ASIC devices designed by Apple. With a ISBN number it would be easy. This project has been done with no documentation ordered. To fully understand and make use of special features it will be needed to order the specific documentation from the device manufacturer.

V Conventions

To make sure there are no problems about definitions over the meaning of terms and words here are how they are used in this document.

List of conventions,

0x1000_0000	A hexadecimal value.
(C) header file	A file used for declarations, defining types and constants usable in multiple object files. “Always” with the extension .h.
(C) object file	The source file containing the implementation code, the code that needs to be compiled to create the object for the system to run. For C programmes this is the file with extension .c.
a.out	It defines a executable program (or process) image. It contains various fields, at least the sizes of the text, data and bss section of the program.
Activation record	The stack layout (memory structure) of a running function, defining the locations of the return address, the first argument, first local variable and more (defined by the architecture ABI).
BIOS	Where BIOS is used it refers to the boot firmware or bootROM used by the IBM PC compatible to prepare the machine to boot the operating system
Click (memory)	block of memory, unit of memory wherein memory is allocated. For the bigger machines (4GiB address space) 4096 bytes. Memory pages are also allocated in this size.
Stackframe, CPU state	The minimal information to suspend a process and (re)start it later.
Development machine	A Apple iBook G4 1333Mhz, late 2004.
Interrupt, Exception	There are two causes of by which the stream of instructions can be broken, by interrupt or by exception. For the system there are few difference between them, and in the power documentation there are only exceptions. For this document a exception means a error in the system has occurred and a interrupt is a “normal” system operation.
Kernel image	Or kernel process image, the first process image in the system image.
Library, Archive	A collection of pre-compiled source files or objects collected in one (big) file, most of the time with the extension .a.
Load address	The physical start address to were the program section is placed.
Machine, Architecture, Platform	When used define a complete system, in PowerPC “architecture” all, hardware needed for the PowerPC CPU to work and from a complete computer system.
Makefile	Input file for the “make” program this programs aids in compiling

	and linking programmes consisting of multiple source files. It also keep track of changes to source files and only recompile them if needed.
MINIX	The MINIX OS in general.
Open Firmware	Used by the PowerPC platform to provide the same as function the BIOS does for the IBM PC, although open firmware is more advanced than the usual BIOS program. It has support for (a) file system(s) and can execute files using elf32 or COFF format.
Physical address	The address a virtual or effective address translates to (when the MMU contains the mapping).
POWER, PPC, POWER CPU, PowerPC	All these terms refer to the IBM POWER architecture, although the PowerPC architecture is (technically) a derivative of the POWER architecture.
Process context	The "memory map" when the process runs. Every machine capable of protected mode can create a "process context" by defining the memory the process can access using segments.
Process image	The information the OS needs to load in memory to start a process running. This is almost always the executable file read from disk.
Section	A part of a program that can be loaded into memory or stored on disk. For the a.out executable format there are three major sections, 'text', 'data' and 'bss'.
Segment	Region in physical memory with access protection and type.
Segment identifier	The "number" used in translation to indicate the segment used. This is the same number as the VSID.
Supervisor	A program or process with all rights to the system, has the ability to use every instruction of the architecture.
Symbols	Used in our context are the "addresses" of functions and variables used by the compiler. Sometimes functions and variables are together called "the symbols" of a program.
System	In a global context the complete operating system, kernel, tasks drivers and library.
System image	This is the complete "image" that's needed for the monitor to read form disk, it contains the kernel image (just another process image), the process management image and more.
Text, Code	Text and code are used to indicate the same program section. The machine instructions.
User	A program or process that runs in a shielded environment and has limited rights. Can not use every instruction of the architecture.
Virtual address, Effective address	The address the machine instruction is using.

VI Acronyms and Abbreviations

Acronyms and abbreviations used throughout the text, giving the page of the first occurrence.

Abbreviation	*Stands for*	*First page found*
ACK	**A**msterdam **C**ompiler **K**it	4
ADB	**A**pple **D**esktop **B**us	19
API	**A**pplication **P**rogramming **I**nterface	4
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit	19
BAT	**B**lock **A**ddress **T**ranslation	23
BIOS	**B**asic **I**nput **O**utput **S**ystem	17
CISC	**C**omplex **I**nstruction **S**et **C**omputer	17
CPU	**C**entral **P**rocessing **U**nit	5
DSI	**D**ata **S**et **I**nstruction	23
IA	**I**ntel **A**rchitecture	1
IA-32	**I**ntel **A**rchitecture-**32**bit	1
IBM	**I**ndustrial **B**usiness **M**achines	1
ISI	**I**nstruction **S**et **I**nstruction	23
MDC	**M**achine **D**epended **C**ode	4
MIC	**M**achine **I**ndependent **C**ode	4
MMU	**M**emory **M**anagement **U**nit	23
MSR	**M**achine **S**tatus **R**egister	11
OS	**O**perating **S**ystem	4
PAT	**P**age **A**ddress **T**ranslation	21
PMU	**P**ower **M**anagement **U**nit	19
POWER (CPU/PC)	**P**erformance **O**ptimization **W**ith **E**nhanced **R**ISC	1
RISC	**R**educed **I**nstruction **S**et **C**omputer	17
USB	**U**niversal **S**erial **B**US	1
VIA	**V**ersatile **I**nterface **A**dapter	19
VSID	**V**irtual **S**egment **Id**entifier	26
VU	**v**rije **U**niversiteit (Amsterdam)	1

1

Introduction

1.1 About MINIX

The MINIX operating system was created to provide a practical tool for students of computer science to program and study a complete operating system. Other available operating systems at the time like UNIX were commercialized and had licences that prohibited use in class or viewing (and altering) the source code. Today there are various open source operating systems available, most notably GNU/Linux and BSD. Where BSD is one of many flavours. For students today (like myself) it would be extremely hard to know and understand operating systems by studying one of these systems. They have a large array of features and are therefore very complex. There are many professionals working on these systems, the community is very large and is still growing.

MINIX keeps "things" simpler, it is small, efficient and fast. MINIX is build up of modular components to isolate code and keep the overall system scalable and stable. In the early approach it didn't use all hardware functionalities in the kernel. To keep compatibility with simpler (older) systems and increase portability to other systems. As example, virtual memory is not supported (at time of this writing) although most systems today provide hardware support which simplifies software implementation. But MINIX features and supported programs are growing. Work is done on virtual memory support and virtual file systems, these new features are needed to make MINIX accessible for a greater public.

The number of people working on MINIX at the VU is viewable in one browser page and all fit on one floor. This has advantages and disadvantages. Some advantages are, that communication lines are small so a simple decision is made quickly. All knowledge about the system is available and if needed you can blame someone you know by face.

But the small number of people limit the development speed. If someone has particular hardware it could be that it's not supported by MINIX, it has a limited driver base. For MINIX to get USB drivers it could mean that work on the "kernel" gets suspended. Also fewer people are less creative, no matter how good they are.

1.2 MINIX v3

The PowerPC port was originally done from the MINIX 3.1.1 tree. This is the default tree for the IBM PC compatible and downloadable from the www.minix3.org site. In this document we will refer to this as the "IA-32 code base" or simply "IA-32." It has support for older architectures, like the Motorola 68000, early Macintosh, SUN4 and ATARI. Some of the older architectures are not fully supported as development for these have stood still for quite some time now.

People familiar with MINIX know that MINIX is implemented using a micro-kernel design. This means that the kernel has a minimum of functionality and system processes are moved to user space. This increases reliability by "less" critical code. It uses separate processes to do various system tasks. This forces modular design. For communication between parts of the system it uses messages. Non-critical processes as the file system and process manager are run in user space. Starting with MINIX v3 even drivers are run in user space.

To make a small introduction, there are three processes, called tasks that run in kernel space, the IDLE, CLOCK and SYSTEM task. IDLE is the process that's running when there are no other processes to run. The CLOCK task is called for every clock tick generated by the interrupt system. It keeps MINIX alive by scheduling processes and provides a interface for the system time and timers. System calls are handled by the SYSTEM task. It does remote memory mapping, forking of processes or device I/O. Not all processes can make the same number of system calls. For example server processes have the possibility to do device I/O a "normal" user process doesn't.

The MINIX v3.1.1 system used in the port consist of these processes,

- IDLE idle task
- CLOCK clock task
- SYSTEM system task
- HARDWARE pseudo process, kernel task
- pm process manager server
- fs file system server
- rs reincarnation server
- memory memory driver
- log log driver
- tty terminal driver
- at_wini ATA-IDE driver
- ds data store server
- init first user process

There are drivers newly developed or rewritten while the port was in progress. The new drivers are needed for the system or better modularity and increase of portability. Not all of

these drivers are needed for MinixPPC to function. The “dbg” driver is only used for debugging and usable while kernel development is in progress. It enables the developer to access (and print) the data structures of the kernel. When the MinixPPC kernel is stable enough it should be removed. The “pcim” driver is developed to help PCI drivers get information about PCI devices in the system. At the moment it is only used by the MinixPPC “at_wini” driver. If the “pcim” or its design ideas get a place in the final MINIX tree is to be seen. The “macio”driver is used to access simple input (and output) devices of the iBook, and is critical for MinixPPC. This driver will always be needed in some form or another.

Current MinixPPC only drivers,

- macio MacIO driver
- pcim PCI manager
- dbg debug process

While working on this project, development on the IA-32 base code continued. There were new modularities built and code upgraded, by increasing functionality or removing bugs. When doing a project with a large timespan it is inevitable that the code base used for the (first) port is developed further (and faster, as more people work on it). So the initial version of MinixPPC is lower than MinixX86. In the end the new modularities and processes need to be merged or re-ported which isn't always a trivial process.

When porting a operating system you always start with existing code. If you are lucky this code is layout that architecture dependent code is seen quickly so there are no problems identifying the entries to machine independent code. Porting is easy and fast then, if not, a reasonable approach can be made to set up rules identifying code, but experience and common sense help the most. Chapter 2 “Porting an OS” will give a method for creating portable code from scratch in a way that all entry points are identified. Also guidelines are given to convert existing code to the same layout as creating portable code from scratch.

2

Porting an OS

2.1 A complete system

A operating system consist of more than “a kernel.” It has lots of programs and utilities to manage and work with the computer. A decent operating system would provide a software development environment to create programs. IA-32 MINIX provides a C compiler (ACK) and libraries with a API that conforms to the POSIX standard. When porting a system not only the kernel but also the system libraries must be ported. They are used by the systems compiler to build the programs and utilities, including the compiler itself.

System software is (most of the time) written in a low and high level language. The low level language is the assembler code for the target and the high level code could be C, C++ or a mix. Before porting it is wise to see if there is a compiler for your target that uses the same high level language as the system itself. MINIX is written in (ANSI) C, there is a good compiler for it from the GNU project, GCC. Some people call C an architecture independent language as almost all hardware has a C compiler for it. There are even compilers that accept “language X” to compile it to a C program and then call the system C compiler to generate the final executable.

Most problems arise when advanced compiler functionality is used, exotic types or alignment features like bit fields. If possible always try to avoid non-standard tricks, it will save lots of trouble in future ports.

All program code is dividable in two parts; machine dependent and machine independent code. If it's not, you have either universal code that works always, like compiled Java byte code that uses a virtual machine (that is compiled for the target) or something like assembler code that only works for one architecture. The rule is, the closer the language to the hardware the less portable it is. Object code (compiled source files) is the glue between the hardware (via assembler instructions) and software (C) so it's never portable.

Porting is all about rewriting Machine Dependent Code (MDC) from one architecture to the target architecture. The creation of portable code is the isolation of the MDC from the Machine Independent Code (MIC) so the MDC is clear and rewriting as easy as possible.

The MINIX source tree consists of two types of code, Intel x86 assembler and C. Although the MINIX source tree contains assembler code for 16bit and 8Bit Intel processors we will use IA-32 to name all Intel type assembler (and architectures). All the assembler code is sure not to compile for the PowerPC architecture and every functionality provided by assembler code needs to be ported to either (high level) C or PowerPC assembler.

For C code there is a catch; even with C code that would compile for every architecture machine dependencies can exist. Most obvious because of different hardware implementations that only look the same for the software's point of view.

For example the implementation of device I/O. Because of the differences between the

IA and PowerPC architecture in the use of port I/O. The details will be in chapter 5.4.4 “I/O.” But for I/O these problems are excepted, code does compile without any problem (not even a warning) but does not work in any way. Luckily this incompatibility is easily found and the solution is simple.

To sum it up, for the MINIX kernel to work it needs certain “key” services from the hardware. These are found in the interrupt, memory and clock system. These services need to be isolated from the “general” management kernel code, like the code that does process scheduling or interprocess communication. The original kernel source did not separate the driver code of the “in kernel” drivers with the rest (management). The isolation method used for MinixPPC is explained in the next chapters. Starting with “creating portable code”, the driver model used to separate MDC from MIC and ending with a example.

2.2 Creating portable code

To create a portable system you have to make a separation between machine dependent code (MDC) and machine independent code (MIC). The better the separation, the easier to port the system. Taking the kernel as an example, in a ideal situation there are two parts that could be compiled without each other and then linked to form the kernel for the architecture. This does not mean that IA-32 compiled MIC could be linked with PowerPC MDC as they have different machine instructions. Compiled MDC and MIC parts must match architecturally.

This split in code increases modular design which is a good thing and when done correctly, increases scalability and security in the kernel itself. When multiple people work on the kernel, each can have his/her own speciality and modules can only used “exported” functions of each other.

There are essentially two situations determining if code is machine dependent. All assembler type code and all code directly accessing system hardware is machine dependent. Code directly accessing hardware is (most of the time) located in a device driver. For the kernel this includes functions that are using special CPU registers, and all in-kernel device drivers. For “general” system drivers it is normal to assume that they are system dependent. Although MINIX uses a “log” and “memory” driver that don't directly access hardware so they can be taken as machine independent. Keeping code separate creates more files. To keep them organized directory structures are created. In all systems this starts with keeping architectural code in different directories, usually <*./kernel/arch/xxx/*> at the kernel tree. With *'xxx'* as the target directory. The general source files needed for every architecture are in the root directory <*./kernel/*>. A view of the MINIX kernel tree,

```
./kernel            general kernel files
|-- arch
|   |-- ppc         implementation PPC code
|   `-- x86         implementation IA code
`-- system          implementation system calls
```

Figure 2.1: *Kernel tree (simplified).*

Updates to one of the files in the PowerPC MDC would require a recompile of the code in the <*'./kernel/arch/ppc/'*> directory and a re-link of (in this case) the kernel process. The recompile creates a new library that contains the MDC for the PowerPC kernel. This library provides all entry's (functions) for the MIC of the kernel to the hardware.

For the MIC and MDC to link together and work as intended, an interface must be provided. This interface describes the incoming arguments and result(s) of every function in the MDC library. You could see it as the "libc" system library, but then as "Kernel Hardware Interface" (KHI) instead of a system API. Figure 2.2 shows the KHI between the kernel and the hardware. The MDC part of the kernel could be called 'arch.a' and the MIC 'kernel.a'. Most of the time when building the kernel, 'kernel.a' would not be created, all objects are linked into the final executable right away.

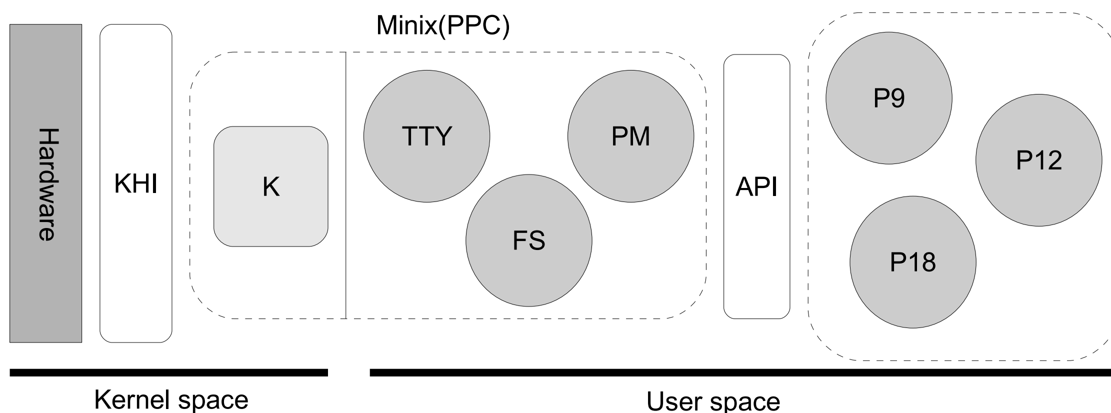


Figure 2.2: *Kernel Hardware Interface (KHI) location.*

To define and recognize interface functions, a "driver model" is used. It is called this way as it is the default way of building a driver front-end in ANSI C. More generally, the programming construct is used to encapsulate code within a logical module (or object). The driver model used is discussed in the next sub chapters.

If one considers the CPU to be a (general) device, the code directly accessing the CPU registers is nothing more than a device driver. This would mean that all MDC for the complete system is located in device drivers. Using the device driver model for all functionality needed from the CPU isolates the CPU and will aid portability. It also gives a uniform programming style throughout the system code. This however makes a lot of parts [MIC, MDC] to keep track off. Makefiles shall help a lot in keeping the system easy to compile.

There are three major ways to create portable code; from the preferred to the less preferred way,

a) Create (extra) files, reimplement functions,
b) Using type (re)definitions,
c) Using compiler directives.

Figure 2.3: *Ways to create portable code.*

Using the the first (a), the programmer would make at least two object files. One containing the MDC and one containing the MIC. The actual driver is the file containing the MDC. For every other port there would be a MDC file added. A interface defined in a common header file would give the MIC object, using the driver, access to functionality provided by the MDC object. This is the preferred way because it introduces the isolation of driver code to a file. In essence you are redefining functions.

The interface file would contain a list of prototype functions. When compiling for one type of architecture the file <*file1.c*> is always needed and compiled giving <*./file1.o*>. For <*file2.c*> there are as many versions as there are architectures, only the version for the target architecture is compiled, providing the functions for the prototypes in the interface header in <*./interface.h*>. Next are two source files that are present for one driver, it's defined by the interface, and used by file <*file1.c*>*,*

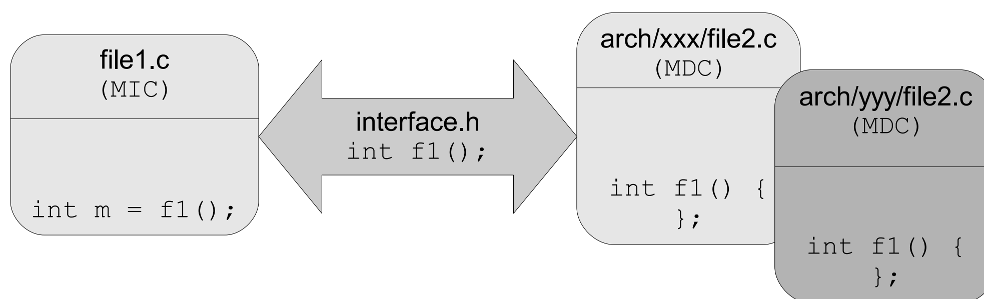


Figure 2.4: *Multiple files providing a body for function "f1()."*

The second programming way (b) is usable with careful programming, but implemented very powerful, redefining types. This does "not" include redefining the simple or standard types used by the compiler. Only advanced data types created with the C language keyword "struct." This data structure could contain members that are simple types or that are advanced again.

To be most effective this method requires that corresponding members of the (data) structures have the same name and semantics. When using the data structure in handwritten assembler files the layout in memory is very important. Then when redefining the data structure the "key" members must be in the same order. Otherwise the same offset into the data structure points to another member. When the data structure is only used in C code the layout of the members is calculated with every compile. The "main" reason for using this paradigm is automatic calculation of the addresses and size needed by the data in the object or memory.

MinixPPC uses this construct in the definition of the process table. The process table entries contain two key (advanced) structures that define the state of the processor when the process was switched out. The first of these structures 'stackframe_t', contains the process CPU registers like, stack pointer and program counter but also a array of general purpose registers. The other 'segframe_t' contains the segment identifiers, defining the memory settings for the process. Some members of the stack frame structure are used in certain system

calls like “do_exec” so these members need to have general names. Luckily every system must have some of the members in the stackframe, “all” systems have a program counter, stack pointer and more similarities. These general names must be provided so code in the MIC part could alter these when needed. Most of the general purpose registers aren't touched but they need to be saved as well, taking different sizes for every architecture, so room must exist in the process table entry. The amount of CPU data saved for the (current) PowerPC process switch is 408 bytes ('stackframe_t') + 64 bytes ('segframe_t'). For the IA-32 switch it is 64 bytes + 8 bytes. A clear indication the PowerPC is a register machine. The redefinition takes care in allocation the memory in the page table with every (re)compile per architecture. Further details about the CPU state save and process switching is found in chapter 5.5 “Exceptions and context switching.”

So using type redefinitions “automatically” creates room in the process table. The only problem is how to redefine the type on compilation of the target architecture.

Files (most of the time header files, but could be any) can be included in two ways “relative” from the source file position and “searchable” in a directory list. This is indicated by the way they are included in the source file, within quotes or within angled brackets. We don't want the relative include as this would make us write out the path of the header file and limit the include to one file. Using the “search” include we can limit the search to headers with the same name but in the directory we like.

Most compilers come with a option to add directories to the search list. This is what we use to include another header file while keeping the include directive in the source file intact. It is important that we don't need to alter the source file in any way. This method requires a separate architectural “include” directory besides the standard system include directory. Only the directory for the target architecture is added to the list. If you accidentally include files both defining a type you would get a redefinition warning and all problems can be solved, but compiling source A with header X and compiling source B with header Y would cause very nasty bugs.

Next all the files needed to redefine a (data structure) type,

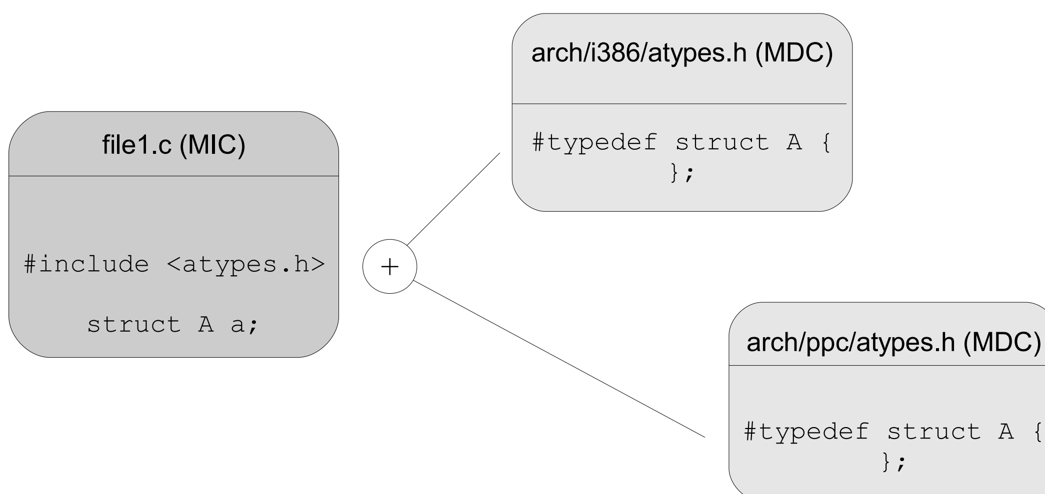


Figure 2.5: *Including header files with the same name.*

The GNU C compiler has the option '-I' to add a directory to the list of directories where the compiler searches for included files. So when compiling for IA-32 we would add “-I ./minix/arch/i386/” to the compiler command. Looking at figure 2.5 “#include <atypes.h>” in <*file1.c*> would resolve to the file <*./minix/arch/i386/atypes.h*>. For the PowerPC compile we use “-I./minix/arch/ppc” and resolve to the file <*./minix/arch/ppc/atypes.h*>**.**

These directories can't be in the standard (or system) include directory like <*/usr/include/*>, as search conflicts could occur. Most compilers deal with this by taking the first match found. This could introduce programming errors like above that are hard to find because no warning is given.

The last way (c) to switch code that need to be compiled is using compiler directives like “#ifdef, #if and #endif.” By using a global definition these directives tells the compiler if code should be included or just ignored.

Looking at the code in listing 2.1 MACHINE tells we are building for the IBM PC or PowerPC, it is defined in a global file so every source file shares the same definition. It needs to be global to make sure the definition of MACHINE is the same in every file of the project. Getting different definitions for MACHINE will introduce problems because functions or definitions for different architectures could be used together.

In the next listing we are using “MACHINE” to indicate the target we are compiling for,

```
1. #define MACHINE      POWERPC     /* or IBM_PC */
2.
3. typedef unsigned int reg_t;
4.
5. #if ( MACHINE == POWERPC )
6.   typedef struct A {
7.     reg_t stkframe_space[102];
8.     reg_t segframe_space[16];
9.   } A_t;
10.#endif /* #if ( MACHINE == POWERPC ) */
11.
12.#if ( MACHINE == IBM_PC )
13.  typedef struct A {
14.    reg_t stkframe_space[16];
15.    reg_t segframe_space[2];
16.#endif /* #if ( MACHINE == IBM_PC ) */
```

Listing 2.1: *Code fragment illustrating the “#ifdef” directive.*

Note, the types in listing 2.1 are just for illustrating, and not used in MINIX. Also take note at the commenting on line 10, “`/* ( MACHINE == POWERPC ) */`” this indicates the end of the corresponding “#ifdef” statement above.

There is no reason that compiler directives don't work, but it has one big disadvantage, it is not scalable. When we port to another architecture we need a extra “#ifdef” block. Probably for every “#ifdef (MACHINE == XXX)” there is already. This makes us updating the same source files again for every port that we make, something we don't want. Ideally it should be possible to create a new port without touching the general source files.

Luckily here the number of lines between the directives is small but most of the time code spans a reasonable amount. That brings another disadvantage of using this construct,

readability, the most important quality of source code (otherwise we would have stick to assembler). Readability drops when using compiler directives through blocks of code. You could be in the assumption that code is executed while it's between directives just scrolled off the screen, and it's not compiled at all. Using this in code blocks should be avoided at all cost, as it introduces dead code. Programmers new to the source code would have a hard and long time reading through it.

If use is inevitable it should be used in a way it doesn't span more than a few lines. If possible split the file and include the file needed with a include directive between the conditional directives. Create a number of files equal to the number of conditions and move the code from the conditions to one of the new files and include that file where the code was located.

Also the effects should stay local and must be clearly seen. Machine models could have other devices or need different parameters per model. Most of the time the impact on the code are only a few lines. Using the directives is the solution, but in fact you are just programming locally (in MDC of the architecture).

But to be complete, there are some exceptions though, the library headers use compiler directives as well. These are mostly to define basic system types when being compiled with certain flags; no “user” should ever need to look at them.

2.2.1 Creating portable code from existing files

This is a bit tricky as you have to look at the original file and determine if the “function” or “definition” used, is architectural. Most of the time it's easier to determine that code is used in management then for initialization or communication with “some” architectural option. The first thing done is removing all “for sure” architectural code (MDC) leaving the management code (MIC) behind. Trying to compile and link the management code will list the dependencies, functions and symbols that are undefined. Giving a hint on what the MDC was providing.

Then create a new source file with the removed MDC, the symbols and functions taken from the original. Define the interface of functions in a “mutual” header file. It is best to avoid global symbols and definitions. The “global” definition of the port number X of device D can be totally different between architectures. The architecture could not have a device D, the port could be at M etc. A good interface should not have to change between architectures. If it does, it wasn't good. Updating a interface should not have to be a problem updating the “older” architecture to accept the interface function can fix it, the “older” architecture just ignores it. Whenever code is found between compiler directives as in listing 2.1, this code should be moved to the new MDC source file. Note that dependencies of the moved MDC should move to if they are not from the system libraries.

In the end both source files should be able to compile without each other. The only thing in common is the include of the interface file, with a line like this “#include <interface.h>.” The creation of portable code from scratch should be no different. Just follow the same rules. Keep management to the left and architecture to the right. For MinixPPC the driver programming model” is used to keep left and right apart. A example is presented in chapter 2.3.1 “Clock system example.”

2.3 Driver programming model

The driver model presented is used for the in-kernel drivers to guarantee the separation of machine dependent and machine independent code. Recalling points 3 and 4 in the preface, we also use it to isolate the CPU from the kernel. The (low-level) driver model used for MinixPPC follows the following criteria; ease of future porting, logical partitioning, no overhead and a clear view of the code purpose. The required points have a deep impact on the structure of a driver. Although they could be considered related, a set of rules can be used to guarantee separation between them. As stated earlier, there are two types of code, dependent and machine independent code. Which automatically gives rise to the second point. The final "real" driver only contains MDC accessible via the driver interface, making the logical partitioning.

The goal of using the driver model in the kernel is the isolation of the parts that are not kernel related (MDC). The writing of a register in the OpenPIC controller to disable the "8259A pass through" is not kernel related. What is kernel related is the way how the kernel enables access to memory, schedules processes and when it enables or disables interrupts. "Machine dependency" is only located on the end of a implementation, for example the last function called (sometimes in assembler) that sets the flag to disable the external interrupts in the MSR.

The best way to explain the driver model and porting of existing code is by a running example. As example we take the MINIX v3.1.1 clock system. It was originally written for the IA which used port I/O to access the clock hardware located in the 8259A interrupt controller. This controller (or compatible) is located in every IBM PC compatible. The <*./kernel/clock.c*> file provides the clock task for MINIX and also keeps track of several (watchdog) timers. The file consists of two sets of functions. A set that does hardware I/O and a set that does management, this includes the clock interrupt handler.

Clock hardware is accessed by the software on three occasions,

1. Initializing the clock hardware
2. Reading the current clock count
3. Stopping the clock

Listing 2.2: *Clock hardware access.*

The management functions are used by processes to set up and use the watchdog timers. The file also provides an interrupt handler. It is called at every clock tick by the interrupt driver and updates timers. As a side effect, process accounting is also done in this file. All the code inside the <*clock.c*> file is C code and would compile without any warning for the PowerPC architecture, but it is sure not to work as the PowerPC (used for development) doesn't use the 8259A interrupt controller to generate a timer tick. This calls for a rewrite of the <*clock.c*> file. We start with the three points in the file where the hardware is accessed, these need to be isolated with the driver model.

The driver model isolates these parts of code and is providing a interface definition with predefined functions for the final step to access the hardware.

2.3.1 Clock system example

The number of files needed for a driver created via the driver model varies. The actual driver will be only “two” files. The header defining the interface, that gives the function prototypes, and the source file providing the function implementations. Extra files are needed for bigger drivers (like the MacIO driver, chapter 5.7.1). The original clock file (<*./kernel/clock.c*>) had the clock task, timer management, process accounting and hardware access code all in one file.

The example clock system shall consist of three files,

1. <*kernel/clock.c*> machine independent code [MIC]
2. <*kernel/interface.h*> providing the clock driver interface
3. <*kernel/arch/xxx/clock.c*> machine dependent code [MDC] (the driver).

Listing 2.3: *Clock system files.*

The (altered) <*./kernel/clock.c*> file will still contain the clock task, timer management and process accounting code. These are an integral part of MINIX itself. The interface file <*./kernel/interface.h*> would contain (in this example, four) functions inside a structure declaration for maximum security and isolation. The hardware access functions are going to move to the new file <*./kernel/arch/xxx/clock.c*> providing the implementation of the functions described in the interface. The definition of the interface states what functionality MINIX needs from the hardware to function. It should be clear that the parts of code that accesses the clock hardware is going to be moved to the <*./kernel/arch/xxx/clock.c*> file.

The isolation of the driver code gives the developer much freedom to program the driver implementation, as long as the interface is respected. The PowerPC architecture uses the OpenPIC standard for it's interrupt system, it has four general purpose programmable timers, also the PowerPC CPU has a programmable clock timer that produces a periodic interrupt (decrementer). MINIX only needs one timer. So the programmer is free to chose which timer he or she is going to use to generate the clock interrupt. One of the four OpenPIC timers (t0) is set to produce an external interrupt 60 times per second.

A possible interface definition for the clock driver in <*./kernel/interface.h*> could look like this (note that this is not the interface used by MinixPPC),

```
1. /* Example clock functions.
2.  */
3. typedef struct if_clock_s {
4.   int     (*init)(void);  /* init the clock hardware. */
5.   int     (*stop)(void);  /* stop the clock (if possible). */
6.   int     (*start)(void); /* start the clock (if possible). */
7.   clock_t (*read)(void);  /* read the current count. */
8. } clock_interface_t, if_clock_t;
```

Listing 2.4: *Possible interface file for a clock driver, in <interface.h>.*

In listing 2.4 the interface prototype functions are placed inside a data structure, 'if_clock_t'. This groups functions to the “one” driver they belong. The way functions in data structures are called provides a clear indication for the programmer that he or she is using a machine dependent function. We will see how a function is called from a data structure in a moment.

A careful reader would say that giving all functions prefixes like “clock_init( );” and “clock_stop( );” would get the same result and groups functions to the clock driver as well. This is true and works but it does not isolate the code. A file without “knowing” the interface definition could access the driver functions with little effort. All that is needed is the name of the function and a “correct” argument list and the functions are free to be used all over the program. The grouping of access to functions inside a data structure give an extra step in using the (driver) functions. It makes sure the source file that implements the functions must know about the data structure by including the interface definition.

Continuing the clock example, starting with a snippet of MDC.

```
1. FORWARD _PROTOTYPE( int init, (void) );
2.
3. /* The only way to access the functions.
4.  */
5. PUBLIC const if_clock_t Clock = {
6.   info:info,       /* info about this system */
7.   init:init,       /* clock initialization function */
8.   start:start,     /* start the clock */
9.   ...
10.};
11.
12./* Init the systems clock/timer hardware here.
13. */
14.PRIVATE int init(void) {
15.  /* set the timebase to zero */
16.  mttb(0, 0);
17.  opic_timer_write_vp(OPIC_TIMER0, OPIC_UNMASK,
18.                      (DEFAULT_IRQ_PRIORITY + 1), CLOCK_IRQ);
19.  ...
20.  return 0;
21.}
22.PRIVATE void start(void) {
23.  /* (re)Read the base count value in the count register
24.   * and continue counting.
25.   */
26.  opic_timer_set_count_inhibit(OPIC_TIMER0, OPIC_TIMER_FREE);
27.}
```

Listing 2.5: *Implementation of MDC.*

The interface in listing 2.4 needs to be respected by both sides of the code. Listing 2.5 is a fragment of code from the *<./kernel/arch/xxx/clock.c>* file. Note the private and public usages on lines 5, 14 and 22. All functions (and file global variables) inside the MDC file(s) should be private so they are only accessible via the members of the (public) data

structure or interface. Then defining the structure as a “constant” (line 5) the function references can never change after compilation. This is recommended for “security” reasons while compiling the kernel. When we look at line 16 we see a typical machine dependent call. This is a call to a function written in PowerPC assembler and zeros out the 64 bit timebase register, that is used to keep the uptime of the system. At line 17 the clock hardware is initialized by using a function directly accessing the OpenPIC registers, to produce a 60 HZ interrupt.

In MIC these functions are called in the following way,

```
1. /* Hook to the system dependent structure, the structure
2.  * defined and initialized in de 'arch/xxx/clock.c' file.
3.  */
4.
5. #include “kernel/interface.h”
6.
7. extern const if_clock_t Clock;  /* Clock driver data structure */
8.
9. Clock.init();     /* init the clock hardware (MDC) */
10.  ...
11.Clock.start();    /* starting the clock hardware to tick. */
```

Listing 2.6: *Calling a “driver function”, to initialize the clock hardware.*

Looking at line 7 and 9 of listing 2.6, line 7 hooks the MIC source to the data structure defined in listing 2.5 (line 5). Line 9 is the actual call of the initialization function in listing 2.5 (line 14). The form of the call, 'Clock.' is prefixed to the actual function, and looks a lot like object oriented programming. It should be familiar with many programmers. Also the call form indicates the user and reader that this is a call into the MDC part of the (in this case) clock system. Note that the code includes the interface definition on line 5, otherwise it would be impossible to access the function(s).

With the driver programming model, porting to a new architecture is easy. All “kernel” related files needed to be rewritten are located in the <*./kernel/arch/xxx/*> tree. Choose the architecture closed to the new target and the work should be clear. Write the drivers keeping to the interfaces and MINIX should work right away. The hard part in this model is defining interfaces that work for all systems out there. Unfortunately we only know how good they are when several architectures are ported. Keeping interfaces to a bare minimum should help to keep incompatibilities to a minimum as well. Performance does not suffer from this model, only the compile time could increase as more files and data structures are used. The “references” in the data structures are not more then the addresses of the function that was otherwise private. Note it aren't function “wrappers.”

This method is made possible by the linker of the “tool chain” (compiler and linker). It's used to link the compiled objects into the final program, giving the possibility to insert any function as long as its symbol (start address of the function body) is provided. To view which files (objects) provide which symbols appendix H “Kernel symbol listing” lists all symbols in the kernel linkage. The linkage to the architectural code is marked in “**boldface**.” To start developing MDC you need to know your target hardware. The next chapter, 3 “Knowing your architectures” tells in fair details about the PowerPC architecture.

3

Knowing your architectures

3.1 Start

The first thing to do is knowing all there is about your target platform. In this case a PowerPC based system, more precisely an “Apple iBook G4.” The second thing to understand is MINIX itself. When porting you would frequently run into problems generated by MINIX MIC, that runs into problems created by a function in the MDC, the stuff you are making. The faster you find the failing function, the easier the port. Tracing a system function to the problem could take some time, knowing where it goes from MIC to MDC helps.

You also have to know about the make up of the current system, it's not needed to know it in the same detail as the target system. For example, on an IBM PC compatible it is possible, that software uses the BIOS to get hard drive parameters instead of the ATA interface. Experience is the magic word here. Anyone who has no experience should have patience, it doesn't come overnight or by reading a book or two.

Getting the general information about the iBook was easy, nowadays the Internet is the best source for that. Luckily there are open source ports made to the platform. This provided a great information source. Finding people with experience helps to build your own. However it doesn't guarantee that your project will have success. It turned out that the more specific information is not easily shared or available. Problems are so specific that it's not easy to formulate them for anyone to help, and there are components used in the iBook where almost no documentation is found for. This includes the MacIO peripheral, which drives the keyboard, mouse, power management and various other system functions.

A small table with some of the related parts and terms between the two architectures,

IBM PC compatible	***PowerPC CHRP (New World)***
BIOS	OpenFirmware
8259A compatible interrupt controller	OpenPIC compatible interrupt controller
CISC	RISC
Processor state word (PSW)	Machine status register (MSR)
6 (16bit) Segment registers	16 (32Bit) Segment registers
Local and global descriptor table and paging	Page table
Port I/O an memory mapped I/O	Only memory mapped I/O

Table 3.1: *IBM PC compatible vs. PowerPC CHRP.*

To get familiar with the differences of the two architectures in this thesis a small introduction is made to the IBM PC architecture. The PowerPC architecture is covered in more detail as this is our target. After the architecture descriptions we will look at the implementation of MinixPPC in chapter 5.

To indicate one of the differences between a CISC and RISC listing 3.1 shows a few lines of assembler code doing exactly the same; incrementing a byte in main memory.

Incrementing a byte in memory (at address 'k_reenter') with one,

```
/* A IA-32 assembler, */
1. incb       (k_reenter)          /* k_reenter += 1 */

/* B PowerPC assembler, */
1. lis        R2, k_reenter@ha     /* load high part of address */
2. addi       R2, R2, k_reenter@l  /* load lower part of address */
3. lbz        R1, 0(R2)            /* load k_reenter to R1 */
4. addi       R1, R1, 1            /* add one */
5. stb        R1, 0(R2)            /* overwrite k_reenter */
```

Listing 3.1: *Differences between assembler code for CISC and RISC.*

These pieces of code seem highly unlikely to do the same thing but they do. They increment the variable 'k_reenter' with one. Part 'A' shows x86 assembler and part 'B' PowerPC assembler. The one Intel instruction does essentially the same as all the PowerPC instructions because you can't increase memory in place. The CPU is needed to do the arithmetic, so the steps shown by the PowerPC assembler show what's done. First the value is loaded from memory to a CPU register, incremented and written back to memory.

The Intel instruction hides the steps enabling hidden hardware optimizations to fast access memory and use extra registers. As seen in the PowerPC part, we use 'R2' as address, keeping this register intact lets us use it again when writing the new value 'R1' back.

The only “extra” instructions the PowerPC assembler needs is to load the address of the variable to a register (somehow IA does it to). Because all PowerPC instructions are 4 bytes this leaves no room for a 32 bit address value. The two instructions on line 1 and 2 are used to load a 32 bit value in two steps, in this case the address of 'k_reenter'.

In performance there should not be much difference, as the above Intel instruction would take (a lot) more time to execute than one of the PowerPC instructions.

Some of the PowerPC assembler sequences are always the same. Ideal for the use of small macros. The file *<./minix/arch/ppc/asm.h>* contains (most of) the used macros in the MinixPPC assembler code.

3.2 IBM PC compatible

Information about the IBM PC or “IBM PC compatible” is widely available, every piece of hardware follows a open standard. Lots of (free) information is easily found on the Internet. The open architecture of the IBM PC compatible is why it's is the most successful per-

sonal computer at the time. If the documentation is free and accessible there are always people willing to program for it. All IBM PC compatibles are CISC systems and use the Intel x86 and compatible architecture.

3.3 PowerPC

The architecture used to build the iBook G4 is know as the “New World” architecture. It means it uses at least version 3.0 of Open Firmware for initialization and it's architecture is derived from the PowerPC “Common Hardware Reference Platform” (CHRP) [3]. Not all devices listed in the CHRP are available in the iBook. There is no SCSI or serial controller (at least not on the outside). There are several devices built in to the iBook not listed in the CHRP documentation: firewire, USB, Modem and a ATA controller. This is not strange as the document is from 1995 and the iBook model is from late 2004. The next devices are needed and used by MinixPPC.

The OpenPIC standard is used for the interrupt system. The documentation [5] is available for it. Some systems “still” use the Apple Desktop Bus (ADB) to communicate to simple and low speed devices like keyboard and mouse. The ADB is being phased out and replaced by USB, but our iBook still uses it. Access to the ADB, Power Management Unit (PMU) and NVRAM goes via a Versatile Interface Adapter (VIA).

The iBook has almost all devices located in a Application Specific Integrated Circuit (ASIC) designed by Apple itself. This chip is commonly referred to as the MacIO chip and contains at least the ADB controller, OpenPIC interrupt controller and VIA. At the moment MINIX is able to use the PMU to power-off and reset the iBook.

The logical view used for building the drivers for the MacIO chips “components” is presented in chapter “5.7.1 MacIO.” Building these was quite a challenge.

3.3.1 Booting

When the iBook is powered on, the first software run is located in the bootROM, the Open Firmware interpreter. At the moment the machine is set up to stay at the interpreter (showing a prompt) and wait for user input. See appendix J “Open Firmware” for the Open Firmware welcome screen and some typical commands used to view and set variables.

The boot process of the iBook can be divided into three stages; bootROM software executing (Open Firmware), the boot monitor loading the operating system image into memory, and operating system booting. The software initializes and preforms diagnostics on the system hardware. It probes the system buses for devices and builds the “devices tree” with the devices it finds. It query's I/O devices and maps them with the memory they need.

The Open Firmware “shell“ is able to interpret programs written in the Forth [6, part I] language. For the port it was not necessary to study and use the language. Open Firmware has three types of interfaces, the user interface, the client interface and the device interface. The Open Firmware interfaces are shown on the next page,

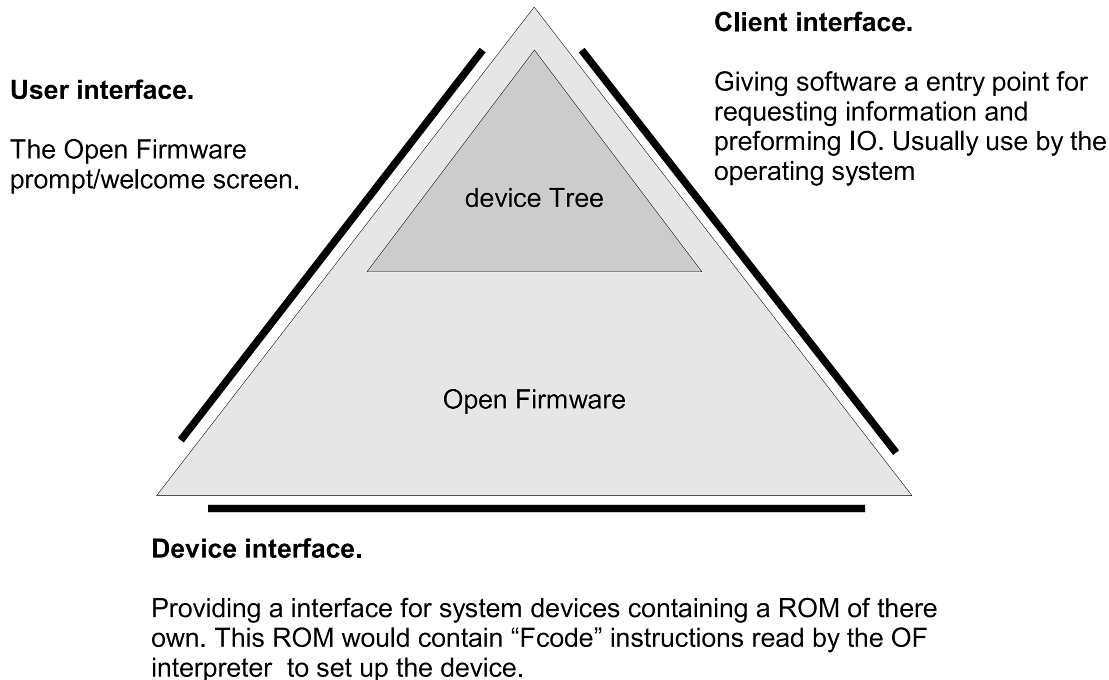


Figure 3.1: *Open Firmware, user, client and device interface.*

Peripherals, like network or video controllers, are located on one of the system buses and use the device interface. They could have a small Forth program or script in their ROM to be executed by the Open Firmware, customizing initialization of the device. Normally a "user" would not see anything from the Open Firmware interpreter and would "just" see the OS loading. With the "user" interface it is possible to view all system information and devices in the system, what resources they use and where they are. This information is gathered by Open Firmware into a device tree, which is browsable like a file system. The device tree can grow pretty big. For every device in the system a sizeable list of properties are presented, which varies from device to device.

The third interface is the client interface. This is a software entry point. This interface is used for example by the OS device drivers to request information about the devices in the device tree. Like the register locations and IRQ line of the ATA controller or the start of the memory space of the OpenPIC interrupt controller. The Open Firmware entry takes a variable number of arguments and is used in every information request. There is a library written that makes a front-end to the Open Firmware entry. It is possible to search for devices by there type or name in the device tree. Typically one would get back the device node or device handle and would use that for further request of information.

After the request for device information to initialize the driver, Open Firmware must release its hold on it. We will see later how MinixPPC drivers get the information about the device they must "drive."

Open Firmware has support for the "New World" boot block file system type, so it can access the hard disk on a file basis. Open Firmware can be upgraded by installing software packages into the bootROM. Depending on the packages installed, it supports execution of

multiple executable file formats including Elf32 (but no a.out, package is installed by Apple). As we will see later the support for Elf32 is used for executing the (second stage boot program) monitor booting MinixPPC.

After device initialization Open Firmware will search for a file named tbxi, (toolbox info) to complete the first stage of the boot process. In the toolbox file further information is provided how to boot the system, extra initialization code for devices and driver locations on disk. In our case the <*tbxi*> file would also tell where the boot monitor for MINIX is located, where this project really begins.

3.3.2 PowerPC CPU details

To port an OS you must know the details of the CPU used in the target system. When the OS does a context switch, you need to know which registers to save and which you can ignore or use while saving. The code that saves the CPU state has to be written in assembler for reasons that will soon be clear.

The heart of the iBook G4 is a PowerPC CPU of the fourth generation (type 740). It is a 32 bit CPU and can address 4 GB of linear physical memory. This CPU is, compared to the IA-32 CPU, loaded with registers. The CPU uses branch prediction and out-off-order execution of instructions. It can preform several instructions in parallel and all instructions are of the same size, four bytes. The processor supports big-endian and little endian addressing modes but MinixPPC keeps the default addressing mode, big-endian.

The CPU uses three basic data types, “8 bit byte”, “16 bit half word” and “32 bit word.” This could give confusion about data sizes as “word” is sometimes used as 16 bit value. Besides the simple types it also supports double and single precision floats. In this document and MinixPPC source the MINIX sizes are used, byte 8 bit, word 16 bit and long or other 32 bit.

There are two operating modes; user and supervisor. The operating mode defines the usable register and instruction set. A extension to the “user” instruction and register set is present in limited access (read-only) to certain registers.

The architecture defines three levels,

a) UISA User Instruction Set Architecture
b) VEA Virtual Environment Architecture and
c) OEA Operating Environment Architecture

Figure 3.2: *PowerPC architecture levels.*

A high level programming language like C shields the CPU registers from the programmer. It is simply not possible to access the content of a register with “normal” C programming. Also under normal circumstances, the compiler generates code that only uses institutions from the UISA. So if we need to use OEA instructions (and registers) we need to use handwritten assembler functions. Usually these stay short, simple and for a specific “task.” When for example, setting a flag in the MSR. When we want to make a CPU save, we need

to use special instructions as well, accessing registers that only the supervisor can do frequently. This means that at least “basic” assembler for the target system must be learned.

Shown in figure 3.3 is the PowerPC register set, from most privileged to less privileged register set,

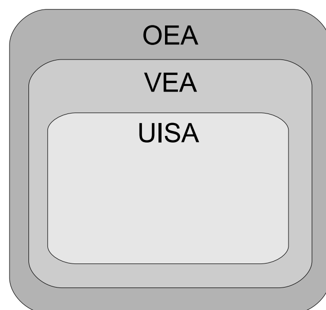


Figure 3.3: *PowerPC instruction set, OEA { VEA { UISA } }.*

Next are the registers present per level, [32] means a total register count of 32, range 0 – 31, and (32) indicates the size of the registers in bits. Note that register names, sizes and count can vary among CPU types.

Listing the registers in the “three” sets shown in figure 3.3,

```
USIA
      GPR[32] (32)
      FPR[32] (64)
      Condition register (32)
      Floating point status control register (32)
      Extended result register (32)
      Link register (32)
      Count register

VEA
      + USIA
      Time base lower (read) (32)
      Time base upper (read) (32)

OEA
      + VEA
      Machine status register (32)
      Processor version register (32)
      Memory management registers:
            BAT, configuration register[16] (32)
            PAT, Segment descriptor register 1 (32)
            PAT, Segment register[16] (32)
```

(***Listing 3.2:*** *continued on next page)*

```
Exception handling registers:
      Data address register (32)
      Data/instruction information register (32)
      Save restore register[0 – 1] (32)
      Special purpose register general [0 – 3] (32)
      Floating point information register (32)
Miscellaneous Registers:
Time base register lower (write) (32)
Time base register upper (write) (32)
Decrementer register (32)
Processor identification register (32)
Data address breakpoint register (32)
External Access register (32)
```

Listing 3.2: *PowerPC register names.*

The virtual environment only spans the time base and UISA, the user is able to read the timebase registers but can not write them. When a process is switched out, at least all the UISA registers need to be saved. We will see in chapter 5.5 “Exceptions and context switching” which additional registers are saved for MinixPPC.

The CPU has 17 registers for the PAT system and 16 for the Block Address Translation (BAT) system. The Memory Management Unit (MMU) consists of the PAT and BAT system. They are used to translate a effective (or logical) address to a physical address, supporting protection mode. The registers are used to set up the translation mechanism. Memory management details are discussed below.

The Floating Point Unit (FPU) of the PowerPC is impressive. It can be switched off via a flag in the MSR to disable support, including exceptions generated by it. MinixPPC supports use of floating point registers. Although no optimizations are used, at the moment all floating point state is saved in the process state (remember the large number of bytes (408) saved to the process table, defining the process state).

3.3.3 Memory management

This chapter covers the memory management for the PowerPC CPU in the iBook G4, not all PowerPC CPUs have the same capabilities. Our CPU can use address translation for both data access and instruction fetch. Not all CPUs have the instruction address translation. The memory translation process is done by the CPU Memory Management Unit (MMU). The MMU can be turned off by setting two flags in the MSR, one for data and one for instruction address translation. Killing both translations also disables all exceptions from the MMU.

There are two types of addresses in the machine, physical and effective addresses (also called logical or virtual address). The MMU takes an effective addresses and “produces” an physical addresses. Every effective address used should translate to a physical address in RAM or mapped (device) registers otherwise a “machine check exception” occurs.

A “quick” reader could have seen at the CPU register layout that there are three ways for the CPU to resolve a address; no translation, block address translation (BAT) and page ad-

dress translation (PAT). BAT and PAT can be used at the same time, when an effective address is resolvable via BAT it precedes PAT, this test is done in parallel. If then no translation can be made a ISI or DSI exception is made.

Giving a simplified MMU overview,

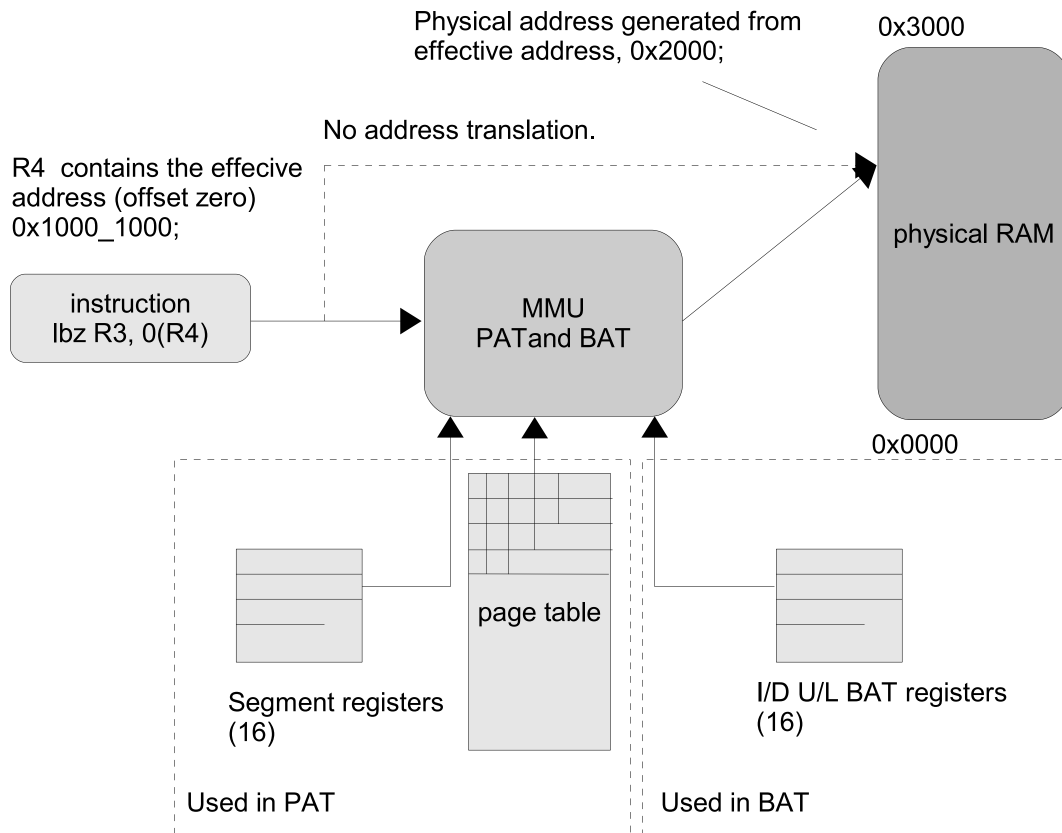


Figure 3.4: *Simple MMU overview.*

The PowerPC architecture is of RISC type. As a consequence the only way memory is accessible, is by store and load operations. Every general purpose register can be used to reference memory and can serve as a source or destination to load or store to memory. When a store or load is done the instruction uses the value inside a register as address to the memory. It is this value that is called the effective address. With this value (address) the translation starts. How this value translates to the physical address we will see next.

As shown in figure 3.4, the translation starts with a load instruction loading a byte from memory to (GP)R3. The value of R4 is used as address, with zero offset. It could be that the value of R4 is 0x1000_1000 but translated via BAT or PAT results to a physical address of 0x2000. So in fact the byte at physical address 0x2000 is loaded.

The PowerPC architecture doesn't use ports and has no “special” I/O port instructions like the Intel x86 instruction set has. Instead the PowerPC uses memory mapped I/O and

“general” load and store instructions to read or write from the registers. There are however instructions that guarantee sequence, order and completion of previous instructions in the instruction stream. To further aid memory mapped I/O, memory can be mapped with access arguments that force the MMU to keep coherence, force write through, inhibit caching and guard access. We will see later that caching of data in memory can give big problems when ignored.

There are (configurable) regions in main memory that are predefined. Otherwise the system memory is presented to the CPU as one linear address space. First there is the exception vector space. The CPU reacts to exceptions by jumping to a specific address. For all exceptions these lie within 0x100 of each other. It is possible to define a offset to the start of the exception space, with a flag in the MSR. Either it starts at 0x100 and continues to 0xFFF or it starts at 0xFFF0_000 and continues to 0xFFF0_0EFF. The second space is right after the configured exception space and is 0x2000 in size, this is implementation specific memory. It could be used for software trap code. The last region is the page table. This can be anywhere in memory, as long as its starting address is a multiple of its size.

3.3.3.1 Block Address Translation

The BAT system gives the possibility to map a contiguous block of effective addresses onto a contiguous block of physical addresses. With memory access and protection bits defined for the whole range. The set up of the BAT system contains 16 registers and no more. These are 8 data and 8 instruction BAT registers. The registers are combined in pairs, a upper and lower (I/D)BAT register. So it is possible to define four regions for data and instruction space at one time. The set up of these registers is straightforward. The smallest block one could map is 128 KB and the largest is 256 MB.

When the BAT system is set up in a way a effective addresses is translated by more than one region it is considered a programming error, which could result in exceptions.

Simple BAT address translation overview,

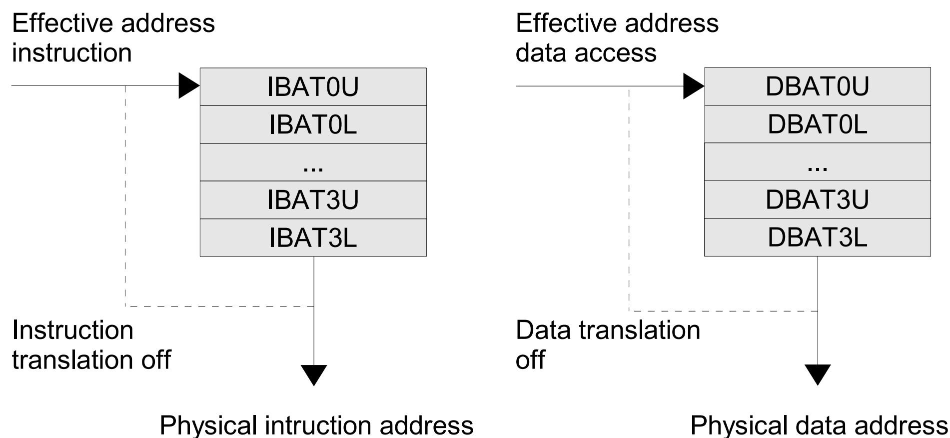


Figure 3.5: *BAT translation.*

3.3.3.2 Page Address Translation

The segment translation system consists of 16 segment registers (SR[0-15]) one base segment register descriptor (SDR1), and the Page Table (PT). The page table must be defined somewhere in physical memory, this is done with the SDR1 register. When a system has “only” 256 MB installed it cannot be on a physical address higher than 0x1000_0000. The size of the page table depends on the installed (or supported) memory. The page table is build up of page table entries (PTE), and these are grouped to form Page Table Entry Groups (PTEG's). A PTE is build up of two words, making a total of 8 bytes. A PTEG is created from 8 PTE's. The architecture defines two types of PTEG's the primary PTEG and secondary PTEG. The updates are made by calculating the addresses of the PPTEG and SPTEG form the segment register and effective address. This gives a maximum of 16 PTE's that have to be searched when a software update to the PT is made or an address translated. To optimize performance the hardware would access (and search) the PTE's in parallel.

Shown in figure 3.6 is the (simplified) page table, primary PTEG, secondary PTEG and PTE layout,

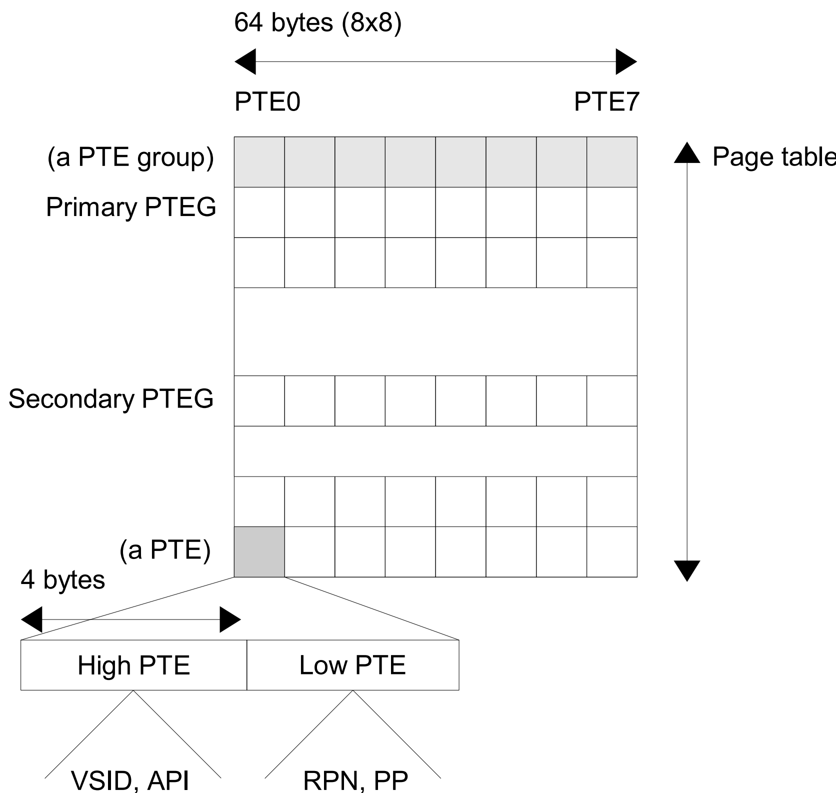


Figure 3.6: *Page table layout.*

Every PTE translates to a 4 KB page in (physical) memory. A PTE contains among others two important fields, the Virtual Segment IDentifier (VSID) and Real Page Number (RPN). The segment register contains the VSID part which must match that in the PTE for a match. The RPN in the low part of the PTE gives the page location in physical memory. The PTE also defines access protection, and memory control settings, so these can be set per page. The protection indicates the level of accesses granted, RO, RW, WO, or NO ACCESS for user or/and supervisor. The memory control fields defines the caching and access type.

Showing the effective to physical address resolution,

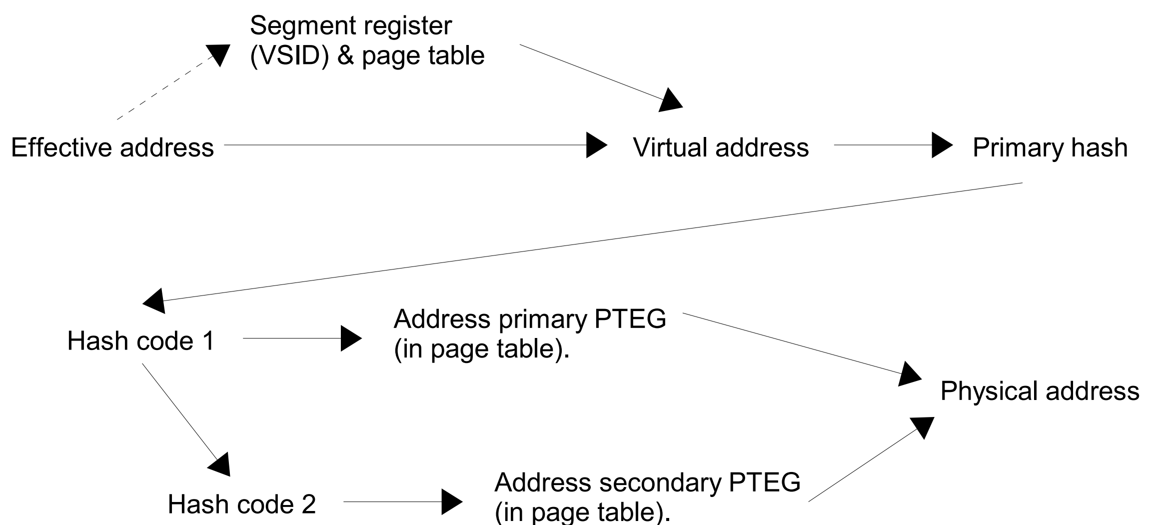


Figure 3.7: *PAT translation, from effective to physical address.*

Looking at the above figure, the translation starts with the effective address. The effective address is split up in to three parts, the segment register selector [0 - 3], abbreviated page index (API) [4 - 19] and the byte offset into the page[20 - 31]. Note that the byte offset never changes. The translation then creates the virtual address with the aid of a segment register. The four bits to select the segment register give 16 possibilities, hence the 16 on-chip segment registers. Every effective address selects a segment register this way. The (maximum) amount of memory one segment translates (or maps) to is 256 MB (4 GB/16). The virtual address length is 52 bits and gives a virtual address space of 4 TB (2^{52}).

The PAT then uses the high part of the virtual address to calculate the address of the PTEG in the PT, with the intermediate values listed in above. To simplify, access to the page table results in a hit if the generated address into the PT delivers a PTEG that contains a PTE with a VSID (and API) that is equal to the VSID in the segment register. For a complete overview of how the PAT system calculates the intermediate values see chapter 7 (page 7-55) “Memory management” in “The Programming Environments for 32-Bit Microprocessors [9].”

To create memory regions bigger than 256 MB, one could map two or more segments continuously. For example, when using segment[3] and [4], 0x3FFF_FFF0 then allocation

0x16 bytes would give a end address 0x4000_0006. Effectively going into the segment above. So the process would have segment register[4] mapped as well. When the OS is going to support programs that use more than 4 GB of memory or more than is installed, virtual memory support is needed. Also all the processes loaded at one time in memory could require more memory than installed. Virtual memory management is supported by the PAT system. The PTE also contains fields that indicate if the page is referenced and if changed. A virtual memory manager would use these to decide when to save a page and when a page could be overwritten.

Every memory access would use the page table to get the PTE to finish translation. Memory access is very slow compared to the speed of the CPU. To speed up memory access Translation Lookaside Buffers (TLB) are used. These are on-chip buffers with recent used address translations. Be sure to invalidate the buffers for the effective address just (re)mapped. Showing the relation between the effective address, segment identifier and physical memory and giving a extra example,

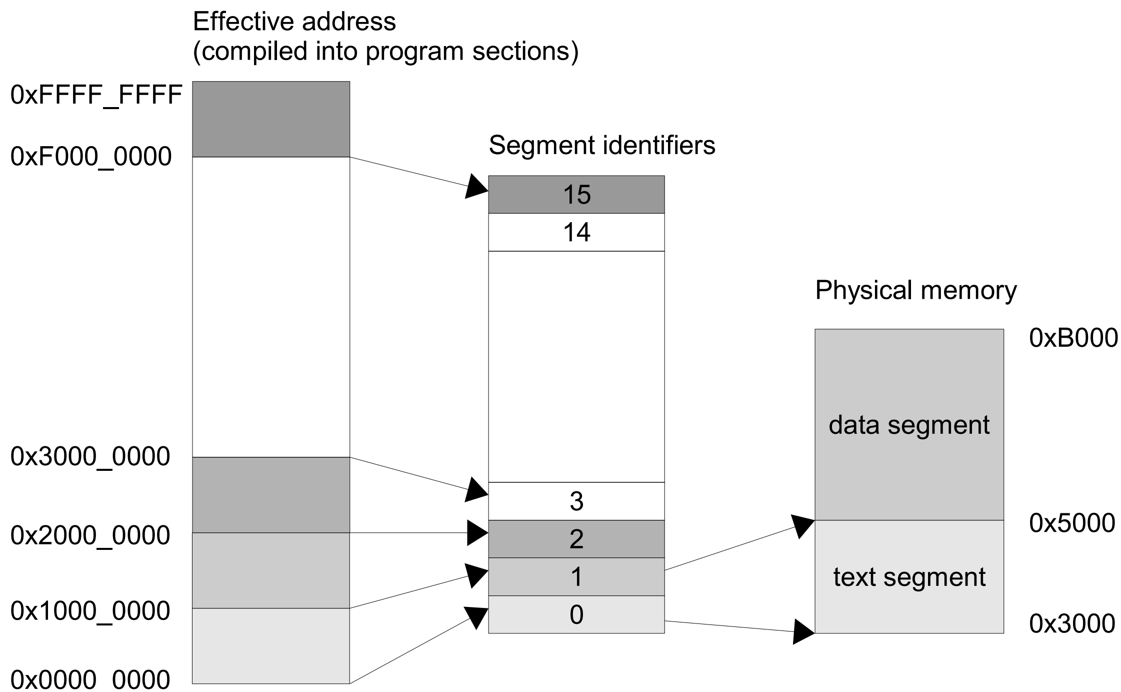


Figure 3.8: *Conceptual view of segment (identifier) to physical memory.*

In figure 3.8 a address used by the program (effective address) like 0x100 would translate to 0x3100 physical, using segment register 0. A address of 0x2001 would give a exception as the mapped segment (0) is only 0x2000 (two pages) big. An effective address of 0x1000_1000 would translate to 0x6000 physical, and 0x1000_6001 would trigger a exception. Note that MinixPPC reserves three segments of which two are actually used, the rest is available for remote segment mapping.

3.3.4 I/O

Communication with external devices is called I/O. These devices are located on the system expansion bus. Systems like IA-32 have special port I/O instructions for writing and reading data to devices on the system bus. As mention earlier in this document the PowerPC doesn't used port I/O; all I/O is memory mapped. External devices are mapped by the Open Firmware software and most of the time located on well known places. The “well known” places could differ form computer model to model though. Most of the time the only thing a device driver needs to know is, what the “base address” is of the device memory map. The device driver itself will usually know how big the mapped memory region is. Unfortunately this eats addresses from the main memory address space.

The (OpenPIC) interrupt controller and several “general purpose” input/output (like the MacIO device) devices in the iBook use memory mapped I/O. The device drivers (processes) “know” where they are in main memory and map there registers as one block into there addressing space using PAT with the right memory protection and access parameters for I/O. The ATA controller uses memory mapped I/O, but on the Intel architecture port mapped I/O. In the Intel architecture the port number would be somewhere between 0 and 2^{16}. For the PowerPC the same number would be interpreted as address but could be anywhere in 4 GB. How problems relating to I/O are fixed for MinixPPC is seen in chapter 5.4.4 “I/O.” The video memory is also mapped as bitmap into the main memory by a (very) simple driver, but good enough for MinixPPC in it's development phase.

The default (black and white) video memory mapping for the iBook,

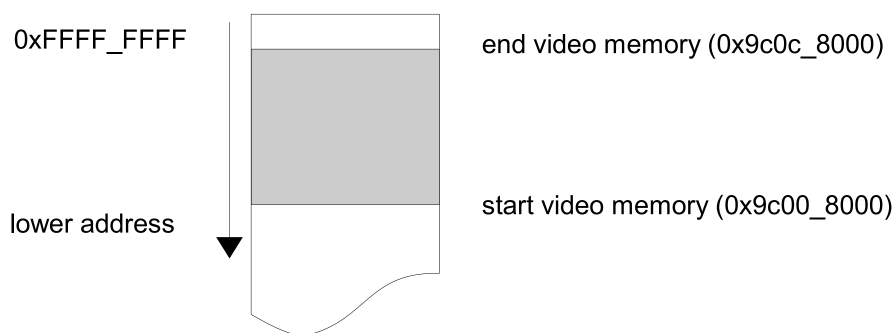


Figure 3.9: *Video memory map for iBook, used as bitmap.*

Writing to the memory anywhere between 0x9c00_8000 and 0x9c0c_8000 would “print” to the (LCD) screen of the iBook. The current screen driver for MinixPPC is very crude and only supports white text. For every pixel on the screen one byte is used, creating a video buffer of 0xc_0000 in size. Via the Open Firmware client interface it is possible to tell the basic video settings, resolution and start address of the video memory.

The usual attributes for mapping a page as I/O are caching inhibited and guarded. The “caching inhibiting” makes sure all writes really go into the (video) memory and reads are not from cache. Guarded makes sure that the memory is written and read in the sequence the program does, and not possibly out of order by the CPU. If we don't map the memory

like this we end up with strange artefacts on the screen and that's not acceptable.

In the next example we see the differences between the x86 and PowerPC assembler functions responsible for I/O. To use these functions it is not needed to map the memory with the default I/O attributes. There are two parts of code listed in listing 3.4. Part 'A' is PowerPC assembler and part 'B' is x86 assembler.

```
1. A) _GLOBAL_F(_outb)
2.       stb      R4, 0(R3) # write the byte to the address in R3
3.       eieio              # make sure of ordering and completeness
4.       sync               # be sure^sure.
5.       blr
6.
7. B) _outb:
8.       push  ebp
9.       mov   ebp, esp
10.      mov   edx, 8(ebp)    ! port
11.      mov   eax, 8+4(ebp)  ! value
12.      outb  dx             ! output 1 byte
13.      pop   ebp
14.      ret
```

Listing 3.3: *Example assembler I/O for the PowerPC (A) and x86 (B).*

In part 'A' of listing 3.4 the “stb” instruction on line 2 is a general store instruction, only the “eieio” instruction on the next line forces the PowerPC CPU to make sure that the store completes before a new store instruction (to the same address) starts. The sync instruction on line 4 makes sure the “eieio” instruction is seen by the CPU before any other instruction.

In part 'B' the general “mov” instruction on line 9 it is the equivalent of “stb.” In this case the “outb” instruction writes the value in register “ax” to the port number located in the “dx” register. The “out(x)” and “in(x)” instructions are protected, the CPU makes sure that before another instruction on the bus is taken the first is completed or not interfered. The “out(x)” and “in(x)” instructions are only used with port I/O, memory mapped I/O is done with the “general” mov instructions.

3.3.5 Interrupts and exceptions

The semantics of an exception are different from those of an interrupt; exceptions occur only when the system has a error, interrupts can occur at any time. Confusion could come from the fact that they are presented to the software in exactly the same way as interrupts, and much literature doesn't make a effort on keeping them apart. In fact in the PowerPC documentation no difference is made, everything is an exception.

Exceptions and interrupts cause the PowerPC CPU to save two “special” registers and for certain exceptions updates status registers as well. For example, saving the effective address that the MMU could not translate and flags indicating what type of instruction caused

the exception. Every exception or interrupt is recognised by the continuing of execution at a specific vector (or address).

The two special registers saved are the MSR from before the exception (or interrupt) and the address of the instruction that would have executed next if no exception (or interrupt) occurred. In other words the return address. It is guaranteed that every instruction before the return address is executed. There is however a catch: sometimes the return address is of an instruction partially executed, this is indicated by a flag so appropriate action can be taken.

3.3.5.1 Interrupts

The PowerPC CPU supports two sources for a external interrupt: the processors decrementer and the interrupt controller. Both external interrupts can be masked by one flag in the MSR. The decrementer can be used to generate a periodic interrupt. This is a 32 bit register that continually counts down. When passing though zero it will trigger a external interrupt, continuing execution at vector 0x900.

The second source for an external interrupt, the (OpenPIC) interrupt controller,

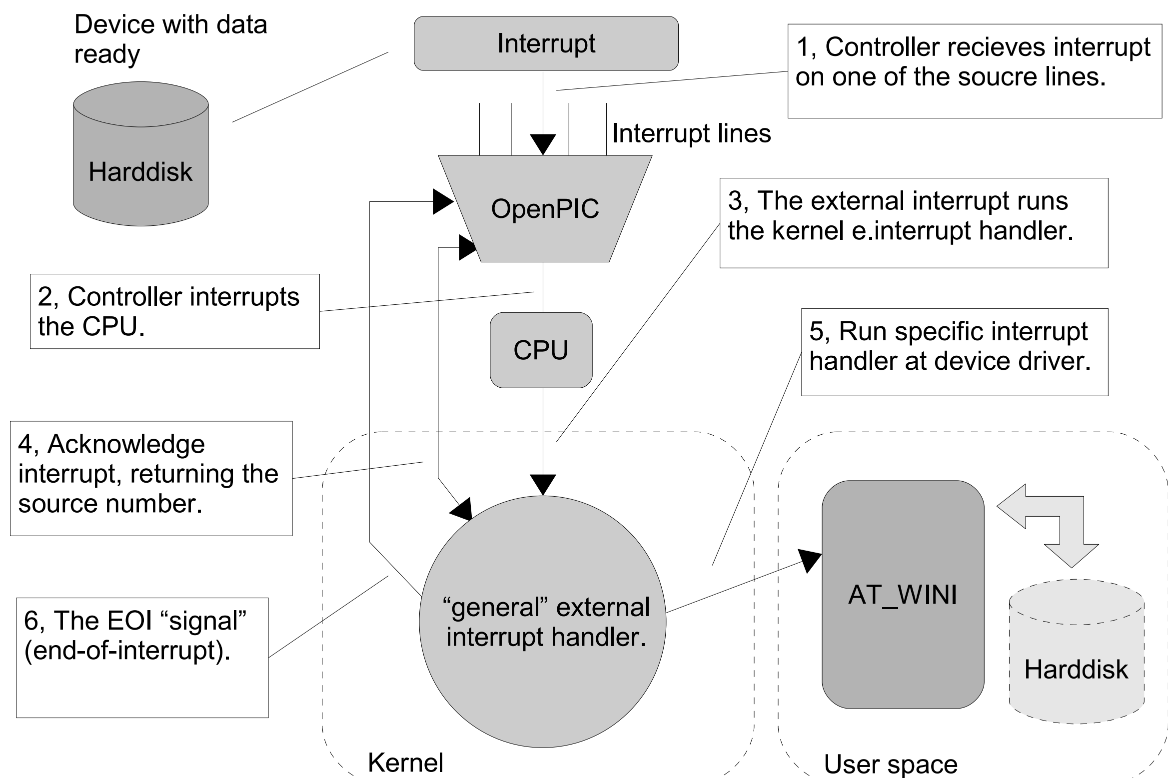


Figure 3.10: *PowerPC, from interrupt source to device driver path.*

The second (and last) source where a external interrupt can originate from is the (multi-processor) OpenPIC compatible interrupt controller. The interrupt controller identifies which device does the IRQ to the CPU. This is a “real” external interrupt and continues execution at 0x500. When it occurs the interrupt handler in the kernel would request (by acknowledge) at the interrupt controller the IRQ number (or vector) and a second specialized handler or service routine (in MINIX located in a dedicated process) registered for that IRQ number would read the device data. When the kernel handler returns “end of interrupt” (EOI) is indicated to the controller and the system is ready for a new IRQ.

The sequence is shown in figure 3.10 and shows a typical interrupt handling using the OpenPIC controller (and most other controllers). The controller supports four timers, 64 interrupt sources (including the timers) and four CPUs. Every interrupt source can be set to a priority and mapped to a CPU. The complete OpenPIC specification (1.2) [5] was available so it was easy to provide a library for the controller.

The way interrupts are handled is straightforward and differs “not” from the Intel architecture. The major difference is the supported numbers and the ability to program the OpenPIC controller to generate a specific vector for a source line. For example (source) line 39 could generate a vector of 1.

3.3.5.2 Exceptions

As there is no “real” difference between interrupts and exceptions, the only way to tell is by the vector. The PowerPC CPU can generate 9 exceptions,

The vectors are without offset,

Exception	*vector*
SYSTEM RESET	0x100
MACHINE CHECK	0x200
DSI (Data Storage Instruction)	0x300
ISI (Instruction Storage Instruction)	0x400
ALIGNMENT	0x600
PROGRAM	0x700
FLOATING POINT UNAVAILABLE	0x800
TRACE	0xD00
FLOATING POINT ASSIST	0xE00

Table 3.2: *PowerPC exception vectors.*

When control continues at the vector (or address), the systems runs in supervisor state with translation off. It runs with a “new” MSR (hence the saving of the old). The previous

MSR is located in special purpose register SRR1 and the SRR0 register contains the return address. These registers are only readable and writeable in the supervisor state (OEA).

3.3.5.3 System call

The system call is more like an interrupt than an exception, as it's nowhere near a "error." Note that a system call is always synchronous as it follows from a instruction and thus is predictable. Hardware interrupts asynchronous, they can occur at any time.

A system call will continue execution at 0xC00. System calls are needed for the user to send a request to the OS, to let print a character to the screen or other device.

In MINIX the system call is used for sending and receiving messages, and it's the only "interrupt" using arguments and having a return value.

3.3.5.4 Exception and interrupt return

Returning from an exception or interrupt is straightforward. Just make sure that Special Purpose Registers (SPR) SRR0 and SRR1 are loaded with the return address and "MSR" you need, then executed a "rfi" (Return From Interrupt) instruction. This would load the "MSR" with the value in SPR SRR1 and starts executing from the address in SPR SRR0, all in one go. This simple construct allows the programmer to change context (set the MSR) and start execution at a new address in a atomic action. By changing the MSR (in the right way) we effectively do a context switch, from supervisor to user (note that the MSR also changes when we go from user to supervisor, with an exception, interrupt or system call). Of course a "complete or real" context switch between processes requires more than a update of SPR SRR0 and SPR SRR1, but that's discussed in chapter 5.5 "Exceptions and context switch."

3.4 Software

A small introduction into the world of "ABI" or Application Binary Interface follows. The ABI should not be confused with the API (Application Programming interface). In terms of levels the ABI lies closer to the hardware then the API. As a result the API does not change when changing platform. The API is the standard chosen by the system developers by which user programs can use the system services. For example providing a set of (library) functions to open, close, read and write files. For MINIX the API is defined by the POSIX standard, that's common for UNIX like environments. On systems supporting the same API, the underlying hardware could be as different as RISC vs CISC but the same program source would compile and run at both.

The ABI defines the relation between a high level language and generated assembler

code. The way the assembler is generated by the compiler defines how the (hardware) CPU registers are used. It is the bridge between software and hardware. In our case C code and generated PowerPC assembler. The definition includes the calling convention and how the stack is used. What a function activation record looks like and which registers are volatile (between function calls) and which should keep their contents.

For MinixPPC the GNU/C compiler is used. The easiest way to see what assembler the compiler generates is by creating a “empty” program (only function bodies) and skip the linking process. It is one way to learn basic assembler and as a bonus you are learning and understanding the ABI. The ABI “used” for the PowerPC is described in [1], chapter 3 “Low-level system information.” This document was printed in 1995, so there can be many special features introduced by the GCC compiler designers till now, but for the port no specialities where needed or found.

Providing a example of generated assembler by the GNU/C compiler. The first file is a C source file familiar for all readers, note that we are only compiling and not linking. The second listing is the generated assembler file by the compiler on the development machine.

```
1. /* empty_file.c */
2.
3. char global[] = "This is a global string";
4.
5. int FA(int a, int b) {
6.   return a + b;
7. }
8.
9. int FB(int b) {
10.  return FA(2, 5) + b;
11.}
```

Listing 3.4: *C source file <empty_file.c>, only two functions FA( ) and FB( ).*

In fact the only purpose of the above source file is semantics, we can read (and understand) C. So we also understand what the assembler does. Note that the compiler reuses a lot of names in the generated assembler listing.

Corresponding assembler code,

```
1.     .file "empty_file.c"
2.     .globl global
3.     .section    ".data"
4.     .align 2
5.     .type global, @object
6.     .size global, 24
7. global:
8.     .string     "This is a global string"
9.     .section    ".text"
10.    .align 2
```

(***Listing 3.5:*** *continued on the next page)*

```
11.   .globl FA
12.   .type FA, @function
13.FA:
14.   stwu 1,-16(1)
15.   add 3,3,4
16.   addi 1,1,16
17.   blr
18.   .size FA, .-FA
19.   .align 2
20.   .globl FB
21.   .type FB, @function
22.FB:
23.   mflr 0
24.   stwu 1,-32(1)
25.   stmw 29,20(1)
26.   li 4,5
27.   mr 29,3
28.   li 3,2
29.   stw 0,36(1)
30.   bl FA
31.   lwz 0,36(1)
32.   add 3,3,29
33.   lmw 29,20(1)
34.   addi 1,1,32
35.   mtlr 0
36.   blr
37.   .size FB, .-FB
38.   .section     .note.GNU-stack,"",@progbits
39.   .ident       "GCC: (GNU) 3.4.6 (Gentoo 3.4.6-r1, ssp-3.4.5-1.0,
40.                 pie-8.7.9)"
```

Listing 3.5: *The assembler from the source code in listing 3.4.*

The assembler code in listing 3.5 is generated with the following command:

"gcc -O2 -fomit-frame-pointer -S empty_file.c -o empty_file.c.s"

The compiler options used influence the generated assembler, they limit the generated code, that's good as we want as little as possible (at least in the beginning). The "-fomit-frame-pointer" option tells the compiler to skip using a "fixed" pointer into a function stackframe (or activation record) to access function local data. Now only the stack pointer is used for that. The option '-O2' optimizes to level two, removing even more assembler instructions.

We start by looking at the simplest function in listing 3.4, FA( ) spanning from line 13 to line 21. Some of the lines can be ignored such as lines 18,19,20 and 21. These are needed by the assembler and don't tell about the workings of the ABI. Clear to see in FA( ) is the use of GPR1 at lines 14 and 16. There "some" value is decreased and increased again, looks a lot like stack handling. Also the instruction at line 17 is "special", peeking at function FB( ), line 36 we see it's there too. Looks like a function return (O and it's at the end of the function too). This leaves only one instruction left on line 15, doing the actual work of FA( ).

When we look at FB( ) we see the call to function FA( ) on line 30. This instruction is

used to do a function call, a branch. Looking at the documentation we see that this function saves the address of the next instruction into the CPU's link register just before the branch, so this is the “return” address. Looking back at the function end we see, “blr.” This function uses the link register to branch too, making the circle complete.

Just before it we see how the parameters are prepared. We know the stack pointer is at GPR1 and we see it's not used. We see that GPR3 and GPR4 are loaded with 2 and 5, looking at the C code these are our parameters. To be sure one could alter the values in the C and regenerate the assembler listing and check again.

This is a powerful method to learn the target assembler and understand the ABI at the same time, it has been used a lot while developing MinixPPC. It should not be underestimated especially when you are new to the target architecture.

From the software point of view, the biggest difference in architecture between the PowerPC and Intel x86 is in the stack. The Intel x86 architecture is build around the usage of the stack. It has instructions to “push” and “pop” values (register contents) to and from the stack. The CPU has a dedicated stack pointer register, that is decremented and incremented when using the stack instructions. The PowerPC CPU has not. All notion of “the stack” is defined by the ABI. The dedicated stack pointer register could be any of the 32 general purpose registers, the PowerPC ABI takes register one. There are no special instructions pushing or popping registers from the stack. Stack access is just like any other memory access. In other words the CPU itself is in theory free of the stack concept. A good second is the way arguments are passed between functions or modules of code. For few arguments the ABI uses registers GPR3 to PGR10 and FPR1 to FPR8. The location of the argument is determined by the data type. If more arguments need to be passed a fall back to the stack frame is made. In the same way all arguments are passed in the x86 architecture. The place of the return value is also determined by the data type an could be in GPR3 inclusive GPR4 if needed or in FPR1.

For systems to be compatible on a ABI level, it would mean that a executable compiled and on system A would run on system B without recompiling and OS(A) != OS(B). Also pre-compiled libraries could be mixed. But most of the time it is just backwards compatibility between versions of one OS.

You might think; “isn't this extra work, why should I learn the ABI for the target architecture, I am not planning to use assembler code.” Then you are right, but it is impossible to write a operating system without using handwritten assembler code. It is at least needed in the context switching code. Then the other extreme of thought, “I am not going to use C code.” Then you are right again, and I wish you good luck.

So it's inevitable that C code and handwritten assembler code are used together developing for a operating system so one must know what the compiler generates. We will see later in the context switching code and signal handling why these concepts must be known. In the end to successfully port a operating system, knowledge and understanding of the target assembler code and used ABI is crucial.

One of the personal questions of the author was; how “people” develop kernels, what “machinery” is needed or how does there lab looks like. The next chapter, 4 “Development environment” tries to answer that question. Looking back its quit “weird” realizing that everything was “at home” except the iBook. And only requiring information, and even more information.

4

Development environment

4.1 Why

A good environment can speed up development considerably. This small chapter will give an overview of the environment MinixPPC was developed in.

The first thing any developer wants is a way to get code to be executed on the CPU with no restrictions (supervisor). To know that his or her code is running to some degree you need results back from the system. This can be somewhat of a problem as you might need to write code to get the screen to work first. Some (development) systems have serial ports that can be used to hook up a terminal, upload code and run it. Then there would only be the problem of getting output. Remembering a small project some time ago, something as “simple” as getting a LED switching on and off, by a timer could be very rewarding. Now you know your code is executing and then the real programming can begin.

When you are lucky (like me) there are Open Source running systems, so there is code that works which can be viewed and used as bootstrap to write you own. It's all about keeping the space where “problems” can occur to a minimum.

For example, when first trying to load the development kernel, the bootloader for the Linux PowerPC kernel can be used. You know that when there is a problem it can't be the bootloader as it is able to load the Linux kernel. So it has to be in your code. When your code is finally producing output it is time to write you own bootloader. The next problem could not be in your development code as the first boot loader was able to load it. This iterative process with small updates every time should get you to the final solution in the end.

4.2 Two computer setup

For development, two computers were used. I shiver at the thought of using only the iBook as development system. There were times the system was reset more than 30 times a day (note, after 30 mounts, (per default) the ext2 FS wants to do a file system check, I soon switched to ReiserFS). You always need to wait for the system to boot up again, but time can be spared. My own PC has the GNU/Linux OS installed that comes with NFS client and server support “out-of-the-box.” The iBook also has the GNU/Linux OS installed and has NFS client and server support enabled. By mounting the MinixPPC tree as exported directory from the iBook on my own PC I could access the source files with my favourite text editor. To compile source code a remote shell is used.

Now the trick comes in, when using NFS you can reboot the iBook without having to

“close” the files in the text editor, they stay sort of “open.” So after editing a source file, compiling, testing and rebooting the text editor would get back to life and continues as if nothing has happened, as soon as the network is brought up again. When you try to edit the file when the network is (still) gone the text editor would block on file access. So when using NFS saves reopening the file as well.

Long live NFS, it saved loads of time. Naturally one could use a console text editor like VI, nano or emacs but most of the time console fonts are “big” and ugly. I would like to see more than ~40 lines of code on my screen and in many many colours, using a window manager. The text editor used is “nedit” (Nirvana Editor) and NFSv3 on kernel 2.6.

The next chapter, 5 “MinixPPC,” will focus on the implementation of the MinixPPC kernel, from how it's booted to which (new) drivers are developed for it. It also covers the necessary utilities needed to compile and “use” MinxPPC in it's immature state.

5

MinixPPC

Here we get to the implementation of MinixPPC: from power-on to shut-down. Starting with the new (PowerPC) libraries then the boot monitor, kernel organization, memory management, exceptions and interrupts, signals, new drivers and as last utilities needed to convert and install MinixPPC programs.

The location of project files is determined by the project root directory <*minix/*>. If not mentioned otherwise all mentioned directories are below the project root. Not all directories should stay in the final version of MinixPPC (or in general). The <*./fs.img/*> directory should be removed when it is possible to get updates to runtime files besides using the utilities. The same for the library directories. Keep in mind this is the development tree used on the host OS.

A complete list of the project tree, and a short description of the content in every directory,

```
./minix
  |-- arch              architectural includes
  |-- commands          programs and utilities for MINIX OS
  |-- drivers           MINIX general and architecture drivers
  |-- fs.img            MINIX v3 file system image
  |-- image             system image (kernel + servers + drivers)
  |-- include           standard system include directory
  |-- kernel            the MINIX kernel
  |-- lds               linker scripts
  |-- lib               system libraries + PPC library
  |-- misc              test programs and files
  |-- servers           MINIX servers
  |-- stdlib            MINIX standard library archives
  |-- syslib            MINIX system library archives
  `-- util              programs to help creating the MINIX system
```

Figure 5.1: *MinixPPC development source tree.*

The the directories <*./kernel/*> and <*./lib/*> contain a hierarchy of there own and will be handled later in this chapter. The <*./drivers/*> directory contains system independent drivers in the root and system independent drivers below the <*./drivers/arch/xxx/*> directory. Some system dependent drivers are the “macio” and “tty” drivers.

It is possible to split (most) the of the drivers in to two parts, MDC and MIC. Some are (re)written with this in mind. Especially the tty driver has been rewritten like this it has a large bulk of MIC and only needs a screen and keyboard (sub)driver to do its task. For MinixPPC this is already the case for the keyboard, the MacIO driver.

The “macio” driver is an architecture only driver, it has no use on any architecture other than PowerPC. It primary task in MinixPPC is to get scancodes from the keystrokes via the ADB (and everything in between). At the moment drivers like the PCI manager are still PowerPC only but in the future it could become a default way to get information from PCI devices in the system.

The <*./arch/*> directory contains include files per architecture. It is a extension of the system include directory to support redefinition of types. Building for the PowerPC architecture would use the <*./arch/ppc/*> directory to redefine types.

Run time programs and utilities are located in the <*./commands/*> directory. Most of the programs are located in the path they have in the MINIX file system. The more advanced or bigger programs (like the shell) have a separate directory.

At the moment the MinixPPC file system (v3) is created from a tree in the host file system (Linux, Reiserfs) to a image file. The root of the host tree is <*./minix/*> and contains all files needed for MinixPPC. The directory <*./fs.img/*> contains a few simple scripts to create and install (as root) the MINIX file system to the system. More about how to create and use the MINIX file system for MinixPPC in chapter 5.9.3 “mkffs.”

5.1 Libraries

Besides the “general” system libraries that are needed to provide the POSIX standard, there is also a PowerPC-only library created. It includes functions that could be used a various places in one program or in multiple programs. Most of the functions can only be used by the supervisor, these include the functions for the MMU, OpenPIC device and MSR. Other libraries are, int64, libppc, I/O, RTS (Run Time System) and string. These libraries needed to be ported because they contain MDC. For the MMU and OpenPIC this is clear but for some it's not. The other libraries are machine dependent because they are written in assembler code. String must be rewritten in C code, writing it in assembler was good for practice. The int64 library sort of stands out. The PowerPC version does not use assembler code, only C code. For some architectures it could be difficult to handle 64 bit types and handwritten assembler functions are needed.

The systems libc library, that contains (most of) the API, could be compiled almost right away. The compiler gave warnings about unused identifiers and incompatible pointer assignments. At the moment the biggest bulk of the MINIX library tree compiles without warning. A few modifications have been done to get variable argumentation to work, used by the “printf” type functions. All these changes to the library code have to be tested by compiling it with the default MINIX compiler the ACK (Amsterdam Compiler Kit) for the IA-32 architecture. Appendix D “Library notes” tells shows to where the library parts (directories) are archived too.

On the next page the tree for the PowerPC library, not seen here are the files <*libppc.S*> and <*delay.c*>. The first file provides functions for forced memory writing through the cache, and copying memory from different pointer context (effective to physical). The last functions do this by reading a value for the source address into a register, then switching the MMU state, and then using the destination address to write (with the new MMU state) the content of the register to memory. Note that not all functions are used by MinixPPC,

but programmed as “extra.” The second file provides high resolution delaying with the on-chip timebase. Any user program should be able to use it as it only read from the registers and doesn't write them (which is a supervisor only instruction), note that the delaying can be interrupted if the process is unscheduled. The tree for the PowerPC library,

```
./minix/lib/ppc
  |-- bt          contains a very simple screen driver
  |-- cpu         access CPU registers (MSR etc)
  |-- int64       to handle and convert 64bit types
  |-- io          simple byte/half/word IO routines, like outb()
  |-- mmu         setting and updating BAT or PAT
  |-- of          Open Firmware front-end functions
  |-- rts         send/receive primitives as well as long jump code
  |-- string      memcpy(), memset() the default C string routines
  `-- sys         has the OpenPIC library
```

Figure 5.2: *PowerPC library tree.*

For ease of use all these objects are archived together to form a single library archive, <*libppc.a*>, that is placed in the <*./stdlib/*> directory at the project root.

Every program compiled for MinixPPC needs a small bootstrap object linked before all program objects. The PowerPC library contains the <*crtso.o*> object for that purpose. It is located in the <*rts/*> directory. Looking is this directory one will find more runtime code include the IPC primitives, long jump code and dynamic memory support code.

5.2 Boot monitor

The purpose of the MINIX monitor is to load the processes in the system image into memory and create a environment for the kernel to start with. The environment includes information about the video mode, type of processor etc. Also information about the loaded processes are passed in a array of a.out executable headers. The kernel get arguments that tells were the kernel environment is located and where the process headers start in (physical) memory. The boot loader is system dependent and located in the <*./drivers/arch//ppc/monitor/*> directory. The monitor is only usable on a PowerPC with at least the Open Firmware (v3) bootROM.

When the iBook boots it will stop at the Open Firmware prompt. The next command could be anything from booting from a CD to booting from a network, “boot” is good enough for us (if no one has been smart). With the default action it will load a Forth script file and that will execute the monitor program. To this point there is nothing that can be done by “us”, we can alter the script file, but before that all is defined by the software of the bootROM and that is designed by Apple to work with Mac OS X.

The current monitor contains two boot paths, to load and execute a Linux kernel and to load and execute a MinixPPC image. The two paths are needed as the GNU/Linux OS is still used to compile every program for MinixPPC. Of course the last path is of our interest.

The first thing done is loading the MinixPPC image from disk to memory. The MinixPPC image is (still) located on a ReiserFS. The monitor is derived form the “Yaboot loader[13].” It uses a ReiserFS interface to read the Linux kernel from disk, we also use it to read the MinixPPC image into memory. In the future, when it is possible to compile MinixPPC in MinixPPC, ReiserFS should be replaced by MINIX v3 FS. It can be done at the moment but we still need the GNU/Linux environment and needed to save time.

The monitor is compiled a lot like other MinixPPC programs, but stays in Elf32 executable format, our Open Firmware doesn't support the a.out executable format. The monitor is compiled to a 10 MB address, Open Firmware uses “virtual = physical” memory mapping so the monitor program is loaded at 10 MB. Why this is important we will see below.

The monitor is the only software that uses the Open Firmware client interface. To get input from the user, use the screen and read the disk image. It “should” also be used to gather information from the device tree into the kernel environment. Device drivers would then use the kernel environment to read the properties of the device it's written for. This way the kernel and drivers them self stay clear of interfacing with Open Firmware. Even if programmed only in the MDC there could be compatibility problems with machine models. At the moment the device drivers are programmed with default values.

When the monitor gets more mature, valid environment strings will be passed to the kernel, and drivers should be updated to use them. Also by design it is not possible for the kernel to use the client interface. The exception vectors used by Open Firmware are overwritten as soon the kernel process is loaded into place, as we will see in the next chapters.

5.2.1 Image format

All parts (processes) of the system image are stripped of their a.out header and prefixed with a new header that has the name of the process and the striped a.out header.

Note that in the next figure the “struct exec” defines the build up of the a.out executable header.

```
1. /* A process header.
2.  *
3.  * There is but one image, build of multiple
4.  * processes (the kernel is process[0]).
5.  */
6. typedef struct process_header_s {
7.   char name[IM_NAME_MAX +1];  /* Null terminated .*/
8.   struct exec process;        /* a.out header */
9. } process_hdr_t;
```

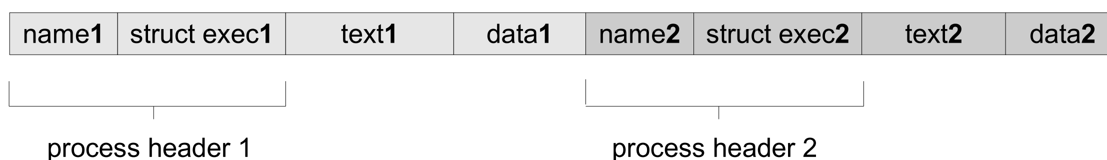


Figure 5.3: *Process headers in image of two processes.*

The creation of the system image for MinixPPC is done with a utility program called “mkimage,” it's source is located at <*./util/mkimage*>. The program takes a image destination name and takes Elf32 or a.out programs as input. The steps done by the program are simple. For every program create a new process header, copy the name (given on the command prompt) and copy (or create) the a.out header to it. Then write the process header, text and data sections to the image file. More details about the “mkimage” program in chapter 5.9 “Utilities.”

5.2.2 Loading and executing the kernel

The MINIX boot path in the monitor has two phases. Loading the image into memory “somewhere high”, out of the way and copy the kernel and the processes inside the image to there “first” location. Preparing text, data and bss.

The physical memory layout at the end of phase one, in clicks.

Figure 5.4: *Physical memory layout at end of phase one.*

From left to right, the first (small) block is the exception vector space, the second block the monitor program then the system image and as last the location of the a.out header array (but still empty as the image isn't scan yet). The a.out header array is built by the monitor while scanning the system image for processes to load. They are copies of the a.out headers in the process headers. After the end of phase one, the monitor presents a overview of where the processes are going to be loaded in memory. Once the loading starts there is no turning back as the Open Firmware exception vectors are going to be overwritten.

After system image scanning and copying the memory layout looks as this.

Figure 5.5: *The first MB's of the physical memory.*

With the copying and setting of memory regions special memory functions are used from the <*libppc.a*> library. These are the “memfcpy(w)( )” and “memfset(w)( )” functions. These copy and set through the caching of the CPU, to make sure the data is written to the system main memory. Making sure that on later access no stall values are read.

The first physical memory is overwritten with the kernel process and every other process in the system image. Every process in the system image is striped from the process header and it's text and data sections are copied to there place. Except for the kernel process, the 'bss' + 'stack' space is added and the base address for the next process is aligned to the next click.

As soon as the copying is done, the monitor sets the kernel segment registers. These are always the same, and calculated like this,

```
1. SR[0] = 0x0;                     /* kernel text */
2. SR[n] = ( SR[n - 1] + 0x10 );   /* every next segment ID */
```

Listing 5.1: *Kernel segment register ID calculation.*

This makes it possible to “calculate” the segment registers for the kernel on a context switch, we don't need to save them to restore. The monitor maps a memory block using data BAT registers 0. It is used by the kernel for physical addressing. The size of the block can be set in the monitor file, and defaults to 64MB. The kernel text is mapped on instruction BAT registers 0 and is at first 32MB (to include monitor instruction fetch) but reset by the kernel itself to 256KB. The kernel data is mapped to virtual address 0x1000_0000 using PAT. How the virtual addressing effects the way we need to compile programs is seen in chapter 5.4 “Memory management.”

To start MinixPPC, the only thing left to do is to jump to the entry point of the kernel process, it's at the start of the assembler file <*./kernel/arch/ppc/minix.S*>.

This is what the entry point looks like as an (ordinary) C function,

```
1. /* The MinixPPC entry.
2.  */
3. typedef int (*minixppc_t)(segdesc_t    cs,     /* text SR */
4.                           segdesc_t    ds,     /* data SR */
5.                           char*        params, /* start params */
6.                           size_t       len,    /* size params */
7.                           struct exec* aout,   /* a.out hdrs */
8.                           u32_t        sd);    /* start data */
9.
10....
11.
12.minixppc_t entry = \
13.              (minixppc_t)(process[IMAGE_PROCN_KERNEL].entry);
```

Listing 5.2: *The MinixPPC kernel entry definition and entry point assignment.*

The monitor defines a variable of type 'minixppc_t' (on line 12) and sets it to the entry

point of the kernel process. For MinixPPC this is acutely always zero, but it could be any value (or reference). To jump to the kernel a call must be made to entry( ). The parameters needed for the kernel entry point are free to program.

But the MinixPPC entry needs these,

```
1. entry(mfsr(T), mfsr(D), params1, sizeof(params1), aout, \
2.       process[IMAGE_PROCN_KERNEL].data.addr);
```

Listing 5.3: *Entering the kernel just like any other function call.*

Among other things the MinixPPC kernel wishes to know where the a.out headers are located in physical memory, which is the fifth parameter. These are needed for the kernel to make the very first memory map. Recall that this lists the map of the processes (already) loaded in memory by the boot monitor.

The “params1” and “sizeof(params1)” are used to transfer the kernel environment from the monitor to kernel space. The mfsr(T) and mfsr(D) parameters are used to calculate the physical addresses of certain variables, in future releases an effort should be made to let the kernel do without these. The last parameter indicates the (physical) load address of the kernel data segment. It's used in fast calculation of a virtual to physical address.

5.3 Kernel organisation

The first process ported was the kernel. The original kernel was located in a “flat kernel tree” that didn't have architectural subdirectories. The kernel code was a bit organised to be used on multiple architectures by separate header files containing only IBM PC compatible definitions. Compiler directives where used to include blocks of code or files. Although other (older) architectures are supported the original MINIX kernel is targeted for the IBM PC compatible.

For this project some alterations have been made to the default MINIX source tree. Some changes could be permanent while others are not, but for the MinixPPC project these where logical. The new kernel tree is expanded and divisible in two parts, the root is the first part and has all MIC and the second part is where <*./kernel/arch/xxx/*> starts and has all the MDC. The MDC for the PowerPC architecture is located in <*./kernel/arch/ppc/*>, directory, typing “make” here would create the <*./kernel/arch/ppc/arch.a*> library. This is needed when linking the MinixPPC kernel. Typing “make” at <*./kernel/*> would create the MIC part object(s) and would link directly with the <*arch.a*> file to form the kernel process.

The organisation of the kernel and the place where the source files are located in the kernel tree are related. System dependent header files are located in <*./arch/ppc/*>, files in this directory have names like <*atypes.h*>. These files are included when the kernel gets compiled to (re)define “special” types needed by the MinixPPC kernel. The reason they can't be below the kernel tree is that other processes use the header files as well.

Next is the overview of the old and new kernel trees. The root of both trees include only

the source and header files which belong to kernel MIC. Directories are printed **boldface** and the x86 root is empty as the architecture isn't back ported yet. Note that the directories <*./kernel/system/*> and <**/debug/*> don't have there content listed but aren't empty.

```
./kernel
|-- system
|-- Makefile
|-- clock.c
|-- config.h
|-- const.h
|-- debug.c
|-- debug.h
|-- exception.c
|-- glo.h
|-- i8259.c
|-- ipc.h
|-- kernel.h
|-- klib.s
|-- klib386.s
|-- klib88.s
|-- main.c
|-- mpx.s
|-- mpx386.s
|-- mpx88.s
|-- priv.h
|-- proc.c
|-- proc.h
|-- protect.c
|-- protect.h
|-- proto.h
|-- sconst.h
|-- start.c
|-- system.c
|-- system.h
|-- table.c
|-- type.h
`-- utility.c
```

```
./kernel
|-- arch
|   |-- ppc
|   |   |-- debug
|   |   |-- Makefile
|   |   |-- clock.c
|   |   |-- exception.c
|   |   |-- interrupt.c
|   |   |-- klibppc.c
|   |   |-- memory.c
|   |   |-- minix.S
|   |   |-- system.c
|   |   `-- table.c
|   `-- x86
|-- debug
|-- system
|-- Makefile
|-- clock.c
|-- config.h
|-- const.h
|-- debug.c
|-- debug.h
|-- glo.h
|-- interface.h
|-- interrupt.c
|-- ipc.h
|-- kernel.h
|-- klib.c
|-- main.c
|-- priv.h
|-- proc.c
|-- proc.h
|-- proto.h
|-- sconst.h
|-- sendmask.h
|-- start.c
|-- system.c
|-- system.h
|-- type.h
`-- utility.c
```

Figure 5.6: *Old (left) and new(right) kernel tree organization.*

As listing 5.6 shows the current tree supports only two architectures, 'ppc' for PowerPC and 'x86' for Intel. All future ports should get a place there too.

To port the existing kernel code first all machine dependencies had to be found. This is done by using the method described earlier (chapter 2.2.1 “Creating portable code from ex-

isting files”). To provide a indication of the compile time dependencies found for kernel files these listed in appendix E.1 “Missing symbols.” Note that not all architectural code is found by listing the missing symbols when trying to compile. The table is only listing the “most” interesting files (right column), assembler files are per default architecture dependent. Symbols and definitions not removed but known to be architectural dependent are printed in bold.

Changes are not limited to the files listed in appendix E.1, existing header files were changed and new ones created. Most changes were the result of symbols and definitions that were moved or function prototypes that weren't needed any more or out of place. The changes are trivial, too numerous and small to be listed all. Also certain changes have nothing to do with system dependability they where done to add or change functionality. Looking at the first file, <*clock.c*>, the two missing functions and three undeclared constants result from the removal of the header file <*./include/ibm/prorts.h*> and I/O functions (that are known to be architectural dependent). This is not the only MDC used in <*clock.c*>. The bold lines are definitions that the compiler accepts, they are just definitions, but the programmer knows that they are out-of-place as these definitions are used in the initialization of and communication to the timer hardware. They have nothing to do with the clock management so the lines must be moved to the MDC file (for the x86 architecture in this case). The file <*clock.c*> is split in two files, <*./kernel/clock.c*> and <*./kernel/arch/ppc/clock.c*>, keeping the same name links there purpose!

The <*i8259.c*> file was named after the interrupt controller in the IBM PC compatible. It not only contains code for initializing and communicating with the interrupt controller, it also contains code for setting and using the MINIX interrupt hook system (interrupt management). The split was easy as almost all hardware code is located in the initialization function. The file is renamed to <*interrupt.c*> keeping only the hook system and placing all device code into the architecture file. The functions “enable_irq( )” and “disable_irq( )” where originally located in the <*./kernel/klib386.s*> file, the functions are now located in the <*./kernel/interrupt.c*> file.

Some system dependencies are needed to support (external) programs. Programs like the monitor are implemented for the PowerPC and IBM PC architecture, but only their names are alike. The programs function very differently, as defined by the system architecture. MINIX can exit to the monitor on the IBM PC compatible, but not on the PowerPC. It's simply not supported there. Therefore the <*start.c*> file is cleared of monitor support. As seen in appendix E, at the file <*system.c*> there is a similar problem with the system architecture BIOS access.

Isolation of code touches one of the problems of the previous kernel layout. Not all device driver code was isolated as well. The next code is from the original kernel source shows two lines that write to the same port but they are separated over two files with very different context,

```
1. /*line1, i8259.c */
2. outb(INT_CTLMASK, mine ? IRQ0_VECTOR : BIOS_IRQ0_VEC);
3.
4. /*line2, main.c */
5. outb(INT_CTLMASK, ~0);
```

Listing 5.4: *Setting the i8259 controller interrupt mask register.*

The <*i8259.c*> file contains code to write directly to the port(s) of the interrupt controller and so does the <*main.c*> file. Luckily this was limited to the two files for the in-kernel clock and interrupt driver code. But this is bad for portability, references and communication to a device need to be found easily and be grouped together. These problems are solved with the driver model as we will see in the next chapter. This “overview” shows what needs to be ported, only the device direct communication, the driver. If we ignore the memory and other system special code, most of the original dependencies for the kernel come from the clock driver and interrupt driver. Recall we “see” the CPU as device too and are going to writing a driver for it. The isolation of driver code needs to be done for every architecture, for the PowerPC this is towards the <*./kernel/arch/ppc/*> directory.

At the moment the directory for the PowerPC architecture contains the files listed in appendix E.2 “PPC architecture files.” The files containing interface implementations are discussed in the next chapter. Taking the PowerPC file list as a example appendix E.2 gives a clear indication of what needs to be ported of the kernel process to a new target, every file.

5.3.1 Kernel driver model

We want to split the kernel in MIC and MDC. As mentioned earlier we do this by defining all the (special) system parts as devices. Then building device drivers for every device, using the driver model, creates the MIC/MDC separation we need. Creating an in-kernel device driver is usual for the interrupt controller. Most of the time this includes the clock hardware as well. The clock driver is used and set up by the clock task, so in fact it isn't a in-kernel driver but the kernel does use a direct call to the clock driver to stop the clock when shutting down.

To view the CPU as a “device” is new. For MINIX the CPU device is logically split in two devices and driven by two drivers, a memory and system driver. The current (MinixPPC) kernel uses four interfaces in total to access the systems hardware,

1. System interface
2. Memory interface
3. Interrupt interface and
4. the Clock interface

Listing 5.5: *Kernel interfaces.*

The interfaces listed in listing 5.5, are defined in the <*./kernel/interface.h*> file. One of the problems faced porting the system was searching for all system dependent functions. Making a list of them and determining what goes in and what comes out. Now (simply said) the <*interface.h*> file contains the list of functions needed to port for the new architecture. The functionality needed by the interface(s) are such that every architecture should be able to provide. There is always the possibility that a new system requires a function not listed or maybe an existing function definition needs to be updated. In the end porting to more architectures should help making the interfaces more robust.

Showing the definition of the clock interface in the file *<./kernel/interface.h>*. The complete file is included at appendix I.1 “Kernel interfaces.”

```
1. /* The clock functions MinixPPC needs (for the clock task).
2.  */
3. typedef struct if_clock_s {
4.   const char* info;
5.   int          (*init)(void);
6.   void         (*stop)(void);
7.   void         (*start)(void);
8.   u32_t        (*frequency)(void);  /* hardware clock (in Hz!) */
9.   clock_t      (*read)(void);
10.} if_clock_t;
```

Listing 5.6: *The C type definition of the clock interface.*

This type definition needs to be known by both the kernel MIC and MDC. The MIC will use the functions listed in the clock interface to initialize and manage the clock hardware. The MDC will only make it “happen.” Every function (name) listed in one of the interfaces uses system dependent code, like port numbers, register addresses or a driver that's only present on the target architecture. Some references are the start of a very small assembler function, like the I/O functions.

Identifying system dependent code or calls throughout the kernel code is made easy. The *<./kernel/glo.h>* file contains the following declarations,

```
1. /* We need to use the interface for accessing the machine
2.  * parts.
3.  */
4. extern const if_system_t    System;
5. extern const if_memory_t    Memory;
6. extern const if_clock_t     Clock;
7. extern const if_interrupt_t Interrupt;
```

Listing 5.7: *Kernel hardware interface access points.*

These are the references to the data structures containing the list of functions listed in listing 5.8. A call to disable external interrupts at the CPU goes via the system data structure, 'System.'. A call to allocate the memory map inside a process structure is done via the memory data structure 'Memory.'

Recalling the driver model, these two calls look like this,

```
System.lock();
Memory.alloc_segments(process_ptr);
```

Figure 5.7: *Disabling external interrupts and allocating a memory map.*

Figure 5.8 shows the full list of functions needed by the MinixPPC kernel. Every architecture should be able to provide these functionalities.

The "needs" of the (project) kernel from the hardware,

Kernel

Architecture dependent

System
info
(*init)
(*syscall)
(*lock)
(*unlock)
(*locked)
(*shutdown)
(*reset)
(*idle)
(*sputc)
(*read_tsc)

Memory
info
(*init)
(*alloc_segments)
(*alloc_remote_segment)
(*copy_message)
(*phys_copy)
(*phys_memset)
(*seg2phys)

/* I/O */
(*outb)
(*outw
(*outl)
(*inb)
(*inw)
(*inl)

(*be_outw)
(*be_outl)
(*be_inw)
(*be_inl)
(*le_outw)
(*le_outl)
(*le_inw)
(*le_inl)

/* I/O streams. */
(*insb)
(*insw)
(*insl)
(*outsb)
(*outsw)
(*outsl)

Interrupt
info
(*init)
(*get_vector)
(*set_vector)
(*ack_vector)
(*enable)
(*disable)

Clock
info
(*init)
(*stop)
(*start)
(*frequency)
(*read)

Figure 5.8: *Kernel hardware interface (MinixPPC).*

The impact of the driver model on the original kernel MIC is small. It “only” requires a renaming of the original function. The device I/O system call uses port I/O functions.

Looking at the <*./kernel/system/do_devio.c*> file this gives the following lines,

```
1. /* A */
2. case DIO_BYTE: m_ptr->DIO_VALUE = inb(m_ptr->DIO_PORT);
3. case DIO_WORD: m_ptr->DIO_VALUE = inw(m_ptr->DIO_PORT);
4. case DIO_LONG: m_ptr->DIO_VALUE = inl(m_ptr->DIO_PORT);
5.
6. /* B */
7. case DIO_BYTE: m_ptr->DIO_VALUE = Memory.inb(m_ptr->DIO_PORT);
8. case DIO_WORD: m_ptr->DIO_VALUE = Memory.inw(m_ptr->DIO_PORT);
9. case DIO_LONG: m_ptr->DIO_VALUE = Memory.inl(m_ptr->DIO_PORT);
```

Listing 5.8: *Original lines in part A and replacement lines in part B.*

Looking at listing 5.8, the changes needed for code part A (lines 2, 3 and 4) are minimal, and result in part B (lines 7, 8 and 9).

As seen on line 2 and 7, 'inb' becomes 'Memory.inb'. The prefix 'Memory.' is clearly a sign that the 'inb' function is using architecture dependent code. In this case the 'inb' function is the reference to the assembler function for the PowerPC using special I/O instructions. To see how the interface data structures are used by the linker look at appendix H “Kernel symbol listing.”

Additional code is been added to the MIC of the kernel as well. Sometimes it replaced early initialization functions like “prot_init( )” (to initialize memory protection), but otherwise it was needed for the new programming style.

The system initialization at the <*./kernel/cstart.c*> file (MIC) in MinixPPC,

```
1.   /* Setup minix memory space by setting up the hardware so
2.    * memory can be mapped and used.
3.    */
4.   Memory.init();
5.
6.   /* Initialize the hardware of the (external) interrupt system.
7.    * This function returns with external interrupts disabled at
8.    * CPU and at all sources lines.
9.    *
10.   */
11.  Interrupt.init();
12.
13.  /* Initialize the system interface.
14.   */
15.  System.init();
16.
17.  /* Note, the Clock interface is initialized by the
18.   * Clock task (clock.c).
19.   */
```

Listing 5.9: *How the architectural layer is initialized.*

At the moment the “System.init( )” function at listing 5.9, line 15, doesn't do much, but maybe future architectures needs a “general” initialization method. The first two initialization functions should be clear though.

Another example of system dependency is at the MINIX IDLE task. Usually it contains power saving instructions, this makes it system dependent. Somewhere in the MIC idle loop it will call the “System.idle( )” function providing the power saving functions. Also in the end the shut-down and reset sequence come down to a system dependent call, for the IBM PC compatible a BIOS call/interrupt and for the PowerPC a request to the MacIO driver.

Interfaces need to be defined, there input and output should be “standard” over all architectures. The definition of the four interfaces needed by MinixPPC are listed in appendix I “Kernel interfaces”, keep to the interface and you kernel should work. These interfaces don't include the work for the drivers located in <*./drivers/arch/xxx/*>. Some of these need to be ported as well and have there own “internal” interface definitions.

5.4 Memory management

The memory management consist of two parts, the policy and the mechanism ([8] chapter 1, page 51). The policy is located in and defined by the process management process (PM). The mechanism is located in the kernel. The mechanism for MinixPPC is developed from scratch, it only provides the functionality needed for MINIX. In this chapter we focus on the implementation of the mechanism.

The memory space available to a program is defined by the 16 on-chip segment registers loaded when the program is running. Recalling the workings of the PowerPC MMU, every segment register contains several mode flags and a segment identifier (VSID).

The segment ID and page table entry define the physical address where the program (virtual) addresses translates to. It is important to realize that every time a context switch is made, from one process to another the segment registers need to be updated to the values of the new process. Unless processes are sharing memory they never have even one segment register the same. The number of segment identifiers is limited, 2^{24} to be exact. This space is divided by the segment ID's in use at one moment. Every process ID space is 16 segment ID's, this gives 2^{20} segment ID spaces, or 2^{20} process can run at the same time.

In MinixPPC for every process the virtual segment ID's are calculated using its process number,

```
1. SR[0] = 0x100;               /* kernel offset */
2. SR[0] += PID * 0x100;        /* text segment */
3. SR[n] = (SR[n - 1] + 0x10);  /* every next segment */
```

Listing 5.10: *Calculation of segment ID with PID.*

Line 1 in figure listing 5.10 is needed to give the (new) ID space a offset above the kernel ID space. Line 2 gives the offset to jump over every “used” process below the PID and

line 3 creates every ID for the ID space. It is possible to have process 21 mapped in and process 15 mapped out. When a new process is created it will use the ID space defined by it's process number, it could get 15 or 22.

The mapping code (the code that eventually sets the page table entries) is located in the PowerPC MMU library. The kernel memory driver provides a function that takes a process (pointer) as input and does the section to segment mapping, as seen in listing 5.11,

```
1. int Memory.alloc_segments(struct proc* p);
```

Listing 5.11: *Segment allocation interface.*

The allocation function is called with the process to allocate for as parameter. The process contains a partially valid memory map. It has the start and size of the 'text' and 'data' (+ bss) sections in physical memory. The size of the 'data' segment must be updated to include the stack (+ gap) if appropriate. At the moment the function doesn't support the stack (+ gap) in its “own” segment.

The first thing the “alloc_segments( )” function does is check to see if the process is a kernel task, these include the IDLE, CLOCK and SYSTEM task. The tasks run in the same memory space as the kernel. So the segment identifiers are the same as for the kernel. For every other process first the segment identifiers are calculated as above (listing 5.10) and then the sections are mapped. The “alloc_segments( )” function checks if the given stack pointer is zero. A zero stack pointer is impossible for MinixPPC, if so, its invalid otherwise its considered valid. If invalid “alloc_segments( ) will define the stack pointer to the end of the process memory. The reason why we need this check and not always default the stack pointer to the end of the process memory map follows.

For a “normal” process the stack pointer is defined on compilation (a.out format) by the amount of “extra” memory reserved above the 'bss' section. The operating system will allocate the extra memory and initialize the stack pointer to the end of it. But the stack pointer for a kernel task isn't defined on compilation. The tasks are linked into one executable, the kernel process. The stack spaces for the processes (and kernel) are located in the data segment of the kernel. The stack space for the kernel itself is initialized in the file *<./arch/xxx/minix.S>*, at lines 420, 699-717. When filling the process table, the stack pointers for the tasks are calculated by taking a bites from the total task stack space, defined in *<./arch/xxx/table.c>* at line 51. So for tasks “alloc_segments( )” is called, with a valid stack pointer. For new processes this is not the case and the stack pointer is kept zero.

Recall that every address in a process is actually two, a effective address and a physical address. This relation is defined in the process memory map.

```
1. /* Memory map for local text, stack, data segments. */
2. struct mem_map {
3.   vir_clicks  mem_vir;   /* virtual address */
4.   phys_clicks mem_phys;  /* physical address */
5.   vir_clicks  mem_len;   /* length */
6. };
```

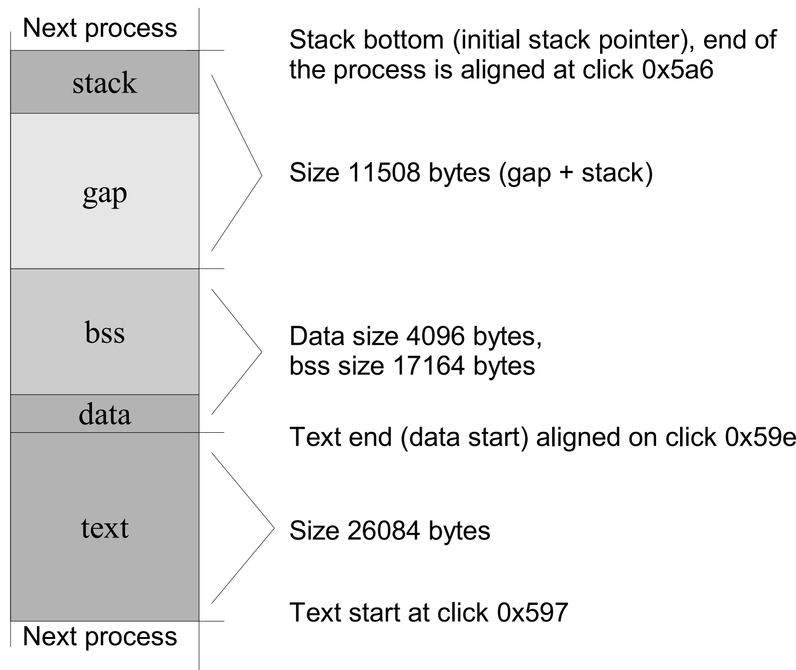
Listing 5.12: *Data structure used in any memory map, for every section.*

The data structure in listing 5.12 is used to build the memory map, one for text and one for the data with stack section. Note that MINIX processes are prepared with extra data structures so the stack could get a own structure (and corresponding memory segment).

The PowerPC architecture allows the same segment identifier to exists in multiple PTE's in one PTEG. Effectively allowing more than one translation for a effective address. The PTE selected for translation is undefined but it's is probably “solved” by letting the hardware take the first it finds.

The memory mapping function in the PowerPC library only supports one occurrence of a segment identifier in a PTEG. By searching and invalidating for the “to map” segment identifier before mapping “it.” Now we don't need to manually delete the previous mapped pages for that segment identifier, and saves the memory interface from a function like “free_segments( ).” Although future ports will show if the decision can be kept.

The next figure shows the memory map of a process to map, address are in clicks and sizes in bytes.



Result virtual address mapping, for above process physical memory map.

Section	SR	VA offset(in clicks)	PA	Size
TEXT	0	0x0	0x597	7 pages
DATA + BSS + GAP + STACK	1	0x1000_0	0x59e	8 pages
(not used)	2	(not used)	(not used)	(not used)

Figure 5.9: *Virtual to physical memory map.*

At the moment MinixPPC only supports the data, bss, gap and stack sections in one segment. To support multiple segment mapping (for example the stack and gap to segment 2) considerable changes need to be made to the “break” code supporting dynamic memory mapping (on exec or malloc call). This code is “trivial” to the “alloc_segments( )” function but changes to the “break” code located in the process manager would change MIC, and that has to be approved by a greater audience.

Large programs (text bigger then 256 MB) are supported. When a program requires more than 256 MB the virtual address space in the text section would cross a segment register mapping by increasing addresses. Effectively going to the next segment, giving the MMU a new segment ID for translating a physical address for the same text section. The allocation algorithm takes care of this by aligning the virtual address to 0x1000_0 (in clicks), and then do the mapping.

The current programs are compiled with the data section at the “next” address multiple of 0x1000_0000, defining the mapped segment. How this is forced is handled in chapter 6.2 “Linker scripts.”

5.4.1 Mapping a new process.

A “fundamental” change has been made in the creation of a new process. Although the code impact is not that big, the consequences are. To explain where, why and how first a little introduction to the “exec( )” call, the only way to execute a new process.

The way to create a new process is doing a fork and letting the child process do the execute call. This call “destroys” the original process (the child) by transforming to the new process and starts the new process when it's rescheduled again.

The exec call would make the PM read the file image from disk, create and allocate a physical memory map (it manages the physical memory) and loads the sections into the allocated memory. It is here where the update of the process memory map is done, with the “sys_newmap( )” system call. The (original) system call was one-way. The memory map of the process was completely calculated at the PM, this includes the virtual addresses. This introduced a problem with virtual addressing, the only place where these addresses should be calculated is in MDC and the PM should stay clear from MDC. But the PM must know the virtual addresses eventually. The solution was simple.

The “sys_newmap( )” call used a pointer to the process memory map at the PM to read the initial (physical) memory map. Recall that the “alloc_segments( )” function only needs valid physical memory addresses (and sizes), and it updates the virtual addresses (to use) in the process memory map. Now the system call simply overwrites the memory map at the PM with the one that is updated locally (in the kernel process table) by the “alloc_segments( )” function. This approach is needed as every architecture could use different virtual address definitions and it should work every time.

The fork code did not need any modification but at the “sys_newmap( )” call, a virtual copy is added right after the allocation function. The copy writes the memory map back to the PM. The code at the PM is essentially still the same. The original file it contains compiler directives that selected between IA and others. For the PowerPC a extra select is added, but this should replace all other selects.

5.4.2 Remote segments

For MinixPPC remote segments are a powerful tool used by drivers when doing I/O. Most devices present there complete register map in blocks of memory, for example the VIA (this includes access to the ADB and PMU) of the MacIO device uses memory range [0x8001_6000 – 0x8002_A000], a memory block of 80 KB. Remote segments for big programs that are using more than one segment for text or data are not supported at the moment.

The driver would map the memory range with a remote segment request. Mapping the complete block it will be able to access the device registers in its own address space. The driver can use simple I/O functions to write and read device registers. The kernel does not have to be between the driver and device doing the I/O call, improving performance.

The PowerPC has 16 segment registers of which at least three are reserved (at the moment two are used) text SR(0), data + bss SR(1) and stack SR(2). This leaves 13 segment registers that can be mapped to blocks starting at any physical address. Most drivers however will only require one remote segment.

Using the system call “sys_segctl( )” will return the index, selector and memory offset of the remote segment mapped. A driver for MinixPPC uses the selector and offset to create the virtual address to the start of the physical memory block. Drivers for other architectures should use there own interpretation of the “return” variables.

Next the mapping of a virtual address range to physical address space,

```
1. sys_segctl(&via_rsi, &via_sel, &via_off,
   (phys_bytes)0x80016000, 0x14000);
```

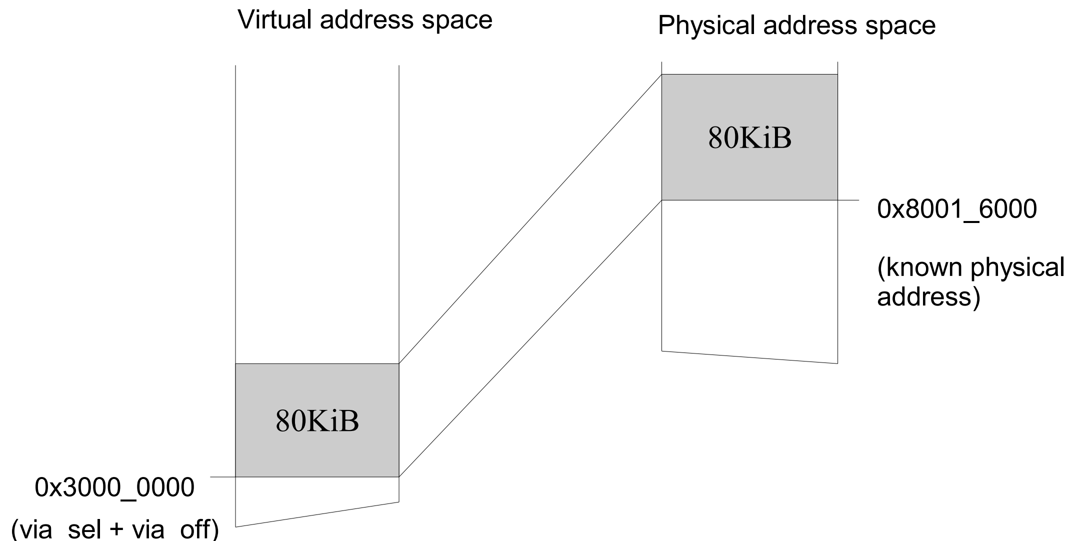


Figure 5.10: *Mapping a remote segment, index 0, start 0x3000_0000.*

If this is the first remote segment mapped for the process via_rsi would return 0. via_sel will return 0x3000_0000 and via_off shall be 0. via_off is zero as the request PA is a multiple of the page size. At the moment this is the only supported mapping, mapping of non-aligned physical addresses isn't supported for security reasons. With the mapping in figure 5.10, a read/write at address 0x3000_0050 would read/write at PA 0x8001_6050. A read/write of an address bigger than 0x3001_4000 would trigger a DSI exception.

The call "sys_segctl( )" uses the "Memory.alloc_remote_segment( )" function located in MDC. The original kernel call code was very Intel specific and has been replaced by it. It should be possible to move the Intel code to the MDC part of the x86 without problems.

5.4.3 MMU library functions

The MMU library for the PowerPC is located in the *<./lib/ppc/mmu>* directory. It contains functions to set and update the page table. The main functions are implemented in the *<mmu.c>* file. The *<libmmu.S>* file contains utility functions to write a PTE, switch off the MMU, read or write the segment registers or invalidate effective addresses. These functions are written in assembler as they use special instructions from the OEA instruction set. The *<mmu.c>* file contains the functions to map a number of pages for PAT or set the BAT. All the functions written for the MMU will have the prefix "mmu_", functions for the PAT system contain infix "pat_" and BAT system infix "bat_." We will focus our attention on the PAT functions as MinixPPC does not use BAT.

Before using the PAT system first a page table must be set up. For MinixPPC this is done by the monitor so the kernel assumes there is a valid PT and starts mapping anything it needs. This is by design, the kernel could redefine the PT if needed.

The line in the monitor initializing the page table,

```
1. /* Init the page table at 128MB (note, the test machine has 256MB
2.  * installed).
3.  */
4. mmu_pat_init_pt(0x08000000, PT_SIZE_SYSMEM_256MB);
```

Listing 5.13: *Page table initialization.*

It should be noted, that when defining or writing to the PT the MMU should be off. Also writing to the PT, should be done using the memory "force" functions to make sure the physical memory is reached.

Mapping one page in the PT takes the steps listed in listing 5.14 on the next page. To map a page one needs a VSID (aka segment identifier, one of 16), EA, PA, protection and access flags. The functions in step 3, 4 (except the searching) and 5 require the MMU to be off. Never forget to invalidate the effective addresses mapped (step 6), otherwise exceptions could occur while accessing only some of the pages, giving much confusion. To map more than one page these step are placed inside a loop and the EA and PA are increased every iteration with one page size, 4 KB (0x1000 bytes).

1. Calculate the address of the PPTEG and SPTEG (these come in pair)
2. Get the PPTEG, search the PPTEG for VSID & API
3. If needed get the SPTEG, search the SPTEG for VSID & API
4. Create PTE for EA, with PA + protection + access
5. Update PTE in PT
6. Invalidate EA

Listing 5.14: *Updating a PTE.*

The monitor also maps the kernel text and data segments. The text segment mapping is actually the whole physical address mapping for the kernel, hence the mapped size (64 MB, independent of the actual text size). Note that the kernel and the tasks are the only processes using physical addressing.

Listing 5.15, the kernel segments mapping by the monitor,

```
1. /* Map the pages for the kernel “text” segment 1:1.
2.  *
3.  * Note, memmap is just like the segmap but it uses the
4.  * current "onchip" segment registers, for mapping inside the PT.
5.  */
6.
7. mmu_pat_memmap(
8.    0,                  /* from ea */
9.    0,                  /* from pa */
10.   0x4000,             /* 0x4000 * 0x1000 (bytes) = 64MB */
11.   PTEL_M,
12.   PTEL_PP_USER_NA,
13.   NO_FORCE);
14.
15./* Map the pages for the kernel data segment, data (and bss)
16. * is linked to the next 256MB so it uses the second segment
17. * register for it's address translation.
18. * (note: the mapping is into the kernel segment).
19. */
20.
21.count = (process[IMAGE_PROCN_KERNEL].end -
22.          process[IMAGE_PROCN_KERNEL].data.addr) / PAGE_SIZE;
23.
24.mmu_pat_memmap(
25.   0x10000000,         /* the second segment.*/
26.   process[IMAGE_PROCN_KERNEL].data.addr, /* just after text. */
27.   count,                    /* page count data + bss */
28.   PTEL_M,             /* defualt memory access */
29.   PTEL_PP_USER_NA,    /* default protection */
30.   NO_FORCE);          /* mapping type */
```

Listing 5.15: *Mapping the kernel text and data segment (done by the monitor).*

On line 7, the text segment is mapped and on line 24 the data segment. Note that the monitor assumes that the kernel text is never bigger than 0x1000_0000 bytes.

Almost all memory management steps are programmed in separate functions. Some library functions return only intermediate results. Like viewing the addresses calculated for the primary PTEG and secondary PTEG. These are “left over” from developing and debugging. For debugging purposes there handy but otherwise not used by the (in kernel) memory driver for MinixPPC.

5.4.4 I/O

There are two ways a driver can do I/O on the MINIX OS via system calls or by remote segment(s). With the first, system independency is guaranteed as a standardized system call is used. The second way is using memory mapped I/O. Usually with mapping remote segments on the device registers. In this case the driver can directly access the device registers in it's own address space. No kernel in between communication, only using a “special purpose” I/O function. For MinixPPC all drivers (and in kernel drivers) are built this way.

If all drivers always used memory mapped I/O, the kernel would not need I/O system calls. But some architectures (IA) must use port I/O, and that is most of the time privileged. So the kernel must be used to do the I/O for the process. But for much reading or writing it doesn't scale to well. Luckily MINIX uses a performance tweak, using a separate system call for I/O streams. To prevent a driver process using a lot of IPC to the requesting process writing a (local) buffer with I/O data. This tweak lets the system task read form a I/O port (or register) and write directly into the requesting process address space.

Lets take the file system as example. The FS process would get a request to read bytes form a file on disk, the disk uses the AT_WINI driver. This driver uses a ATA layer to access to the harddisk. With the standard library (fread) read call a (user) local(!) buffer pointer is given. The call needs the AT driver to place a request for the data blocks from the disk drive via the ATA controller. The commands go via the simple device I/O system calls but the data transfer goes with a “sys_sdevio( )” system call to read from the ATA data port (or register) and write to the (user process) local buffer. So in the actual transfer of data the ATA driver is bypassed. Note that on the PowerPC architecture port numbers are interpreted as (register) addresses.

Getting to our problem the system task must occasionally read from a register address that is unknown. It can be anywhere in the 4 GB memory space. This makes it difficult for the system task to map addresses in advance, it can map the whole 4 GB but that would create a (large) page table with a lot of unnecessary entries. The solution is to let pages be mapped “automatically”, only when needed, which is unfortunately a complicated and not (yet) completely secure process.

When the system task tries to write or read to a page not mapped, a DSI exception will occur. This exception is then used for mapping the page with the (port) address in the page table. The DSI exception handler in file <*./arch/ppc/exception.c*> will check to see if the process causing the exception is the system task and only then update the PT. At the exception we know the effective address making the exception, but we need the physical address to where it translates for the process containing the buffer. The process making the (stream I/O) call has the address of the register mapped (probably by remote segment). We need the segment identifier used by that process to calculate the physical address of the register.

Using the “mmu_pat_seg_ea2pa( )” function from the MMU library and the effective address from the exception we get the physical address. Now we map a page 1:1 to the physical address for the system task (all kernel processes use 1:1 translation). The exception returns and the system task restarts with the page mapped so reading (or writing to) from the address is now possible and the copying can take place.
The figure will show the control flow of mapping a page for the system task (or kernel),

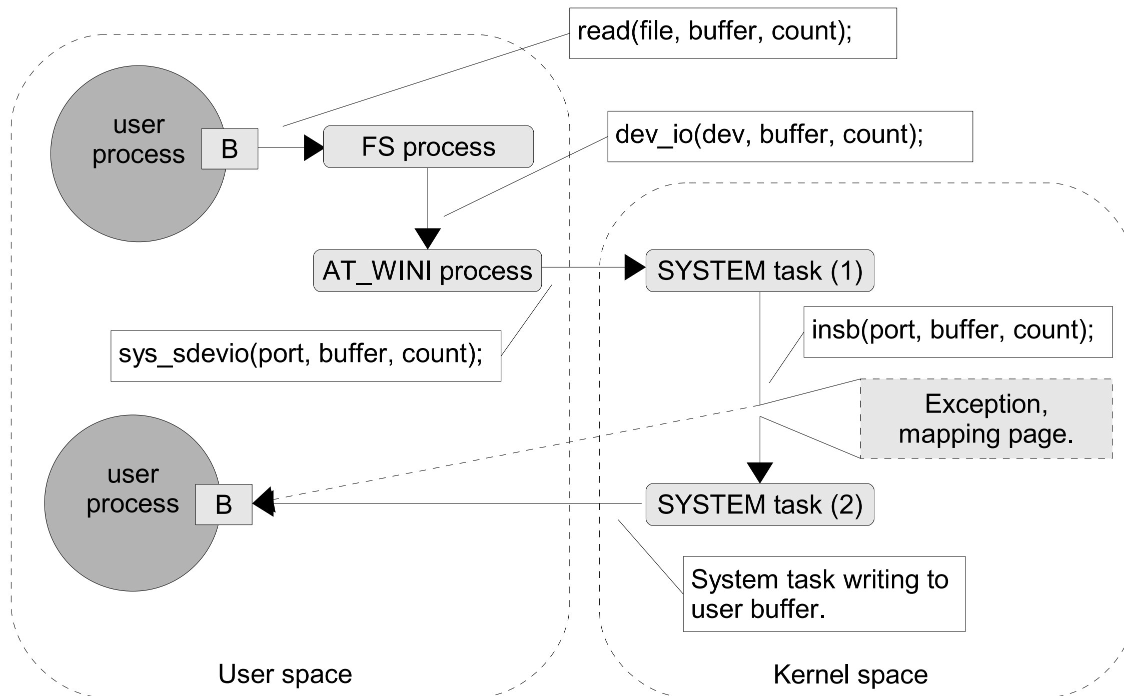


Figure 5.11: *Control flow of user process requesting data for a file on disk.*

Note that the lines don't represent the IPC construct of MINIX, but system function calls. It is important to understand that this flow only occurs when the address is accessed the first time. In every next “sys_sdevio( )” system call the page is already mapped so the second system task process box and exception box would not be needed. The data would follow the solid and then dotted line from the first system task box.
By the introduction of this method security issues are introduced and need consideration. A process could do a “dev_io” system call to some place in memory and the system task would map it, and execute the request with the highest priority. In the newer release of MINIX v3.1.2 I/O security is increased the re-port will show if this solves security problems introduced by this scheme and remote segment allocation. A solution would be to disallow this construct but the memory allocation problem will stay. Mapping memory regions of devices in advance for the system task could be a solution but isn't dynamic.

5.5 Exceptions and context switching

Recall that for the hardware interrupt and exceptions are the same. As a consequence the code handling interrupts and exceptions is the same. Only for the system call a exception is made, as it's the only interrupt using parameters and returns a result. For this chapter interrupts and exceptions together are called exceptions. Except for two, all exceptions are handled at the file <*./kernel/arch/ppc/exception.c*>. The system call and external interrupt are handled in the <*./kernel/proc.c*> and <*./kernel/interrupt.c*> file. All the code for context switching is located in the <*./kernel/arch/ppc/minix.S*> file.

Every exception starts at a exception vector, for MinixPPC these are without offset, so the first exception is at 0x100. The <*./kernel/arch/ppc/minix.S*> assembler file is the object (when assembled) that gets loaded from address 0x0, so it's written over the exception vector space when loaded by the monitor. Knowing this we can force the assembler to assemble code aligned to the vector (address) we want, catching the exception, using the following code snippet,

```
1. # Phase 1 exception c0de.
2. #
3. #define EXCEPTION_PHASE1(ev, handler)       \
4.         . = ev;                             \
5.         ...
```

Listing 5.16: *Assembling code to a exception vector (ev).*

Listing 5.16 is the start code for every exception. The '. = ev;' directive on line 4 instructs the assembler to assemble the next instructions from the 'ev' address.

Memory mapping of “minix.o” object to main memory,

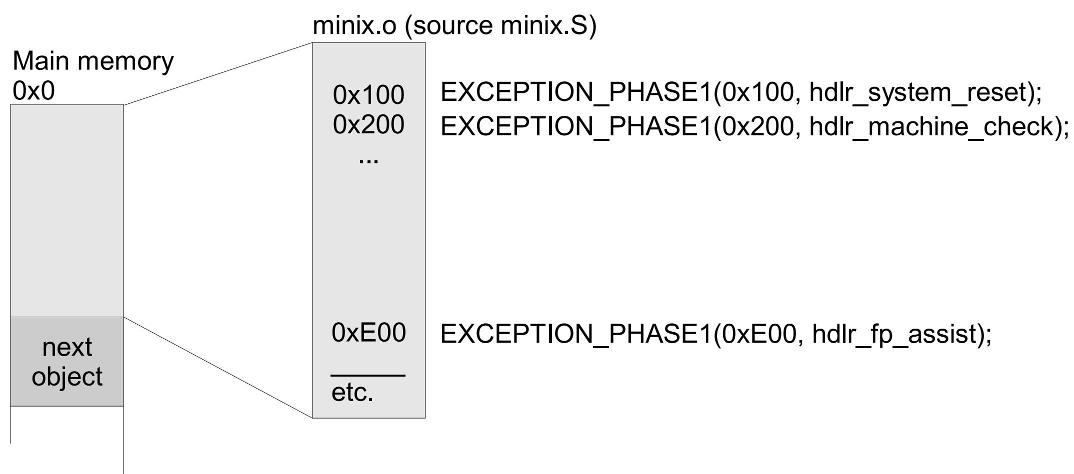


Figure 5.12: *The first bytes in memory, <minix.o> object loaded at 0x0.*

The <*minix.S*> file is part of the kernel process. The Makefile to build the kernel process makes sure the “minix.o” object is always linked before every other object file making the kernel process. The file further contains a few lines of code creating stack space and calling functions to start MinixPPC “rolling.”

As a careful reader would notice, there are only 0x100 bytes between each vector, too small to write a reasonable handler and too big for just a jump so space gets wasted. It is not a “real” problem but we will see later a acceptable solution is found.

We need the same code for almost every exception as a result we use macro's. The <*minix.S*> file has 12 lines describing the first phase for all exception types, of which we have seen the first four lines in listing 5.16 and all in appendix A.1 “Exception phase 1.”

For exception handling the PowerPC CPU has four “Special Purpose Registers General”, SPRG[4]. The only special thing about them is that they are not part of the UISA. They are used for intermediate use when saving the state of the CPU to memory. For the PowerPC it's impossible to access the memory without using a GPR so if the content of that same GPR needs to be saved it has to be temporarily stored somewhere. To make context switching easy and fast it's better to define the use of the SPRG's. For MinixPPC only SPRG[3] has a predefined use, it will always contain the “running” process (physical) pointer into the process table. The rest SPRG[0 – 2], can be used by any of the macro's that save and restore the CPU state. Exceptions are handled in two phases by MinixPPC, where phase two contains 5 steps,

1. Prepare CPU save
2. Do CPU save
   Call the C-handler
   Clean up
   Restore CPU
   Continue or restart next process

Listing 5.17: *Context switch steps.*

The first phase selects where the CPU state is saved; to the kernel stack or to the process table. Normally a exception would be from a user process or clock task to the kernel code. In this case the CPU save is going to be to the process table, via the process pointer in SPRG3. If the exception is produced by the kernel itself (so there is at least one exception taking place already) the save is done to the kernel stack. If the exception is critical we go into panic. For a normal exception the kernel stack is always empty and the handler (external interrupt or system call) would start with a fresh stack. Note that the exceptions become messages, for both the external interrupt and system task.

Looking at the complete code for exception phase 1, appendix A.1 “Exception phase 1.” The first thing it does is freeing up GPR1 and GPR2 so we can use these registers to restore the segment registers for the kernel. A exception would disable the MMU, by loading the default kernel MSR we re-enable the MMU, and have translation again. We need this because the variable 'k_reenter' is located in the kernel data segment and we use a virtual address. If the re-enter count is greater than zero we are re-entering the kernel and should use the kernel stack again. If the re-enter count is zero we are switching context from a user or

task process, which means we need to save it to the process table. Note that saving to the kernel stack prevents a different process to be run next. The kernel always runs again, when the system doesn't halt with a panic. Phase one ends with a call to “handler##_table” or “handler##_stack”, for '##', “handler” in the macro call is substituted. Luckily the code for the phase one is small enough (75% of free space) to fit between the exception vectors, so not much space is wasted.

After restoring GPR1 and GPR2 we enter phase two. Note that the content of the whole USIA register set is exactly the same as on the start of the exception. In the second phase macro the two functions “handler##_stack” and “handler##_table” are defined, see appendix A.2 “Exception phase 2.” Recall the end of phase one, there they are called.

When we need to save to the stack, we increase the current stack pointer with the size of the CPU stack frame. Load the stack pointer (top) to the SPRG2 and call the macro “SAVE_CPU”, the first step of phase 2, with the pointer in SPRG2 as destination and SPRG1 as scratch register. The macro code saves the total USIA register set behind the pointer in SPRG2. After the CPU save, a zero frame is created on the stack. This is needed as the C handler function called next will save it's link register to the above frame, to created a link chain (see ABI [1]). As far the compiler knows this is a ordinary function.

The C handler function is called next and we do step two. When the handler returns we begin step three. If the stack is cleared from the zero frame and the kernel re-enter value is decremented. In the next step the CPU restored and the kernel process is ready to restart. Then the stack is cleared from the stack frame and a Return From Interrupt (RFI) instruction is taken, ending step five. Looking at the code in appendix A.2 “Exception phase 2” line 25, it looks like the stack pointer is restored wrong but it's not. Just before we saved the CPU the stack pointer was increased, decreasing it make it right.

The phases for the process table save are the same except the CPU save is much simpler. The “save space” is already allocated in the process table, and the pointer in SPRG3 is already pointing to it. We use the same “SAVE_CPU” macro as before with the pointer in SPRG3 as destination and SPRG2 as scratch register. After the CPU save we take a fresh kernel stack, “_LA(SP, kstk_bottom)” appendix A.2 line 29, and call the C handler. When the handler returns, a jump to restart is made, starting step 5. This function loads SPRG3 with a valid (physical) pointer into the process table and decrements 'k_reenter. At last the CPU is restored, including the segment registers with the process. A return from interrupt instruction follows, ending step 5.

The process loaded by “restart( )” is determined by the value of 'next_ptr', this variable defines the process (pointer) to be run next. If it is zero (NULL) the current process in SPRG3 is run again. If not it's converted to a physical address and loaded into SPRG3. Note that segment registers are never saved. They are only written to the process entry on process mapping by the “Memory.alloc_segments( )” function.

Every time a exception is taken, it is always the kernel that runs so we need the same values for the segment identifiers. The kernel segment registers (values) are “hard coded” into the “SEGMENT_REGISTERS_RESTORE_KERNEL” macro.

To illustrate a context switch figure 5.13 follows on the next two pages. In the figure user process 1 (file system) is running and is interrupted by the hardware clock. The dark blocks indicate actions taken by the code. The two phases and reset parts of the code are closed inside dotted boxes. Transitions from code parts are indicated by dotted arrows as well. If not indicating a resource used for the action, the solid arrows indicate a context switch. The clock interrupt will switch from user to kernel space and starts the saving of

the running process (P1) and selecting (most of the time) another process to run. In figure 5.13 this will be the clock task, we assume that the file system process has used it's full quantum and needs to be rescheduled. This will make the clock hardware interrupt handler ("clock_handler( )") send a message to the clock task. The result is that the clock task is scheduled and it will do process accounting/scheduling and timer management (in the function "do_clocktick( )"). This is enclosed in the solid box coming from the clock task process. Note that some confusion could occur as the clock handler and clock tick functions are in the same file (<*kernel/clock.c*>) but their context is different although they both run in kernel space.

Time
User space
P1
Context switch from user space to kernel space, on clock interrupt (#63).
Kernel space
SPRG3
-4 | -3 | -2 | -1 | 0 | 1 | 2 | ...,n
Process table
'k_reenter' == -1
e_phase 1
enable MMU
k_reenter += 1
'k_reenter' == 0 so save to the process table.
Saving to slot 1
e_phase 2
Fresh kernel stack for the (interrupt) handler (note k_reenter == 0).
save CPU state
fresh stack pointer
call for intr_handler
kernel stack
Call every function hooked to interrupt #63
restart
This includes "clock_handler()" and that could send a message to the clock task, the clock task would always be ready to receive the message so it's run next.
User space
Kernel space
Time

(***Figure 5.13:*** *continued on the next page)*

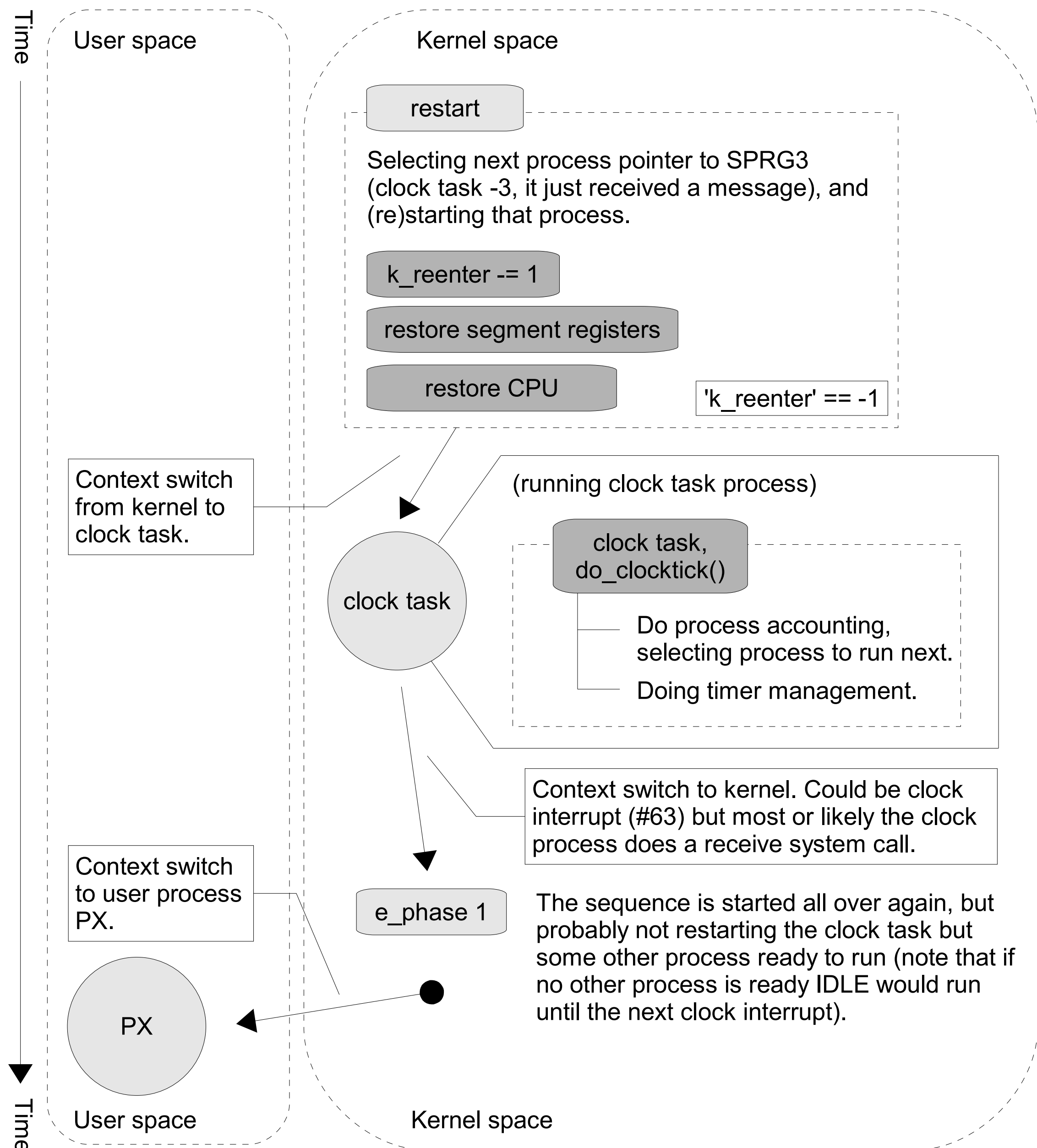


Figure 5.13: *Context switch P1 to clock task, and (re)starting other process.*

5.6 Signals

Signal handling code is pretty machine dependent although the changes needed to make from x86 to PowerPC are not earth shaking. Here a small introduction to the concept of signals is made to illustrate the problems found and changes made. If signals are completely new look for a full explanation look at ([8], chapter 4.7.7).

To implement signal handling you must know the ABI for you target architecture. It defines the layout of an activation record, the layout of "memory" on the stack when a function is running. The normal way a activation record is created is by the called function itself. For signal handling we have to implement a pseudo activation record, to simulate a function call, by writing to the stack space of the signalled process.

The use of signals involves the process manager. For most signals the default action is to kill the receiving process. The next mechanism allows to change the default action. Only for the "kill" signal no new action can be defined.

Signal handling can be divided into three phases,

1. Preparation
2. Response and
3. Clean up

Listing 5.18: *Signal handling phases.*

Preparation means for a process to define the response to the signal. This must be done in advance, by the process itself because signals get sent when the process is not running. For this administration the PM process is used. For every process it will maintain a list of actions for supported signals. Processes usually redefine the default action to run a user defined signal handler function, but the signal could also be (just) ignored. The focus of the port lies on the introduction of the user signal handler.

Taking two processes, user process U and process S sending a signal to process U, the phases look like,

- Process U creates a new action for signal USER1. It tells to call signal handler function "siguser1( )" in process U itself.
- Process S send process U signal USER1. The PM will update the state of process U in a way that the first time it gets scheduled again function "siguser1( )" is called.
- When "siguser1( )" returns the PM must catch this to update process U so it continues execution as if no signal was send.

Listing 5.19: *Signal handling phases for process U and S.*

The programmer of process U would program a function of type 'sighandler_t' in the program. This only means that the function prototype is fixed and it looks like "void siguser1(int);" Then the programmer will install the signal handler using the "sigaction( )" function. This tells the PM to call the function "siguser1( )" when the signal USER1 is send to process U. The preparation will not change process U in any way.

When process S sends signal USER1 to process U, it does this via the PM, and when process U is not running (process S is running). The first action of the PM will be to save a copy of the current CPU state (created by SAVE_CPU) of process U (located in the process table) to the stack of process U. The CPU state is needed after the user signal handler

"siguser1( )" is finished. Recall we need to continue process U as if nothing has happened.

Next it will create the small "pseudo activation record" on the stack, as if function "P( )" called function "siguser1( )." The pseudo record of function "P( )" will contain the parameter for "siguser1( )" and will set the return address for the user signal handler. This will enable the PM to catch the end of the function as it is set to "sigreturn( )." Note that function "P( )" is not programmed by the user and actually never called. Then the program counter of process U will be updated to call "siguser1( )" on the next time the process gets run.

The new stack of process U phase 1, just before it runs the function "siguser1( )" and phase 2 when the user signal handler is "running",

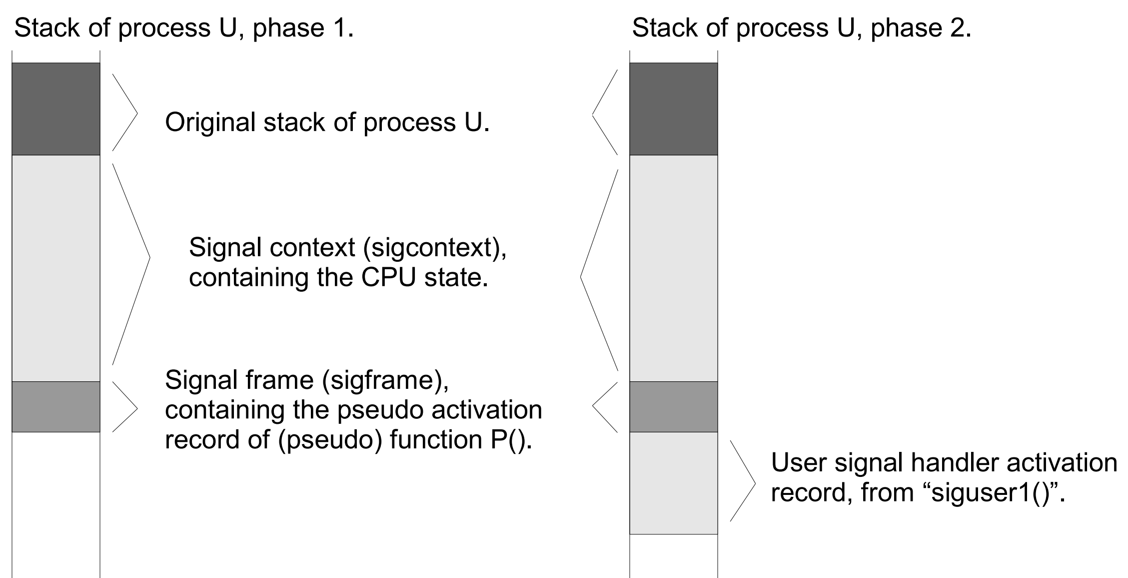


Figure 5.14: *Stack phases of signalled process.*

When "siguser1( )" returns, a call to "sigreturn( )" is made, undetected by the user. Note that this was possible by the pseudo activation record. This function will tell the PM to clean-up process U from the signal and restore to it's original state. This means removing all leftover stack parts in figure 5.14, phase 1, and restoring the original CPU state from the "sigcontext" just removed.

The place where the PowerPC differs with the IA that matters now, is at stack usage during a function call (calling convention). Parameters are not passed via the stack but via registers (see chapter 3.4 "Software") and the function return address is not located on the stack, but in the CPU Link Register (LR). The support for the PowerPC comes in the from of a altered library function "sigreturn( )", and a few extra lines in the kernel system call that prepares the stack and process state. Because a new file is created for the library that's portable, but the update to the system call required very machine dependable alterations and final solution must be worked out when more operating systems are ported. At the moment two lines of compiler directives are used.

For the PowerPC signal support, the signal frame is only used as buffer and parameter

container. Perhaps this is the best view to take, making a standard (MINIX) convention for the “pseudo activation record.” This will call for specialized (assembler) “sigreturn( )” functions for every architecture.

5.7 New drivers and changes

The PowerPC architecture demanded new or changes to existing drivers. Changes are made to the AT_WINI and TTY driver. The TTY driver has been split to be independent of keyboard scancode generation and the AT_WINI driver uses a new ATA layer in programming.

The original AT_WINI driver is updated to supports block offsets and block counts greater that MAX_UINT (64 bit type is used). Partition map support for Macintosh is included, and the driver has a new ATA layer that separates the “driver” code from the way access to the ATA registers is made. The “old” set up would need a “base” address (port) from where all other register addresses are calculated from. This works if the space between the registers is the same for every architecture, but at the PowerPC the ATA registers are spaced differently then at the IA. Besides BIOS access for device info the rest of the AT_WINI driver is mostly intact.

Register spacing is “dealt with” by the initialization of the ATA layer. On initialization the ATA layer will request the ATA PCI device infos from the PCI manager and calculate the addresses and places them in a array. For the AT_WINI ATA register access is now simply by index. In theory it should be sufficient to port the ATA layer instead of the AT_WINI driver.

The TTY driver has the hardware notification replaced by a scancode message. This message comes from a separate driver, “macio.” Such a driver will be needed for every architecture. The only system dependency in the TTY driver is the video driver. At the moment it is still linked in the TTY process itself, but it should be trivial to build in a separate process to print characters to the screen.

5.7.1 MacIO

The MacIO driver is the “General Purpose I/O” (GPIO) driver for MinixPPC. The driver includes support for several low speed and system management devices. Most importantly the ADB and the PMU devices. Besides the ADB and PMU the MacIO ASIC contains the OpenPIC controller, but this is logically a separate device and as a consequence it is driven by a separate in-kernel driver. The communication to the ADB and PMU is though a Versatile Interface Adapter (VIA), “for maximum compatibility and programming effort.” The driver is programmed into three logical parts (drivers) named; ADB, PMU and VIA.

The MacIO driver is located in the *<./drivers/arch/ppc/macio/>* directory and called “macio.” At the moment the PMU driver provides minimal support, only to do a reset or shut-down. The NVRAM is accessed via the PMU driver and is used to store non volatile information like Open Firmware defaults. At the moment this is not implemented but test

code is there. Furthermore the PMU is used to put the system in different power modes and to wakeup again. Also CPU speed and the cover state (in case of the iBook) can be monitored.

The ADB driver supports multiple devices on the ADB; they are identified by there addresses. It can scan the ADB for attached input devices and a specialized "driver" can install it's handler for a ADB address. This handler would then be called by the ADB driver when data is available. The iBook keyboard and trackpad are supported via the ADB, the keyboard is at ADB address 2 and mouse at address 3.

The logical view of the MacIO driver and the files giving the location of the code,

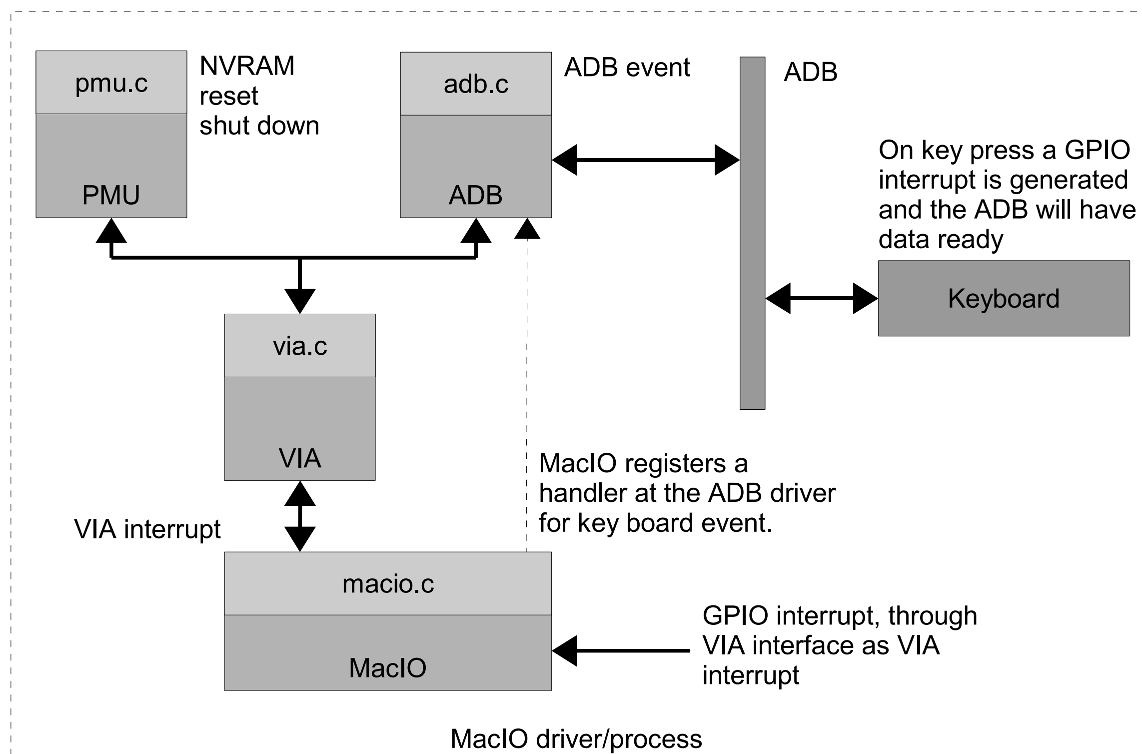


Figure 5.15: *Logical view of MacIO driver.*

The driver is build from three objects providing the interface and management parts for the ADB and PMU. A fourth object is needed to provide the interface for the MacIO process to MINIX.

Processes can now send and receive messages from the driver. If the iBook must be shut-down the MacIO process gets the request and uses the PMU driver to shutdown the system. The same goes for the system reset message.

One of the first things done by the MacIO driver is register a IRQ hook for the GPIO interrupt. So it will receive a hardware notification message from the system. There are various ways a GPIO interrupt is generated by the PMU or by the ADB. The PMU can be set up to generate a interrupt every second, on environmental changes and more. The one we are most interested is the interrupt generated when there is data ready on the ADB. This is

the case when a key is pressed or the mouse is moved. To catch these events there must be a ADB event handler installed at the ADB driver.

The iBook has three devices on the ADB, the trackpad, the “normal” keyboard giving all characters and the “extended” keyboard. Using the “fn” key generates a event from a special key like the volume adjust or eject button for the CD drive.

The ADB events originate from a different addresses and an event handler needs to be installed per address. At the moment the MacIO driver only installs a handler for address two, the default address for a keyboard. The only thing this handler does is convert the ADB event to a scancode message for the TTY driver and send it. This immediately makes the TTY driver system independent of how the scancode is generated.

Considering the original keyboard driver that was located inside the TTY process the new keyboard driver is split in two parts and placed in two processes. One part that “generates” the scancode message and the part that converts the scancode into a character. The last part is still in the TTY driver; it will be the same for “all” architectures, although there could be a different key-map per architecture. As a result the TTY driver is not getting a hardware notification, this is replaced by the message from the MacIO driver containing the scancode.

Here we see the transformation of GPIO interrupt to message, ADB event to scancode and scancode to character.

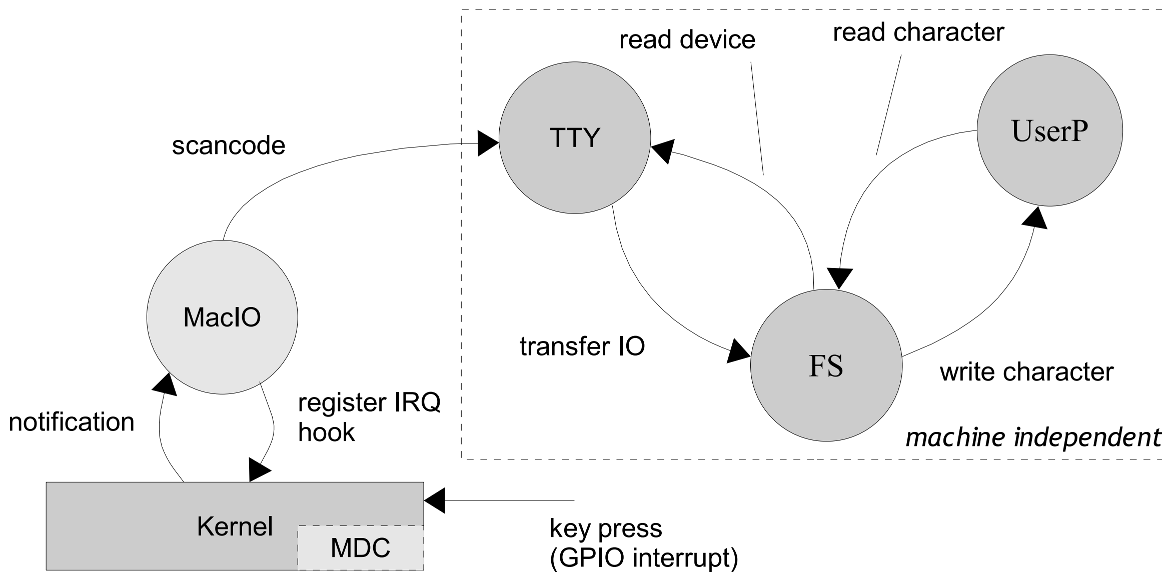


Figure 5.16: *MacIO and TTY driver relation.*

In figure 5.16 the flow of messages is drawn. The scancode message is the only message that goes form MDC to MIC. Although the kernel is viewed here besides the MIC it contains a large part of MIC. The MacIO process is inherent to the PowerPC architecture so it is completely machine dependent.

A similar construct has been developed for the PCI drivers. By letting drivers using PCI

devices gather information from a “standard” process we make the information gathering system independent.

5.7.2 PCI manager

To use devices located on the PCI bus, a PCI manager is needed. This process is created to be used by device drivers that want to manage a device on the PCI bus. The PCI manager (PCIM) is located in the *<./drivers/arch/ppc/pcim/>* directory and is called “pcim.” It will be included in the system image so it is available at all times. It's using the PCI system to gather device info to aid in independency from Open Firmware or the BIOS. At the moment the PCI manager is only used by the AT_WINI driver, but could be used for a Ethernet or video driver as well. The AT_WINI driver provides IDE support via the ATA layer.

The PCI manager of MinixPPC should not be confused with the PCI driver for IA that is included with MINIX from v3.1.2. The two were designed separately, at the same time and the developers were unaware of each other. This document will focus only on the PCI manager included with MinixPPC.

When the PCI manager starts, it scans the PCI bus for devices and creates a list of the devices it finds. For every device found it's data is queried and a entry is made in the list selectable via it's index or ID. The entries contain information about vendor and product type of the PCI device. A driver always “knows” for which device(s) it is built and includes this in a request to the PCI manager. On a match, the PCI manager returns all information found on the PCI device at the scan or device not found. The structure returned (or actually written) by the PCI manager is listed in appendix A.3 “Definition PCI device.”

The next figure shows how the AT_WINI device driver requests the information from the PCI manager about the ATA PCI device,

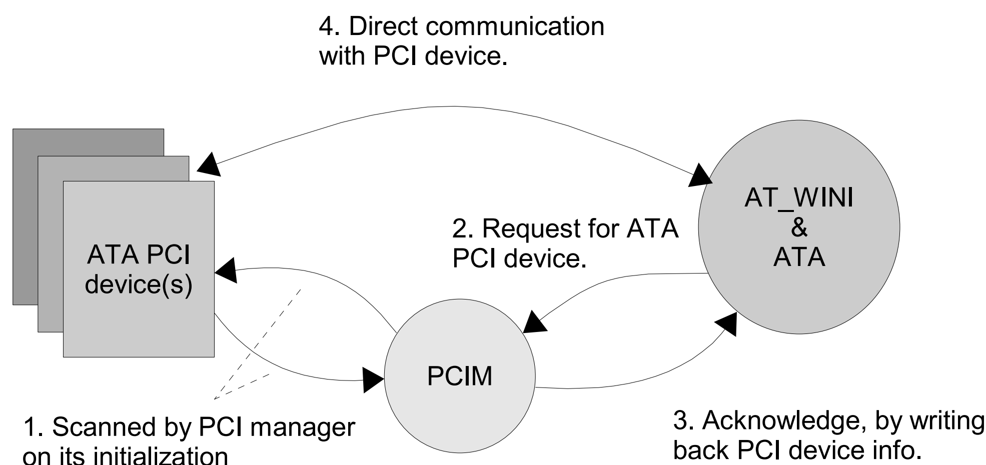


Figure 5.17: *The PCI manager, route to returning information of the PCI device.*

This information includes the physical start addresses of the I/O and MEM registers in the form of memory ranges that are mapped into main memory by the Open Firmware boot software. These registers are used to communicate with the PCI device itself. There context is defined by the device driver not the PCI manager. To use the PCI device the driver would map the memory using remote segments. Then it accesses the device registers with the (simple) I/O functions from the PowerPC library.

When a driver requests the information of a PCI device already mapped to another driver it will return “error device already in use.” This forces the system to have only one driver per PCI device. Only one driver at the time can request information per index until the index is freed again.

To use the PCI manager there is a library created, “libpci.” It includes the functions to request device information, PCI device count and to disable or enable a PCI device. The functions use the “sendrec” IPC system call to keep the driver synchronized with the PCI manager. Beside the functions for the PCI manager there are “general use” PCI functions, like dumping the PCI device information. In combination with the “PCI device request by index” very handy to request and dump all found PCI devices in the system.

5.9 Utilities

For the development of MinixPPC, several utilities were created. These are needed to create a file system image, create the system image or convert a executable from Elf32 to a.out format. The utilities are written in C and use the host operating system, in this case Linux. Where possible the types used are of the MINIX OS, careful programming is needed not to mix things up. Some types are defined differently using the MINIX standard include or Linux standard include directory. For example the a.out definition, at MINIX the size of “struct exec” is 48 bytes and for Linux it's 32 bytes, also the field names are incompatible.

The “elf2aout” program is used to convert a executable from Elf32 format [2] to a.out format. It is needed because the MINIX OS supports only the execution of a.out executables. The Elf32 format is designed to replace the a.out format because it's too limited. For example it does not support load addresses and dynamic linking, which are needed by large systems to keep the size of executables limited to be more efficient with resources. The “mkimage” program is used to create the system image from the kernel, servers and drivers. The “mkffs” program is used to create a MINIX v3 file system from a prototype file system.

5.9.1 elf2aout

The elf2aout program is located in the <*./util/elf2aout/*> directory. It uses the “libabi” to convert the formats. It can take several arguments as parameters to increase or decrease the stack size of the created a.out program, or to output information to the screen.

It starts by loading the Elf32 file into memory. Then it scans the header sections, using the library function in “libabi.” The Elf32 format supports a lot of sections with different

attributes, dynamic loading, static or non static allocation, symbol table, read only or comments. See appendix G.1 “Elf32 section listing of the elf2aout program”, for a typical Elf32 executable section listing, it contains 35(!) section types.

The a.out format supports a fraction of these, the most important are the 'text', 'data' and 'bss' sections. Our goal is to divide the Elf32 sections over the three sections in the a.out format. Luckily for us we can limit the output of sections in the Elf32 format in two ways, that is best explained when compiling the monitor program. First we limit sections by giving compile options and second using linker scripts. Note that the monitor program stays in Elf32 format for the Open Firmware software. The impact on giving compiler options can be viewed in appendix G.2 “With compile-time options.” Although there are still a lot of sections created there are about ten fewer. The options instruct the compiler not to generate certain sections like '.sdata' (no static data section).

Using linker scripts we can limit the output of sections even more, to whatever we need, almost creating the format we seek! The linker script used to limit the sections in the monitor executable is listed in appendix G.3 “Linker script for the monitor” and the linker output in appendix G.4 “With compile-time options and linker script.” As seen, the linker script instructs the linker to place selected sections in one of three main sections, 'text', 'data' or 'bss'. Sections not of interest are discarded. The resulting executable is still in Elf32 format, but with minimal sections, only 7 out of 22. The linker scripts give a powerful tool in creating executables and makes life a lot easier when converting formats.

The final step to create a a.out file from the minimum section count Elf32 file we use the “elf2aout” utility. Showing the usage of the “elf2out” utility in figure 5.18,

```
"Usage: elf32aout [-v] [-S stacksize] src dst;"
"  Need at least two file names as parameters."
"  -v,          verbose, info and summary about the result a.out."
"  -S n[MKwb],  set the stack to a size, see \"man install\"."
"  src,         the source file, elf(32) file to convert."
"  dst,         the destination file, if it exists it will be
               overwritten.");
```

Figure 5.18: *Usage “elf2out” program.*

The '-S' argument has the same context of the MINIX “install -S” argument. The '-v' argument will print a summary of the created a.out executable file. Converting to an a.out file is now a “piece of cake.” All we have to do is create a new file which starts with an a.out header, fill in the offsets and sizes and we are done.

Figure 5.19 on the next page is a typical printout (using the argument -v) of the “elf2aout” utility. Note that “total:” does not mean the total size of the file or sections created, but the total size of the 'data', 'bss' and 'stack' sections. At the moment the utility doesn't include the symbol table and valid extended a.out members, although the maximum size a.out header is used creating new file.

The program converts the Elf32 executable file *<./commands/ash/elf32/ash>* to the file *</minix/bin/sh>* in a.out format. Most will recognize that it's the (default) shell used by the MINIX operating system. All programs and utilities used by MinixPPC have to be converted like this, most of the time right to their intermediate location on the host file system.

Note the argument '-S' defining the stack size of 100 KB, not all programs or utilities need the same stack size. The makefiles in the <*./commands/simple/**> directories contain lines calling "elf32aout" with a reasonable stack size per program.

Elf2aout in action,

```
elf2aout -v -S 100k elf32/ash /minix/bin/sh
Elf(32) to a.out binary file convertor, compiled on "Jun 21 2006,
16:02:25" version 0.1
Loaded source file "elf32/ash" size 224282 bytes, using destination
file "/minix/bin/sh".
Stack updated.
Found the following sizes (bytes) for the a.out sections,
----------------------------------------------------------
  text size:  0x1f7f8 (125 KB)
  data size:  0x3000 (12 KB)
  bss size:   0xd1c (3 KB)

  total:      0x1cd1c (data + bss + stack) (115 KB)
  stack size: 0x19000 (100 KB)
  entry:      0x0
  filesize:   0x22828 (138 KB)
----------------------------------------------------------
```

Figure 5.19: *Converting ash from elf32 to a.out using a stack size of 100 KB.*

When developing the converter program a small problem was found, due to the limitations of the a.out format. The Elf32 format uses load addresses and section sizes to define how segments must be created for every section. The load address and the memory addresses in the section should match the compiled addresses.

For example you could define a load address for the text section at 0x1000_0000 and for the data section at 0x2000_0000. Now we know that we need to map a segment to virtual address 0x1000_0000 to include the text and one for the data section at virtual address 0x2000_0000. The default a.out format does not support this, it considers all addresses compiled relative from the addresses compiled in the text section (most of the time text is compiled from zero).

The problem would occur when the compiler would align the start of the bss section. Effectively creating a gap between the data section end and start of the bss. In the a.out format these two sections must "touch" each other. To make things more complicated, the compiler did not always align on the same multiple, sometimes 0x10 or 0x100.

A (fast) solution was found in taking a "big" enough multiple (0x1000) and use it for alignment and "simulate" the data size to the alignment point. This way the data and bss section concatenate and can be loaded in the same segment without any tricks. The only artefact is a data size that's always a multiple of the click size (0x1000). When creating the system image with the "mkimage" program, it shows in the output at the second column.

5.9.2 mkimage

The “mkimage” program is used to create the system image, containing the kernel, servers and drivers needed to boot MinixPPC. The kernel contains the tasks, IDLE, CLOCK, SYSTEM and HARDWARE (also known as KERNEL). For MinixPPC the servers include the process manager, file system, reincarnation server, data storage server, debug server and INIT process. The drivers needed are memory, log, TTY, macio, pcim and AT_WINI.

The “mkimage” program can take executables in Elf32 and a.out format as arguments. Giving a Elf32 executable it will use a default stack size of 64 KB when converting to a a.out format executable. The sequence in which the processes are given as arguments determines there place in the image. This sequence is kept by the monitor when loading the processes to memory. Make sure the sequence and process numbers are the same as defined in the <*./minix/kernel/arch/table.c*> file. To change the stack size of a process convert it to a.out format first using “elf2aout.”

The workings of “mkimage” are straightforward, it will create the image by the format described in chapter 5.2.1 “Image format.” The typical output of a system image build for MinixPPC, given the next command line,

```
1. ./mkimage -image minixppc \
   /home/i2a/minix/kernel/aout/kernel \
   /home/i2a/minix/servers/pm/aout/pm \
   /home/i2a/minix/servers/fs/aout/fs \
   /home/i2a/minix/servers/rs/aout/rs \
   /home/i2a/minix/drivers/memory/aout/memory \
   /home/i2a/minix/drivers/log/aout/log \
   /home/i2a/minix/drivers/arch/ppc/tty/aout/tty \
   /home/i2a/minix/drivers/arch/ppc/macio/aout/macio \
   /home/i2a/minix/drivers/arch/ppc/pcim/aout/pcim \
   /home/i2a/minix/drivers/arch/ppc/at_wini/aout/at_wini \
   /home/i2a/minix/servers/ds/aout/ds \
   /home/i2a/minix/servers/dbg/aout/dbg \
   /home/i2a/minix/servers/init/aout/init
```

output:

```
1. Make image (mkimage.c), compiled on "Jul 30 2006, 11:13:51" \
2. version 0.1
3.
4. Making image form the following files,
5.   /home/i2a/minix/kernel/aout/kernel, a.out process.
6.   /home/i2a/minix/servers/pm/aout/pm, a.out process.
7.   /home/i2a/minix/servers/fs/aout/fs, a.out process.
8.   /home/i2a/minix/servers/rs/aout/rs, a.out process.
9.   /home/i2a/minix/drivers/memory/aout/memory, a.out process.
10.  /home/i2a/minix/drivers/log/aout/log, a.out process.
11.  /home/i2a/minix/drivers/arch/ppc/tty/aout/tty, a.out process.
12.  /home/i2a/minix/drivers/arch/ppc/macio/aout/macio, a.out process.
13.  /home/i2a/minix/drivers/arch/ppc/pcim/aout/pcim, a.out process.
14.  /home/i2a/minix/drivers/arch/ppc/at_wini/aout/at_wini, a.out process.
15.  /home/i2a/minix/servers/ds/aout/ds, a.out process.
```

(***Listing 5.20:*** *continued on the next page*)

```
16.   /home/i2a/minix/servers/dbg/aout/dbg, a.out process.
17.   /home/i2a/minix/servers/init/aout/init, a.out process.
18.
19. Image file "minixppc" created, 13 processes added size of process \
20.headers 112(0x70) bytes.
21. Section sizes in bytes(0x),
22.      text      data       bss     stack     total
23. --------  --------  --------  --------  ---------
24.     1e2e1      2000     1cac8         0     3cda9  kernel(0)
25.      b104      2000     11094     20000     3e198  pm(0)
26.     13584      2000    4c2bac     42400    51a530  fs(0)
27.      6654      1000      50c0     10000     1c714  rs(0)
28.      68cc      1000       c70      8000     1053c  memory(0)
29.      6ccc      1000      f7d8      8000     1f4a4  log(0)
30.      aa74      3000       6f4      8000     16168  tty(0)
31.      84f8      2000       298      8000     12790  macio(0)
32.      74a4     14000      1064      4000     20508  pcim(0)
33.      a844      2000      8e5c      8000     1d6a0  at_wini(0)
34.      5348      1000       7b8      4000      ab00  ds(0)
35.      7710      2000       4e4      4000      dbf4  dbg(0)
36.      2ec0      1000       568     10000     14428  init(0)
37. --------  --------  --------  --------  ---------
38.     84ac1     28000    511660     b6400    674521  total bytes to load.
39.
40. Image "minixppc" ready, kernel magic numbers,
41.   text 0xc0de4a11,
42.   data 0x0000526f.
43. Image filesize should be 708721(0xad071) bytes (692 KB).
44. End in memory when loaded from zero, 0x684000 (6672 KB).
45. Processes in image, 13
46. `minixppc' -> `/minix/boot/minixppc'
47. `minixppc' -> `minixppc_aout'
```

Listing 5.20: *Typical output from the "mkimage" program.*

In the output listing, the 'bss' section always starts at a page click. This makes the data size a multiple of 0x1000 (the click size). Because the data section (line 30, column two) of the PCI manager contains a "database" of PCI device vendors and models it grows "pretty" big. Note that all sizes are in hexadecimal notation and the entry points of the processes are between brackets after there names. The monitor also uses magic numbers to check if the kernel loaded is correct, these are listed at lines 41, 42.

5.9.3 mkffs

In UNIX like operating systems every I/O operation goes through the file system. At the *</dev/>* directory all used devices (nodes) have to be listed. Simply said, the device node from the device directory doing I/O with tells the file system process which driver to access. So to use MINIX, a file system needs to be created and accessible (with a filled device directory). This is done with the "mkffs" program. This program is able to create a prototype file system with files from the host file system. To do this it needs a file listing; the source file, owner and rights, and the target file in the MINIX file system.

```
boot
2560 768
d--755 0 0
  bin d--755 2 0
       cat          ---755 2 0 /minix/bin/cat
       chroot       ---755 2 0 /minix/bin/chroot
       getty        ---755 2 0 /minix/bin/getty
       install      ---755 0 0 /minix/bin/install
  $
  boot d--755 0 0
       minixppc     ---755 0 0 /minix/boot/minixppc
  $
$
```

Figure 5.20: *Snippet of the file system prototype file.*

A sample listing is given in figure 5.20, this is only a snippet from the prototype file system used for MinixPPC. The real prototype file used for MinixPPC contains many more files than listed here. This snippet defines a root directory with the directories *</bin/>* and *</boot/>*. The directory *</bin/>* contains several files, *<cat>*, *<chroot>*, *<getty>* and *<install>*. The *</boot/>* directory only contains the system image. The “mkffs” program will write a image of 2560 sectors and use 768 inodes for the file system. Every sector is 4096 bytes long so the image will be 10 MB in size. Taking the “cat” program as a example, “mkffs” will take the content of the *</minix/bin/cat>* file on the host FS and place it in *</bin/cat>* in the MINIX v3 file system created in the image. The file in the image will get '0755' as mode bits and '2' as group and '0' as owner. The complete file system listing is printed in appendix F “File system prototype file.” Make sure that the executable files are converted into a.out format and text files in UNIX format.

The “mkffs” program is a port of the MINIX “mkfs” program but runs under the GNU/Linux OS. Type names like “ino_t” and structure names like “struct dirent” are the same for both operating systems but there definitions worlds apart. To make “mkffs” possible the MINIX definitions are to be used. Therefore the header files containing the MINIX types and structures definitions are copied to the directory containing the source of the “mkffs” program and every MINIX type definition has “m_” prefixed. Normally these header files are located in the system include directory and are never redefined or come with the source of a program.

Next the command that could be used to create the file system image,

```
mkffs -l 10MB.img proto.fs
```

Command 5.1: *Create a 10 MB file system in the <10MB.img> file.*

The '-l' option lets the program output the file system content created to the 10 MB *<10MB.img>* image file and *<proto.fs>* contains the used prototype file system listing. The image file must be created before the “mkffs” program is used. It can be any file as long as the size is correct. Note that writing a new file system to the image file destroys the previous content. Any file created when running MinixPPC will be lost.

Next a possible command to create an empty 1 MB or 10 MB file usable as image,

```
dd if=/dev/zero of=1MB.img count=2048;  /* creates 1 MB */
dd if=/dev/zero of=1MB.img count=20480; /* creates 10 MB */
```

Command 5.2: *Create a 1 MB or 10 MB file containing only "zero's."*

The "dd" command read blocks of default size (512 bytes) so for 10 MB (1024 * 1024 * 10) 20480 reads are needed. Reading from the *</dev/zero>* device, reads zeros into the output file. This is the file used to contain the MINIX v3 proto file system. It must be installed to the MinixPPC boot device. The harddisk of the iBook contains 9 partitions,

```
/dev/hda
#                type name          length   base      ( size ) system
1 Apple_partition_map Apple            63 @ 1         ( 31.5k) Partition map
2 Apple_Bootstrap     bootstrap      1600 @ 64        (800.0k) NewWorld bootblock
3 Apple_Bootstrap     bootstrap      1600 @ 1664      (800.0k) NewWorld bootblock
4 Apple_UNIX_SVR2     Swap        1048576 @ 3264      (512.0M) Linux swap
5 Apple_UNIX_SVR2     LinuxRoot  10485760 @ 1051840   (  5.0G) Linux native
6 Apple_UNIX_SVR2     LinuxHome  20971520 @ 11537600  ( 10.0G) Linux native
7 Apple_UNIX_SVR2     LinuxUsr   31457280 @ 32509120  ( 15.0G) Linux native
8 Apple_UNIX_SVR2     MinixPPC    1024000 @ 63966400  (500.0M) Linux native
9 Apple_Free Extra               52219840 @ 64990400  ( 24.9G) Free space

Block size=512, Number of Blocks=117210240
DeviceType=0x0, DeviceId=0x0
```

Figure 5.21: *Current iBook partition table, using the "mac-fdisk" program.*

Figure 5.21 shows the macintosh partition map currently installed on the iBook. The "bootblock" system contains a special file system type that Open Firmware supports. Support has been included in the "AT_WINI" driver to open devices which are partitioned with a Mac partition map. As one can see in the figure 5.21, partition 8 will contain the MINIX v3 file system. It's device number is 0x308 or as special file *</dev/c0d0p8>*. The special file has been included in the prototype file system of course. Note that the original IBM PC partitioning "only" supports 4 primary partitions, the Mac partitioning supports 16.

To install the MINIX file system image the following command can be used,

```
dd if=10MB.img of=/dev/hda8 count=20480
```

Command 5.3: *Installing the MINIX file system.*

The same command as creating the "empty" file is be used to install the prototype file system to its partition. Command 5.3 installs the file *<10MB.img>* to partition 8 of the first harddisk. When the file system process initializes the root file system it uses the kernel environment set by the boot monitor to get the the root device number, for MinixPPC it is 776 (0x308). This way MinixPPC is able to load its root file system and it can access the files created with the "mkffs" program. Next chapter 6 "Compiling MinixPPC."

6

Compiling MinixPPC

The GNU compiler kit is used to compile the project. The main reason: it came with the GNU/Linux distribution installed on the iBook and it follows standards or can be forced to do so by giving compiler options. It is available for a large array of architectures so it could possibly be used in a future port. This could be a advantage when features like link scripts are needed. Besides the GNU/C compiler there will often be a manufacturers C compiler that could be used to compile MINIX.

To compile Makefiles are used so (GNU/)make is needed as well. The features used in the Makefiles are limited so other “make” programs should have no problem using the same files. Although the current makefiles are good enough to compile the project they should be improved in the release version of the software.

With the knowledge of previous chapters it should be easy to use the development environment created for MinixPPC, but the next few chapters provide additional information and the steps to compile and install the current system to the iBook.

6.1 How to compile the system

Almost all processes, utilities and commands have three “default” subdirectories, <*./misc/*>, <*./elf32/*> and <*./aout/*>. Every Makefile contains a linker command telling the linker to output symbol linkage to a file inside <*./misc/*> ending on <*.last.output>. The <*./elf32*> directory would contain the linker output from the Elf32 executable, with the link script and compiler options used to define sections. Sometimes the makefile has a rule that after compilation of the Elf32 file convert it to an a.out executable to the <*./aout/*> directory. Compiling and installing MinixPPC takes about 8 steps, considering where changes are made and new data have to be installed,

1. Compiling the libraries, stdlibs, syslibs, ... ,
2. The commands, including the shell, ... ,
3. The server processes, PM, FS, ... ,
4. The drivers, AT_WINI, Memory, PCIM, ... ,
5. The kernel process and architectural library,
6. Create the system image,
7. Install the system image,
8. Install the MINIX file system.

Listing 6.1: *Steps for building the MinixPPC system.*

Some steps could be done in reverse sequence but the libraries need to be built first. Most of the time the makefiles would catch this and issue a dependable “make” to create the dependency. Start by building the libraries, by issuing a “make” in the <*./lib/*> directory after that every other part can be build.

Commands are located in the <*./command*> directory. The simple commands are divided over the location they get in the MINIX file system. This means certain executables are located in <*./simple/bin/*> and others in <*./simple/usr/bin/*>. Every command build will be converted to a.out and placed in the host prototype file system. That is used by the “mkffs” program when it reads the prototype file system definition file. All server processes are located in the <*./server/*> directory, typing “make all” there would build all server processes. The same goes for the drivers, these are located in <*./drivers/*> and “make ppc” will create them. The kernel makefile is located in the <*./kernel/*> directory making this would create the kernel process and if the 'arch.a' library isn't created it will build that first.

Typing “make” in <*./image/*> creates the system image. It only executes the “mkimage” program that creates the <*minixppc*> image file from the processes. It copies this file in the <*/boot/*> directory of the prototype file system. You need “root” rights to copy the image to the </boot/> directory of the host file system (Linux, ReiserFS). The monitor expects the MinixPPC image to be there (see <./drivers/arch/ppc/monitor/second/monitor.c> line 37).

The install and creation of the file system was handled in the previous chapter 5.9.3 ‟mkffs.” All that is left is a reboot and selecting MINIX when the monitor comes up.

6.2 Link scripts

Link script were mentioned earlier in the chapter about Elf32 to a.out format conversion. The focus there was on the sections; here on other keywords used are handled. The default location of link scripts is the <*./lds/*> directory. It contains a link scripts to compile the kernel, (runtime) executables or the monitor program.

The content of the script directory,

1	<*default.lds*>	Used in linking a executable for the MINIX OS. Also used to link the server and driver processes.
2	<*kernel.lds*>	Used in linking the kernel process.
3	<*monitor.lds*>	Used to link the monitor program, note that this is the only program needed to stay in Elf32 format.

Listing 6.2: *Linker scripts used by the MinixPPC system.*

Every file in listing 6.2 contains a line containing 'ENTRY(...)'. This defines the entry value in the executable header, the start of the program. The kernel has as entry symbol 'MINIX, the monitor '_start' and the default linker script 'crtso'. The 'MINIX symbol is loc-

ated at the start of the <*./kernel/arch/ppc/minix.S*> file. Because the kernel is always linked with the <*minix.o*> as first object this symbol will always be at address 0x0. The monitor '_start' symbol is located in the <*./drivers/arch/ppc/monitor/crt0.S*> file, this does not have to be linked at (the relative) address 0x0.

The default link script defines the code entry point to 'crtso'. It is located in the bootstrap code to run the program. The real start of a program is most of the time not the "main( )" function as a beginner programmer might think, but a little piece of bootstrap code that prepares the start of the program, "argc" and "argv" must mean something. In the end it's the bootstrap code that will call the "main( )" function with valid arguments. This bootstrap code is also responsible to call the program clean up function "exit( )", if the main function should return. The default script forces the bootstrap object to be linked before the rest of the objects of the program.

Except for the <*monitor.lds*> file the data section is always compiled aligned to 0x1000_0000, with the line ".data ALIGN(0x10000000) : {." This effectively forces the data section to be loaded in the segment after the text segment. The 'bss' section is compiled directly behind the data section, no segment offset is needed for it.

The "GNU Compiler Kit" can be set up to use default linker scripts specially created for the target system. When GCC is ported to MinixPPC these scripts should be placed in the "default" linker script directory of GCC. Issuing a command like "gcc -c hallo.c -o hallo" should default to use 'default.lds' as linker script.

6.3 Debugging

At the moment MinixPPC is still in development. The code is littered with debug statements and warnings. To make general debugging easy there is a header file created to include in the file you want to debug. This file is <*warning.h*> and located at <*./arch/ppc/*>, the architectural include. The file contains macro's that need to be supported by the compiler. As MinixPPC can only be compiled (at the moment) by the GCC compiler we can be sure the file is usable, but then it needs to be in the architectural include. To use the macros use the following construct in you source file,

```
1. #define ENABLE_DEBUG
2. #define ENABLE_WARNINGS
3. #define PRINTER printf
4. #include <warning.h>
5.
6. /* ... */
7. char some_function( void ) {
8.   char c = 'A';
9.
10.  if( read(console, &c, 1) != 1) {
11.    warning("error on read, \"%s\".\n", strerror(errno));
12.  }
13.
14.  debug("returning character %c.\n", c);
15.  return c;
16. }
```

(***Listing 6.3:*** *continued on the next page*)

Output would look like:

```
Warning (file.c, 27), "some_function()” error on read, “error_text”.
Debug (file.c, 30), “some_function()” returning character A.
```

Listing 6.3: *Using the debug and warning macro's.*

The <*warning.h*> file defines two macro's that look like functions, “debug( )” and “warning( ).” The macros are to be used like the “printf( )” function. The macros will print the file name, line number, function name and then the output line from where the macro is used. The debug and warning macro are essentially the same but can be switched off by different “#define” directives. To remove the use of the macros just remove or undefine the “ENABLE_DEBUG” or “ENABLE_WARNINGS” definitions at lines 1 and 2. The macros would then become empty.

Care must be taken when using the macros as programming errors could occur when they “disappear.” If for example in listing 6.3 the brackets for the if statement on line 10, where omitted the code would work “correct” with warnings defined but when warnings (and debug) are undefined the function would only return the character on a read error.

It is also possible to define the “printer” function that the macros use. Sometimes to “printf” but most of the time it would be “kprintf” as servers and drivers are still in development stage. Not defining “PRINTER” would default to “kprintf.”

The current version of the MinixPPC kernel contains a little “screen driver” that is able to write to the bitmap buffer of the video device. It is called “kscreen” and located in the PPC driver directory as <*kscreen.a*>. It uses the bottom half of the screen to dump diagnostic messages of the kernel itself. Using “kprintf” inside the kernel code writes to the bottom half of the screen. Removing it is simple, start with the initialization from the <*minix.S*> file, make the rewire of the “kputc( )” function in the file <*./kernel/utility.c*> to the TTY (or log server) and update the TTY driver to use the complete screen.

As mentioned earlier the kernel symbol listing in appendix H “Kernel symbol listing” can be used to find the start of functions. The exception “reports” come with the address of the instruction causing it, the listing can give a indication where in the program the problem is occurring.

Chapter 7 “Aftermath” sums some of the known issues of MinixPPC, it could be that when this document is read some of the issues in the table are solved. A small conclusion is made and examples of other kernels listed.

7

Aftermath

7.1 Other examples

The XNU (Darwin) kernel is maintained by Apple developers and is programmed in C and object C. The object C is used for the I/O kit while the kernel core is in C. Drivers writing in object C are rewritten quicker by reusing code. Much like the goal of the introduced driver model.

7.2 Conclusion

Still a lot of work needs to be done. It has been a one person project to get MinixPPC to run. This is almost inevitable as the (first) problems are very specific and most of the time caused by a chain reaction of programming errors.

The programming model introduced has strong bonds with object oriented programming; this is reasonable as both isolate code to purpose.

The kernel hardware interface is in fact the lowest possible point to get hardware abstraction from the system software, although one or two places exists that need special attention. This is providing the kernel base so any architecture can be supported by MINIX.

7.3 Known issues

There are some known problems and missing support at the moment that did not get fixed or updated to the project within the time period.
Here is a list of known issues (continued on the next pages),

Problem	*Description*
sync( ); call	This call simply doesn't work. Although some debugging is under way, the problem wasn't found.

(***Figure 7.1:*** *continued on the next page*)

Problem	*Description*
PCI manager	It does not find all devices in the system. The system is known to have a network controller, but it is not found when scanning the PCI bus. The used scanning algorithm is very basic and doesn't treat PCI bridges specially. The problem should be local to the scan algorithm of the PCI manager.
Missing support	Not all MINIX services are started, reincarnation server and log server aren't working yet. There should be no problem turning them on. Reincarnation must be debugged first (which should be possible).
New drivers	These are not well tested and don't support all MINIX stability functions. Including “alive” ping and reincarnation. They should mature fast though.
Memory management	The current memory management doesn't allow more that one valid PTE per segment ID. This could be a restriction, but otherwise saves memory mapping from the need to unmap pages to prevent double mapping.
Makefiles	Current Makefiles need an overhaul. Full dependency checking needs to be implemented. For example changes to header files aren't noticed.
RC script	When MinixPPC boots there are several programs run by the *</etc/rc>* script. At the moment not all services are supported.
Argument lists	The standard library is updated to use the “va_list” type instead of “char*” type.
Monitor return (or kernel exit)	Is not supported for MinixPPC, although a attempt can be made by using the exception vector space offset flag in the MSR.
Date and time	Time management is not ready. Uptime is kept but the “time” starts with zero. We need NVRAM access to complete the time functionality.
MacIO	It happened ones or twice during development of the MacIO driver that when the driver hanged, the system keyboard would not react on reboot. The exact problem, ADB, PMU or VIA failure is unknown. But it could take a couple of resets and some time between them for the keyboard to get back to live. With the current version of the MacIO driver no problems occur and it is “rock” solid.
Supported hardware	It is possible that MinixPPC run on other iBook like systems as the Mac Power Book. They uses Open Firmware v3, so the monitor should be compatible. This is not tested though and it is possible that “default” values of drivers are off.

Problem	*Description*
Kernel printer	The kernel print formatting was done by a stand alone kprintf function with limited formatting options at <*/lib/sysutil/*>. For MinixPPC the “kprintf” function has been “rerouted” to use the stdio “doprnt” function (using vsprintf).
Compiler version(s)	Currently two compilers versions of GNU/gcc are installed on the iBook G4. Using “gcc-config -l” one can view the installed versions and set up the default configuration. Recently (27/07) the default configuration is switched to gcc version 4.x.x. Using it to compile MinixPPC has no success, the code generated is unstable, and some “new” warnings occur. Keep to version 3.4.6 until it is clear why this happens. To make sure gcc version 3.4.6 is used the Makefiles are updated to use the command “gcc-3.4.6” instead of “gcc.”
Process numbers at com.h	The boot process numbers are defined in the file <*minix/include/minix/com.h*>. This introduces a problem, MinixPPC used the MacIO driver and so its defined in <*com.h*>. Thus the “system include” contains a system dependency.

Figure 7.1: *Known issues.*

A

Code Listings

A.1 Exception phase1

```
1. # Phase1:
2. #  - free-up R1 and R2,
3. #  - restore kernel segment descriptors,
4. #  - load kernel machine status word (enabling translation),
5. #  - load the re-enter count (needed the translation for that!),
6. #  - figure out where to save the context, in the process table
   or on
7. #    the (kernel) stack,
8. #  - update re-enter count,
9. #  - jump to the needed handler, for a stack save or a process
   save.
10.#
11.# Phase 1 exception c0de.
12.#
13.#define EXCEPTION_PHASE1(ev, handler)       \
14.      . = ev;                              \
15.      mtspr     SPRG1, R1;                 \
16.      mtspr     SPRG2, R2;                 \
17.      SEGMENT_REGISTERS_RESTORE_KERNEL(R1); \
18.      _LV32(R1, KERNEL_PSW);               \
19.      mtmsr     R1;                        \
20.      sync;                                \
21.      isync;                               \
22.      _LW(R1, k_reenter);                  \
23.      addi      R1, R1, 1;                 \
24.      _SW(k_reenter, R1, R2);              \
25.      cmpwi     R1, 0;                     \
26.      mfspr     R1, SPRG1;                 \
27.      mfspr     R2, SPRG2;                 \
28.      beq       handler##_table;           \
29.      b         handler##_stack
```

A.2 Exception phase2

```
1. # Phase2:
2. #  - save the current cpu state, to the stack (leaving SPRG3
   intact) or
3. #    process table,
4. #  - take the fresh kernel stack,
5. #  - handle exception or interrupt,
6. #  - when returning from the kernel stack just restore the
   stackframe,
7. #    clean the stack and return (to the kernel),
8. #  - otherwise "reset/switch" to a new process and go running it.
9. #
10.# Phase 2 exception c0de.
11.#
12.#define EXCEPTION_PHASE2(handler, chandler) \
13.handler##_stack:;                           \
14.      _GROW(STACKFRAME_SIZE);               \
15.      mtspr     SPRG2, SP;                  \
16.      SAVE_CPU(SPRG2, SPRG1);               \
17.      _GROW(FRAME0_SIZE);                   \
18.      bl        chandler;                   \
19.      _SHRINK(FRAME0_SIZE);                 \
20.      _LW(R1, k_reenter);                   \
21.      addi      R1, R1, -1;                 \
22.      _SW(k_reenter, R1, R2);               \
23.      li        R2, STACKFRAME_SIZE;        \
24.      RESTORE_CPU(SPRG2, R2);               \
25.      _SHRINK(STACKFRAME_SIZE);             \
26.      rfi;                                  \
27.handler##_table:;                           \
28.      SAVE_CPU(SPRG3, SPRG2);               \
29.      _LA(SP, kstk_bottom);                 \
30.      bl        chandler;                   \
31.      b         restart
```

A.3 Definition PCI device

The data structure passed to the driver for a PCI device. This information could be extended by simply updating the PCI device query function used by the PCI manager.

The PCI device information is defined in the <*./libpci/*> directory in the drivers directory by the file <*pci_device.h*>.

```
1. /* The PCI device typedef.
2.  */
3. typedef struct pci_device_s pci_device_t;
4.
5. struct pci_device_s {
6.   int nr;                 /* device number */
7.   int enabled;            /* enabled or disabled */
8.   int mastering;          /* enabled or disabled */
9.
10.  pci_tag_t tag;          /* device tag [bus/dev_nr/function] */
11.  pci_reg_t r_vp;         /* vendor/product id (raw cpy of register)
   */
12.  pci_reg_t r_class;      /* device class  (raw cpy of register) */
13.
14.  u8_t  revision;         /* */
15.  u8_t  interface;        /* */
16.  u8_t  class;            /* device PCI base class number */
17.  u8_t  subclass;         /* device PCI sub-class number */
18.  u8_t  function;         /* function number */
19.
20.  char  interrupt_pin;      /* 'A','B','C','D' or zero */
21.  int   interrupt_line;
22.
23.  /* The info for the BARs needed to communicate to the device.
24.   * Gathered from the PCI controller when querying the
25.   * device (pci_query_device.c).
26.   */
27.  pci_device_bar_t bar_io[MAX_PCI_DEVICE_BAR_IO];
28.  int bar_io_count;
29.  pci_device_bar_t bar_mem[MAX_PCI_DEVICE_BAR_MEM];
30.  int bar_mem_count;
31.
32.  pci_device_bar_t rom;
33.  int rom_enabled;
34.};
```

B

Bibliography

Documentation and references used in this project, most of these documents are in electronic form but included (among other documents) on the projects CD-ROM. The documents are found at the project directory in the <*CoreDoc/*> subdirectory.

Mainly used documentation,

[1] SunSoft,
SYSTEM V Application Binary Interface, PowerPC Processor Supplement,
September 1995

[2] Tool Interface Standards (TIS), Portable Format Specification version 1.1,
Executable and linkage format

[3] Apple Computer,
Macintosh Technology in the Common Hardware Reference Platform II
1995 Apple Computer, Inc.

[4] Apple Computer,
Inside Macintosh, Chapter 5 ADB Manager

[5] Advanced Micro Devices and Cyrix Corporation,
The Open Programmable Interrupt Controller (PIC) Register Interface Specification
Revision 1.2
Issue Date: October 1995

[6] Apple Computer, Technical Notes,
TN2001, Running files from a hard drive in Open Firmware
TN1167, The Mac ROM Enters a New World
TN1061, Fundamentals of Open Firmware, Part I: The User Interface
TN1062, Fundamentals of Open Firmware, Part II: The Device Tree

[7] Jorit N. Herder,
Towards a True Microkernel Operating System
February 23, 2005

[8] Andrew S. Tanenbaum and Albert S. Woodhull. Operating Systems
Design and Implementation. Prentice-Hall, Upper Saddle River, NJ
07458, USA, third edition, 2006.

[9] IBM,
PowerPC™ Microprocessor Family, The Programming Environments for 32-Bit Microprocessors, February 2000

[10] IBM,
PowerPC User Instruction Set Architecture, Book I
Version 2.01, September 2003

[11] IBM,
PowerPC Virtual Environment Architecture, Book II
Version 2.01, December 2003

[12] IBM,
PowerPC Operating Environment Architecture, Book III
Version 2.01, December 2003

[13] Yaboot loader howto,
Version 1.04, Feb 22 2004
Chris Tillman

C

Enhancements

At the moment MINIX (MinixPPC) has to know the content of the system image before booting. The content and more information is “recorded” while compiling the kernel process via the boot image table in *<./kernel/arch/xxx/table.c>*. It should be possible to free MINIX from this.

The fixed image table is needed for the definition of the process numbers and masks. For example, all processes need to use the same process number to send to the file system server. One solution is saving the information “normally” located in the boot image into the process headers used when creating the image. The monitor would load the processes into memory in the right order sets a process header instead of a a.out array accordingly.

The kernel would then start without the boot image table but knows the number and sizes of the processes from the process header array array.

Getting MINIX to support (basic) Elf32 format executables would enable developers to use current compilers without the need to convert to the a.out format first. A recompile of most ”public” programs should be enough.

Freeing the stack from the same segment as the data (and bss) segment enables MINIX to detect stack overflow, which increases the security of the system. This includes updates to the memory (dynamic) allocation functions in the PM as well.

Getting the CDROM drive to work, this should be a trivial job and would be a great way to start MinixPPC to use its own file system.

The TTY driver could have a second “slave” process for driving the screen. This process it then the actual screen driver. A standard interface (using the IPC) to write characters and bitmaps would create system independency, although performance could become a issues.

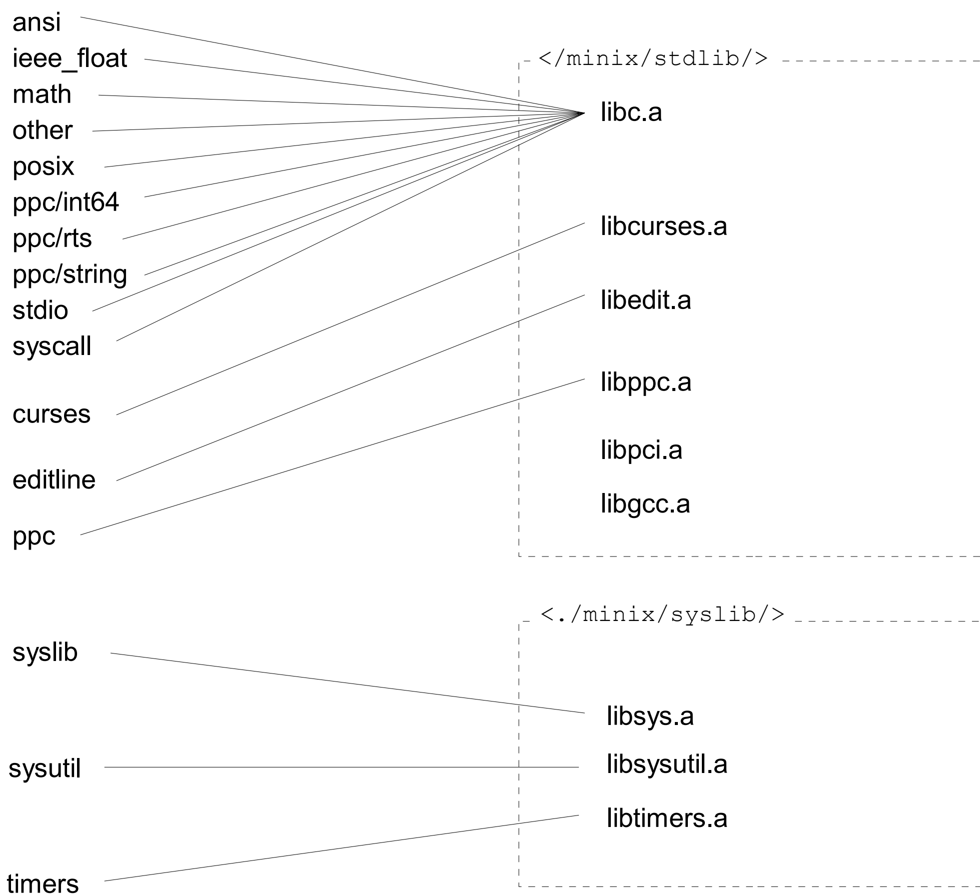
D

Library Notes

The libraries used by MinixPPC are located in the <*minix/lib/*> directory. All directories archive to a new library, or contain the part of a "bigger" library.

The libraries are created to two directories, <*minix/stdlib/*> or <*minix/syslib/*>. The standard lib directory contains the libraries needed to create a program not using special OS features like timers, system utilities or the system library. This directory also contains libraries needed when linking drivers that use the PCI manager and code using bigger numbers (64 bit). At the moment the source for these is located in the driver directory (except for <*libgcc.a*>).

Directories and the library they build (to),



E

Kernel Files

E.1 Missing symbols

Missing symbols after removal of "obvious" system dependent code from existing files.

./kernel/original file [v3.1.1]	***Missing symbols or changes after removal of system dependent code.***
clock.c	1. function `outb', 2. `TIMER_MODE' undeclared, 3. `TIMER0' undeclared, 4. `PORT_B' undeclared, 5. function `inb'. **6. COUNTER_FREQ** **7. LATCH_COUNT** **8. SQUARE_WAVE** **9. TIMER_COUNT** **10. TIMER_FREQ** **11. CLOCK_ACK_BIT**
debug.c	"Completely" MIC, unchanged.
exception.c	"Completely" MDC, move to <*./kernel/arch/xxx/*>.
i8259.c	1. function `intr_disable', 2. function `outb', 3. `INT_CTL' undeclared, 4. `INT_CTLMASK' undeclared, 5. `IRQ0_VECTOR' undeclared, 6. `BIOS_IRQ0_VEC' undeclared, 7. `CASCADE_IRQ' undeclared, 8. `INT2_CTL' undeclared, 9. `INT2_CTLMASK' undeclared, 10. `IRQ8_VECTOR' undeclared, 11. `BIOS_IRQ8_VEC' undeclared, 12. `NR_IRQ_VECTORS' undeclared, 13. `irq_handlers' undeclared. I8259 is renamed to 'interrupt.c'.

./kernel/original file [v3.1.1]	***Missing symbols or changes after removal of system dependent code.***
main.c	1. \`INT_CTLMASK' undeclared, 2. \`INT2_CTLMASK' undeclared, 3. function \`level0', 4. \`monitor' undeclared, 5. \`STOP_MEM_CHECK' undeclared, 6. \`SOFT_RESET_FLAG_ADDR' undeclared, 7. \`SOFT_RESET_FLAG_SIZE' undeclared.
proc.c	"Completely" MIC, except for the (new) copy message function does not contain MDC.
protect.c	"Completely" MDC, move to *<./kernel/arch/xxx/>*.
start.c	Almost MIC. The MINIX monitor support is removed. It looks inherent to IBM PC architecture.
system.c	1. \`BIOS_MEM_BEGIN' undeclared, 2. \`BIOS_MEM_END' undeclared, 3. \`BASE_MEM_TOP' undeclared, 4. \`UPPER_MEM_END' undeclared. The PowerPC does not use a/the "BIOS" like the way the IBM PC compatible does. So these are system dependent, and can't be supported here.
table.c	Contains the system image build-up, the image is not the same for every architecture so move to *<./kernel/arch/xxx/>*. Example, the x86 image would not a MacIO process.
utility.c	"Completely" MIC (almost unchanged).

E.2 PPC architecture files

PowerPC only files.

<./kernel/arch/ppc/> [MDC] ***owned by the <arch.a> library.***	***Content description***
Makefile	The "make" file that creates the library (arch.a) archive from the files in this directory.
arch.a	The interface functions needed by the kernel (four) + the table with the system image definition.
clock.c	Providing the implementation of the clock interface, Clock. Here the PowerPC hardware is set and used to form the functionality that is needed for the periodic interrupt needed by MINIX to do process switching.
exception.c	The exception handling functions, giving error messages and ending with signalling the process.
interrupt.c	Providing the implementation of the interrupt interface, Interrupt. The OpenPIC interrupt controller is initialized and managed from this file. It will set default priorities and vectors for source lines on initialization. Functions to read and acknowledge which interrupt has occurred are included.
klibppc.c	General PowerPC functions (only used in files at this directory).
memory.c	Providing the implementation of the memory interface, Memory. Here all special memory functions are provided, allocation, forced copying, translation and I/O.
minix.S	The first MINIX file, providing the context switching code and kernel start-up.
system.c	Providing the implementation of the system interface, System. Accessing system features like a driver.
table.c	Defining the kernel constants and system image. The image contains system dependent drivers so it must be at the architectural root. (maybe dynamic system image loading should help).

F

File System Prototype File

The file used to create the prototype file system for MinixPPC. It is located in the *<minix/fs.img/>* directory and used by the “mkffs” program. It list every file currently available for the MinixPPC system. All programs listed compile without problems, but not all are tested.

This file is for a 10 MB image file,

```
boot
2560 768
d--755 0 0
  bin d--755 2 0
      cat         ---755 2 0 /minix/bin/cat
      chroot      ---755 2 0 /minix/bin/chroot
      cp          ---755 2 0 /minix/bin/cp
      date        ---755 2 0 /minix/bin/date
      dev2name    ---755 2 0 /minix/bin/dev2name
      first       ---755 2 0 /minix/bin/first
      fsck        ---755 2 0 /minix/bin/fsck
      getty       ---755 2 0 /minix/bin/getty
      install     ---755 0 0 /minix/bin/install
      mined       ---755 2 0 /minix/bin/mined
      mount       ---755 0 0 /minix/bin/mount
      printroot   ---755 2 0 /minix/bin/printroot
      pwd         ---755 2 0 /minix/bin/pwd
      second      ---755 2 0 /minix/bin/second
      sed         ---755 2 0 /minix/bin/sed
      sh          ---755 2 0 /minix/bin/sh
      sync        ---755 2 0 /minix/bin/sync
      sysenv      ---755 2 0 /minix/bin/sysenv
      umount      ---755 0 0 /minix/bin/umount
      ln          ---755 2 0 /minix/bin/cp
      rm          ---755 2 0 /minix/bin/cp
      service     ---755 2 0 /minix/bin/service
  $
  boot d--755 0 0
      minixppc    ---755 0 0 /minix/boot/minixppc
  $
  dev d--755 0 0
      ram         b--555 0 8 1 0
      mem         c--555 0 8 1 1
      kmem        c--555 0 8 1 2
      null        c--555 0 0 1 3
      boot        b--555 0 8 1 4
      zero        c--555 0 8 1 5
      c0d0        b--555 0 0 3 0
```

```
        c0d0p1       b--555 0 0 3 1
        c0d0p2       b--555 0 0 3 2
        c0d0p3       b--555 0 0 3 3
        c0d0p4       b--555 0 0 3 4
        c0d0p5       b--555 0 0 3 5
        c0d0p6       b--555 0 0 3 6
        c0d0p7       b--555 0 0 3 7
        c0d0p8       b--555 0 0 3 8
        c0d0p9       b--555 0 0 3 9
        console      c--555 0 0 4 0
        tty          c--555 0 0 5 0
        lp           c--555 1 1 6 0
        log          c--555 0 0 4 15
        kbd0         c--555 0 0 4 250
        psm0         c--555 0 0 4 251
        ttyc1        c--555 0 0 4 1
        ttyc2        c--555 0 0 4 2
        ttyc3        c--555 0 0 4 3
        tty00        c--555 0 0 4 16
        tty01        c--555 0 0 4 17
        tty02        c--555 0 0 4 18
        tty03        c--555 0 0 4 19
        ttyp0        c--555 0 0 4 128
        ptyp0        c--555 0 0 4 192
        ttyp1        c--555 0 0 4 129
        ptyp1        c--555 0 0 4 193
        ttyp2        c--555 0 0 4 130
        ptyp2        c--555 0 0 4 194
        ttyp3        c--555 0 0 4 131
        ptyp3        c--555 0 0 4 195
        eth0         c--555 0 0 7 0
        ip0          c--555 0 0 7 1
        tcp0         c--555 0 0 7 2
        udp0         c--555 0 0 7 3
        eth          c--555 0 0 7 0
        ip           c--555 0 0 7 1
        tcp          c--555 0 0 7 2
        udp          c--555 0 0 7 3
        klog         c--555 0 0 15 0
        random       c--555 0 0 16 0
        urandom      c--555 0 0 16 0
        cmos         c--555 0 0 17 0
        rescue       b--555 0 0 9 0
    $
    etc d--755 0 0
        binary_sizes      ---755 0 0 /minix/etc/binary_sizes
        binary_sizes.big  ---755 0 0 /minix/etc/binary_sizes.big
        fstab             ---755 0 0 /minix/etc/fstab
        group             ---755 0 0 /minix/etc/group
        hostname.file     ---755 0 0 /minix/etc/hostname.file
        inet.conf         ---755 0 0 /minix/etc/inet.conf
        keymap            ---755 2 0 /minix/etc/keymap
        motd              ---755 0 0 /minix/etc/motd
        mtab              ---755 0 0 /minix/etc/mtab
        org.rc            ---755 0 0 /minix/etc/org.rc
        rc                ---755 0 0 /minix/etc/rc
        rc.cd             ---755 0 0 /minix/etc/rc.cd
```

```
            rc.rescue       ---755 0 0 /minix/etc/rc.rescue
            passwd          ---755 0 0 /minix/etc/passwd
            profile         ---755 0 0 /minix/etc/profile
            proto.fs        ---755 0 0 /minix/etc/proto.fs
            protocols       ---755 0 0 /minix/etc/protocols
            services        ---755 0 0 /minix/etc/services
            shadow          ---755 0 0 /minix/etc/shadow
            termcap         ---755 0 0 /minix/etc/termcap
            ttytab          ---755 0 0 /minix/etc/ttytab
            utmp            ---755 0 0 /minix/etc/utmp
            version         ---755 0 0 /minix/etc/version
        $
        home d--755 2 2
            ast d--755 3 0
            $
            bin d--755 2 2
            $
            i2a d--755 3 0
            $
            test d--755 0 0
                  t10a      ---755 0 0 /minix/home/test/t10a
                  t11a      ---755 0 0 /minix/home/test/t11a
                  t11b      ---755 0 0 /minix/home/test/t11b
                  test1     ---755 0 0 /minix/home/test/test1
                  test10    ---755 0 0 /minix/home/test/test10
                  test11    ---755 0 0 /minix/home/test/test11
                  test12    ---755 0 0 /minix/home/test/test12
                  test13    ---755 0 0 /minix/home/test/test13
                  test14    ---755 0 0 /minix/home/test/test14
                  test16    ---755 0 0 /minix/home/test/test16
                  test17    ---755 0 0 /minix/home/test/test17
                  test18    ---755 0 0 /minix/home/test/test18
                  test19    ---755 0 0 /minix/home/test/test19
                  test2     ---755 0 0 /minix/home/test/test2
                  test20    ---755 0 0 /minix/home/test/test20
                  test21    ---755 0 0 /minix/home/test/test21
                  test22    ---755 0 0 /minix/home/test/test22
                  test23    ---755 0 0 /minix/home/test/test23
                  test24    ---755 0 0 /minix/home/test/test24
                  test25    ---755 0 0 /minix/home/test/test25
                  test26    ---755 0 0 /minix/home/test/test26
                  test27    ---755 0 0 /minix/home/test/test27
                  test28    ---755 0 0 /minix/home/test/test28
                  test29    ---755 0 0 /minix/home/test/test29
                  test3     ---755 0 0 /minix/home/test/test3
                  test30    ---755 0 0 /minix/home/test/test30
                  test31    ---755 0 0 /minix/home/test/test31
                  test32    ---755 0 0 /minix/home/test/test32
                  test33    ---755 0 0 /minix/home/test/test33
                  test34    ---755 0 0 /minix/home/test/test34
                  test35    ---755 0 0 /minix/home/test/test35
                  test36    ---755 0 0 /minix/home/test/test36
                  test37    ---755 0 0 /minix/home/test/test37
                  test38    ---755 0 0 /minix/home/test/test38
                  test39    ---755 0 0 /minix/home/test/test39
                  test4     ---755 0 0 /minix/home/test/test4
                  test40    ---755 0 0 /minix/home/test/test40
```

```
                test5        ---755 0 0 /minix/home/test/test5
                test6        ---755 0 0 /minix/home/test/test6
                test7        ---755 0 0 /minix/home/test/test7
                test8        ---755 0 0 /minix/home/test/test8
                test9        ---755 0 0 /minix/home/test/test9
                testsh1.sh   ---755 0 0 /minix/home/test/testsh1.sh
                testsh2.sh   ---755 0 0 /minix/home/test/testsh2.sh
                run          ---755 0 0 /minix/home/test/run
          $
      $
      lib d--755 0 0
      $
      mnt d--755 0 0
      $
      root d--755 0 0
      $
      sbin d--755 2 0
          at_wini            ---755 2 0 /minix/sbin/at_wini
          log                ---755 2 0 /minix/sbin/log
          memory             ---755 2 0 /minix/sbin/memory
          tty                ---755 2 0 /minix/sbin/tty
      $
      tmp d--755 0 0
      $
      usr d--755 2 2
          bin d--755 2 1
                at           ---755 0 0 /minix/usr/bin/at
                banner       ---755 0 0 /minix/usr/bin/banner
                basename     ---755 0 0 /minix/usr/bin/basename
                cal          ---755 0 0 /minix/usr/bin/cal
                calendar     ---755 0 0 /minix/usr/bin/calendar
                cdiff        ---755 0 0 /minix/usr/bin/cdiff
                cgrep        ---755 0 0 /minix/usr/bin/cgrep
                chmem        ---755 0 0 /minix/usr/bin/chmem
                chmod        ---755 0 0 /minix/usr/bin/chmod
                chown        ---755 0 0 /minix/usr/bin/chown
                ci           ---755 0 0 /minix/usr/bin/ci
                cksum        ---755 0 0 /minix/usr/bin/cksum
                cleantmp     ---755 0 0 /minix/usr/bin/cleantmp
                cmp          ---755 0 0 /minix/usr/bin/cmp
                co           ---755 0 0 /minix/usr/bin/co
                comm         ---755 0 0 /minix/usr/bin/comm
                compress     ---755 0 0 /minix/usr/bin/compress
                cut          ---755 0 0 /minix/usr/bin/cut
                dd           ---755 0 0 /minix/usr/bin/dd
                dhrystone    ---755 0 0 /minix/usr/bin/dhrystone
                diff         ---755 0 0 /minix/usr/bin/diff
                dirname      ---755 0 0 /minix/usr/bin/dirname
                du           ---755 0 0 /minix/usr/bin/du
                ed           ---755 0 0 /minix/usr/bin/ed
                eject        ---755 0 0 /minix/usr/bin/eject
                env          ---755 0 0 /minix/usr/bin/env
                expand       ---755 0 0 /minix/usr/bin/expand
                fgrep        ---755 0 0 /minix/usr/bin/fgrep
                file         ---755 0 0 /minix/usr/bin/file
                find         ---755 0 0 /minix/usr/bin/find
                fix          ---755 0 0 /minix/usr/bin/fix
```

```
fold        ---755 0 0 /minix/usr/bin/fold
fortune     ---755 0 0 /minix/usr/bin/fortune
grep        ---755 0 0 /minix/usr/bin/grep
halt        ---755 0 0 /minix/usr/bin/halt
head        ---755 0 0 /minix/usr/bin/head
id          ---755 0 0 /minix/usr/bin/id
ifdef       ---755 0 0 /minix/usr/bin/ifdef
in.fingerd  ---755 0 0 /minix/usr/bin/in.fingerd
intr        ---755 0 0 /minix/usr/bin/intr
isoread     ---755 0 0 /minix/usr/bin/isoread
join        ---755 0 0 /minix/usr/bin/join
kill        ---755 0 0 /minix/usr/bin/kill
leave       ---755 0 0 /minix/usr/bin/leave
loadramdisk ---755 0 0 /minix/usr/bin/loadramdisk
login       ---755 0 0 /minix/usr/bin/login
look        ---755 0 0 /minix/usr/bin/look
lp          ---755 0 0 /minix/usr/bin/lp
lpd         ---755 0 0 /minix/usr/bin/lpd
ls          ---755 0 0 /minix/usr/bin/ls
man         ---755 0 0 /minix/usr/bin/man
mesg        ---755 0 0 /minix/usr/bin/mesg
mkdir       ---755 0 0 /minix/usr/bin/mkdir
mkfifo      ---755 0 0 /minix/usr/bin/mkfifo
mkfs        ---755 0 0 /minix/usr/bin/mkfs
mknod       ---755 0 0 /minix/usr/bin/mknod
mkproto     ---755 0 0 /minix/usr/bin/mkproto
modem       ---755 0 0 /minix/usr/bin/modem
mt          ---755 0 0 /minix/usr/bin/mt
newroot     ---755 0 0 /minix/usr/bin/newroot
nm          ---755 0 0 /minix/usr/bin/nm
od          ---755 0 0 /minix/usr/bin/od
origmkfs    ---755 0 0 /minix/usr/bin/origmkfs
passwd      ---755 0 0 /minix/usr/bin/passwd
paste       ---755 0 0 /minix/usr/bin/paste
pr          ---755 0 0 /minix/usr/bin/pr
prep        ---755 0 0 /minix/usr/bin/prep
printenv    ---755 0 0 /minix/usr/bin/printenv
printf      ---755 0 0 /minix/usr/bin/printf
progressbar ---755 0 0 /minix/usr/bin/progressbar
proto       ---755 0 0 /minix/usr/bin/proto
ps          ---755 0 0 /minix/usr/bin/ps
rawspeed    ---755 0 0 /minix/usr/bin/rawspeed
readall     ---755 0 0 /minix/usr/bin/readall
readfs      ---755 0 0 /minix/usr/bin/readfs
reboot      ---755 0 0 /minix/usr/bin/reboot
rev         ---755 0 0 /minix/usr/bin/rev
rmdir       ---755 0 0 /minix/usr/bin/rmdir
shar        ---755 0 0 /minix/usr/bin/shar
shutdown    ---755 0 0 /minix/usr/bin/shutdown
size        ---755 0 0 /minix/usr/bin/size
sleep       ---755 0 0 /minix/usr/bin/sleep
slip        ---755 0 0 /minix/usr/bin/slip
sort        ---755 0 0 /minix/usr/bin/sort
split       ---755 0 0 /minix/usr/bin/split
stat        ---755 0 0 /minix/usr/bin/stat
strings     ---755 0 0 /minix/usr/bin/strings
strip       ---755 0 0 /minix/usr/bin/strip
```

```
                stty         ---755 0 0 /minix/usr/bin/stty
                su           ---755 0 0 /minix/usr/bin/su
                sum          ---755 0 0 /minix/usr/bin/sum
                swapfs       ---755 0 0 /minix/usr/bin/swapfs
                tail         ---755 0 0 /minix/usr/bin/tail
                tar          ---755 0 0 /minix/usr/bin/tar
                tee          ---755 0 0 /minix/usr/bin/tee
                term         ---755 0 0 /minix/usr/bin/term
                tget         ---755 0 0 /minix/usr/bin/tget
                time         ---755 0 0 /minix/usr/bin/time
                touch        ---755 0 0 /minix/usr/bin/touch
                tr           ---755 0 0 /minix/usr/bin/tr
                tsort        ---755 0 0 /minix/usr/bin/tsort
                ttt          ---755 0 0 /minix/usr/bin/ttt
                tty          ---755 0 0 /minix/usr/bin/tty
                uname        ---755 0 0 /minix/usr/bin/uname
                unexpand     ---755 0 0 /minix/usr/bin/unexpand
                uniq         ---755 0 0 /minix/usr/bin/uniq
                update       ---755 0 0 /minix/usr/bin/update
                uud          ---755 0 0 /minix/usr/bin/uud
                uue          ---755 0 0 /minix/usr/bin/uue
                who          ---755 0 0 /minix/usr/bin/who
                whoami       ---755 0 0 /minix/usr/bin/whoami
                write        ---755 0 0 /minix/usr/bin/write
                yes          ---755 0 0 /minix/usr/bin/yes
                ps           --2755 2 8 /minix/usr/bin/ps
          $
          lib d--755 2 1
                pwdauth      ---755 0 0 /minix/usr/lib/pwdauth
          $
          sbin d--755 2 1
                dbg          ---755 2 0 /minix/usr/sbin/ds
                ds           ---755 2 0 /minix/usr/sbin/ds
                fs           ---755 2 0 /minix/usr/sbin/fs
                init         ---755 2 0 /minix/usr/sbin/init
                is           ---755 2 0 /minix/usr/sbin/is
                pm           ---755 2 0 /minix/usr/sbin/pm
                rs           ---755 2 0 /minix/usr/sbin/rs
          $
      $
      var d--755 0 0
      $
  $
```

G

Elf32 Section Listings

G.1 Elf32 section listing of the elf2out program

System (in our case Linux) default sections created in a Elf32 executable file.

```
There are 35 section headers, starting at offset 0x2f78:

Section Headers:
  [Nr] Name              Type         Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL         00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS     10000134 000134 00000d 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE         10000144 000144 000020 00   A  0   0  4
  [ 3] .hash             HASH         10000164 000164 0000b0 04   A  4   0  4
  [ 4] .dynsym           DYNSYM       10000214 000214 000190 10   A  5   1  4
  [ 5] .dynstr           STRTAB       100003a4 0003a4 0000f8 00   A  0   0  1
  [ 6] .gnu.version      VERSYM       1000049c 00049c 000032 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED      100004d0 0004d0 000030 00   A  5   1  4
  [ 8] .rela.dyn         RELA         10000500 000500 000018 0c   A  4   0  4
  [ 9] .rela.plt         RELA         10000518 000518 0000f0 0c   A  4  25  4
  [10] .init             PROGBITS     10000608 000608 000028 00  AX  0   0  4
  [11] .text             PROGBITS     10000630 000630 00177c 00  AX  0   0  4
  [12] .fini             PROGBITS     10001dac 001dac 000020 00  AX  0   0  4
  [13] .rodata           PROGBITS     10001dcc 001dcc 0009fc 00   A  0   0  4
  [14] .sdata2           PROGBITS     100027c8 0027c8 000000 00   A  0   0  4
  [15] .eh_frame         PROGBITS     100027c8 0027c8 000004 00   A  0   0  4
  [16] .ctors            PROGBITS     100127cc 0027cc 000008 00  WA  0   0  4
  [17] .dtors            PROGBITS     100127d4 0027d4 000008 00  WA  0   0  4
  [18] .jcr              PROGBITS     100127dc 0027dc 000004 00  WA  0   0  4
  [19] .got2             PROGBITS     100127e0 0027e0 000010 00  WA  0   0  1
  [20] .dynamic          DYNAMIC      100127f0 0027f0 0000c8 08  WA  5   0  4
  [21] .data             PROGBITS     100128b8 0028b8 000064 00  WA  0   0  4
  [22] .got              PROGBITS     1001291c 00291c 000014 04 WAX  0   0  4
  [23] .sdata            PROGBITS     10012930 002930 000000 00  WA  0   0  4
  [24] .sbss             NOBITS       10012930 002930 00000c 00  WA  0   0  4
  [25] .plt              NOBITS       1001293c 002930 000138 00 WAX  0   0  4
  [26] .bss              NOBITS       10012a74 002930 000004 00  WA  0   0  1
  [27] .comment          PROGBITS     00000000 002930 0001f0 00      0   0  1
  [28] .debug_aranges    PROGBITS     00000000 002b20 000058 00      0   0  8
  [29] .debug_info       PROGBITS     00000000 002b78 000164 00      0   0  1
  [30] .debug_abbrev     PROGBITS     00000000 002cdc 000020 00      0   0  1
  [31] .debug_line       PROGBITS     00000000 002cfc 000159 00      0   0  1
  [32] .shstrtab         STRTAB       00000000 002e55 000120 00      0   0  1
  [33] .symtab           SYMTAB       00000000 0034f0 000e50 10     34  67  4
  [34] .strtab           STRTAB       00000000 004340 000afe 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

G.2 With compile-time options

The monitor program uses no OS libraries and is free of sections created by compiling with different options.

Using the following compiler options for every source file,

```
-mpowerpc -m32 -O3 -Wall -Wstrict-prototypes -fomit-frame-pointer \
-Wno-trigraphs -ffreestanding -mno-sdata -fno-builtin -mno-altivec \
-static-libgcc
```

Created sections in the elf32 executable,

```
There are 22 section headers, starting at offset 0x19428:

Section Headers:
  [Nr] Name              Type       Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL       00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS   018000b4 0000b4 013be8 00  AX  0   0  4
  [ 2] .rodata           PROGBITS   01813c9c 013c9c 001503 00   A  0   0  4
  [ 3] .sdata2           PROGBITS   018151a0 0151a0 000000 00   A  0   0  4
  [ 4] .eh_frame         PROGBITS   018151a0 0151a0 0000a4 00   A  0   0  4
  [ 5] .got2             PROGBITS   01825244 015244 000000 00  WA  0   0  1
  [ 6] .data             PROGBITS   01825244 015244 0005b4 00  WA  0   0  4
  [ 7] .sdata            PROGBITS   018257f8 0157f8 000000 00  WA  0   0  4
  [ 8] .sbss             NOBITS     018257f8 0157f8 000008 00  WA  0   0  4
  [ 9] .bss              NOBITS     01825800 0157f8 009528 00  WA  0   0  8
  [10] .comment          PROGBITS   00000000 0157f8 0008a6 00      0   0  1
  [11] .debug_aranges    PROGBITS   00000000 01609e 000080 00      0   0  1
  [12] .debug_pubnames   PROGBITS   00000000 01611e 00007e 00      0   0  1
  [13] .debug_info       PROGBITS   00000000 01619c 001e8d 00      0   0  1
  [14] .debug_abbrev     PROGBITS   00000000 018029 00067f 00      0   0  1
  [15] .debug_line       PROGBITS   00000000 0186a8 000534 00      0   0  1
  [16] .debug_frame      PROGBITS   00000000 018bdc 0000cc 00      0   0  4
  [17] .debug_str        PROGBITS   00000000 018ca8 0002f2 01  MS  0   0  1
  [18] .debug_ranges     PROGBITS   00000000 018f9a 0003c0 00      0   0  1
  [19] .shstrtab         STRTAB     00000000 01935a 0000cd 00      0   0  1
  [20] .symtab           SYMTAB     00000000 019798 002280 10     21 267  4
  [21] .strtab             STRTAB   00000000 01ba18 001937 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

G.3 Linker script for the monitor

This is the monitor linker script file. It is only used when linking the executable. It clearly shows which sections the linker put together to form the final section. A you can see not only section layout can be defined in the script also the start and offset load addresses of the sections in memory.

Show is the content of the file <*minix/lds/monitor.lds*>,

```
OUTPUT_FORMAT("elf32-powerpc", "elf32-powerpc", "elf32-powerpc")
OUTPUT_ARCH(powerpc:common)
ENTRY(_start)
SEARCH_DIR("./");

/* Use this to make the elf header contain lesser sections.
 *
 * By putting (additional) "sections" in one of three sections
 * (text, data and bss) the elf header will only contain these
 * three sections. Note the order.
 *
 * See the bottom of the file for a elf32 section layout before
 * and after.
 *
 * Actually the forcing of lesser sections is not needed as the
 * monitor program stays in elf32 format and the OF interface
 * supports no a.out. But the limitation of sections can be
 * "checked" this way.
 *
 * The link script is still needed as we must compile the monitor
 * program "to" 10MiB. It needs to be loaded there to keep room
 * for the kernel and processes below it. Also we define the
 * entry address to the _start symbol.
 */

SECTIONS
{
/* Text is done normal but with the read-only stuff that the
 * compiler seams to produce, this includes constant string
 * definition and the like.
 */
  .text 0xa00000 : { /* the 10MiB offset */
    __text_start = .;
    *(.text);
    *(.sdata2);
    *(.rodata);
    *(.rodata.*);
    etext = .;
    _etext = .;
    __etext = .;
    __text_end = .;
  }
```

(continuing on next page)

```
/* Data is "done" normally.
 */
  .data : {
    __data_start = .;
    *(.data);
    *(.sdata);
    *(.got);
    *(.got2);
    *(.plt);
    edata = .;
    _edata = .;
    __edata = .;
    __data_end = .;
  }

/* Make sure __bss_end is behind .sbss and COMMON, so is realy the
 * end of the 'sections'. Also I want .bss, .sbss and COMMON beside
 * each other, to make one .bss section for the image.
 *
 * The alignment is handy for making the image and loading it
 * into memory, otherwise it defaults to 0x100. (note it is only
 * needed here)
 */
  .bss ALIGN(0x1000) : { /* the alignment does not seem to hurt. */
    __bss_start = .;
    *(.bss);
    *(.sbss);
    *(COMMON);
    end = .;
    _end = .;
    __end = .;
    __bss_end = .;
  }

/* Stuff I don't want to know about.
 */
  /DISCARD/ : {
    *(.note.GNU-stack);
    *(.comment);
    *(.debug_*);
    *(.eh_frame);
  }

}
```

G.4 With compile-time options and linker script

For this listing the compiler options and the linker script are used for creating the elf32 executable.

```
There are 7 section headers, starting at offset 0x256cc:

Section Headers:
  [Nr] Name              Type       Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL       00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS   00a00000 010000 0150eb 00  AX  0   0  4
  [ 2] .data             PROGBITS   00a150ec 0250ec 0005b4 00  WA  0   0  4
  [ 3] .bss              NOBITS     00a16000 0256a0 009530 00  WA  0   0  8
  [ 4] .shstrtab         STRTAB     00000000 0256a0 00002c 00      0   0  1
  [ 5] .symtab           SYMTAB     00000000 0257e4 002220 10      6 252  4
  [ 6] .strtab           STRTAB     00000000 027a04 00198e 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

H

Kernel Symbol Listing

Listing the kernel symbols, it spans a few pages. Knowing the addresses of linkage can help a great deal when developing the kernel. Most exceptions indicate the address of the instruction creating the exception. Also the effect of linker scripts can been seen through the addresses of the symbols, between the text section and data section addresses. The combination of various sections to combine the 'text', 'data' or 'bss' sections is seen clearly.

Listed bold are the architectural symbols,

```
Archive member included because of file (symbol)

/home/i2a/minix/stdlib/libc.a(atoi.o)
                              start.o (atoi)
/home/i2a/minix/stdlib/libc.a(strtol.o)
                              /home/i2a/minix/stdlib/libc.a(atoi.o) (strtol)
/home/i2a/minix/stdlib/libc.a(errno.o)
                              /home/i2a/minix/stdlib/libc.a(strtol.o) (errno)
/home/i2a/minix/stdlib/libc.a(strncmp.o)
                              klib.o (strncmp)
/home/i2a/minix/stdlib/libc.a(strncpy.o)
                              start.o (strncpy)
/home/i2a/minix/stdlib/libc.a(strcmp.o)
                              start.o (strcmp)
/home/i2a/minix/stdlib/libc.a(strlen.o)
                              utility.o (strlen)
/home/i2a/minix/stdlib/libc.a(ipc.o)
                              clock.o (_receive)
/home/i2a/minix/stdlib/libc.a(vsprintf.o)
                              utility.o (vsprintf)
/home/i2a/minix/stdlib/libc.a(sigaddset.o)
                              system.o (sigaddset)
/home/i2a/minix/stdlib/libc.a(sigismember.o)
                              system.o (sigismember)
/home/i2a/minix/stdlib/libc.a(chartab.o)
                              /home/i2a/minix/stdlib/libc.a(strtol.o) (__ctype)
/home/i2a/minix/stdlib/libc.a(_sigset.o)
                              /home/i2a/minix/stdlib/libc.a(sigaddset.o)
(_sigaddset)
/home/i2a/minix/stdlib/libc.a(doprnt.o)
                              /home/i2a/minix/stdlib/libc.a(vsprintf.o) (_doprnt)
/home/i2a/minix/stdlib/libc.a(flushbuf.o)
                              /home/i2a/minix/stdlib/libc.a(doprnt.o)
(__flushbuf)
/home/i2a/minix/stdlib/libc.a(icompute.o)
                              /home/i2a/minix/stdlib/libc.a(doprnt.o)
(_i_compute)
/home/i2a/minix/stdlib/libc.a(exit.o)
                              /home/i2a/minix/stdlib/libc.a(flushbuf.o) (_clean)
/home/i2a/minix/stdlib/libc.a(malloc.o)
                              /home/i2a/minix/stdlib/libc.a(flushbuf.o) (malloc)
```

```
/home/i2a/minix/stdlib/libc.a(toupper.o)
                              /home/i2a/minix/stdlib/libc.a(doprnt.o) (toupper)
/home/i2a/minix/stdlib/libc.a(fphook.o)
                              /home/i2a/minix/stdlib/libc.a(doprnt.o) (_f_print)
/home/i2a/minix/stdlib/libc.a(_brk.o)
                              /home/i2a/minix/stdlib/libc.a(malloc.o) (_brk)
/home/i2a/minix/stdlib/libc.a(syscall.o)
                              /home/i2a/minix/stdlib/libc.a(_brk.o) (_syscall)
/home/i2a/minix/stdlib/libc.a(_isatty.o)
                              /home/i2a/minix/stdlib/libc.a(flushbuf.o) (_isatty)
/home/i2a/minix/stdlib/libc.a(_lseek.o)
                              /home/i2a/minix/stdlib/libc.a(flushbuf.o) (_lseek)
/home/i2a/minix/stdlib/libc.a(_tcgetattr.o)
                              /home/i2a/minix/stdlib/libc.a(_isatty.o)
(_tcgetattr)
/home/i2a/minix/stdlib/libc.a(_write.o)
                              /home/i2a/minix/stdlib/libc.a(flushbuf.o) (_write)
/home/i2a/minix/stdlib/libc.a(memcpy.o)
                              /home/i2a/minix/stdlib/libc.a(malloc.o) (memcpy)
/home/i2a/minix/stdlib/libc.a(brksize.o)
                              /home/i2a/minix/stdlib/libc.a(_brk.o) (_brksize)
/home/i2a/minix/stdlib/libc.a(data.o)
                              /home/i2a/minix/stdlib/libc.a(flushbuf.o)
(__stdout)
/home/i2a/minix/stdlib/libc.a(ecvt.o)
                              /home/i2a/minix/stdlib/libc.a(fphook.o) (_ecvt)
/home/i2a/minix/stdlib/libc.a(fflush.o)
                              /home/i2a/minix/stdlib/libc.a(flushbuf.o)
(__cleanup)
/home/i2a/minix/stdlib/libc.a(_exit.o)
                              /home/i2a/minix/stdlib/libc.a(exit.o) (_exit)
/home/i2a/minix/stdlib/libc.a(ext_comp.o)
                              /home/i2a/minix/stdlib/libc.a(fphook.o)
(_str_ext_cvt)
/home/i2a/minix/stdlib/libc.a(frexp.o)
                              /home/i2a/minix/stdlib/libc.a(ext_comp.o) (frexp)
/home/i2a/minix/stdlib/libc.a(ldexp.o)
                              /home/i2a/minix/stdlib/libc.a(ext_comp.o) (ldexp)
/home/i2a/minix/stdlib/libc.a(hugeval.o)
                              /home/i2a/minix/stdlib/libc.a(ext_comp.o)
(__huge_val)
/home/i2a/minix/stdlib/libc.a(__exit.o)
                              /home/i2a/minix/stdlib/libc.a(_exit.o) (__exit)
/home/i2a/minix/stdlib/libc.a(_ioctl.o)
                              /home/i2a/minix/stdlib/libc.a(_tcgetattr.o)
(_ioctl)
/home/i2a/minix/stdlib/libppc.a(libmmu.o)
                              ./arch/ppc/minix.o (clr_ibat)
/home/i2a/minix/syslib/libtimers.a(tmrs_set.o)
                              clock.o (tmrs_settimer)
/home/i2a/minix/syslib/libtimers.a(tmrs_clr.o)
                              clock.o (tmrs_clrtimer)
/home/i2a/minix/syslib/libtimers.a(tmrs_exp.o)
                              clock.o (tmrs_exptimers)
system/ksyslib.a(do_devio.o)  system.o (do_devio)
system/ksyslib.a(do_exit.o)   system.o (do_exit)
system/ksyslib.a(do_getksig.o)
                              system.o (do_getksig)
system/ksyslib.a(do_irqctl.o)
                              system.o (do_irqctl)
system/ksyslib.a(do_newmap.o)
                              system.o (do_newmap)
system/ksyslib.a(do_sdevio.o)
```

```
                                system.o (do_sdevio)
system/ksyslib.a(do_sigreturn.o)
                                system.o (do_sigreturn)
system/ksyslib.a(do_trace.o)    system.o (do_trace)
system/ksyslib.a(do_vcopy.o)    system.o (do_vcopy)
system/ksyslib.a(do_abort.o)    system.o (do_abort)
system/ksyslib.a(do_endksig.o)
                                system.o (do_endksig)
system/ksyslib.a(do_fork.o)     system.o (do_fork)
system/ksyslib.a(do_kill.o)     system.o (do_kill)
system/ksyslib.a(do_nice.o)     system.o (do_nice)
system/ksyslib.a(do_segctl.o)
                                system.o (do_segctl)
system/ksyslib.a(do_sigsend.o)
                                system.o (do_sigsend)
system/ksyslib.a(do_umap.o)     system.o (do_umap)
system/ksyslib.a(do_vdevio.o)
                                system.o (do_vdevio)
system/ksyslib.a(do_copy.o)     system.o (do_copy)
system/ksyslib.a(do_exec.o)     system.o (do_exec)
system/ksyslib.a(do_getinfo.o)
                                system.o (do_getinfo)
system/ksyslib.a(do_iopenable.o)
                                system.o (do_iopenable)
system/ksyslib.a(do_memset.o)
                                system.o (do_memset)
system/ksyslib.a(do_privctl.o)
                                system.o (do_privctl)
system/ksyslib.a(do_setalarm.o)
                                system.o (do_setalarm)
system/ksyslib.a(do_times.o)    system.o (do_times)
system/ksyslib.a(do_unused.o)
                                system.o (do_unused)
system/ksyslib.a(do_debug.o)    interrupt.o (do_debug_count_irq)
./arch/ppc/arch.a(debug_mem.o)
                                system/ksyslib.a(do_debug.o)
(debug_pat_print_segment_registers)
./arch/ppc/arch.a(debug_stack.o)
                                system/ksyslib.a(do_debug.o) (debug_stackframe)
./arch/ppc/arch.a(debug_trace.o)
                                system/ksyslib.a(do_debug.o) (trace_reset)
./arch/ppc/arch.a(exception.o)
                                ./arch/ppc/minix.o (chdlr_system_reset)
./arch/ppc/arch.a(clock.o)      main.o (Clock)
./arch/ppc/arch.a(memory.o)     start.o (Memory)
./arch/ppc/arch.a(interrupt.o)
                                start.o (Interrupt)
./arch/ppc/arch.a(system.o)     start.o (System)
./arch/ppc/arch.a(table.o)      main.o (image)
../drivers/arch/ppc/kscreen/kscreen.a(kscreen.o)
                                ./arch/ppc/minix.o (kscreen_init)
/home/i2a/minix/stdlib/libc.a(sigemptyset.o)
                                system/ksyslib.a(do_getksig.o) (sigemptyset)
/home/i2a/minix/stdlib/libppc.a(libppc.o)
                                system/ksyslib.a(do_debug.o) (memcpyw_pa2ea)
/home/i2a/minix/stdlib/libppc.a(bt.o)
                                ../drivers/arch/ppc/kscreen/kscreen.a(kscreen.o)
(bt_init)
/home/i2a/minix/stdlib/libppc.a(register.o)
                                system/ksyslib.a(do_debug.o) (mfspr)
/home/i2a/minix/stdlib/libppc.a(msr.o)
                                ./arch/ppc/arch.a(system.o) (msr_ei_enable)
/home/i2a/minix/stdlib/libppc.a(memio.o)
```

```
                              ./arch/ppc/arch.a(memory.o) (_inb)
/home/i2a/minix/stdlib/libppc.a(mmu.o)
                              ./arch/ppc/arch.a(debug_mem.o) (mmu_pat_seg_ea2pa)
/home/i2a/minix/stdlib/libppc.a(openpic.o)
                              ./arch/ppc/arch.a(interrupt.o)
(opic_write_current_task_priority)

Allocating common symbols
Common symbol       size              file

irq_actids          0x100             ./arch/ppc/arch.a(table.o)
proc                0xed40            ./arch/ppc/arch.a(table.o)
kinfo               0x4c              ./arch/ppc/arch.a(table.o)
rdy_tail            0x40              ./arch/ppc/arch.a(table.o)
do_serial_debug     0x4               ./arch/ppc/arch.a(table.o)
k_reenter           0x4               ./arch/ppc/arch.a(table.o)
mon_return          0x4               ./arch/ppc/arch.a(table.o)
krandom             0x480             ./arch/ppc/arch.a(table.o)
shutdown_started    0x1               ./arch/ppc/arch.a(table.o)
bill_ptr            0x4               ./arch/ppc/arch.a(table.o)
aout                0x4               ./arch/ppc/arch.a(table.o)
__functab           0x80              /home/i2a/minix/stdlib/libc.a(exit.o)
kmess               0x108             ./arch/ppc/arch.a(table.o)
call_vec            0x78              system.o
lost_ticks          0x4               ./arch/ppc/arch.a(table.o)
irq_hooks           0x700             ./arch/ppc/arch.a(table.o)
level0_func         0x4               ./arch/ppc/arch.a(table.o)
proc_ptr            0x4               ./arch/ppc/arch.a(table.o)
timingdata          0x910             debug.o
mon_sp              0x4               ./arch/ppc/arch.a(table.o)
t_stack             0x3c00            ./arch/ppc/arch.a(table.o)
ppriv_addr          0x80              ./arch/ppc/arch.a(table.o)
irq_use             0x4               ./arch/ppc/arch.a(table.o)
priv                0x1a80            ./arch/ppc/arch.a(table.o)
rdy_head            0x40              ./arch/ppc/arch.a(table.o)
kernel_exception    0x1               ./arch/ppc/arch.a(table.o)
next_ptr            0x4               ./arch/ppc/arch.a(table.o)
pproc_addr          0x1a0             ./arch/ppc/arch.a(table.o)
last_sysproc_nr     0x4               ./arch/ppc/arch.a(table.o)
mon_ss              0x4               ./arch/ppc/arch.a(table.o)
machine             0x18              ./arch/ppc/arch.a(table.o)
prev_ptr            0x4               ./arch/ppc/arch.a(table.o)

Memory Configuration

Name             Origin             Length             Attributes
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map


.text           0x0000000000000000    0x1e2e1
                0x0000000000000000            __text_start = .
 *(.text)
 .text          0x0000000000000000     0x64b4 ./arch/ppc/minix.o
                0x0000000000000000            MINIX
                0x0000000000006410            do_some_test
                0x0000000000006468            do_some_test2
                0x00000000000061b4            restart
 .text          0x00000000000064b4      0x494 start.o
                0x00000000000064b4            cstart
 .text          0x0000000000006948      0x558 main.o
                0x0000000000006948            idle_task
```

```
                0x0000000000006974                main
                0x0000000000006d10                prepare_shutdown
 .text          0x0000000000006ea0      0x70c debug.o
                0x0000000000006f90                timer_end
                0x0000000000006ea0                timer_start
                0x00000000000071d4                check_runqueues
 .text          0x00000000000075ac      0x394 utility.o
                0x00000000000076a8                panic
                0x000000000000775c                ttyprintf
                0x00000000000075ac                kprintf
 .text          0x0000000000007940      0x428 clock.o
                0x0000000000007bf8                set_timer
                0x0000000000007be4                get_uptime
                0x0000000000007940                clock_task
                0x0000000000007ca4                reset_timer
                0x0000000000007ad8                clock_stop
                0x0000000000007d3c                read_clock
 .text          0x0000000000007d68     0x1c2c proc.o
                0x000000000000958c                lock_enqueue
                0x0000000000007d68                lock_notify
                0x00000000000083b8                sys_call
                0x00000000000097ac                lock_dequeue
                0x0000000000009294                lock_send
 .text          0x0000000000009994      0x384 interrupt.o
                0x0000000000009b68                intr_handle
                0x0000000000009c38                enable_irq
                0x0000000000009a88                rm_irq_handler
                0x0000000000009994                put_irq_handler
                0x0000000000009ca0                disable_irq
 .text          0x0000000000009d18      0x18c klib.o
                0x0000000000009d44                get_value_other
                0x0000000000009d18                enable_iop
 .text          0x0000000000009ea4      0x8c4 system.o
                0x000000000000a244                get_randomness
                0x000000000000a55c                virtual_copy
                0x000000000000a504                umap_remote
                0x000000000000a46c                umap_local
                0x000000000000a390                cause_sig
                0x000000000000a19c                get_priv
                0x000000000000a340                send_sig
                0x0000000000009ea4                sys_task
 .text          0x000000000000a768       0x28
/home/i2a/minix/stdlib/libc.a(atoi.o)
                0x000000000000a768                atoi
 .text          0x000000000000a790      0x218
/home/i2a/minix/stdlib/libc.a(strtol.o)
                0x000000000000a998                strtoul
                0x000000000000a988                strtol
 .text          0x000000000000a9a8       0x30
/home/i2a/minix/stdlib/libc.a(strncmp.o)
                0x000000000000a9a8                strncmp
 .text          0x000000000000a9d8       0x38
/home/i2a/minix/stdlib/libc.a(strncpy.o)
                0x000000000000a9d8                strncpy
 .text          0x000000000000aa10       0x24
/home/i2a/minix/stdlib/libc.a(strcmp.o)
                0x000000000000aa10                strcmp
 .text          0x000000000000aa34       0x18
/home/i2a/minix/stdlib/libc.a(strlen.o)
                0x000000000000aa34                strlen
 .text          0x000000000000aa4c       0x8c
/home/i2a/minix/stdlib/libc.a(ipc.o)
                0x000000000000aa4c                _echo
```

```
                  0x000000000000aa60                 _notify
                  0x000000000000aac4                 _nb_send
                  0x000000000000aa9c                 _send
                  0x000000000000aa74                 _sendrec
                  0x000000000000aab0                 _nb_receive
                  0x000000000000aa88                 _receive
 .text            0x000000000000aad8       0xbc
/home/i2a/minix/stdlib/libc.a(vsprintf.o)
                  0x000000000000ab34                 vsprintf
                  0x000000000000aad8                 vsnprintf
 .text            0x000000000000ab94        0x4
/home/i2a/minix/stdlib/libc.a(sigaddset.o)
                  0x000000000000ab94                 sigaddset
 .text            0x000000000000ab98        0x4
/home/i2a/minix/stdlib/libc.a(sigismember.o)
                  0x000000000000ab98                 sigismember
 .text            0x000000000000ab9c       0xf4
/home/i2a/minix/stdlib/libc.a(_sigset.o)
                  0x000000000000ac3c                 _sigfillset
                  0x000000000000ac24                 _sigemptyset
                  0x000000000000ab9c                 _sigaddset
                  0x000000000000abe0                 _sigdelset
                  0x000000000000ac58                 _sigismember
 .text            0x000000000000ac90      0xc3c
/home/i2a/minix/stdlib/libc.a(doprnt.o)
                  0x000000000000af74                 _doprnt
 .text            0x000000000000b8cc      0x350
/home/i2a/minix/stdlib/libc.a(flushbuf.o)
                  0x000000000000b8cc                 __flushbuf
 .text            0x000000000000bc1c       0x90
/home/i2a/minix/stdlib/libc.a(icompute.o)
                  0x000000000000bc1c                 _i_compute
 .text            0x000000000000bcac       0xac
/home/i2a/minix/stdlib/libc.a(exit.o)
                  0x000000000000bcac                 exit
 .text            0x000000000000bd58      0x65c
/home/i2a/minix/stdlib/libc.a(malloc.o)
                  0x000000000000bdf8                 malloc
                  0x000000000000c078                 realloc
                  0x000000000000bd58                 free
 .text            0x000000000000c3b4       0x28
/home/i2a/minix/stdlib/libc.a(toupper.o)
                  0x000000000000c3b4                 toupper
 .text            0x000000000000c3dc      0x77c
/home/i2a/minix/stdlib/libc.a(fphook.o)
                  0x000000000000cb20                 strtod
                  0x000000000000c3dc                 _f_print
 .text            0x000000000000cb58      0x124
/home/i2a/minix/stdlib/libc.a(_brk.o)
                  0x000000000000cbbc                 _sbrk
                  0x000000000000cb58                 _brk
 .text            0x000000000000cc7c       0x64
/home/i2a/minix/stdlib/libc.a(syscall.o)
                  0x000000000000cc7c                 _syscall
 .text            0x000000000000cce0       0x2c
/home/i2a/minix/stdlib/libc.a(_isatty.o)
                  0x000000000000cce0                 _isatty
 .text            0x000000000000cd0c       0x48
/home/i2a/minix/stdlib/libc.a(_lseek.o)
                  0x000000000000cd0c                 _lseek
 .text            0x000000000000cd54       0x2c
/home/i2a/minix/stdlib/libc.a(_tcgetattr.o)
                  0x000000000000cd54                 _tcgetattr
```

```
 .text          0x000000000000cd80       0x38
/home/i2a/minix/stdlib/libc.a(_write.o)
                0x000000000000cd80                _write
 .text          0x000000000000cdb8       0x9c
/home/i2a/minix/stdlib/libc.a(memcpy.o)
                0x000000000000cdb8                memcpy
 .text          0x000000000000ce54       0xd8
/home/i2a/minix/stdlib/libc.a(ecvt.o)
                0x000000000000ce54                _ecvt
                0x000000000000cec0                _fcvt
 .text          0x000000000000cf2c      0x250
/home/i2a/minix/stdlib/libc.a(fflush.o)
                0x000000000000cf2c                fflush
                0x000000000000d10c                __cleanup
 .text          0x000000000000d17c        0x4
/home/i2a/minix/stdlib/libc.a(_exit.o)
                0x000000000000d17c                _exit
 .text          0x000000000000d180     0x2080
/home/i2a/minix/stdlib/libc.a(ext_comp.o)
                0x000000000000e03c                _ext_str_cvt
                0x000000000000ef54                _ext_dbl_cvt
                0x000000000000d5e0                _str_ext_cvt
                0x000000000000ee14                _dbl_ext_cvt
 .text          0x000000000000f200       0x98
/home/i2a/minix/stdlib/libc.a(frexp.o)
                0x000000000000f200                frexp
 .text          0x000000000000f298      0x130
/home/i2a/minix/stdlib/libc.a(ldexp.o)
                0x000000000000f298                ldexp
 .text          0x000000000000f3c8       0x18
/home/i2a/minix/stdlib/libc.a(hugeval.o)
                0x000000000000f3c8                __huge_val
 .text          0x000000000000f3e0       0x30
/home/i2a/minix/stdlib/libc.a(__exit.o)
                0x000000000000f3e0                __exit
 .text          0x000000000000f410       0x38
/home/i2a/minix/stdlib/libc.a(_ioctl.o)
                0x000000000000f410                _ioctl
 .text          0x000000000000f448      0x55c
/home/i2a/minix/stdlib/libppc.a(libmmu.o)
                0x000000000000f670                mmu_off_rtn
                0x000000000000f6e4                clr_ibat
                0x000000000000f7f4                mfibatu
                0x000000000000f6ac                mmu_on_simple
                0x000000000000f6c8                mmu_off_simple
                0x000000000000f8d4                mfdbatl
                0x000000000000f79c                mtdbat
                0x000000000000f91c                tlbia_off
                0x000000000000f714                clr_dbat
                0x000000000000f744                mtibat
                0x000000000000f980                tlbie
                0x000000000000f4b8                set_pte
                0x000000000000f5a8                mfsr
                0x000000000000f994                mfsdr1
                0x000000000000f66c                mmu_on_rtn
                0x000000000000f448                get_set_pteg
                0x000000000000f83c                mfibatl
                0x000000000000f99c                mtsdr1
                0x000000000000f65c                mmu_off
                0x000000000000f504                mtsr
                0x000000000000f88c                mfdbatu
                0x000000000000f64c                mmu_on
 .text          0x000000000000f9a4       0xe0
```

```
/home/i2a/minix/syslib/libtimers.a(tmrs_set.o)
                0x000000000000f9a4                tmrs_settimer
 .text          0x000000000000fa84        0x90
/home/i2a/minix/syslib/libtimers.a(tmrs_clr.o)
                0x000000000000fa84                tmrs_clrtimer
 .text          0x000000000000fb14        0xe0
/home/i2a/minix/syslib/libtimers.a(tmrs_exp.o)
                0x000000000000fb14                tmrs_exptimers
 .text          0x000000000000fbf4       0x118 system/ksyslib.a(do_devio.o)
                0x000000000000fbf4                do_devio
 .text          0x000000000000fd0c       0x328 system/ksyslib.a(do_exit.o)
                0x000000000000ffa0                do_exit
 .text          0x0000000000010034        0xa0 system/ksyslib.a(do_getksig.o)
                0x0000000000010034                do_getksig
 .text          0x00000000000100d4       0x224 system/ksyslib.a(do_irqctl.o)
                0x00000000000100d4                do_irqctl
 .text          0x00000000000102f8       0x1e8 system/ksyslib.a(do_newmap.o)
                0x00000000000102f8                do_newmap
 .text          0x00000000000104e0       0x144 system/ksyslib.a(do_sdevio.o)
                0x00000000000104e0                do_sdevio
 .text          0x0000000000010624       0x154 system/ksyslib.a(do_sigreturn.o)
                0x0000000000010624                do_sigreturn
 .text          0x0000000000010778       0x2c4 system/ksyslib.a(do_trace.o)
                0x0000000000010778                do_trace
 .text          0x0000000000010a3c       0x168 system/ksyslib.a(do_vcopy.o)
                0x0000000000010a3c                do_vcopy
 .text          0x0000000000010ba4        0xd0 system/ksyslib.a(do_abort.o)
                0x0000000000010ba4                do_abort
 .text          0x0000000000010c74        0x6c system/ksyslib.a(do_endksig.o)
                0x0000000000010c74                do_endksig
 .text          0x0000000000010ce0       0x130 system/ksyslib.a(do_fork.o)
                0x0000000000010ce0                do_fork
 .text          0x0000000000010e10       0x134 system/ksyslib.a(do_kill.o)
                0x0000000000010e10                do_kill
 .text          0x0000000000010f44       0x104 system/ksyslib.a(do_nice.o)
                0x0000000000010f44                do_nice
 .text          0x0000000000011048       0x10c system/ksyslib.a(do_segctl.o)
                0x0000000000011048                do_segctl
 .text          0x0000000000011154       0x1e0 system/ksyslib.a(do_sigsend.o)
                0x0000000000011154                do_sigsend
 .text          0x0000000000011334        0xd8 system/ksyslib.a(do_umap.o)
                0x0000000000011334                do_umap
 .text          0x000000000001140c       0x3e8 system/ksyslib.a(do_vdevio.o)
                0x000000000001140c                do_vdevio
 .text          0x00000000000117f4        0xd4 system/ksyslib.a(do_copy.o)
                0x00000000000117f4                do_copy
 .text          0x00000000000118c8       0x12c system/ksyslib.a(do_exec.o)
                0x00000000000118c8                do_exec
 .text          0x00000000000119f4       0x47c system/ksyslib.a(do_getinfo.o)
                0x00000000000119f4                do_getinfo
 .text          0x0000000000011e70        0x74 system/ksyslib.a(do_iopenable.o)
                0x0000000000011e70                do_iopenable
 .text          0x0000000000011ee4        0x54 system/ksyslib.a(do_memset.o)
                0x0000000000011ee4                do_memset
 .text          0x0000000000011f38       0x1d8 system/ksyslib.a(do_privctl.o)
                0x0000000000011f38                do_privctl
 .text          0x0000000000012110       0x208 system/ksyslib.a(do_setalarm.o)
                0x0000000000012110                do_setalarm
 .text          0x0000000000012318        0x80 system/ksyslib.a(do_times.o)
                0x0000000000012318                do_times
 .text          0x0000000000012398        0x38 system/ksyslib.a(do_unused.o)
                0x0000000000012398                do_unused
 .text          0x00000000000123d0      0x221c system/ksyslib.a(do_debug.o)
```

```
                0x0000000000013ac0                do_debug_pt_debug_dmp
                0x0000000000013d00                do_debug_farmem_dmp
                0x00000000000124b4                do_debug_monparams_dmp
                0x000000000001295c                do_debug_kenv_dmp
                0x00000000000134a4                do_debug_trace_rst
                0x00000000000138c0
do_debug_current_process_cpu_dmp
                0x00000000000134c4                do_debug_print_process_info
                0x0000000000012404                do_debug_count_irq
                0x0000000000013978                do_debug_view_mem
                0x0000000000013240                do_debug_memmap_dmp
                0x0000000000012428                do_debug_print_irq_stats
                0x00000000000123d0                do_debug_uptime_dmp
                0x00000000000139e4                do_debug_detailed_process_dmp
                0x0000000000013e04                do_debug
                0x0000000000013a74                do_debug_effective_address_dmp
                0x0000000000013c6c                do_debug_cpu_dmp
                0x0000000000012bb8                do_debug_privileges_dmp
                0x000000000001281c                do_debug_sched_dmp
                0x0000000000012f6c                do_debug_proctab_dmp
                0x00000000000135f8                do_debug_kernel_info_dmp
                0x0000000000013484                do_debug_trace_dmp
                0x0000000000012550                do_debug_image_dmp
                0x0000000000013b50                do_debug_pt_print_pte_info
                0x0000000000012f40                do_debug_sendmask_dmp
                0x00000000000133e4                do_debug_misc_dmp
 .text          0x00000000000145ec      0xce0 ./arch/ppc/arch.a(debug_mem.o)
                0x00000000000145ec
debug_pat_print_segment_registers
                0x0000000000014890                debug_pat_print_pt_small
                0x0000000000014b8c                debug_pat_seg_address
                0x0000000000015040                debug_memviewh
                0x0000000000014c38                debug_pat_address
                0x00000000000149c0                debug_pat
                0x00000000000146e0                debug_pat_print_pt
                0x0000000000014cec                debug_pat_print_pte_info
                0x0000000000014ed0                debug_memvieww
 .text          0x00000000000152cc      0x380 ./arch/ppc/arch.a(debug_stack.o)
                0x000000000001545c                debug_signal_frame
                0x00000000000152cc                debug_stackframe
                0x00000000000154a4                debug_signal_context
 .text          0x000000000001564c      0x600 ./arch/ppc/arch.a(debug_trace.o)
                0x00000000000156d0                trace_add1
                0x00000000000159b0                trace_rm
                0x000000000001564c                trace_add
                0x00000000000159e8                trace_print
                0x0000000000015bc8                trace_return
                0x0000000000015820                trace_add3
                0x00000000000158e0                trace_add4
                0x0000000000015770                trace_add2
                0x00000000000159cc                trace_reset
 .text          0x0000000000015c4c     0x12ac ./arch/ppc/arch.a(exception.o)
                0x0000000000015c4c                chdlr_system_reset
                0x00000000000164e4                chdlr_external_interrupt
                0x0000000000016908                chdlr_fp_unavailable
                0x0000000000016ab0                chdlr_decrementer
                0x0000000000015f40                chdlr_dsi
                0x0000000000015d98                chdlr_machine_check
                0x0000000000016ba8                chdlr_trace
                0x0000000000016304                chdlr_isi
                0x0000000000016b10                chdlr_system_call
                0x0000000000016704                chdlr_program
                0x0000000000016d50                chdlr_fp_assist
```

```
                0x000000000001655c                chdlr_alignment
 .text          0x0000000000016ef8      0x100 ./arch/ppc/arch.a(clock.o)
 .text          0x0000000000016ff8      0x5d8 ./arch/ppc/arch.a(memory.o)
 .text          0x00000000000175d0      0x32c ./arch/ppc/arch.a(interrupt.o)
 .text          0x00000000000178fc      0x260 ./arch/ppc/arch.a(system.o)
 .text          0x0000000000017b5c      0x118
../drivers/arch/ppc/kscreen/kscreen.a(kscreen.o)
                0x0000000000017b5c                kscreen_init
                0x0000000000017bd8                kscreen_memmap_init
 .text          0x0000000000017c74        0x4
/home/i2a/minix/stdlib/libc.a(sigemptyset.o)
                0x0000000000017c74                sigemptyset
 .text          0x0000000000017c78      0x43c
/home/i2a/minix/stdlib/libppc.a(libppc.o)
                0x0000000000018038                memfcpyw
                0x000000000001808c                memfsetw
                0x0000000000017db4                memcpyw_ea2ea
                0x0000000000017d5c                memcpy_pa2ea
                0x0000000000017eec                memset_ea
                0x000000000001800c                memfcpy
                0x0000000000017f2c                memset_pa
                0x0000000000017d04                memcpy_ea2pa
                0x0000000000017c78                memcpy_ea2ea
                0x0000000000017f6c                memsetw_ea
                0x0000000000017e3c                memcpyw_ea2pa
                0x0000000000017fac                memsetw_pa
                0x0000000000017df8                memcpyw_pa2pa
                0x0000000000017fec                running_address
                0x0000000000018064                memfset
                0x0000000000017e94                memcpyw_pa2ea
                0x0000000000017cbc                memcpy_pa2pa
 .text          0x00000000000180b4      0x708
/home/i2a/minix/stdlib/libppc.a(bt.o)
                0x0000000000018144                bt_init
                0x0000000000018514                bt_putc
                0x00000000000182a8                bt_flush
                0x00000000000181d4                bt_clear
                0x00000000000180b4                bt_init_default
 .text          0x00000000000187bc      0x6d8
/home/i2a/minix/stdlib/libppc.a(register.o)
                0x0000000000018e3c                mtdec
                0x0000000000018e8c                mtsprg3
                0x00000000000188e4                mfgpr
                0x0000000000018e74                mtsprg0
                0x0000000000018e7c                mtsprg1
                0x0000000000018e34                mfmsr
                0x00000000000187bc                mtgpr
                0x0000000000018a14                mfspr
                0x0000000000018e5c                mttb
                0x0000000000018e48                mfdec
                0x0000000000018e20                mfsp
                0x0000000000018e84                mtsprg2
                0x0000000000018e28                mtmsr
                0x0000000000018e54                nop
 .text          0x0000000000018e94      0x348
/home/i2a/minix/stdlib/libppc.a(msr.o)
                0x000000000001916c                msr_ri_enable
                0x0000000000018ee8                msr_ile_disable
                0x0000000000018f20                msr_ei_disable
                0x00000000000190e0                msr_ip_disable
                0x0000000000018fc8                msr_mc_disable
                0x0000000000019188                msr_ri_disable
                0x0000000000019000                msr_fe0_disable
```

```
                0x0000000000019038                msr_se_disable
                0x0000000000019134                msr_dr_enable
                0x0000000000018e94                msr_pow_enable
                0x0000000000018eb0                msr_pow_disable
                0x00000000000190c4                msr_ip_enable
                0x000000000001908c                msr_fe1_enable
                0x0000000000018fac                msr_mc_enable
                0x0000000000018f74                msr_fp_enable
                0x0000000000018f58                msr_pr_disable
                0x0000000000019118                msr_ir_disable
                0x00000000000191a4                msr_le_enable
                0x0000000000018f04                msr_ei_enable
                0x00000000000190a8                msr_fe1_disable
                0x0000000000018f3c                msr_pr_enable
                0x000000000001901c                msr_se_enable
                0x0000000000019070                msr_be_disable
                0x00000000000191c0                msr_le_disable
                0x0000000000018fe4                msr_fe0_enable
                0x00000000000190fc                msr_ir_enable
                0x0000000000018ecc                msr_ile_enable
                0x0000000000019150                msr_dr_disable
                0x0000000000019054                msr_be_enable
                0x0000000000018f90                msr_fp_disable
 .text          0x00000000000191dc      0x200
/home/i2a/minix/stdlib/libppc.a(memio.o)
                0x0000000000019304                _le_insl
                0x00000000000192bc                _insw
                0x0000000000019304                _le_insw
                0x000000000001923c                _le_inh
                0x0000000000019304                _br_insl
                0x0000000000019370                _outsw
                0x0000000000019268                _le_outw
                0x0000000000019304                _br_insw
                0x00000000000193b8                _le_outsw
                0x0000000000019298                _insh
                0x00000000000193b8                _br_outsl
                0x0000000000019370                _outsl
                0x00000000000191fc                _inw
                0x000000000001922c                _outw
                0x00000000000191dc                _inb
                0x00000000000193b8                _le_outsl
                0x00000000000193b8                _br_outsw
                0x00000000000192e0                _le_insh
                0x000000000001925c                _le_outh
                0x0000000000019394                _br_outsh
                0x00000000000191ec                _inh
                0x0000000000019274                _insb
                0x0000000000019328                _outsb
                0x000000000001924c                _le_inw
                0x0000000000019394                _le_outsh
                0x00000000000192e0                _br_insh
                0x000000000001934c                _outsh
                0x000000000001920c                _outb
                0x000000000001921c                _outh
                0x00000000000192bc                _insl
 .text          0x00000000000193dc      0xba8
/home/i2a/minix/stdlib/libppc.a(mmu.o)
                0x0000000000019e84                mmu_dbat_ea2pa
                0x0000000000019698                mmu_pat_seg_ea2pteg
                0x0000000000019438                mmu_pat_seg_ea2pa
                0x0000000000019c34                mmu_pat_tlbir
                0x0000000000019600                mmu_pat_ea2pteg
                0x0000000000019c24                mmu_pat_update_pp
```

```
                 0x00000000000195c4              mmu_pat_ea2pa
                 0x0000000000019d18              mmu_pat_make_segdesc
                 0x0000000000019764              mmu_pat_segmap
                 0x0000000000019c98              mmu_pat_init_pt
                 0x00000000000193dc              mmu_pat_make_pte
                 0x0000000000019d04              mmu_pat_make_sdr1
                 0x0000000000019728              tlbia
                 0x00000000000199ac              mmu_pat_memmap_new
                 0x0000000000019a38              mmu_pat_seginv
                 0x0000000000019d40              mmu_bat_make_entry
                 0x0000000000019d84              mmu_ibat_ea2pa
 .text           0x0000000000019f84      0x1758
/home/i2a/minix/stdlib/libppc.a(openpic.o)
                 0x000000000001b5b8              opic_is_read_destination
                 0x000000000001b304              opic_is_get_polarity
                 0x000000000001ac30              opic_timer_read_vp
                 0x000000000001a984              opic_write_spurious_vector
                 0x000000000001a574              opic_read_vendor_id
                 0x000000000001a7a4              opic_ipi_set_mask
                 0x000000000001a624              opic_ipi_read_vp
                 0x000000000001aa0c              opic_timer_read_current_count
                 0x000000000001af74              opic_timer_read_destination
                 0x000000000001ad7c              opic_timer_set_mask
                 0x000000000001b4b8              opic_is_read_vector
                 0x000000000001ae44              opic_timer_read_priority
                 0x000000000001aecc              opic_timer_write_destination
                 0x000000000001a4a4              opic_write_base_address
                 0x000000000001a9e0              opic_timer_frequency
                 0x000000000001a530              opic_read_vendor_device_id
                 0x000000000001a664              opic_ipi_write_vp
                 0x000000000001aa60              opic_timer_get_togglebit
                 0x000000000001ab68              opic_timer_get_count_inhibit
                 0x000000000001a474              opic_read_base_address
                 0x000000000001b01c              opic_is_write_vp
                 0x000000000001afc4              opic_is_read_vp
                 0x000000000001abac              opic_timer_set_count_inhibit
                 0x000000000001a500              opic_read_stepping
                 0x000000000001a17c              opic_write_eoi
                 0x000000000001a130              opic_read_interrupt_acknowlage
                 0x000000000001b420              opic_is_get_sense
                 0x000000000001a330              opic_write_config_register
                 0x000000000001a1cc              opic_read_eoi
                 0x000000000001aae8              opic_timer_write_base_count
                 0x000000000001a904              opic_ipi_read_vector
                 0x000000000001a86c              opic_ipi_get_act
                 0x000000000001a958              opic_read_spurious_vector
                 0x000000000001a5a4              opic_read_processor_init_reg
                 0x000000000001b46c              opic_is_read_priority
                 0x000000000001a5d0              opic_write_processor_init_reg
                 0x000000000001a2b8              opic_get_version_id
                 0x0000000000019f84              opic_write_ipi_dispatch
                 0x000000000001a034
opic_write_current_task_priority
                 0x000000000001a2e0              opic_read_config_register
                 0x000000000001b234              opic_is_set_polarity
                 0x000000000001ae88              opic_timer_read_vector
                 0x000000000001a218              opic_read_feature_register
                 0x000000000001a42c              opic_enable_i8259
                 0x000000000001a268              opic_get_is_count
                 0x000000000001a290              opic_get_cpu_count
                 0x000000000001aaa4              opic_timer_read_base_count
                 0x000000000001a3a0              opic_reset
                 0x000000000001a0b8              opic_read_current_task_priority
```

```
                 0x000000000001a760                opic_ipi_get_mask
                 0x000000000001a3e4                opic_disable_i8259
                 0x000000000001b118                opic_is_set_mask
                 0x000000000001a104                opic_read_who_am_i
                 0x000000000001b614                opic_init
                 0x000000000001b504                opic_is_write_destination
                 0x000000000001b1e8                opic_is_get_mask
                 0x000000000001a8b0                opic_ipi_read_priority
                 0x000000000001b350                opic_is_set_sense
                 0x000000000001ac80                opic_timer_write_vp
 *(.sdata2)
 *(.rodata)
 .rodata        0x000000000001b6dc        0x8 utility.o
 .rodata        0x000000000001b6e4       0x28 clock.o
 .rodata        0x000000000001b70c       0x24 proc.o
 .rodata        0x000000000001b730      0x198
/home/i2a/minix/stdlib/libc.a(doprnt.o)
 .rodata        0x000000000001b8c8       0x8c
/home/i2a/minix/stdlib/libc.a(fphook.o)
 .rodata        0x000000000001b954        0xc system/ksyslib.a(do_newmap.o)
 .rodata        0x000000000001b960       0x10 system/ksyslib.a(do_sigreturn.o)
 .rodata        0x000000000001b970       0x2c system/ksyslib.a(do_trace.o)
 .rodata        0x000000000001b99c        0x8 system/ksyslib.a(do_kill.o)
 .rodata        0x000000000001b9a4       0x3c system/ksyslib.a(do_getinfo.o)
 .rodata        0x000000000001b9e0       0x18 system/ksyslib.a(do_setalarm.o)
 .rodata        0x000000000001b9f8      0x128 system/ksyslib.a(do_debug.o)
 .rodata        0x000000000001bb20       0x94 ./arch/ppc/arch.a(exception.o)
 .rodata        0x000000000001bbb4       0x18 ./arch/ppc/arch.a(clock.o)
                0x000000000001bbb4                Clock
 .rodata        0x000000000001bbcc       0xa0 ./arch/ppc/arch.a(memory.o)
                0x000000000001bbcc                Memory
 .rodata        0x000000000001bc6c       0x1c ./arch/ppc/arch.a(interrupt.o)
                0x000000000001bc6c                Interrupt
 .rodata        0x000000000001bc88       0x40 ./arch/ppc/arch.a(system.o)
                0x000000000001bc88                System
 .rodata        0x000000000001bcc8       0x64
/home/i2a/minix/stdlib/libppc.a(bt.o)
 *(.rodata.*)
 .rodata.str1.4
                0x000000000001bd2c       0x30 start.o
 .rodata.str1.4
                0x000000000001bd5c      0x13b main.o
                                        0x144 (size before relaxing)
 *fill*         0x000000000001be97        0x1 00
 .rodata.str1.4
                0x000000000001be98      0x1f1 debug.o
                                        0x1f4 (size before relaxing)
 *fill*         0x000000000001c089        0x3 00
 .rodata.cst4   0x000000000001c08c        0x4 debug.o
 .rodata.str1.4
                0x000000000001c090       0x18 utility.o
                                         0x1c (size before relaxing)
 .rodata.str1.4
                0x000000000001c0a8       0xb5 clock.o
                                         0xb8 (size before relaxing)
 *fill*         0x000000000001c15d        0x3 00
 .rodata.str1.4
                0x000000000001c160      0x1ed proc.o
                                        0x1f0 (size before relaxing)
 *fill*         0x000000000001c34d        0x3 00
 .rodata.str1.4
                0x000000000001c350       0x5b interrupt.o
                                         0x5c (size before relaxing)
```

```
 *fill*         0x000000000001c3ab        0x1 00
 .rodata.str1.4
                0x000000000001c3ac       0x45 klib.o
                                         0x48 (size before relaxing)
 *fill*         0x000000000001c3f1        0x3 00
 .rodata.str1.4
                0x000000000001c3f4       0x6c system.o
 .rodata.str1.4
                0x000000000001c460        0x7
/home/i2a/minix/stdlib/libc.a(doprnt.o)
                                          0x8 (size before relaxing)
 *fill*         0x000000000001c467        0x1 00
 .rodata.cst8   0x000000000001c468        0x8
/home/i2a/minix/stdlib/libc.a(fphook.o)
 .rodata.cst8   0x000000000001c470       0x20
/home/i2a/minix/stdlib/libc.a(ext_comp.o)
                                         0x48 (size before relaxing)
 .rodata.cst8   0x000000000001c490        0x8
/home/i2a/minix/stdlib/libc.a(frexp.o)
 .rodata.cst8   0x000000000001c498        0x0
/home/i2a/minix/stdlib/libc.a(ldexp.o)
                                         0x10 (size before relaxing)
 .rodata.cst8   0x000000000001c498        0x8
/home/i2a/minix/stdlib/libc.a(hugeval.o)
 .rodata.str1.4
                0x000000000001c4a0       0x5e system/ksyslib.a(do_exit.o)
                                         0x60 (size before relaxing)
 *fill*         0x000000000001c4fe        0x2 00
 .rodata.str1.4
                0x000000000001c500       0x85 system/ksyslib.a(do_newmap.o)
                                         0x88 (size before relaxing)
 *fill*         0x000000000001c585        0x3 00
 .rodata.str1.4
                0x000000000001c588       0x24 system/ksyslib.a(do_sigreturn.o)
                                         0x40 (size before relaxing)
 .rodata.str1.4
                0x000000000001c5ac       0x51 system/ksyslib.a(do_kill.o)
                                         0x6c (size before relaxing)
 *fill*         0x000000000001c5fd        0x3 00
 .rodata.str1.4
                0x000000000001c600        0xa system/ksyslib.a(do_vdevio.o)
                                          0xc (size before relaxing)
 *fill*         0x000000000001c60a        0x2 00
 .rodata.str1.4
                0x000000000001c60c        0x8 system/ksyslib.a(do_exec.o)
 .rodata.str1.4
                0x000000000001c614       0x47 system/ksyslib.a(do_setalarm.o)
                                         0x60 (size before relaxing)
 *fill*         0x000000000001c65b        0x1 00
 .rodata.str1.4
                0x000000000001c65c       0x26 system/ksyslib.a(do_unused.o)
                                         0x28 (size before relaxing)
 *fill*         0x000000000001c682        0x2 00
 .rodata.str1.4
                0x000000000001c684      0xbef system/ksyslib.a(do_debug.o)
                                        0xc50 (size before relaxing)
 *fill*         0x000000000001d273        0x1 00
 .rodata.str1.4
                0x000000000001d274      0x4be ./arch/ppc/arch.a(debug_mem.o)
                                        0x4d8 (size before relaxing)
 *fill*         0x000000000001d732        0x2 00
 .rodata.str1.4
                0x000000000001d734      0x1e3 ./arch/ppc/arch.a(debug_stack.o)
```

```
                                                0x1e8 (size before relaxing)
 *fill*         0x000000000001d917               0x1 00
 .rodata.str1.4
                0x000000000001d918              0x1a ./arch/ppc/arch.a(debug_trace.o)
                                                0x20 (size before relaxing)
 *fill*         0x000000000001d932               0x2 00
 .rodata.str1.4
                0x000000000001d934             0x739 ./arch/ppc/arch.a(exception.o)
                                               0x778 (size before relaxing)
 *fill*         0x000000000001e06d               0x3 00
 .rodata.str1.4
                0x000000000001e070             0x1b1 ./arch/ppc/arch.a(memory.o)
                                               0x1d0 (size before relaxing)
 *fill*         0x000000000001e221               0x3 00
 .rodata.str1.4
                0x000000000001e224               0x5 ./arch/ppc/arch.a(interrupt.o)
                                                 0x8 (size before relaxing)
 *fill*         0x000000000001e229               0x3 00
 .rodata.str1.4
                0x000000000001e22c              0x79 ./arch/ppc/arch.a(system.o)
                                                0x98 (size before relaxing)
 *fill*         0x000000000001e2a5               0x3 00
 .rodata.str1.4
                0x000000000001e2a8              0x39
../drivers/arch/ppc/kscreen/kscreen.a(kscreen.o)
                                                0x3c (size before relaxing)
                0x000000000001e2e1                       etext = .
                0x000000000001e2e1                       _etext = .
                0x000000000001e2e1                       __etext = .
                0x000000000001e2e1                       __text_end = .

.data           0x0000000010000000            0x1b2c
                0x0000000010000000                       __data_start = .
 *(.data)
 .data          0x0000000010000000              0x20 ./arch/ppc/minix.o
                0x0000000010000014                       ca_data_end
                0x000000001000001c                       ca_bss_end
                0x0000000010000018                       ca_bss_start
                0x000000001000000c                       ca_text_end
                0x0000000010000010                       ca_data_start
                0x0000000010000008                       ca_text_start
 .data          0x0000000010000020             0x104
/home/i2a/minix/stdlib/libc.a(chartab.o)
                0x0000000010000020                       __ctype
 .data          0x0000000010000124               0x4
/home/i2a/minix/stdlib/libc.a(fphook.o)
                0x0000000010000124                       _fp_hook
 .data          0x0000000010000128               0x4
/home/i2a/minix/stdlib/libc.a(brksize.o)
                0x0000000010000128                       __brksize
                0x0000000010000128                       _brksize
 .data          0x000000001000012c              0x98
/home/i2a/minix/stdlib/libc.a(data.o)
                0x000000001000017c                       __stderr
                0x000000001000012c                       __iotab
                0x00000000100001ac                       __stdin
                0x0000000010000194                       __stdout
 .data          0x00000000100001c4             0x480
/home/i2a/minix/stdlib/libc.a(ext_comp.o)
 .data          0x0000000010000644               0xc system/ksyslib.a(do_vdevio.o)
 .data          0x0000000010000650               0xc system/ksyslib.a(do_debug.o)
 .data          0x000000001000065c              0xa4 ./arch/ppc/arch.a(exception.o)
 .data          0x0000000010000700              0x24 ./arch/ppc/arch.a(clock.o)
```

```
 .data           0x0000000010000724       0x30 ./arch/ppc/arch.a(memory.o)
 .data           0x0000000010000754       0x54 ./arch/ppc/arch.a(interrupt.o)
 .data           0x00000000100007a8       0x50 ./arch/ppc/arch.a(system.o)
 .data           0x00000000100007f8      0x280 ./arch/ppc/arch.a(table.o)
                 0x00000000100007f8                image
 .data           0x0000000010000a78       0x64
../drivers/arch/ppc/kscreen/kscreen.a(kscreen.o)
 .data           0x0000000010000adc     0x1050
/home/i2a/minix/stdlib/libppc.a(bt.o)
 *(.sdata)
 *(.got)
 *(.got2)
 *(.plt)
                 0x0000000010001b2c                edata = .
                 0x0000000010001b2c                _edata = .
                 0x0000000010001b2c                __edata = .
                 0x0000000010001b2c                __data_end = .

.bss             0x0000000010002000    0x1cac8
                 0x0000000010002000                __bss_start = .
 *(.bss)
 .bss            0x0000000010002000     0x4030 ./arch/ppc/minix.o
                 0x0000000010002000                la_data_start
                 0x0000000010006010                kstk_bottom
                 0x0000000010002010                kstk_top
 .bss            0x0000000010006030      0x200 start.o
 .bss            0x0000000010006230       0x14 main.o
 .bss            0x0000000010006244       0xa4 debug.o
 .bss            0x00000000100062e8        0x4 utility.o
 .bss            0x00000000100062ec       0x2c clock.o
 .bss            0x0000000010006318        0x4 proc.o
 .bss            0x000000001000631c      0x100 interrupt.o
 .bss            0x000000001000641c       0x24 system.o
 .bss            0x0000000010006440        0x4
/home/i2a/minix/stdlib/libc.a(errno.o)
                 0x0000000010006440                errno
 .bss            0x0000000010006444        0x8
/home/i2a/minix/stdlib/libc.a(exit.o)
                 0x0000000010006444                _clean
                 0x0000000010006448                __funccnt
 .bss            0x000000001000644c        0xc
/home/i2a/minix/stdlib/libc.a(malloc.o)
 .bss            0x0000000010006458       0x90
/home/i2a/minix/stdlib/libc.a(ext_comp.o)
 .bss            0x00000000100064e8      0x1c0 system/ksyslib.a(do_vcopy.o)
 .bss            0x00000000100066a8       0x40 system/ksyslib.a(do_vdevio.o)
 .bss            0x00000000100066e8     0x1488 system/ksyslib.a(do_getinfo.o)
 .bss            0x0000000010007b70        0x4 system/ksyslib.a(do_setalarm.o)
 .bss            0x0000000010007b74      0x16c system/ksyslib.a(do_debug.o)
 .bss            0x0000000010007ce0      0xf04 ./arch/ppc/arch.a(debug_trace.o)
 .bss            0x0000000010008be4        0xc ./arch/ppc/arch.a(system.o)
                 0x0000000010008bec                interrupt_system_ready
                 0x0000000010008be4                idle_calls
                 0x0000000010008be8                memory_system_ready
 .bss            0x0000000010008bf0       0x34
/home/i2a/minix/stdlib/libppc.a(bt.o)
 .bss            0x0000000010008c24       0x10
/home/i2a/minix/stdlib/libppc.a(openpic.o)
 *(.sbss)
 .sbss           0x0000000010008c34       0x40 ./arch/ppc/arch.a(table.o)
                 0x0000000010008c34                do_serial_debug
                 0x0000000010008c38                k_reenter
                 0x0000000010008c3c                mon_return
```

```
                0x0000000010008c40                shutdown_started
                0x0000000010008c44                bill_ptr
                0x0000000010008c48                aout
                0x0000000010008c4c                lost_ticks
                0x0000000010008c50                level0_func
                0x0000000010008c54                proc_ptr
                0x0000000010008c58                mon_sp
                0x0000000010008c5c                irq_use
                0x0000000010008c60                kernel_exception
                0x0000000010008c64                next_ptr
                0x0000000010008c68                last_sysproc_nr
                0x0000000010008c6c                mon_ss
                0x0000000010008c70                prev_ptr
 *(COMMON)
 COMMON         0x0000000010008c74      0x910 debug.o
                0x0000000010008c74                timingdata
 COMMON         0x0000000010009584       0x78 system.o
                0x0000000010009584                call_vec
 COMMON         0x00000000100095fc       0x80
/home/i2a/minix/stdlib/libc.a(exit.o)
                0x00000000100095fc                __functab
 COMMON         0x000000001000967c    0x1544c ./arch/ppc/arch.a(table.o)
                0x000000001000967c                irq_actids
                0x000000001000977c                proc
                0x00000000100184bc                kinfo
                0x0000000010018508                rdy_tail
                0x0000000010018548                krandom
                0x00000000100189c8                kmess
                0x0000000010018ad0                irq_hooks
                0x00000000100191d0                t_stack
                0x000000001001cdd0                ppriv_addr
                0x000000001001ce50                priv
                0x000000001001e8d0                rdy_head
                0x000000001001e910                pproc_addr
                0x000000001001eab0                machine
                0x000000001001eac8                end = .
                0x000000001001eac8                _end = .
                0x000000001001eac8                __end = .
                0x000000001001eac8                __bss_end = .

.rela.dyn

/DISCARD/
 *(.note.GNU-stack)
 *(.comment)
 *(.debug_*)
 *(.eh_frame)
LOAD ./arch/ppc/minix.o
START GROUP
LOAD start.o
LOAD main.o
LOAD debug.o
LOAD utility.o
LOAD clock.o
LOAD proc.o
LOAD interrupt.o
LOAD klib.o
LOAD system.o
LOAD /home/i2a/minix/stdlib/libc.a
LOAD /home/i2a/minix/stdlib/libppc.a
LOAD /home/i2a/minix/stdlib/libgcc.a
LOAD /home/i2a/minix/syslib/libtimers.a
LOAD /home/i2a/minix/syslib/libsys.a
```

```
LOAD /home/i2a/minix/syslib/libsysutil.a
LOAD system/ksyslib.a
LOAD ./arch/ppc/arch.a
LOAD ../drivers/arch/ppc/kscreen/kscreen.a
END GROUP
OUTPUT(elf32/kernel elf32-powerpc)
```

I

Kernel Interfaces

This appendix will list the current hardware interface. It will list of every interface the member of the data structure and additional information. Keeping to C terminology and syntax. This is the “first” version of the hardware interface and the only architecture ported at this moment is the PowerPC. Although the interfaces are not architectural dependent sometimes a note to the PowerPC architecture is made.

I.1 interface.h

```
1   /*********************************************************************\
2     minix/kernel/interface.h
3
4
5     All interfaces the kernel needs.
6
7     The variables (declared const) all start with a capital letter to
8     indicate they are "sort of" special.
9
10    Remember interfaces always work in two ways.
11
12  \*********************************************************************/
13  #ifndef _KERNEL_IF_H_
14  #define _KERNEL_IF_H_
15
16  /* The interrupt functions MinixPPC needs.
17   */
18  typedef struct if_interrupt_s {
19    const char* info;
20    int     (*init)              (void);
21    int     (*get_vector)(void);
22    int     (*set_vector)(int source, int priority, int vector);
23    int     (*ack_vector)(void);
24    int     (*enable)            (int source);
25    int     (*disable)           (int source);
26  } if_interrupt_t;
27
28  /* - Some info about this interface/"low level" driver (could be NULL).
29   * - Do system (hardware) initialization.
30   * - Read out which vector has occurred.
31   * - Set hardware interrupt source to vector mapping (source disabled).
32   * - After handling a interrupt MinixPPC calls this function to tell
33   *   the hardware it is finished with the handling of the interrupt.
34   *   also called eoi {end-of-interrupt} )
35   * - Enable this source (irq line).
36   * - Disable this source (irq line).
37   */
```

```
92./* The clock functions MinixPPC needs (for the clock task).
93. */
94.typedef struct if_clock_s {
95.   const char* info;
96.   int             (*init)       (void);
97.   void    (*stop)       (void);
98.   void    (*start)      (void);
99.   u32_t   (*frequency) (void); /* in Hz (hardware clock(!)) */
100.  clock_t (*read)       (void);
101.} if_clock_t;
102.
103./* - Could point to some info else NULL.
104. * - Init the clock hardware.
105. * - Ticks per second the clock generates.
106. * - Stop the clock (if possible).
107. * - Start the clock (if possible).
108. * - Read the current clock count.
109. * - Function to return the uptime (from the hardware).
110. */
111.
112./* This interface provides functions for use in the whole kernel.
113. */
114.typedef struct if_system_s {
115.  const char* info;
116.  int     (*init)       (void);
117.  void    (*syscall)    (void); /* preform system call */
118.  void    (*lock)       (void);
119.  void    (*unlock)     (void);
120.  int     (*locked)     (void);
121.  void    (*shutdown)   (void);
122.  void    (*reset)      (void);
123.  void    (*idle)       (void);
124.  void    (*sputc)      (int c);
125.  void    (*read_tsc)   (unsigned long* high, unsigned long* low);
126.} if_system_t;
127./* Need to know that this one is going to exist some time from now.
128. */
129.struct proc;
130.
131.typedef struct if_memory_s {
132.  const char* info;
133.  int          (*init)(void);
134.  int          (*alloc_segments)(struct proc* p);
135.  vir_bytes    (*alloc_remote_segment)(u32_t* selector, segframe_t* s,
136.                                        int index, phys_bytes phys,
137.                                        vir_bytes size, int privilege);
138.  void    (*copy_message)(int source, struct proc* p_src,
139.                          message* m_src, struct proc* p_dst,
140.                          message* m_dst);
141.  void    (*phys_copy)(phys_bytes src, phys_bytes dst,
142.                       phys_bytes count);
143.  void    (*phys_memset)(phys_bytes src, int pattern,
144.                         phys_bytes count);
145.  phys_bytes  (*seg2phys)(segdesc_t segdesc);
```

(continuing on next page)

```
146.  /* I/O
147.   *
148.   * These next are endianness by system default. (ea. should big
149.   * for PowerPC, little for Intel). The different functions are
150.   * needed as hardware could be using another endianess than
151.   * the system.
152.   */
153.  void       (*outb)     (volatile u8_t* a, u8_t v);
154.  void       (*outw)     (volatile u16_t* a, u16_t v);
155.  void       (*outl)     (volatile u32_t* a, u32_t v);
156.  u8_t       (*inb)      (volatile u8_t* a);
157.  u16_t       (*inw)     (volatile u16_t* a);
158.  u32_t       (*inl)     (volatile u32_t* a);
159.
160.  void       (*be_outw)  (volatile u16_t* a, u16_t v);
161.  void       (*be_outl)  (volatile u32_t* a, u32_t v);
162.  u16_t      (*be_inw)   (volatile u16_t* a);
163.  u32_t      (*be_inl)   (volatile u32_t* a);
164.  void       (*le_outw)  (volatile u16_t* a, u16_t v);
165.  void       (*le_outl)  (volatile u32_t* a, u32_t v);
166.  u16_t      (*le_inw)   (volatile u16_t* a);
167.  u32_t      (*le_inl)   (volatile u32_t* a);
168.
169.  /* Streams (system default).
170.   */
171.  void   (*insb)      (volatile u8_t* addr, void* b, size_t c);
172.  void   (*insw)      (volatile u16_t* addr, void* b, size_t c);
173.  void   (*insl)      (volatile u32_t* addr, void* b, size_t c);
174.  void   (*outsb)     (volatile u8_t* addr, void* b, size_t c);
175.  void   (*outsw)     (volatile u16_t* addr, void* b, size_t c);
176.  void   (*outsl)     (volatile u32_t* addr, void* b, size_t c);
177.} if_memory_t;
178.
179.#endif /* #ifndef _KERNEL_IF_H_ */
```

I.2 System

Name: info	
Definition: const char* info;	
Parameters: non.	
Return value: non.	
Remarks: Info should be NULL or a pointer to the start of a string of characters describing the driver/hardware the interface is for.	

Example:
For the PowerPC system interface,
`printf("%s.\n", System.info);`
output:
`"system.c PowerPC system interface, v0.01bu compiled on 26 06 06 15:27."`	

Name: init	
Definition: int (*init)(void);	
Parameters: non.	
Return value: 0 on no error, < 0 if a error occurs.	
Remarks: Init is used to initialize the system hardware once. It is called before any other interface function is used. It could be empty and should then return 0.	
Example: `System.init();`	

Name: syscall	
Definition: void (*syscall)(void);	
Parameters: non.	
Return value: non.	
Remarks: Use this to do a system call without parameters. At the moment not used.	
Example: `System.syscall();`	

Name: lock	
Definition: void (*lock)(void);	
Parameters: non.	
Return value: non.	
Remarks: Use this call to disable external interrupts (at the CPU). Note that in MINIX the macro lock is defined. If you need to be sure this call is taken make sure lock is undefined, by "#undef lock".	

Example:
`#ifdef lock`
`#undef lock`
`#endif`
`System.lock();`	

Name: unlock	
Definition: void (*unlock)(void);	
Parameters: non.	
Return value: non.	
Remarks: Use this call to enable external interrupts (at the CPU). Note that in MINIX the macro unlock is defined. If you need to be sure this call is taken make sure unlock is undefined, as in the example.	

Example:
`#ifdef unlock`
`#undef unlock`
`#endif`
`System.unlock();`	

Name: locked	
Definition: void (*locked)(void);	
Parameters: non.	
Return value: 0 if external interrupts are disabled, 1 if external interrupts are enabled.	
Remarks: Use this call to check external interrupts (at the CPU).	

Example:
`if(! System.locked()) printf("external interrupts are disabled.\n");`
`else`
`printf("External interrupts are enabled, jippie.\n");`	

Name: shutdown	
Definition: void (*shutdown)(void);	
Parameters: non.	
Return value: non.	
Remarks: Use this call to shutdown (or power down) the system. It does nothing more than that, calling this function is a no return.	

Example:
`System.shutdown();`
`printf("never reached.\n");`	

Name: reset	
Definition: void (*reset)(void);	
Parameters: non.	
Return value: non.	
Remarks: Use this call to reset the system. The same a with shutdown, point of no return.	

Example:
`System.reset();`
`printf("never reached.\n");`	

Name: idle	
Definition: void (*idle)(void);	
Parameters: non.	
Return value: non.	
Remarks: The idle call should let the CPU call power saving instructions and methods. It should be called somewhere in the IDLE task (loop) that the OS has.	

Example:
`/* some where in the IDLE task. */`
`System.idle();`	

Name: sputc	
Definition: void (*sputc)(int c);	
Parameters: int c, character to be printed to the system output.	
Return value: non.	
Remarks: If the system has a general output output device, serial line or (like the ibook) a simple screen driver. The sputc function will print or send the character.	
Example: `System.sputc('A');`	

Name: read_tsc	
Definition: void (*read_tsc)(unsigned long* high, unsigned long* low);	
Parameters: unsigned long* high, address for high part of 64 bit return value, unsigned long* low, address for low part of 64 bit return value,	
Return value: non.	
Remarks: "All" system have a constant timer that increases a register with "1" every tick. In the PowerPC architecture call "time base". It can be used to save/record the uptime of the system. This is for the current modern systems a 64 bit register.	

Example:
`unsigned long tbh;`
`unsigned long tbl;`
`System.read_tsc(&tbh, &tbl);`	

I.3 Memory

Name: info	
Definition: const char* info;	
Parameters: non.	
Return value: non.	
Remarks: see System.info();	
Example: `/* see System.info; */`	

Name: init	
Definition: int (*init)(void);	
Parameters: non.	
Return value: 0 on no error, < 0 if a error occurs.	
Remarks: This function needs to map all extra needed memory for the kernel. For the PowerPC the register map for the OpenPIC interrupt controller is mapped here.	
Example: `Memory.init();`	

Name: alloc_segments	
Definition: int (*alloc_segments)(struct proc* p);	
Parameters: struct proc* p, pointer to the process that needs to be mapped (most of the time a into the process table).	
Return value: Total number of pages mapped.	
Remarks: This function updates the process segment registers and maps pages using page address translation. It uses the physical memory map of the given process to define the initial stack pointer and virtual addresses. These are then updated to the process. After a call to this process the virtual start address of the text/data and stack segment are updated and if the stack-pointer was zero entering this function a new stack pointer as well. Only after a use of this function the process is runnable.	

Example:
`struct proc* rp = proc_addr(1);  /* get INIT process pointer */`
`Memory.alloc_segments(rp);  /* map INIT process.`	

Name: alloc_remote_segment	

Definition:
vir_bytes (*alloc_remote_segment) (u32_t* selector, segframe_t* s, int index, phys_bytes phys, vir_bytes size, int privilege);

Parameters:

u32_t* selector,	Value to select the segment the remote segment.
segframe_t* s,	The remote segment frame of the process mapping for.
int index,	Segment index to use, any segment above the stack segment will do.
phys_bytes phys,	Physical address to map.
vir_bytes size,	Size of memory block to map in bytes.
int privilege,	User or supervisor privilege.

Return value:
The offset into the mapped segment to get the physical address requested, if the physical address requested is on a page click this offset is zero.
The return values can be implementation specific. The return selector at the PowerPC is a number that is a multiple of 0x1000_0000 while on the x86 it will be a index into the LDT.
The code requesting the remote segment “knows” on which architecture it runs, so it should handle the return values in the right way.

Remarks:
This function shall map any physical memory to a segment that the “calling” process can access.

Example:
```
/* How the value are to be interpreted for the PowerPC
architecture.
 * The PowerPC code should use the return values to create the
 * virtual address to use the segment.
 */
u32_t selector;
u32_t offset;
u32_t vir_address;
struct proc* rp = proc_addr(m_ptr->m_source); /* process to map for
*/

offset = Memory.alloc_remote_segment(&selector, &rp->p_seg, 5,
0xf5000000,
                                     0x10000, USER_PRIVILEGE);
vir_address = offset + selector;

/* vir_address is using remote segment index 5 so selector is
0x5000_0000,
 * using this code at the caller
 */
int* tmp = (int*)0x5000_0000;

tmp = 0x1234; / this sets the integer at physical address
0xf500_0000 */
```

Name: copy_message	
Definition: void (*copy_message) (int source, struct proc* p_src, message* m_src, struct proc* p_dst, message* m_dst);	
Parameters: int source, Source process, struct proc* p_src, Source process pointer, message* m_src, Source message pointer in source process context, struct proc* p_dst, Destination process, message* m_dst, Destination message pointer in destination process context.	
Return value: non.	
Remarks: This function should be able to copy a message content form any process context (in data or on the stack) to any process context. Note: this function is only used in the './kernel/proc.c' file.	
Example: `Memory.copy_mess(caller_ptr->p_nr, caller_ptr, m_ptr, caller_ptr, m_ptr);`	

Name: phys_copy	
Definition: void (*phys_copy)(phys_bytes src, phys_bytes dst, phys_bytes count);	
Parameters: phys_bytes src, Physical source address. phys_bytes dst, Physical destination address. phys_bytes count, Byte count.	
Return value: non.	
Remarks: This function copies count bytes from src address to dst address, through the "caching" of the architecture making sure that the data is written to RAM.	
Example: `phys_copy(0x100, 0x2000, 20);`	

_Name: phys_memset_	
Definition: void (*phys_memset)(phys_bytes src, int pattern, phys_bytes count);	
Parameters: phys_bytes src, Physical source address. int pattern, Pattern to write per byte. phys_bytes count, Byte count.	
Return value: non.	
Remarks: Setting the memory as pattern for count bytes. Although pattern is a integer is is truncated to a byte.	
Example: `phys_copy(0x100, 'A', 20);    /* setting 20 bytes to 'A' */`	

Name: seg2phys	
Definition: phys_bytes (*seg2phys)(segdesc_t segdesc);	
Parameters: segdesc_t segdesc, Segment descriptor.	
Return value: start physical address of segment.	
Remarks: Given segment descriptor segdesc return the physical start address. Note that the segdesc_t is defined in the architectural include.	

Example:
`/* getting the physical start address of segment 4 (the fifth segment). */`
`segdesc_t segdesc = mfsr(4);`
`phys_bytes phys_addr = seg2phys(segdesc);`	

Name: (be),(le_)outx_	**_For x; b (byte 8), w (word 16), l (long 32)._**
Definition: void (*out**x**)(volatile u**x**_t* a, u**x**_t v);	
Parameters: volatile u**x**_t* a, Destination address, u**x**_t v, Value	
Return value: non.	
Remarks: Writing a byte (8 bit), word (16 bit) or long (32 bit) value v to address a. The prefix le_ or be_ indicate the byte order of the function, little endian or big endian. Note that the byte version does not have a prefix.	

Example:
`outb(0x10000, 'A'); /* write byte */`
`le_outl(0x10000, 0xdeadc0de); /* write in little endian */`	

Name: (be_),(le_)inx	***For x; b (byte 8), w (word 16), l (long 32).***
Definition: u**x**_t (*in**x**)(volatile u**x**_t* a);	
Parameters: volatile u**x**_t* a, Source address.	
Return value: A byte, word or long size value.	
Remarks: Reading a byte (8 bit), word (16 bit) or long (32 bit) value from address a. The prefix le_ or be_ indicate the byte order of the function, little endian or big endian. Note that the byte version does not have a prefix.	

Example:
`u32_t v1 = inl(0x10000);     /* system default */`
`u32_t v3 = be_inl(ox10000); /* big endian */`	

Name: insx	***For x; b (byte 8), w (word 16), l (long 32).***
Definition: u**x**_t (*ins**x**)(volatile u**x**_t* addr, void* b, size_t c);	
Parameters: volatile u**x**_t* addr, Source address, void* b, Destination buffer, size_t c, Count.	
Return value: non.	
Remarks: Reading byte (8 bit), word (16 bit) or long (32 bit) values from the address 'addr' to buffer 'b', until count 'c' elements have been done.	

Example:
`u8_t buffer[12];`
`insl(0x10000, buffer, 3); /* read three longs from address 0x10000 to buffer */`	

***Name: outs*x**	***For x; b (byte 8), w (word 16), l (long 32).***
Definition: u**x**_t (*outs**x**)(volatile u**x**_t* addr, void* b, size_t c);	
Parameters: volatile u**x**_t* addr, Destination address, void* b, Source buffer, size_t c, Count.	
Return value: non.	
Remarks: Writing byte (8 bit), word (16 bit) or long (32 bit) values from the buffer 'b' to address 'addr', until count 'c' elements have been done.	

Example:
`u8_t buffer[12];`
`outsb(0x10000, buffer, 12); /* writing 12 bytes from buffer to address 0x10000 */`	

I.4 Interrupt

Name: info	
Definition: const char* info;	
Parameters: non.	
Return value: non.	
Remarks: see System.info();	
Example: `/* see System.info; */`	

Name: init	
Definition: int (*init)(void);	
Parameters: non.	
Return value: 0 on no error, < 0 if a error occurs.	
Remarks: All sources are mapped to produce the same vector as line number with priority 8 (of 0 -15), ending with all timers and interrupts disabled.	
Example: `Interrupt.init();`	

Name: get_vector	
Definition: int (*get_vector)(void);	
Parameters: non.	
Return value: The interrupt source that is interrupting.	
Remarks: When the "general" interrupt occurs (the CPU external interrupt) use this function to request the interrupt source.	
Example:	

```
irq_hook_t* hook = NULL;
int irq;

irq = Interrupt.get_vector();
hook = irq_handlers[irq];  /* get the irq hook list for this vector
*/
```

Name: set_vector	
Definition: int (*set_vector)(int source, int priority, int vector);	
Parameters: int source, Source line to set, int priority, Priority of vector produced, int vector, Vector to produce on interrupt.	
Return value: 0 on no error, < 0 if a error occurs.	
Remarks: Map a source line, line as in the hardware line in the system, to produce the vector on interrupt. So line 12 could produce vector 61. (not used at the moment, no redefinition of default).	

Example:
`irq_hook_t* hook = NULL;`
`int irq;`
 	
`irq = Interrupt.get_vector();`
`hook = irq_handlers[irq];  /* get the irq hook list for this vector */`	

Name: ack_vector	
Definition: int (*ack_vector)(void);	
Parameters: non.	
Return value: 0 on no error, < 0 if a error occurs.	
Remarks: Ack the last pending interrupt that has been get via get_vector so a new could arrive.	

Example:
`do_interrupt_handler(hook);   /* first handle IRQ */`
`Interrupt.ack_vector();       /* then acknowledge */`	

Name: enable	
Definition: int (*enable)(int source);	
Parameters: line or source number.	
Return value: 0 on no error, < 0 if a error occurs.	
Remarks: Use this function to enable (or unmask) the interrupt source line at the interrupt controller. You need to use this function first to get a external interrupt from the “device”.	
Example: `Interrupt.enable(GPIO); /* enable the MacIO gpio interrupt */`	

Name: disable	
Definition: int (*disable)(int source);	
Parameters: line or source number.	
Return value: 0 on no error, < 0 if a error occurs.	
Remarks: Use this function to disable (or mask) the interrupt source line at the interrupt controller.	
Example: `Interrupt.disable(GPIO); /* disable the MacIO gpio interrupt */`	

I.5 Clock

Name: info

Definition:
const char* info;

Parameters:
non.

Return value:
non.

Remarks:
see System.info();

Example:
```
/* see System.info; */
```

Name: init

Definition:
int (*init)(void);

Parameters:
non.

Return value:
0 on no error,
< 0 if a error occurs.

Remarks:
This function will initialize the clock hardware to set up a periodic (extern) interrupt every 1/60 second or 60 Hz. For the PowerPC one of the timers on the OpenPIC controllers is used. Note that the timer should be running when init is done.

Example:
```
Clock.init();
```

Name: stop

Definition:
void (*stop)(void);

Parameters:
non.

Return value:
non.

Remarks:
Stop the clock with running, not masking but inhibiting the counting process.

Example:
```
Clock.stop();
```

Name: start	
Definition: void (*start)(void);	
Parameters: non.	
Return value: non.	
Remarks: Start the clock, running with whatever is in the count register.	
Example: `Clock.start();`	

Name: frequency	
Definition: u32_t (*frequency)(void);	
Parameters: non.	
Return value: The speed in Hz of the hardware clock(!).	
Remarks: Use this function to know how fast the hardware clock is running. (not used at the moment).	
Example: `Clock.frequency();`	

Name: read	
Definition: u32_t (*read)(void);	
Parameters: non.	
Return value: The current value of the count down register.	
Remarks: Use this function to read the value of the register that is counting down and when zero produces the clock interrupt.	
Example: `Clock.read();`	

J

Open Firmware

J.1 Welcome screen

```
Apple PowerBook6,5 4.8.7f1 BootROM built on 09/23/04 at 16:13:38
Copyright 1994-2004 Apple Computer, Inc.
All Rights Reserved.

Welcome to Open Firmware, the system time and date is: 01:05:31
04/16/1904
  The battery capacity is: 87 precent

To continue booting, type "mac-boot" and press return.
To shut down, type "shut-down" and press return.

 ok
0 > _
```

Note, the system time is incorrect :).

J.2 Some useful commands

All commands are from the Apple technotes, but these are the ones used the most (by the author).

boot	Boot using the default boot device.
dev /	Change to the "root" of the device tree.
dev [device]	Change to device.
.properties	View node properties.
ls	List devices on current node.

/* THE END */