

Title

Linux 2.2 on Ericsson MPC860 platforms

Author

UAB/D/SF Leif Lindholm



Abstract

This report describes the process of porting the Linux 2.2 kernel to Ericsson mpc860 platforms in general and the DCP1 / IPONAX Access Unit in particular.

The report describes the problems encountered, the solutions, the results and ideas for continued development towards a complete stand-alone Linux system on this hardware.

It is assumed that the reader has at least some kind of experience with Unix systems, preferably (but not necessarily) on the administrative side.

The reader should also be familiar with the PowerPC processor family – or at least have a users manual within reach.

Someone who wants to find out more about Linux on the PowerPC will hopefully find something interesting/helpful in here.



Title

Linux 2.2 on Ericsson MPC860 platforms

Author

UAB/D/SF Leif Lindholm



Preface

The work behind this bachelor thesis was my first real attempt at both Linux kernel development and low-level hardware programming. During these few months, I have learned a lot about the PowerPC processors, general computer architecture and about the hardware issues of a customized platform.

I would like to thank the following people:

Jonas Andersson (D/HG) - for finding me the assignment, helping me get started and repairing the network interface on the DCP1.

Mårten Sundling (manager D/SF, my supervisor) - for letting me do something that some may still perceive as "not serious" and supporting me through the entire process.

Lars Jönsson (D/SF) - for answering my millions of questions and for the source code for hexsplit.

Daniel Lundqvist - for helping with the loader.

Mats Sandberg (D/SF) - for TGC.

and of course:

Linus Torvalds - for starting it all

Cort Dougan - for writing the MBX port

and all the others involved in the Linux on PPC development.

All penguins in this report were created by Larry Ewing (lewing@isc.tamu.edu).

"_Sometimes_ it happens that I just see the disassembled code sequence from the panic, and I know immediately where it's coming from. That's when I get worried that I've been doing this for too long"

Linus Torvalds



Title

Linux 2.2 on Ericsson MPC860 platforms

Author

UAB/D/SF Leif Lindholm



Title



Linux 2.2 on Ericsson MPC860 platforms

Author

UAB/D/SF Leif Lindholm

Contents

1	INTRODUCTION.....	7
1.1	NAMES.....	7
1.2	FINDING THE ASSIGNMENT	7
1.3	THE ASSIGNMENT	8
1.4	CONVENTIONS.....	8
2	WHAT IS LINUX?	9
2.1	DISTRIBUTIONS.....	9
2.2	AVAILABLE SOFTWARE	10
2.3	TECHNICAL ASPECTS.....	11
2.3.1	<i>Process handling.....</i>	<i>11</i>
2.3.2	<i>Memory.....</i>	<i>12</i>
2.3.3	<i>Networking.....</i>	<i>12</i>
2.4	THE DEVELOPMENT ENVIRONMENT	13
2.4.1	<i>The kernel source tree.....</i>	<i>13</i>
3	THE TARGET HARDWARE.....	15
3.1	MPC860.....	15
3.2	DCP1.....	15
3.3	IPONAX ACCESS UNIT	15
4	THE WORK ITSELF	16
4.1	THE APPROACH	16
4.2	A HEAD START	16
4.2.1	<i>Setting up a development environment.....</i>	<i>16</i>
4.2.2	<i>Beginning the porting.....</i>	<i>17</i>
4.3	THE "FIRST" DAY	17
4.4	GETTING THROUGH THE LOADER.....	18
4.5	GETTING PAST THE ARCHITECTURE SETUP CODE	19
4.6	INSIDE START_KERNEL.....	20
4.6.1	<i>Getting the console to work</i>	<i>20</i>
4.6.2	<i>Memory and device initialization.....</i>	<i>21</i>
4.6.3	<i>Network.....</i>	<i>22</i>
4.7	(NOT) STARTING INIT	23
4.8	MOVING TO 2.2.5	24
4.9	RUNNING THE SYSTEM	25
5	THE RESULT.....	26
5.1	CURRENT STATUS	26
5.2	DEMONSTRATION	26
6	WHAT NEXT?.....	28
6.1	PLANNED FOLLOW-UP	28
6.2	WHAT REMAINS TO BE DONE.....	28
6.3	KNOWN BUGS	28
6.4	TESTING	28
7	GLOSSARY AND TERMS	29

ERICSSON 	6(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

8	REFERENCES	30
8.1	ELECTRONICALLY AVAILABLE	30
8.2	PRINTED	30
9	THE HARDWARE.....	31
9.1	LAUTERBACH	31
9.2	WORKSTATIONS	31
10	THE SOFTWARE.....	32
10.1	OPERATING SYSTEMS.....	32
10.2	PBOOT.....	32
10.3	COMPILING ENVIRONMENT	32
10.4	THE NFS-ROOT FILESYSTEM.....	32
10.5	OTHER SOFTWARE	33



1 INTRODUCTION

The work took place at Ericsson UAB in Älvsjö, Sweden - starting March 15:th and ending May 28:th 1999.

1.1 NAMES

The people figuring in this report (besides me) are:

Jonas Andersson	SDF ¹ member. Hardware developer. One of the constructors of the DCP1.
Mårten Sundling	Manager UAB/D/SF, my supervisor
Lars Jönsson	Programmer/general guru at UAB/D/SF
Michael Laajanen	Consultant from Enea ² . Another DCP1 constructor.
Daniel Lundqvist	System administrator at Dagens Nyheter ³
Linus Torvalds	Creator of Linux
Cort Dougan	Author of the Motorola MBX Linux port
Richard Stallman (RMS)	Founder of the GNU project and the Free Software Foundation

1.2 FINDING THE ASSIGNMENT



I've been a member of SDF since 1994, and there I have learned much of what I know today about computers. One day in the beginning of 1999 I had a conversation with some other SDF members. They told me that Jonas wanted someone to port Linux to some sort of telephone equipment he had designed at Ericsson UAB. I thought this sounded perfect for my thesis project, which was up in a couple of months, so I called him immediately. He thought this sounded perfect and would check with his manager.

A few weeks later, Jonas told me he had spoken with Mårten Sundling, the manager of the operating systems department, and said that I should meet with him to discuss the details. A few days later, I did. He seemed very positive about it, and we decided that I would begin working on the 15:th of March.

¹ Södertälje DataFörening - a computer club in Södertälje, running old/not-so-old UNIX and VMS computers.
– <http://www.sdf.nu/>

² Enea Data - <http://www.enea.se/>

³ Swedish newspaper – <http://www.dn.se/>

ERICSSON 	8(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

1.3 THE ASSIGNMENT

- To, as a demonstration / an experiment, port a Linux kernel to a mpc860 board constructed by Ericsson.
- On this system, demonstrate something “visibly productive”.

This project will provide a basis for a more formal evaluation of an embedded system based on Linux.

1.4 CONVENTIONS

The following typographic conventions are used throughout this report:

PowerPC ***REGISTERS*** and ***instructions***.

Kernel **functions** and **variables**.

Console *in-/output*.

Kernel CONFIG_URATION_OPTION (Configuration options in the Linux kernel are called CONFIG_xxx, where xxx is a name for the support the option provides).

Reference to documentation inside square brackets – “Linux FAQ”[1].



2 WHAT IS LINUX?

The short-short version is available in section 1.1 of the “Linux FAQ”[1]:

Linux is the free Unix written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers from across the Internet. Linux aims towards POSIX compliance, and has all of the features you would expect of a modern, fully fledged Unix: true multitasking, virtual memory, shared libraries, demand loading, shared, copy-on-write executables, proper memory management, and TCP/IP networking.

However, that is a rather thin explanation, so this section will explain it more thoroughly.

Linux was originally written for Intel x86 family of processors and compatibles, but is now also available for many other processors. The more stable ones of these at this point would be the DEC Alpha, the ARM, the Sun SPARC and UltraSPARC, the Motorola 68k and the PowerPC.

Linux is distributed under the GNU¹ Public License, mostly referred to as GPL. It is available at <http://www.gnu.org/copyleft/gpl.html>.

Today it has developed into a system able to take on any task given it as a workstation, server or router/firewall. The development towards embedded systems is well underway - its main advantage for this market being a small and efficient kernel (for an operating system – compared to most real-time kernels it’s probably huge). Other very important reasons are its extreme flexibility (which it of course shares with all open-source operating systems) and its range of available software – free and commercial.

2.1 DISTRIBUTIONS



Linux is really the name of the operating system kernel, but has come to represent any operating system using this kernel and the GNU system software. Such systems are called *Linux distributions*, although there are some people (led by Richard Stallman) who insist that they should be called GNU/Linux.

The largest of these distributions are:

- **Red Hat:** <http://www.redhat.com/>
- **Debian:** <http://www.debian.org/>
- **SuSE:** <http://www.suse.com/>
- **Caldera:** <http://www.calderasystems.com/>
- **Slackware:** <http://www.slackware.com/>

Also worth mentioning is the relatively new “Yellow Dog” (<http://www.yellowdoglinux.com>) which is dedicated to making Linux on PowerPC systems more stable and up to date.

¹ Gnu’s Not Unix – <http://www.gnu.org/>

ERICSSON 	10(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

There are also other distributions (none worth mentioning here), most of them based on the Red Hat distribution and then modified or/and specialized in some way.

There are really two different sorts of Linux distributions: the free and the commercial. The free versions are available from all of the manufacturers, including the commercial ones. Since most of the software is distributed under the GPL (or similar) it may not be charged for, and must be freely (and easily) available. Because of this, you can download any Linux distribution directly from the Internet. If you want it on CD, you have to pay for it - it is legal to charge for the material, work and transport, normally a low cost. Some vendors also bundle a CD (or many) with a manual, and are then free to charge what they want.

The commercial versions are just free versions bundled with some commercial software – like an accelerated X-server or a word processor. Some also include customer support and thick manuals.

The least commercially oriented distributions are Slackware and Debian. Debian is very closely knit to GNU and because of this, it has adopted GNU's view of things - taking the official name Debian GNU/Linux. Slackware has strong connections to the CD manufacturer Walnut Creek, while Debian has no real commercial connections at all.

2.2 AVAILABLE SOFTWARE

Today, practically everything one may need is available for the x86 version of Linux. If the software in question is distributed in source form, it can usually just be recompiled for any Linux platform that uses a compatible set of libraries. If it is distributed in binary form, one has to convince the creators to recompile it to another platform.

Software freely available for Linux in source form includes (but is in no way limited to):

- Samba – for acting as a file/printer server for windows machines
- Apache – the worlds most used web server software
- Sendmail – the (at least on UNIX systems) most used mail server software
- DHCPD – for dynamically assigning IP-addresses and other network parameters to workstations
- DHCPD – for requesting the same information from a DHCP server
- The GNU development tools – compilers, cross-compilers, debuggers and more
- XFree86 – a freely redistributable implementation of the X Window System
- BIND – The most used internet DNS server
- Staroffice, Applixware, Word Perfect – Different office-programs, compatible with the most common document formats.



Title
Linux 2.2 on Ericsson MPC860 platforms

Author
UAB/D/SF Leif Lindholm

2.3 TECHNICAL ASPECTS

This section provides a short, technical description of the Linux kernel and operating system.

A much more complete explanation is found in “The Linux Kernel”[3] – this can be considered the short-short version.

2.3.1 Process handling

When the Linux system boots up it spawns the init thread (which becomes process #1), which in turn spawns a few kernel daemon threads and executes the init program residing in the root filesystem. The init program then starts a number of new processes as configured. The original thread that spawned init becomes the idle loop, process #0 (and creates idle loops for the rest of the processors on SMP¹ systems).

Each process has a *task_struct* data structure describing it. This structure contains information about:



- PID – process identifier (unique)
- UID and GID – user and group identifiers: used for access control
- State – runnable, waiting, stopped or zombie (dead process that still has a *task_struct*)
- Scheduling – information for the scheduler (next/previous pointers, priorities)
- Relationships – parent, siblings (processes with the same parent) and children
- Virtual memory – processes virtual address space (none for kernel threads and daemons)
- Filesystem – open files, current working directory and the directory where the program is located in the filesystem
- Time and timers – creation time, CPU time and interval timers
- CPU info – (on SMP systems) the currently used CPU and the last used CPU

All of these data structures are collected into the *task* array, which by default has 512 possible entries. This makes 512 the maximum number of active processes, but that is easily reconfigured. Each process can access its own *task_struct* through a *current* pointer.

Apart from normal processes, Linux also support real time processes – which are treated different from other processes by the scheduler, bypassing all other processes in the system. Two types of real time processes are supported: “round robin” and “first in first out”.

The scheduler decides which process deserves running by comparing various parameters. It is not a very complicated scheduler, but it gets the job done. It compares for instance priorities and time since last execution. On SMP systems, it also gives a “bonus” to a runnable process that last executed on a CPU that is now idle (since switching CPUs generate overhead). When a process has to wait for something, the scheduler takes over, giving the CPU to the most deserving runnable process. To not allow runaway (or CPU-intensive) tasks to take hold of the CPU, the scheduler has been made pre-emptive, re-scheduling after a *time-slice* unit.

¹ Symmetric MultiProcessing

ERICSSON 	12(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

All of this makes Linux a very efficient multitasking operating system. It has no problems running for example a rather frequently accessed web-/mail-/nameserver on a 486 with 16 Mb of RAM.

2.3.2 Memory

Linux fully supports virtual memory, dividing the 32-bit address space into user space (0x0 to 0xbfffffff) and kernel space (0xc0000000 and up). This gives kernel threads access to all of the virtual memory, while processes in “userland” cannot mess up the kernel (without help from the inside). On 64-bit platforms, the implementation is different - but that is beyond the scope of this report.

The kernel uses 4 or 8kb pages depending on architecture, 4 on ppc. If the system runs out of memory and contains some form of secondary storage, pages not recently used may be swapped out to this - freeing RAM for running processes.

A new user process is mapped in at a virtual user space address, having no access to any other memory in the system. This memory protection makes it impossible for one process to overwrite memory belonging to another process.

Linux also supports shared libraries and executables. The shared libraries are collections of commonly used functions that all applications can access. These are loaded into memory only once no matter how many processes are using it, saving a lot of memory. The executables can also share common parts of memory. This allows for example a web-server to spawn new servers to deal with many connections, without having duplicate machine code in memory. Only the data areas of the processes are separate.

When a program is started, only the first small section is actually loaded from disk. As the process executes, it will generate page faults when trying to access code not yet in primary memory. At this point, the requested block is loaded into RAM. This technique is called demand loading.

2.3.3 Networking

Linux is, because of its open source policy, always very up to date as to new features, implementations and fixes of exploits.

When for example the “ping of death”¹ bug was discovered it was a matter of hours before users were able to download the patch, while users of most other operating systems had to wait for weeks.

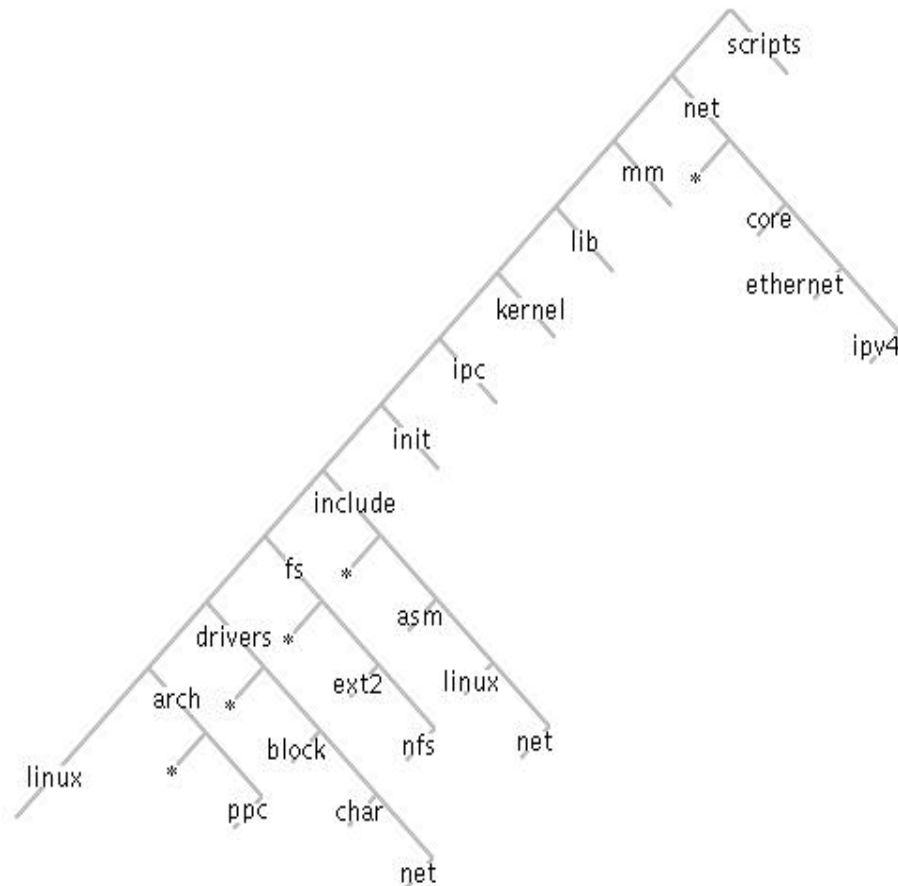
This also means that Linux always is very up to date with new protocols; it has had support for IPV6 for years. It supports for example TCP/IP, IPX, AppleTalk and X25. It contains support for routing, firewalling, masquerading (address translation), bridging and tunnelling. This has made Linux a very popular candidate for routers/firewalls.

¹ In the end of 1996, a programming error in the networking code of most operating systems was discovered. This bug allowed anyone to in some cases crash a remote computer.



2.4 THE DEVELOPMENT ENVIRONMENT

2.4.1 The kernel source tree



* = Directories of no importance for this project.

Description of directory structure in alphabetical order:

- **arch:** Subdirectories for the various architectures. These, in turn, contain the architecture specific source code.
- **drivers:** Subdirectories for various device drivers. The ones used in this project are located in **block** (for block devices), **char** (for character devices) or **net** (for networking drivers). The hardware network drivers for mpc8xx are located in the architecture specific section, but the configuration scripts here are necessary.
- **fs:** Subdirectories for the various filesystems supported by Linux. Also contains the general filesystem code.
- **include:** Subdirectories for asm-<architecture>, of which the current architecture is symbolically linked to **asm**. It also contains the architecture independent **linux** and **net**.

Title

Linux 2.2 on Ericsson MPC860 platforms

Author

UAB/D/SF Leif Lindholm



- **init:** Only the two files **main.c** and **version.c**. All that the version.c file does is to create a character string presenting the version of the kernel and information about its compilation. In main.c, all of the architecture independent setup is performed – some of it calling code in the architecture specific section.
- **ipc:** The Linux implementation of interprocess communication. Completely architecture independent.
- **kernel:** Internals of the kernel. This includes process control, scheduling, time handling and error handling (including **panic()**).
- **lib:** Mostly some overridable, generic functions for architectures that don't have optimised assembly implementations of these. It also contains some functions that no architecture has implemented optimised – like **vsprintf()**.
- **mm:** All of the architecture independent memory management code.
- **net:** Network protocol drivers – both at the media and the network level. Directories used by this 860 port are **core**, **Ethernet** and **ipv4**.
- **scripts:** Various scripts and binaries used/created during configuration and compilation.



3 THE TARGET HARDWARE

This section describes the hardware targeted for the porting.

3.1 MPC860

The Motorola MPC860 is a derivate of the MC68360 QUICC (Quad Integrated Communications Controller) and it is also known as the PowerQUICC. It is a 32-bit PowerPC integrated communications controller. It consists of a PowerPC core (without floating point support) and a communications processor module (*CPM*).

The *CPM* contains six communication circuits: four *SCCs* and two *SMCs*. The Serial Communication Controllers are full duplex and support for example Ethernet, UART, IrDA and transparent operation. The Serial Management Controllers are also full duplex, but support only UART, GCI (ISDN) and transparent operation.

The 860 also have a PCMCIA controller (2 sockets), an enhanced memory controller and a real-time clock integrated.

3.2 DCP1

The DCP1 is an Ericsson mpc860 board, which was used for the first 4-5 weeks of the project. It is clocked at 36MHz and has 16Mb RAM, 4Mb Flash memory, four DSPs and capabilities for both 10baseT and 10base2 Ethernet (although not at the same time).

Its 10baseT interface had never been verified, and was in fact defective. This was corrected during a day of studying sketches and measuring pin voltages and 5 minutes of soldering.

The DCP1 uses the Ericsson PBOOT boot monitor as primary boot program.



3.3 IPONAX¹ ACCESS UNIT

The IPONAX AU was the targeted platform for the porting. Due to a shortage of boards, a DCP1 was used during the first part of the project. It is much like the DCP1, but the clock was at 48MHz on the board used. It has only one DSP, but it is faster than the ones on the DCP1.

It does not have any direct support for 10base2, but has a connector for a second network interface (for simultaneous connection to two different networks).

The AU also uses the PBOOT, although a more recent version.

¹ <http://iponax.uab.ericsson.se/>

ERICSSON 	16(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

4 THE WORK ITSELF

This section describes the actual work of the porting in the form of a project diary.

4.1 THE APPROACH

Jonas told me about a port already made for the Motorola MBX embedded controller (by Cort Dougan), and said that was a good starting point. Both these boards use the Motorola PowerPC 860 (PowerQUICC) integrated microprocessor, and make use of the built-in interfaces for Ethernet and serial communications.

Of course, many things differ. However, the majority of these are beyond what is necessary to get a Linux system up and running. The largest difference was in that the MBX was initialized by the Motorola EPPCBUG monitor, while our card had the Ericsson PBOOT/BABS manager. This was the source of much of my problems (/work).

What caused the most of the problems was that according to the PReP¹, the monitor program should build up a data structure later used by the OS. This struct is called “board information struct” or “residual data” and is configured by the BIOS/boot monitor. It will then be passed to the operating system as a pointer in register 3. When I started the porting, I didn’t know how much of the kernel that depended on this information – which led me to believe that it could just be ignored. Later I learned that the system clock, Ethernet hardware address, console speed and memory configuration were among the things placed into it.

4.2 A HEAD START

4.2.1 Setting up a development environment

Before the porting could begin, a working cross compiler and some debugging tools were needed.

Some weeks before the official start of the project, Jonas downloaded zImage.mbx from linuxppc.cs.nmt.edu. He then tried running it by loading it into the Lauterbach Trace32 debugging program on a DCP1 board. This board was connected to the parallel port of a Toshiba Libretto through a Lauterbach BDM. We also had the console port of the board connected to a nearby Sun SPARCstation running tip against its serial port. A very nice feature of the Trace32 program was that it displayed the ELF symbols, making it much easier to keep track of the execution.

This kernel would start running, execute a few instructions, before it died in **serial_init**. Stepping over, without running, **serial_init** made it continue to the first **udelay** (delay for given number of microseconds), which it never left.

¹ Powerpc Reference Platform – see glossary



We then started trying to get a working cross compiler (linux-x86 -> linux-ppc), which we wanted to be done before the "real" start of the project. This wasn't the easiest thing to do since the documentation available was incorrect on many accounts. It wasn't until we found the "crossgcc-patch" at ftp.cygnum.com that we were even able to get it to compile. Even after that, there was some work (performed in parallel by me at home on my Debian GNU/Linux 2.0 and by Jonas at Ericsson on a Red Hat Linux 5.2) before we got all the way through the compiling.

After that, the Makefile in the top directory of the linux-vger-2.2.1-990216 source tree needed some tweaking to make it understand the kernel was to be cross-compiled.

This kernel did not compile at all - it failed, complaining a lot about Coda¹. Jonas removed all the faulting references until the compile went through. After that, the kernel acted much the same as the zImage.mbx kernel.

4.2.2 Beginning the porting

The first thing to do was to get a working kernel loader. The task of the Linux loader is to unpack the kernel to the correct location and perform the hardware setup necessary so that the kernel can make it from there on its own. Corts MBX loader was used as a basis and it was modified until it seemed to work. The loader source code resides in a subdirectory to /arch/ppc – in this case one called "mbxboot".

On the 9:th of March, we started our first serious work session (still before the official start). Me, Jonas and Daniel sat down for some hours, finding out where things went wrong. We removed the call to **serial_init**, since we knew the PBOOT had already set up the serial ports. After an hour or two of small changes in the assembler source file head.S, we got our first console output. The loader still didn't make it past **udelay**. We kept at it for a few more hours and stepping over the call to **udelay**, we finally got to "Uncompressing Linux..." but then it said "bad gzipped data." and branched to **hang**.



On the 12:th, I was back and got **udelay** working. It seems that PBOOT doesn't initiate the timebase counter (which, looking at the source, EPPCBUG obviously does). Initiated the timebase, and got through the loop correctly. The "bad gzipped data." was still there, but at least there had been some noticeable progress.

4.3 THE "FIRST" DAY

Time to settle in and decide the course of the project.

On the morning of the 15:th, I came in early (8:15), to get the necessary papers signed and to get my access card. After that, I had a small getting-started-meeting with Mårten, which Jonas attended. Here we discussed the duration of the project, the expected result and the approach to the problem. We also talked about follow-up meetings.

¹ An advanced distributed network filesystem developed by the people at CMU. Info available at: <http://www.coda.cs.cmu.edu/>

ERICSSON 	18(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

The planned time for the job was 10 weeks, but in an emergency, an extension by a couple of more weeks was possible.

The outcome of the project should be that something visually productive (a web-server for example) should be running on the board.

The first approach was decided to be getting the complete development environment (TRACE32, compilers, serial console) running on a Solaris workstation. After that, the main thing was to get the loader working so that I could start on the actual kernel.

It was decided that every Friday 15:45 there would be a follow-up meeting. On these we would discuss progress/setbacks during the week, if I were still on schedule and what the following weeks goals would be.

By the afternoon of that first day, I had a working cross-development environment set up.

4.4 GETTING THROUGH THE LOADER

Since the project was now officially underway, it was time to get through the loader to get a look at the real kernel – which should be the main part of the project.

The first approach was to solve the "bad gzipped data." problem. After a while, we discovered that the part of the zImage that was the compressed kernel didn't get loaded into memory. After a tip from Lars, we examined it with ddump¹. When using that to dump the zImage to a SREC file, it too ignored the compressed kernel part.

Discovering that, we decided to manually load the compressed kernel (vmlinux.gz) directly on the address where the loader expected to find it. Since this address was 0x16000 and the kernel was uncompressed to address 0x0, the uncompressing failed after a while (the uncompressed kernel was overwriting the compressed source).

We decided then to bypass the uncompressing function, loading the uncompressed kernel directly at address 0x0. The loader could always be rewritten later (which it was).

At this point, we reached the end of the loader, where it read a jump address from address 0x0 (the first word of the kernel) into the link register, and branched to this address.

At this point, we were completely out of the code in the boot-loader directory and had jumped to the file /arch/ppc/kernel/head.S. This (assembly language) file set up the exception vectors and memory management and then jumped to **start_kernel** in linux/init/main.c.

¹ From Diab Data- <http://www.ddi.com/>



Title
Linux 2.2 on Ericsson MPC860 platforms

Author
UAB/D/SF Leif Lindholm

4.5 GETTING PAST THE ARCHITECTURE SETUP CODE

Once past the loader, the main work of porting the actual kernel can begin. The source code for the first part of the kernel is located in the /arch/ppc/kernel directory. Most of the code here is generic (though sometimes `#ifdef:d`) PowerPC source.

At this point, two problems with the Lauterbach debugging environment were discovered:



- It uses `SRR0` for its stepping functions. The kernel uses `rfi` (which branches to the value of `SRR0`) to get into the `start_here` function. Because of this, any stepping after `SRR0` was set to the address of `start_here` would inevitably crash the kernel.
- It is incapable of using address translation, understanding only physical addresses. This lead to bus errors when trying to step through branches in the kernel code after translation was enabled.

Still, it was invaluable for allowing examination of the contents of memory anywhere and for the machine-code-listing window. The latter allowed looking at the `PC` and see just which instruction had caused an exception/a crash. The register view was just as important, and without any of these functions, it would have taken much longer to finish.

The first thing that happens in (this) `head.S` is that the caches are enabled, and that an 8Mb page of memory at physical address 0x0 is mapped to 0xc0000000, the base address of kernel memory.

After this basic memory setup, the kernel jumps to `start_here`, using a `rfi`. When the kernel was released, it got a machine check exception in `machine_restart` (which is used to reboot the system). This was found by comparing the `PC` against the entries `System.map` file, a technique used throughout the project. The `System.map` is a dump of all symbols in the kernel, together with addresses and types.

Somewhere around this period, a message on the ppc-dev mailing list warned that the standard gcc compiler could generate very faulty code. This made me pause in my porting to get an egcs cross up and running (they also mentioned this to be the preferred ppc compiler).

ERICSSON 	20(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

Since **machine_restart** was in a .c file, I thought I would take my first shot at using **printk**. It didn't work; I didn't think it would, but I had to try. Decided to find out just why **machine_restart** generated a checkstop. Found reason for this - it tried to read memory at a reserved address up in the internal memory map. Apparently, this would generate a reset if the **DER** was set correctly, but the Trace32 set this to generate a checkstop instead. Now the problem was why it ended up inside **machine_restart**.

4.6 INSIDE START_KERNEL

*Now the kernel ended up somewhere inside **start_kernel**, in **/init/main.c**. This is outside the architecture specific section of the kernel source tree. The **start_kernel** function calls many platform specific functions, but also a lot of generic ones – for memory management, network protocols, generic hardware and such.*

4.6.1 Getting the console to work

*There was still no output from the kernel, which complicated the porting quite a lot. Because of this, making the console work was now the most important problem to solve. The console driver consists of a generic and a architecture specific part. The architecture specific code to be manipulated was in the **/arch/ppc/8xx_io** directory.*

At this point, I was already inside the **start_kernel** function, for the first time outside the architecture specific tree. I still had no **printk**, and had to find my way by moving a while(1) around until the **PC** was where I expected. Using this method, I found that somewhere inside **init_IRQ**, the kernel jumped to **panic**. The reason for this was easily explained; the function tried to initialize an Intel i8259 external interrupt controller (which exists on the MBX, but not on the DCP1 or AU).

Since I now had a few variables I just had to know the value of, I decided to copy the **puts** and **puthex** from the loader into the kernel to get something out. This allowed me to inspect the variables **mem_start** and **mem_end**, which had completely absurd values. Soon after this, these functions stopped working - probably because the kernel made it all the way to **m8xx_cpm_reset** (Which resets the communication processor). .

I now found a reproducible bug I hadn't noticed previously, since I'd been told the prototype boards might be unstable. Whenever cold-booting the board, it immediately went into reset and displayed the PBOOT prompt again. I sat down and debugged the loader code, to find a typo in the assembler file (a **lis** had become a **li**), submitted a patch to Cort. I got a reply in the mail the next day thanking for the patch. This bug probably didn't matter when having hardware initiated by the EPPCBUG monitor, otherwise it would have been fixed long ago.

Since I had now noticed enough differences between Ericssons 860 boards and the MBX, I decided to start moving my files into a separate IPONAX directory and creating an IPONAX choice in the configuration files.

Now, whenever stopping the kernel, it was in **timer_interrupt** - which seemed like something relatively positive, but I had no idea as to how it got there.

Title

Linux 2.2 on Ericsson MPC860 platforms

Author

UAB/D/SF Leif Lindholm



Having gotten this far, I decided to put all my efforts in getting **printk** to work. An interesting feature here is that **printk** is called a few times before the console is initiated, buffering the output until then. This meant that it was possible that getting the console to work could help me with some problems that happened before this. Didn't get anywhere with this for a while, but found that the kernel was currently failing in **paging_init**. Since I knew the memory wasn't correctly set up, this was no big surprise.

Michael helped me to get the LEDs on the DCP1 working. I wrote the functions **init_inde**, **ind_on** and **ind_off**. With these, I now had three bits of debugging (seven steps) instead of the one with `while(1)`.

With this new debugging help, I traced the **paging_init** problem down to an attempted access of unavailable memory in **free_area_init** due to misconfigured memory availability.



After some debugging with the LEDs, I drew the conclusion that the error with **printk** was an effect of incorrect initialization of the SMC. The kernel died in **serial_console_write**. Continued working on this until Jonas got the idea to check on **log_buf**, the buffer used by **printk**. So I did, and found a reference to an incorrect initialization of the real-time clock. When I fixed this, I saw "Calibrating delay loop...47.51 BogoMIPS" (in the buffer).

We brought an oscilloscope to my room to check if there was anything at all coming out on the serial Tx pin. Jonas had mentioned that there was a possibility that tip wouldn't show anything if the communication parameters/speed were completely wrong. That was not the case - there was nothing coming out. Later that day I found that the **set_baudrate** function read the system clock frequency from the residual struct that was set up for all of the PREP-systems, but not on mine. A bit of backwards calculating by Jonas while comparing to the value that PBOOT used to initialize 9600 bps showed that the system clock was 36 Mhz. When I hard-coded this, I got my first output - we had a working **printk**. The kernel still panicked almost immediately, but now I had a working debug console.

4.6.2 Memory and device initialization

With the console working, it was now easy to print out extensive debugging information. This made it practically possible to start working out the rest of the hardware initialization: first the memory, then the devices – to finally mount the root filesystem and hopefully have a working system. The memory configuration functions were located in the /arch/ppc/mm/init.c file, and the device drivers in multiple files under /drivers/char and /drivers/block.

The next step was to get the memory correctly initialized/detected. Once I found the line where it set the amount of available memory and hard-coded that too, that seemed to work just fine. This had been another case of reading from the (nonexistent) residual struct. I now decided that I needed to implement some sort of buildup of residual data in the boot loader. This would greatly reduce the excessive amount of work that would otherwise be necessary to get any later version of the kernel to work against this board.

ERICSSON 	22(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

When I had the memory working, I dropped all the way down to the character device initialization. Here I had to comment out the call to **kbd_init** (which was called if you selected CONFIG_VT at kernel configuration, which was set by default for MBX). I also had to bypass **rs_init** which followed with CONFIG_SERIAL, since it insisted on finding three (nonexistent) 16450 uarts. There really should be some choice for "serial non-standard". This happened on a Friday, so we (Jonas and me) decided to work that Saturday too. After a couple of hour's work, we got it to try mounting the root filesystem (which failed since we didn't have any).

The following Monday, I mounted a ramdisk I downloaded, some sort of Red Hat installation image. It didn't work very well, but it didn't die either – stopping it revealed that it was often inside the **schedule** function.

I tried for a while to build together a working root filesystem in a ramdisk image. I even got so far as getting init to prompt for a runlevel, but not much farther. Not really getting anywhere with this, I decided to put my efforts on the networking part – since it would be much easier to experiment with a root filesystem if it was on the hard drive of some computer.

4.6.3 Network

With a now almost completely ported kernel, it was time to get the network up and running. The networking (Ethernet) code consists of two parts: the Ethernet part, and the interface part. The Ethernet driver is generic and did not have to be tampered with. The interface code was located in some files under /arch/ppc/8xx_io.

I had to make a few redefines of some pins and clocks, since the MBX was a bit differently constructed (here too). At first, the kernel got a 0x300 exception – which is generated for "other software errors" - when I included Ethernet support. I found that this was because the driver source code didn't include any board-specific headers other than <asm/fads.h>. I changed this to <asm/iponax.h> and now the kernel got all the way to "Sending RARP and BOOTP requests".

The next problem was the fact that this board didn't have any network connector - the only pins were CompactPCI. Jonas gave me two cables for this (a straight and a crossed one), and I borrowed a hub from the network lab. I also found me a SPARCstation 2 which I used for BOOTP server (and sniffing with tcpdump).

Unfortunately, the Ethernet didn't seem to work – tcpdump didn't show anything when the board said it was sending BOOTP, and the link LED in the hub didn't light up. Another interesting observation is that it seems to somewhat randomize the MAC-address, which isn't really a very big surprise - it's supposed to be read from the residual struct.

Jonas located a circuit just before the Ethernet connector that wasn't correctly mounted. He fixed this down in the lab, but it still didn't work. At this point, we brought even more hardware analyzing equipment up to my room to find out just what was going on with the network circuits. Studying drawings and measuring voltages, we found that clock 1, the receive clock, was used for a whole lot of other things (for instance the 4 DSPs). Jonas re-soldered and I re-programmed it to use clock 4 instead, which is actually what the MBX does.

Title

Linux 2.2 on Ericsson MPC860 platforms

Author

UAB/D/SF Leif Lindholm



Now the Tx LED (and the COL) blinked once as the first RARP (or BOOTP) packet was to be transmitted. Tcpdump showed no traffic though, and I wasn't convinced that the hardware was working yet. I was promised an AU, and decided not to continue trying to work this out until I had hardware with a verified Ethernet functionality. I got my AU, but didn't have a console cable yet.

To have something to do, I decided to download the embedded-2.2.5 kernel, which had been announced on the ppc-dev list a few days earlier. Since I saw that this one was much better organized (and modularized) I started a bit at porting this one. A depressing fact was that I had to correct some syntax errors (like missing semicolons) to get it to compile even with CONFIG_MBX. I didn't get very far with the new porting. I learned some about the new structure of the kernel and the internal functions of the Config.in and Makefiles.



I got my console cable and decided just to try to load the existing 2.2.1 kernel, which gave a lot of strange output - the frequency problem again. Changed the frequency to the correct 48MHz, and got as far as I had on the DCP1. One problem I noticed now was that I no longer had a link to the hub. On the DCP1, the connection from the CPU to the Ethernet transceiver chip had been hard-wired. On the AU, the configuration pins were connected to its port D. I had to browse through some board support package source code from Enea to find out which those pins were. This way, I easily found out how they were supposed to be configured. When I configured my port D the same way, the link LED on the hub lit up when the kernel configured the network interface.

4.7 (NOT) STARTING INIT

When a Linux system has mounted its root filesystem, it tries to start the executable /sbin/init which then brings the system up. If this fails, it tries with some other executables and unless it finds one, it panics. The next step was now to try to start init (or something else that worked) to have a working, running system.

After this, the system would send correct BOOTP packages (and receive replies), but it seemed to freeze a while later, when performing some multicast setup (in `setup_multicast_list`). After uncommenting the call to that function, the kernel was able to mount an NFS root filesystem (and panic since there was nothing on it). This had been performed on the DCP1, but the change between the kernels was limited to four places, and I soon had it working on the AU.

I downloaded the file mbxroot.full.tgz from ftp.cs.nmt.edu and unpacked it on the SPARCstation 2, which would act as a NFS server. When mounting this filesystem, the kernel freaked out after "Freeing unused kernel memory" saying "Page fault in interrupt handler" - which is rarely a good sign. This seemed to be happening in various UDP handling functions, which made me wonder if there wasn't a network problem. I sat down to repair whatever was wrong with the multicasting. Found a change I had made from an udelay to the users manual example of while(flag_set), despite a comment in the code by Cort saying that it was needed. This was probably a bug in the 860. Restored this to its original state, which made the function work, it still panicked a few packages into loading the init executable.

ERICSSON 	24(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

4.8 MOVING TO 2.2.5

There was obviously something wrong with the kernel. The 2.2.1 kernel was not even very stable on the standard x86 platform, and the source code for 2.2.5 was much better organized. This presented the possibility that a version upgrade would be enough to solve the problem, and would also simplify later version upgrades.

I got tired of the lack of success, and I now knew from using it at home on x86 that the network functions (and some others) in 2.2.1 weren't very reliable. Because of this, I decided to start working on the 2.2.5 instead. First, I started by getting it to compile on a standard MBX configuration, which didn't work at first. There were some typos and syntax errors that needed to be fixed before it would go all the way through. At least it didn't complain about CODA this time.

Then I started re-writing the loader to a much more specific implementation. I started reading parameters from the PBOOT, and build up the residual data structure that was used in so many places in the kernel. Since I saw a comment about this in the MBX code, I put the struct at the top of memory, with the command line just before it. After this, I replaced the serial put/get/check functions in the loader with those in the PBOOT. This created another problem: the PBOOT code resides very low in memory (starts at 0x2000) and is therefore overwritten early in the unzipping procedure. I first tried solving this by copying the PBOOT code to a high location and executing it from there. This failed because there were some absolute jumps in there. After this, I did the only (ugly TM) thing I could think of. I unzipped the kernel to a high location and then, (last in the **decompress_kernel** function, about 10 instructions from the jump to the real kernel) performed a **memcpy** on it down to 0x0.

Since I now had a working loader, I decided to load it into flash on the board. Unfortunately, PBOOT only has support for loading files in Motorola SREC format. This made things a bit harder for me since the entire compressed kernel is a plain data file (for which I had no converter). The executables (loader) were easily converted with ddump. I got the source code for hexsplit from Lars. Hexsplit is a program that takes SREC files and split them up into multiple parts for storing a program over multiple PROMs. I started re-working this (keeping the checksum calculation and linked list functions) to get it to do what I needed. This took about a day, but after that, there was no problem loading the kernel over the serial port of the board.

When running the kernel for the first time, it got in some trouble. It got a machine check when copying the residual struct into the kernel memory area. I wondered why, until I realized (while discussing this with Lars) that at that point I had only the first 8 of the 16 megabytes of memory available. I moved the struct down to just below the 8Mb, and tried again.

This time I got an unexplainable kernel stack overflow. I traced the execution (moving a while(1)), finding it happened at the call to **m8xx_cpm_reset**. This was completely unexplainable and when I did a "make dep ; make clean ; make zImage", to completely rebuild the kernel, the overflow disappeared. It had been a compilation error.

Title
Linux 2.2 on Ericsson MPC860 platforms

Author
UAB/D/SF Leif Lindholm



This time it was quite amusing to watch the output from **printk** - "Calibrating delay loop...392.12 BogoMIPS". It seems the PREP specifies the clock frequency to be specified in megahertz, while the Ericsson standard is hertz. Since it obviously wasn't a good idea to try to convince the board that it was running at 48THz, I implemented a /1024 in the loader residual data construction. What was interesting was that even after the change the BogoMIPS was slightly higher than it had been with 2.2.1. It had changed from 47.51 to 48.84, which wasn't much, but still interesting.

The next error was the same as the first time - **keyb_init**, which was only included if the kernel had CONFIG_VT. I had assumed that since this was included in the MBX default config, it was necessary for something. It wasn't; I removed CONFIG_VT and compiled, and it worked just fine. After this it started sending BOOTP requests (trying) and panicked when it couldn't mount a root. It still detected ttyS0/1/2 as uart16450 (in **rs_init** – although it didn't panic this time) and then found them correctly as SMC/SCC. This is based on something I consider a structural error in the Linux configuration, but I still just commented out the **rs_init**.



After this, it was merely the issue of adapting the Ethernet code (pins/clocks). When I had done this, it mounted a root filesystem and displayed a "# " (with init=/bin/ash).

4.9 RUNNING THE SYSTEM

Now, the system was running. It was time to see if it was working, and if it was possible to get it to do something constructive or if extensive debugging was needed.

At first, everything seemed wrong: "more" and "less" didn't work and other things were strange too. This was corrected by running the /fix shellsript, which remounts the root read/write and then mounts /proc. After that, things seemed more stable, but `more` and `less` still generated an unhandled software emulation exception.

Some investigating showed that both programs at some point tried to execute the **lfs** instruction, which wasn't implemented in the softemu code. I implemented it and sent a patch to the ppc-dev mailing list. The next day I received a reply from Cort Dougan saying he'd merge it in – which it is at least in 2.3.4. This is still my only contribution to the official code (since the MBX loader was completely rewritten in 2.2.5).

ERICSSON 	26(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

5 THE RESULT

5.1 CURRENT STATUS

The system will boot up, ask for an IP address via BOOTP, mount a NFS root filesystem and start up /bin/sh in stead of init. It needs to be run in debug mode (the switch on "1"), for the internal watchdog not to reboot the board after 2.7 seconds. This also means that it doesn't go into Linux automatically on power-on – it has to be started with a ">>run bootstrap".

It can also mount a ramdisk loaded into the PBOOT as program "initrd". Unfortunately, there is only 1 Mb of flash available, so when the loader and kernel is saved there isn't any room for a useful ramdisk image.

5.2 DEMONSTRATION

These basic things have been tested, using the mbxroot.full.tgz filesystem mounted over NFS from the Sun SPARCstation 2:

- Native compiling the first program:

```
# gcc -o hello hello.c
# ./hello
Hello WORLD!!!
#
```

- Checking the system uptime:

```
# uptime
11:31pm up 2 days, 18:34, 0 users, load average: 0.00, 0.00, 0.00
#
```

- Looking at the systems idea of its CPU:

```
# cat /proc/cpuinfo
processor       : 0
cpu           : 860
clock         : 49MHz
bus clock     : 49MHz
revision      : 0.0
bogomips      : 48.84
zero pages    : total 0 (0Kb) current: 0 (0Kb) hits: 0/12 (0%)
#
```



Title

Linux 2.2 on Ericsson MPC860 platforms

Author



UAB/D/SF Leif Lindholm

- Inspecting the systems view of its memory:

```
# cat /proc/meminfo
```

```
      total:      used:      free:    shared: buffers:    cached:
Mem:  15691776  3194880 12496896    856064     4096  1224704
Swap:           0           0           0
MemTotal:       15324 kB
MemFree:        12204 kB
MemShared:       836 kB
Buffers:         4 kB
Cached:         1196 kB
SwapTotal:       0 kB
SwapFree:        0 kB
#
```

I have also compiled, and run, Dieter httpd by Christian Johansson of SDF. This is a very small, very low-feature and not yet very stable server, but it manages enough to send an HTML file and a JPEG image over the network (or at least the first 64k of it :).

ERICSSON 	28(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

6 WHAT NEXT?

6.1 PLANNED FOLLOW-UP

If possible, to convert this into a fully functional system – being able to run completely standalone and booting directly into Linux on power-on. Then this system is to be benchmarked, looking into the performance of the scheduler, interrupt latency and such.

When this goal is reached, perhaps the IPONAX software will be ported – making it perform the same tasks as the original (which runs OSE from Enea), but this is in no way decided.

6.2 WHAT REMAINS TO BE DONE

- The mpc860 watchdog needs to be tickled regularly, preferably by an early started kernel thread. The loader uses PBOOT functions, which does this, so what is needed is for it to start less than 2.7 seconds after the loader has jumped to the kernel. This should not be a problem when not using BOOTP to resolve the systems IP address.
- Write support for running the root filesystem on the 3Mb of flash that's unused, with a /tmp (and possibly a /var) on a ramdisk.
- Create a root filesystem, no larger than 3Mb. This filesystem needs to put the board in multi-user mode at boot time. This should not be a problem, since the Linux Router Project can do it on about 1Mb.
- Implement all of the FPU instructions in the softemu code, for completeness and improved stability. Today, only five are implemented – of which one is incorrectly so.

6.3 KNOWN BUGS

When under heavy NFS load, the system can become highly unstable. It uses /bin/ash in stead of /sbin/init, which (by definition) is not healthy for a UNIX system. It should be running a real init binary.



6.4 TESTING

Since the system isn't finished yet, there has been no "real" testing - meaning that there hasn't been much effort put into making it crash. In addition, the system errors that have been encountered may very well have originated from the binaries in the root filesystem (which isn't tested at all). No serious testing will be meaningful until there is a verifiably correct root filesystem.

The system has been tested with crashme for a few minutes. Crashme is a program that generates random code and then executes it. This is an excellent way to test the stability of a system. There were many warnings, but it didn't crash. The recommendation however is for it to go for at least a few hours and preferably a few days.



Term	Explanation
CHRP	Common Hardware Reference Platform. An open computer platform agreed upon by Apple, IBM and Motorola. A superset of PReP.
CMU	Carnegie Mellon University http://www.cmu.edu/
Coda	An advanced distributed network filesystem, allowing among other things disconnected operation. http://www.coda.cs.cmu.edu/
DCP1	An Ericsson-constructed mpc860 based board, which was used for the porting since there was a shortage of IPONAX AUs. A more detailed description is available in the "Hardware" section.
egcs	Used to be a spin-off to GCC, but since April 1999 it is in charge of the entire GCC development project. http://egcs.cygnus.com/
ELF	Executable and Linkable Format: A binary executable format, the current standard format in Linux. Replaced the old a.out format.
gcc	Gnu C Compiler – recently renamed to Gnu Compiler Collection, when integrated with the EGCS project. http://www.gnu.org/software/gcc/gcc.html
GNU	Gnu's Not Unix. A project started by Richard Stallman to create a complete Unix-compatible software system released under GPL. http://www.gnu.org/gnu/thegnuproject.html
GPL	Gnu Public License. A license under which all of the GNU and a lot of other software, including the Linux kernel, is distributed. http://www.gnu.org/copyleft/gpl.html
IPONAX AU	An Ericsson-constructed mpc860 based board, part of the IP telephony project IPONAX. The original target for the porting. A more detailed description is available in the "Hardware" section.
PReP	PowerPC Reference Platform. A system standard designed by IBM to insure compatibility among PowerPC-based systems from different manufacturers/developers.
RMS	Short version of "Richard M. Stallman"
SMP	Symmetric MultiProcessing – A standard for multiple CPU systems.
SREC	S record. Text file format for saving executables. This is the only format useable for loading programs into the PBOOT.

ERICSSON 	30(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

8 REFERENCES

The following literature was of vital value either for the report or for the porting project itself.

8.1 ELECTRONICALLY AVAILABLE

#	Title	Information
1	Linux FAQ	Kiesling, Robert Available at http://metalab.unc.edu/LDP/FAQ/Linux-FAQ.html
2	Linux kernel mailing list FAQ	Available at: http://www.tux.org/lkml/
3	The Linux Kernel	Rusling, David A Available at : http://metalab.unc.edu/LDP/LDP/tlk/tlk.html
4	Linux kernel source code ;-)	Latest version pointed out at: http://www.linuxhq.com/

8.2 PRINTED

#	Title	Information
5	Inside TCP/IP	Siyan, S. Karanjit Third edition. New Riders Publishing, 1997 ISBN: 1-56205-714-6
6	Linux Device Drivers	Rubini, Alessandro O'Reilly & Associates, 1998 ISBN: 1-56592-292-1
7	MPC860 PowerQUICC Users Manual	Motorola Inc, 1996 MPC860UM/AD
8	PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors	Motorola Inc, 1997 MPCFPE32B/AD



9 THE HARDWARE

This is a description of the hardware used during the project, the debugging hardware and the workstations/servers used.

9.1 LAUTERBACH



The equipment from Lauterbach Datentechnik used was the ICD (in circuit debugger). It was connected via the POBBUS->parallel interface to a Toshiba Libretto at first, and later via an ECU32 in the 32-bit system controller to Ethernet (to be used from anywhere within the local network).

9.2 WORKSTATIONS

All of the work, with exception for the head start, was performed on a Sun SPARCstation 5 (Solaris 2.5.1). Since this is not a very fast computer, a Sun Ultra 10 (Solaris 2.6) was used for compiling the kernel.

Later on, a Sun SPARCstation 2 (Red Hat Linux 5.2) was used for a BOOTP and NFS server.

The first two (unofficial) days a Sun SPARCstation 5 (Solaris 2.6), a Dell OptiPlex Gxi (Red Hat Linux 5.2) and a Toshiba Libretto (windows) were used.

ERICSSON 	32(33)	
Title Linux 2.2 on Ericsson MPC860 platforms		
Author UAB/D/SF Leif Lindholm		

10 THE SOFTWARE

This section describes the software used in the project – the development/debugging software as well as the operating systems used on the computers.

10.1 OPERATING SYSTEMS

- The SPARCstation 5 runs Solaris 2.5.1
- The Ultra 10 runs Solaris 2.6
- The SPARCstation 2 runs Red Hat Linux 5.2
- The Dell OptiPlex runs Red Hat Linux 5.2
- The Toshiba Libretto runs Windows 95

10.2 PBOOT

PBOOT is the monitor program that sets up the hardware, manages the flash memory and launches the loader. The loader also uses some PBOOT functions, to read system parameters and write characters to the serial port.

10.3 COMPILING ENVIRONMENT

At first, the regular gcc 2.8.1 was used - a cross compiler version from sparc-solaris to powerpc-linux. After a week, it was replaced by egcs 1.1.1 after warnings about regular gcc generating faulty code for PowerPC. No such problems were encountered, but it's best to play safe.

Gnu binutils, make and bourne again shell (bash) were also used for compiling.

10.4 THE NFS-ROOT FILESYSTEM

Cort Dougans mbxroot.full.tgz (slightly stripped) was used for a root filesystem. It was downloaded from ftp.cs.nmt.edu. This filesystem contains compilers, text-editors and all the standard GNU utilities.

Title

Linux 2.2 on Ericsson MPC860 platforms

Author

UAB/D/SF Leif Lindholm



10.5 OTHER SOFTWARE

- **ddump** from Diab Data was used for examination of miscellaneous executable files and for converting ELF binaries into SREC format, the format required for loading programs into PBOOT
- **bin2srec**, which I wrote myself (using parts of the source code from Lars Jönssons hexsplit) to convert data files to SREC for loading them into PBOOT.