



Studienarbeit

Quantitative analysis of system initialization in embedded Linux systems

André Puschmann

`andre.puschmann@stud.tu-ilmenau.de`

January 26, 2009

Supervisor (TU-Ilmenau):	Dr.-Ing. Bernd Däne
Supervisor (IMMS):	Dipl.-Inf. Rolf Peukert
Field of study:	computer science
Registration number:	36790

Contents

1	Introduction	3
2	Hardware and Software Platform	4
2.1	Gumstix verdex XM4	4
2.2	Software platform	5
3	The boot process	6
3.1	Bootloader phase	6
3.2	Kernel phase	6
3.3	Application phase	7
4	Instrumentation	8
4.1	Measuring	8
4.2	Reference system configuration	9
5	Boot time reduction	11
5.1	Universal techniques	11
5.1.1	File and code size	11
5.1.2	Compression	11
5.1.3	Memory configuration	12
5.1.4	Execution in Place (XIP)	14
5.2	Bootloader layer techniques	15
5.3	Kernel layer techniques	15
5.3.1	Kernel image type	15
5.3.2	Console output	16
5.3.3	The loops_per_jiffy (lpj) value	17
5.3.4	Device drivers	17
5.4	Application layer techniques	19
5.4.1	Filesystem	19
5.4.2	Parallel init script execution	21
5.4.3	Device node population	21
5.4.4	System libraries	22
6	Guidelines to speed up the boot process	24
7	Conclusion and outlook	25
	References	26
	List of Symbols	28
A	Root filesystem listing	29
B	DVD content listing	31

1 Introduction

Since the first release of the Linux kernel for AT-386-compatible personal computers in October 1991 [1], the system has been ported onto many platforms. In the last couple of years, especially embedded consumer electronic devices turned out to be a perfect target for Linux. Unfortunately, standard Linux systems used on desktop machines typically show a very poor boot performance. This is not acceptable for most of the embedded systems. The time a system needs from power-on to a "usable" system is a critical user satisfaction component [2]. Consider a car audio system that needs five minutes to power up. Would you buy it?

The main purpose of this work is to identify the most critical parts during the boot process of embedded Linux systems. Furthermore, we would like to illustrate some solutions and techniques to reduce the boot time. Almost all concepts can be reproduced on similar systems. In this work, the experiments gave insight to understanding the ideas on a XScale PXA270 board. In order to evaluate the advantages and possible savings of a new technique, a oscilloscope was used to measure all timings. Moreover, we investigated the advantages of execution in place, also known as XIP. As a result of this paper we would like to generate a set of general rules which can be used by embedded Linux system engineers, in order to review and improve the boot time performance of their systems.

2 Hardware and Software Platform

This section describes the hardware and software background used for this project. While the first part introduces the hardware, part two is focused on the software platform. Figure 1 illustrates the hardware setup. It shows the *Gumstix verdex XM* board, the power supply and the oscilloscope (Tektronix TDS 3034B) with probes connected to the mainboard.

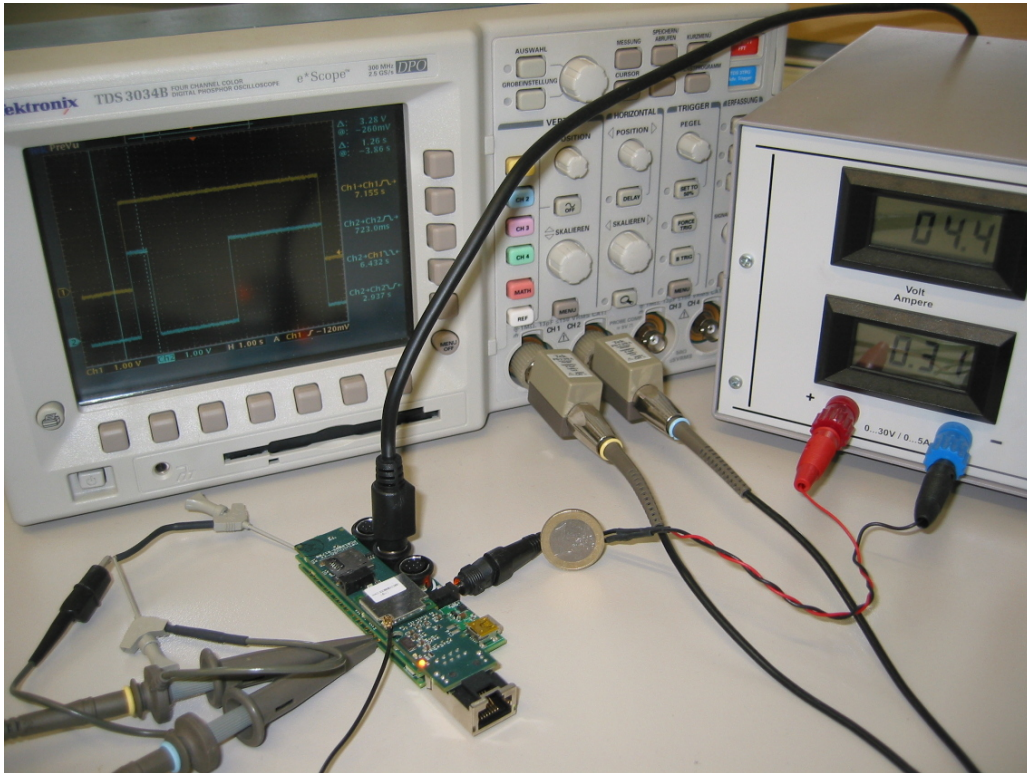


Figure 1: Hardware setup

2.1 Gumstix verdex XM4

As a hardware platform for this research project, the Gumstix verdex XM4 mainboard has been chosen. The small and very flexible layout makes it an ideal object for this project. The main processing core is a powerful XScale PXA270 clocked at 400MHz [3] which complies with the ARM Architecture V5TE. The system-on-chip (SoC) microprocessor has a variety of typical embedded system, like a real-time clock (RTC), general purpose timers (GPT), several UART (universal asynchronous receiver/transmitter), three SSP ports (Synchronous Serial Protocol, also known as SPI), a I²C bus interface, USB, a DMA controller and even a LCD display controller, already included.

In addition to these on-chip components, the verdex XM board features 64MB of RAM and 16MB external Intel NOR flash [4] which is the main storage device. The single mainboard only

has the most important components mounted on it. Additional features, like a JTAG interface, LCD display, GPS receiver or Ethernet, can be added with the help of expansion boards.

2.2 Software platform

The software platform for this project was built with the help of the *Gumstix buildroot* environment [5], which itself is based on *buildroot* [6]. This custom board support package (BSP) for the *Gumstix* hardware includes everything needed to work with a Linux based computer system. The package contains a GCC based cross-compiler, the bootloader *u-boot*, the Linux kernel itself (version 2.6.21) and a lot of additional libraries and useful tools like *busybox*.

Unfortunately, *Gumstix Inc.* maintains their own software repository. Many changes made for this board have not been pushed back into the main project repository. Mainly, this applies to the Linux kernel and the *u-boot* bootloader. Thus, some additional effort has been spend in order to include the *Gumstix verdex* board into the mainline Linux kernel. Recently, *Gumstix* switched over to support *OpenEmbedded* as their main environment.

3 The boot process

The word boot or booting is an abbreviation of bootstrapping which describes a process in the field of computer systems where a small system activates a more complicated system [7]. We define a systems boot time as the time from power-on to a "usable" system. In order to compare different techniques and solutions it is necessary to clarify the state from which a system is considered to be usable (please refer to section 4.2) . This can be a very complex task as it is a very application and vendor specific part. Moreover, the varying interpretation of this term makes a quantitative analysis very difficult. Basically, the process of booting a computer can be divided into three phases which find a closer description in this section. Most operations can be clearly assigned to one of these stages. On the other hand, there are parts, like decompressing the kernel, that can be accomplished by either the bootloader or by the Linux kernel itself.

3.1 Bootloader phase

The bootloader is the first program that is executed after a systems power reset. After a short clock calibration and initialization time the CPU core fetches the first instructions from a hard-wired base address. In our case, the main processor loads the first instructions (from external flash) at physical address zero. This is feasible as NOR flash allows direct addressing and code execution.

The main task of bootloaders in production systems is to load some operating system. But they can also be very helpful during development, though. Moreover, performing a low-level hardware initialization is also a very fundamental task for bootloaders. Typical operations a bootloader needs to perform prior loading any other system are:

- configuring the processor pins (GPIO)
- initializing the core clock system
- memory and memory timing setup
- resetting hardware components and on-chip devices.

Therefore, a bootloader in the world of embedded systems can be compared with a BIOS in desktop environments. After passing the responsibility to the next layer, the kernel, the bootloader typically never runs again.

3.2 Kernel phase

The kernel is the most intrinsic part in operating systems (OS). A monolithic OS kernel is responsible for managing all system resources. This includes the CPU, memory and all devices. All interaction between a user application and one of these resources passes the managing unit: the kernel. Figure 2 illustrates this architecture.

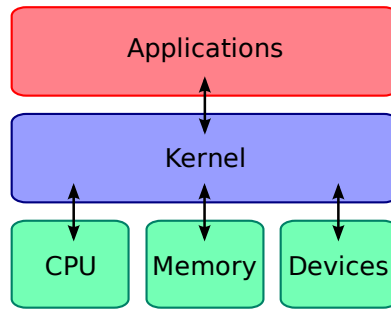


Figure 2: Monolithic operating system architecture [8]

During the boot process the kernel initializes all required hardware and software components. This includes all devices and device drivers, interrupt handling, system timers, filesystems and also network communication stacks (like TCP/IP). As a next step, the kernel mounts the root filesystem. After that, it invokes the first userspace process: the *init* process. The latter two actions are considered to be part of the application phase in this work. Thus, they are discussed in the following paragraph.

3.3 Application phase

The third stage of the entire boot process is called application or userspace phase. It is the longest part and therefore has a great impact measured relatively to the total boot time. Although most of this phase is very application specific, it also has some static components that can be found in many embedded Linux systems.

Hereby, the filesystem has a great influence on the overall system performance. Depending on its architecture and design goals it might support features like compression or execution in place. Critical system parameters like speed or space limitations make the choice of the right filesystem very application specific. However, many system engineers chose compressed read-only filesystems as their primary root filesystem. On the other hand, they use writable partitions of their storage medium to store volatile data. A general discussion about compression can be found in part 5.1.2. The first process invoked by the kernel after mounting the root filesystem is *init*. Usually, *init* successively runs a set of scripts (*init scripts*) that handle the userspace initialization. This includes mounting other filesystems like */sys* or */proc*, populating the device tree (*/dev*), initializing a network connection and starting log daemons like *klogd* and *syslogd*. Typically, right after doing this basic system environment setup the application process is spawned. Depending on its complexity this might take a while. Loading symbols from shared libraries or establishing a network connection to an application server might add an extra overhead which should not be underestimated.

4 Instrumentation

4.1 Measuring

In order to evaluate potential savings of a certain technique it is necessary to have exact knowledge about time. In-system timing information, like for instance the `printk` system that the Linux kernel provides, are not always available. Measuring time spans in low-level operation (like during bootloader startup) can be achieved by extending the software component (i.e. bootloader) to make use of the SoC build-in functions (like general purpose timers). The disadvantage of this solution is that it adds an additional overhead at many places inside the source code. Furthermore, the code becomes more complex and less maintainable, the user need a deep understanding of the system to avoid general mistakes and misinterpretations of the generated results. Due to these circumstances, we chose to do the measuring off the systems. This can be achieved with the help of one or more of the GPIO-pins (general purpose input/output) of the CPU. A logic analyzer or a storage oscilloscope can than be used to measure time spans between two or more logical values. Software based test points that toggle a GPIO pin can be added nearly everywhere in source code. Thus, this technique provides a flexible and very accurate method to measure time. Of course, the disadvantages are the increased hardware requirements.

To measure the time the systems spends in a particular initialization phase (refer section 3) we established measuring points before and after each phase. A further measuring point has been added in the kernel phase, which allows to distinguish between kernel copy/decompression time and kernel execution time. The application phase was also divided to allow to measure the filesystem mount time.

Figure 3 displays the time flow of a sample boot process. The chart below gives further information on the measuring points.

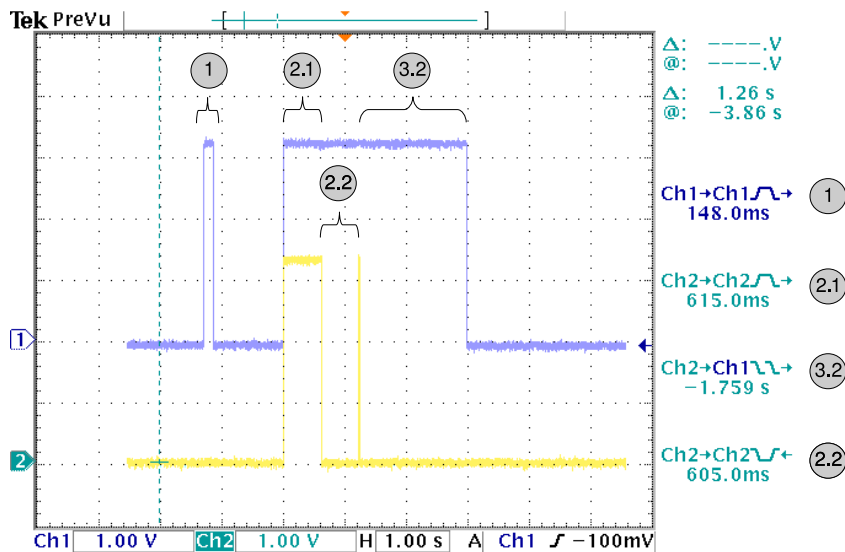


Figure 3: Sample measurement with the oscilloscope

1. Bootloader phase (u-boot)

start: after GPIO initialization in `memsetup.S`
stop: before entering `main_loop()` in `board.c`

2. Kernel phase

2.1 Kernel decompression/copy phase

start: via *u-boot* script before executing the **bootm** command
stop: before scheduler init in function `start_kernel()` in `init/main.c`

2.2 Kernel execution phase

start: right after end of kernel decompression/copy phase
stop: before attempting to mount root filesystem in `init()` in `init/main.c`

3. Application phase

3.1 Mounting root filesystem

start: right after end of kernel execution phase
stop: before executing the *init* process in `init_post()`

3.2 Initializing userspace

start: right after mounting the root filesystem
stop: script invoked by *init*

All measurements have been done three times. The mean value has been calculated over all measured values. The calculated deviation from the mean value of this magnitude can be neglected and is, thus, not presented in this work.

4.2 Reference system configuration

As a basis to compare different system configurations as well as the impact of modifications it is necessary to create a system model which describes a real world scenario as realistic as possible. Of course, different products and customers all have their own requirements and limitations. For the experiments, we developed a general purpose system configuration for the *Gumstix* platform, which includes a bootloader and a Linux kernel as well as a root filesystem. The latter features the most important system libraries like *libc* and *libstdc++*, the C/C++ runtime libraries. Furthermore, it includes a set of configuration files and initialization scripts and *busybox*: "The Swiss Army Knife of Embedded Linux". Please note, that this reference system is not intended to make use of all of the platform features (like bluetooth or wireless LAN). In fact, we tried to model a lightweight system which just provides the most common features. For a realistic filesystem comparison, which is described in section 5.4.1, we extended the system configuration to include a further *init* script. This script starts an application which draws three rectangles on

the framebuffer device. After that, the init script transfers 512 *KiB* of random data from flash to RAM. These steps should simulate the load which typical applications produce while being loaded or while accessing data from flash. A overview of the root filesystem can be found in appendix A. The complete filesystem listing as well as the kernel configuration can be found on the enclosed DVD.

5 Boot time reduction

An ideal startup time of a given system can be achieved if both, IO and CPU workload, are 100 percent during the whole process of booting [9].

The goal of this chapter is to present several techniques and concepts that aim to reduce the boot time. Furthermore, we evaluate the proposals and discuss the potential savings. This part is divided into four subsections. Whereas the first one describes universal techniques and general concepts that can't be assigned to a certain phase in the boot process, the latter three sections in this chapter will refer to each individual boot phase in detail.

5.1 Universal techniques

5.1.1 File and code size

In contrast to common general purpose computing systems like desktop workstations, embedded systems are often designed for a certain dedicated use-case. Many devices are produced in large quantities which further on justifies the effort to create highly optimized device. This opens opportunities to build them cheaper, if for instance less flash memory is needed due to the use of a compressed file system. File and code size are fundamental factors which very much influence the hard- and software system design. On the other hand, many vendors also use third-party mainboard modules, like the *Gumstix* products, as a basis in order to build their own products on top of them. These boards often come with a BSP (board support package) which, by default, enables all available features.

A general advice to system engineers is to review the configuration of all standard system components. Potentially, the largest savings during the boot process can be achieved with a thin Linux kernel configuration, an optimized system libraries and root filesystem (including init scripts). However, even the necessity of standard components is sometimes dubious. Many systems don't run C++ applications and therefore don't require a C++ runtime library. Once again, this is a very application specific process which needs to be done for every new system setup.

5.1.2 Compression

Compression, in general, is a process aiming to reduce the space needed to store a certain amount of information. In this section, the impact of compression in relation to boot time is discussed. Compressing data will generally cause less space to be needed. At a first glance, that implies shorter load time as less data is being transmitted. This is not necessarily the case as compressed data needs to be decompressed before it is being used. This process requires a huge computational effort and is usually done in software on the CPU. Therefore, the decompression speed very much depends on the algorithm, its implementation and the CPU it runs on. In order to answer the question whether compression is useful, the user has to take read- and decompression speed into

account [10]. The following two equations are considered to be a very basic approximation of the problem.

$$t_{compression} = \frac{filesize}{bandwidth_{read}} + \frac{filesize}{bandwidth_{decompression}} \quad (1)$$

$$t_{nocompression} = \frac{filesize}{bandwidth_{read}} \quad (2)$$

Consider this example: loading a kernel image from flash to RAM. The following values are given: the uncompressed image size is 5 MiB , the compressed image is 3 MiB in size, read bandwidth is 20 MiB/s and decompression bandwidth is 30 MiB/s . If compression is not used, loading the image from flash to RAM depends on read speed only. Thus, the process takes $\frac{5\text{ MiB}}{20\text{ MiB/s}} = 0.25\text{ s}$. On the other hand, if compression is enabled, loading and decompressing the (compressed) image from flash to RAM also takes $\frac{3\text{ MiB}}{20\text{ MiB/s}} + \frac{3\text{ MiB}}{30\text{ MiB/s}} = 0.15\text{ s} + 0.1\text{ s} = 0.25\text{ s}$. This example shows that compression does not necessarily has to be faster. If the algorithm achieves a good compression factor and a huge decompression speed enabling it might be a good idea. Unfortunately, the proposed calculation is not always as simple as this. A addition of both times like shown in equation 1 is not practical, if for instance DMA (direct memory access) is used for reading the image and the decompression is done block-wise on the fly without prior copying the whole data.

5.1.3 Memory configuration

In order to access internal or external memory like RAM or flash, the SoC's memory controller needs to be programmed. The configuration routine covers address mappings, cache policies, memory access modes (like synchronous or asynchronous mode, page- or burst read mode), clock settings and memory timing settings (like access delays) as well as programmable power-down modes for saving power. In order to achieve low system startup times and an overall good throughput it becomes even more important to fully use the capabilities of the memory chips. This paragraph only describes the configuration of the external flash chip as it is supposed to be the most influencing bottleneck during the boot process (because RAM is at least several orders of magnitude faster). The flash mounted on the *Gumstix verdex XM* board is used in asynchronous page mode (which is also the default mode) in our configuration. That means no additional clock signal (CLK) is needed. Using the flash in synchronous burst-mode, however, promises even higher transfer rates. Unfortunately, this was not tested on the Gumstix platform as trying to configure the flash in synchronous mode always failed. Maybe this happened due to a missing CLK signal wired to the flash chip.

In order to evaluate the flash performance through the mtd (memory technology devices) abstraction layer in Linux, we used a tool called *flanattoo* (A Flash Analysis Tool) written by *Daniel Parthey* at *Chemnitz University of Technology* during his diploma thesis [11]. Based on version 1.0 we extended *flanattoo* in order to support the ARM architecture. In condition as

supplied to the customer, the flash read performance was around 1.45 MB/s as you can see in figure 4. Another observation is that the read bandwidth is relatively static at lower block sizes. Starting at 32 KiB block size the throughput slightly falls down to 1.32 MB/s .

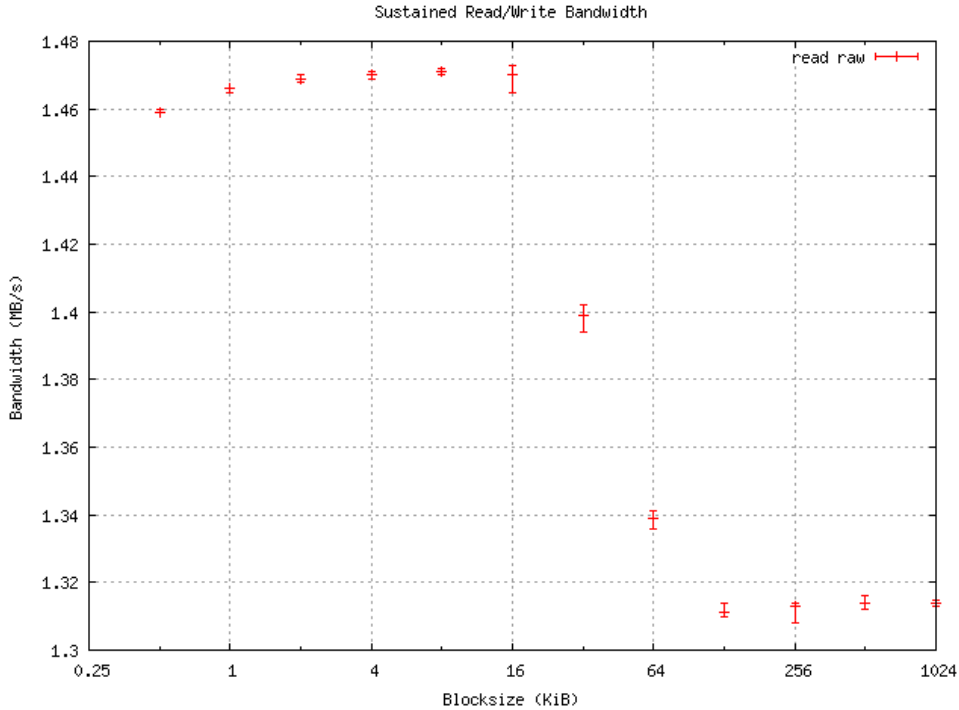


Figure 4: unoptimized flash read bandwidth

First, we investigated the flash timing configuration which is set during the early initialization phase in u-boot. We found out that the default values in static memory control register for chip select 0 (MSC0) have been assigned quite conservative and burst-mode was even disabled. Reducing the flash recovery time (which is the number of memory clock cycles from the time that chipselect is deasserted after a read or write until the next chip select is asserted), the first and next access delay and enabling burst mode achieved a slightly better throughput of around 1.9 MB/s (refer [3] page 309ff for a detailed register description). The slightly throughput degradation (at 32 KiB block size) can also be observed. The second step was to review the cache configuration for the flash memory. Caches have a great performance impact as they reduce the number of data accesses to and from external memory (data caches) or the number of instruction fetches from external memory (instruction caches). We found out, that the Linux kernel (the flash driver in particular), by default, bypassed the data cache while accessing the flash chip. This circumstance reduced the achievable bandwidth dramatically. As illustrated in figure 5, enabled caches yield transfer rates up to 19 MB/s .

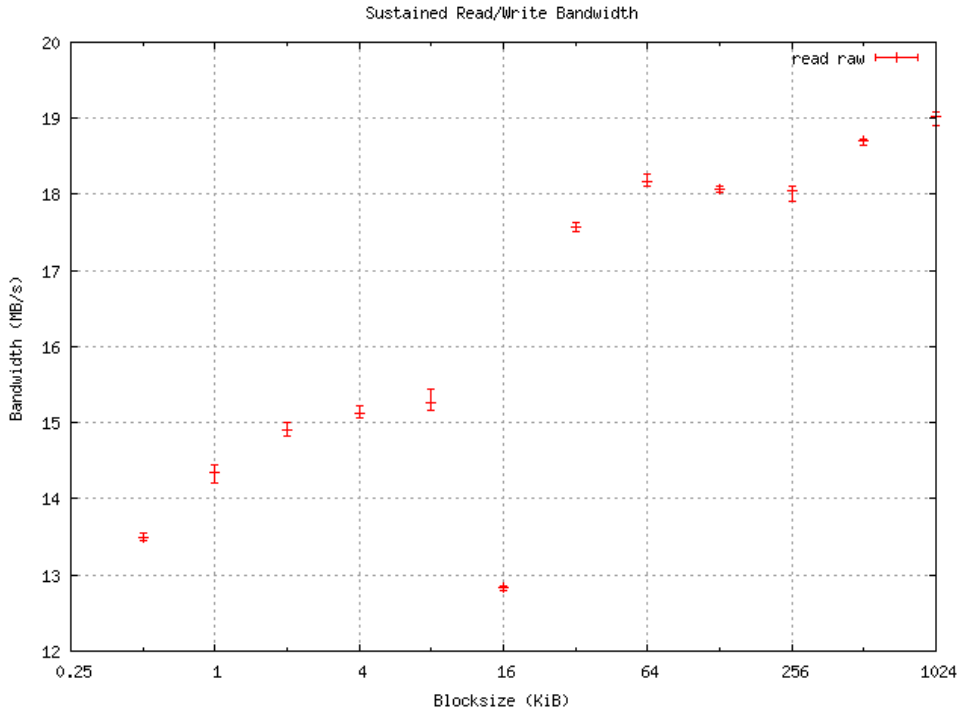


Figure 5: optimized flash read bandwidth

One can observe, starting at 32 KiB block-size, the overall read bandwidth, with enabled caches, slightly improves compared to lower block sizes. The fact, that the data caches of the XScale core are indeed 32 KiB in size, explains this further boost.

All together, these steps improved the linear flash read performance through the mtd subsystem from 1.45 MiB/s to 19 MiB/s . However, in contrast to the memory configuration, which remains the same once it was configured, memory needs to be marked as cachable explicitly, if a new process attempts to map the region.

5.1.4 Execution in Place (XIP)

Execution in place, also known as XIP, is a technique where code is being directly executed from a non-volatile storage media, like flash or ROM. In typical desktop Linux systems, the compressed kernel is stored on a hard-drive. A bootloader, like *lilo* or *grub*, loads this kernel into RAM from where it gets executed. Obviously, this process takes some time. The main idea behind XIP is to omit this extra overhead. XIP is not only related to embedded Linux but it became very popular in this field since most embedded systems use it (and even rely on it) anyway. Flash is often much larger than RAM. Therefore, holding the whole operating system in RAM is impossible. Execution in place requires linear read access and naturally implies uncompressed images. Generally, XIP is usable in conjunction with different memory types, such as NOR flash, RAM or ROM. One can distinguish between three different forms: bootloader XIP, kernel XIP and application XIP.

The main advantages of XIP are less RAM and power consumption. The latter can be a system critical parameter, for instance in a battery driven environment. On the other hand, XIP requires slightly more flash space, as images can't be compressed. Furthermore, NOR flash is, by default, at least one order of magnitude slower than RAM. This might lead to a lower overall throughput when using XIP, although starting applications is typically a bit faster. The following will summarize the advantages and disadvantages of execution in place [12]:

advantages

- less power and RAM consumption
- lower time-to-start

disadvantages

- needs expensive NOR flash
- requires more flash space
- possible lower overall throughput

Section 5.3.1 investigates the influence of kernel XIP, whereas section 5.4.1 studies the impact of XIP in filesystems.

5.2 Bootloader layer techniques

As the bootloader is responsible for the initial hard- and software configuration, this section is considered to be very important. On the other hand, compared with the complete boot process, it is only of short duration and therefore has only little impact. In the default configuration, loading *u-boot* took roughly 150 *ms*. With the optimized memory configuration we were able to reduce the load time to 135 *ms*. Furthermore, we tried to replace *u-boot's* standard byte-by-byte `memcpy()` implementation with the faster Linux version. This step only brought a marginal improvement, especially when copying only small pieces of data.

In general, optimizing this layer is a complex process and promises only marginal improvements. According to the *Pareto principle* (80-20 rule), it is not to be recommended to spend too much time on bootloader optimization, unless there is good reason.

5.3 Kernel layer techniques

5.3.1 Kernel image type

The main goal of this part of the paper is to discuss the impact of kernel XIP. Therefore, three different kernel image types have been compared. The first kernel is a non-XIP kernel which uses its build-in decompression algorithm. The second kernel was configured for XIP. As

the uncompressed kernel is stored in flash there is no need to copy the kernel text segment to RAM. Finally, kernel image three was build with kernel configuration flag `CONFIG_ZBOOT_ROM`. This means, the decompressor, which transfers the uncompressed image to RAM, is executed directly from flash. Of course, all kernels have been build with the same configuration base (`kernel_config_1` in appendix B). The results are presented in table 1.

	normal (non-XIP)	XIP kernel	ZBOOT_ROM kernel
size	885.76 <i>kB</i>	1,826.63 <i>kB</i>	885.76 <i>kB</i>
decompress/copy	730 <i>ms</i>	65 <i>ms</i>	616 <i>ms</i>
kernel init	275 <i>ms</i>	535 <i>ms</i>	275 <i>ms</i>
mount/start rootfs	679 <i>ms</i>	1,156 <i>ms</i>	679 <i>ms</i>
total	1,684 <i>ms</i>	1,756 <i>ms</i>	1,570 <i>ms</i>

Table 1: Kernel image type comparison

As the results show, in our configuration, a XIP kernel boots faster than non-XIP kernels do. Because of the omitted time-wasting decompression phase, the kernel starts executing right after asking to do so. However, as executing code directly from flash is much slower than from RAM, it might still be a good idea to accept the additional overhead. As one can see, the kernel init phase is nearly two times lower in XIP kernels than in non-XIP kernels. And even worse, mounting the root filesystem and executing the init scripts is also almost half a second slower, which results in an overall lower boot time. Based on these observations, we can not recommend using kernel XIP if only the overall system boot time is relevant. Furthermore, we also expect a lower overall system throughput when using kernel XIP. Later on, when the system is running, the user pays for a slightly faster kernel execution. However, if only kernel boot time is the most critical influencing factor, using the XIP configuration might still be an option. Note that *Tim Bird* ([13], slide 20ff) argues that kernel XIP doesn't have a huge negative impact on kernel execution time. Furthermore, he observed only a marginal throughput degradation measured with *lmbench*. His observations have been made on a OMAP board (ARM9, 168MHz). Unfortunately, he gives no details about his flash configuration and performance, which is supposed to be the potential reason for his good results.

5.3.2 Console output

During the process of developing embedded systems, the serial console is one of the most important debugging facilities. If a JTAG (joint test action group) debugging port is not available, sometimes it is even the only way to interact with the target device. Using the console output is therefore strongly recommended. It not only gives the user a good feedback about what is currently going on, it might also saves a lot of time while fixing bugs. On embedded systems in production use, however, avoiding the console output eliminates unnecessary latencies. If for instance the serial port is used for console output, every invocation of `printk()` also implies calling the UART output function which is a waste of time in productive environments. The

easiest way to suppress the kernel console output, while booting, is to append the **quiet** parameter to the kernel command line. After the system finished booting, the kernel output can still be examined with **dmesg**. To measure the performance improvement we booted the same kernel with and without the **quiet** command line parameter. Booting the latter configuration took 1.665 s, whereas booting the same kernel without console output only took 1.354 s. In our case that means a reduction of fairly 300 ms on the *Gumstix verdex XM* board. On desktop computers (or even on some embedded devices), however, the standard console is typically a VGA or framebuffer console. Nevertheless, the same rules apply here, too. An unaccelerated framebuffer implementation can slow down things even more. Therefore, we strongly recommend to disable console output in productive environments, if boot time matters.

5.3.3 The `loops_per_jiffy (lpj)` value

On many hardware platforms the implementation of kernel's `udelay()` is a software loop based on incrementing a counter value that has been calculated at boot time. The cost of calculating this value is not related to the CPU frequency and only depends on the `CONFIG_HZ` value, which is defined to 100 in our configuration for the PXA architecture. As defined in `init/calibrate.c` of the Linux source code, the calculation should take around 0.12 s for this system, presuming a duration of $1.5/\text{CONFIG_HZ}$ seconds for every bit at a precision of eight bits. As described, these costs can be avoided once the `lpj` value has been calculated for a certain system.

First, we have to determine the correct value for a current configuration. This can be achieved by appending **loglevel=8** to your current kernel command line. In our case, the kernel output looks like this:

```
Calibrating delay loop... 415.33 BogoMIPS (lpj=2076672)
```

The calculated value itself might be added to the kernel command line to omit the calculation overhead at every subsequent boot. At our system, appending **lpj=2076672** as an additional parameter yields the following output:

```
Calibrating delay loop (skipped)... 415.33 BogoMIPS preset
```

The kernel initialization of our PXA270-based custom Linux system took 475 ms including the `loops_per_jiffy` calculation. Booting the same kernel with the fixed `lpj` value only took 275 ms. As a result, we suggest avoiding the calculation of the `loops_per_jiffy` value in production systems. In our configuration, this saves about 200 ms at every single boot.

5.3.4 Device drivers

"Device drivers take on a special role in the Linux kernel. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. [...] Drivers can be built separately

from the rest of the kernel and "plugged in" at runtime when needed." ([14], page 1).

If load time matters, however, compiling build-in device drivers is generally faster as additional overhead, such as linking the drivers object code at runtime, is being avoided. This is not always true. During the startup phase, the Linux kernel initializes every subsystem and invokes every device driver init function successively. Therefore, every single load time influences the total load time calculation linearly. Especially, if device drivers perform time exhausting actions, such as hardware probing or firmware loading, this often adds additional underestimated overhead. Building modules instead of build-in drivers in such cases is therefore the better choice, as deferred and concurrent driver initialization in userspace context might boost things up. The benefit, however, is driver specific. In our case, for instance, the network device driver interrupted the system startup for fairly two and a half seconds. This is mainly due to the auto negotiation that is being performed in order to detect the correct link configuration. Of course, this behavior can also be switched off but this step would also reduce the system comfort. A further way to bypass this circumstance is to load the network driver subsequently during the application layer init phase which allows parallel execution. But note that in that case the user can't presume a working e.g. Ethernet connection after kernel startup. Applications that rely on network access can be synchronized with the network device driver load script in such circumstances, though.

5.4 Application layer techniques

5.4.1 Filesystem

Many embedded systems use flash memory as their storage media. There are many different filesystems available which all address different targets. Because size is always a critical parameter, some of them put their main focus on compression. Other setups may require a fast startup time. Thus, some have the ability to directly execute files from flash memory. This section gives a short overview about the most common filesystems used in embedded Linux systems. Furthermore, we try to evaluate the boot time of these as this is the main concern of this work. We created each filesystem image out of a predefined set of files and init scripts. A detailed setup description can be found in section 4.2, the filesystem listing can be found in appendix A.

CramFS (Compressed ROM filesystem) *CramFS* is a free read-only filesystem, mainly used in embedded environments. Its main focus is on simplicity and good compression. On the other hand, it also has some limitations: the filesystem size is restricted to roughly 256 *MiB* whereas the filesize is limited to 16 *MiB*. Normally, the kernel accesses the filesystem through the mtd subsystem (I). *Linear CramFS* (II .. V) was initially developed by *Montavista*. By directly accessing the flash chip via its physical address the *CramFS* driver bypasses the mtd abstraction layer. Therefore, the user has to specify the **physaddr** flag at the kernel command line. *Linear CramFS* support is also a prerequisite for application XIP as instruction fetches rely on directly addressable memory. Filesystem III has XIP disabled, whereas IV and V do have binaries marked for execution in place.

AXFS (Advanced XIP filesystem) *AXFS* is a quite new read-only filesystem with its main focus on execution in place. It is not part of the mainline kernel yet but the developers strongly aspire its integration. The new idea behind *AXFS* is to not only allow single files to be executed in place or to be compressed (like *Linear CramFS*) but to do this separation at finer granularity, namely on a by-page basis. *AXFS* therefore integrates a profiling system which monitors a typical system startup and "gives designers a way to measure which pages to XIP" [15]. Filesystem VI represents a fully compressed *AXFS* filesystem with profiling switched off (to hide the profiling overhead during start up). Unfortunately, we were not able to mount a partly compressed image with XIP enabled as the *AXFS* driver for kernel version 2.6.21 had a major bug at the time of testing (but should be fixed now).

SquashFS *SquashFS* is another read-only filesystem for Linux which compresses data, inodes and directories. It can use different block-sizes (up to 1 MB) for compression and allows files up to 2^{64} bytes in size. Furthermore it supports big and little endian systems and has been tested on PowerPC, i586, Sparc and ARM architectures. In the comparison, filesystem VII is a *SquashFS* filesystem mounted via the mtd abstraction layer.

- I *CramFS*, access through mtd layer [~ 2.2 MiB]
- II *Linear CramFS*, XIP disabled [~ 2.2 MiB]
- III *Linear CramFS*, busybox marked as XIP [~ 2.2 MiB]
- IV *Linear CramFS*, busybox and fbtest marked as XIP [~ 2.7 MiB]
- V *Linear CramFS*, busybox, fbtest and uClibc.so marked as XIP [~ 2.7 MiB]
- VI *AXFS*, XIP disabled, profiling off, mounted via physaddr-flag [~ 2.2 MiB]
- VII *SquashFS*, access through mtd layer [~ 2.0 MiB]
- VIII *JFFS2*, access through mtd layer [~ 2.3 MiB]

	I	II	III	IV	V	VI	VII	VIII
mount fs	0.016 s	0.024 s	0.024 s	0.024 s	0.024 s	0.012 s	0.024 s	0.135 s
start scripts	1.119 s	0.892 s	0.935 s	1.010 s	1.168 s	1.019 s	1.046 s	0.968 s
start appl.	0.140 s	0.100 s	0.150 s	0.060 s	0.010 s	0.110 s	0.230 s	0.140 s
copy file	0.150 s	0.110 s	0.060 s	0.060 s	0.070 s	0.050 s	0.120 s	0.080 s
total time	1.425 s	1.126 s	1.169 s	1.154 s	1.272 s	1.191 s	1.420 s	1.323 s

Table 2: Filesystem comparison

JFFS2 (Journaling flash filesystem 2) *JFFS2* is a log-structured filesystem targeted for flash memory mainly used in the field of embedded systems. In contrast to all other filesystems, *JFFS2* also supports writing. To protect its content from data damage or corruption due to a loss of power, *JFFS2* implements a log to track changes to the filesystem. If a file transaction fails due to a power loss, the filesystem can restore the previous state.

Comparison Table 2 illustrates the boot time measurements we have made with different filesystems and configurations. For the sake of clarity, the first part shows only the system configuration.

Discussion The first observation is that a filesystem mounted through the mtd layer generally is a bit slower (compare I and II), although the process of mounting itself is surprisingly faster. *Linear CramFS* with execution in place enabled (III - V) doesn't bring a notable performance boost. Even worse, filesystem V showed the overall highest startup time, although starting the framebuffer test application was the fastest in that setup. The slightly larger filesystem image is caused by the uncompressed files marked for XIP. The relatively slow flash (compared to RAM) may explain the difference in that case. Mounting a *AXFS* filesystem is very fast (VI). *AXFS* also shows a quite good overall performance which is comparable with plain *Linear CramFS*. *SquashFS* offers the best compression but the total boot time is quite long (VII). Due to the fact, that a *JFFS2* filesystem has to scan the whole flash while being mounted this process consumes comparatively much time (VIII). The overall performance, however, was surprisingly good. In

summary, it is fair to say that executions in place (XIP) have not brought the anticipated effect. The evaluation of *AXFS* with XIP enabled, however, is still desired for future work.

5.4.2 Parallel init script execution

In many desktop Linux systems as well as in embedded distributions the application (or userspace) booting phase consumes a large amount of time due to the serial execution of init scripts. Probing for certain devices or loading kernel modules like a Ethernet driver might take a relatively long time. The overall load time can be reduced, if these actions would be performed in parallel. This assumption is only true, if scripts or processes do not depend on each other. In desktop environments, applications like *bootchart* [16] can help to visualize the complex userspace boot process in order to identify opportunities for optimization. In the embedded field, however, the use of this tool is not recommended as it is written in the interpreted language Perl. Furthermore, a lot of processes are created that all add an extra overhead, which might skew the results. Although the author of [2] announced *embootchart*, a version for embedded systems, there haven't been any releases so far. While writing this report, we found a Google code project with the promising name *bootchart-lite*. Unfortunately, we didn't have the time to try it out but we encourage this for future work.

In this work, the *busybox* implementation of *init* was replaced with *myinit* [17]. *Myinit* is a lightweight implementation which supports parallel execution and dependencies via reference counting. The migration process is easy to perform. The scripts can remain the same except a small syntax change which defines the dependency keyword. The attribute allows every script to synchronize itself with one or more other scripts. In fact, this means: if *script A* claims to depend on *script B*, the *myinit* scheduler first starts *script B* and waits for its complete execution before starting *script A*. This feature allows the user to create a hierarchy of script but also permits the parallel execution of scripts that don't rely on each other.

As the sample system configuration only consists of five init scripts, the dependency tree is relatively small as one can see in figure 6. Experiments showed, that the parallel execution reduced the application phase duration from 1,047 *ms* to 798 *ms*. The performance gain is application specific here, too.

5.4.3 Device node population

Modern Linux systems with kernel version 2.6 feature support for dynamically adding and removing devices and their respective device nodes. This feature is also known as hot-plugging. After the kernel startup phase, a userspace hot-plugging application manages the creation of device nodes that have been internally generated during boot. During normal system operation the kernel also informs the hot-plug daemon about changes, i.e. newly added or recently removed devices. Interested users can also inspect the */sys*-path in their Linux environment. This virtual filesystem (*sysfs*) holds all information and attributes about devices, drivers, modules and

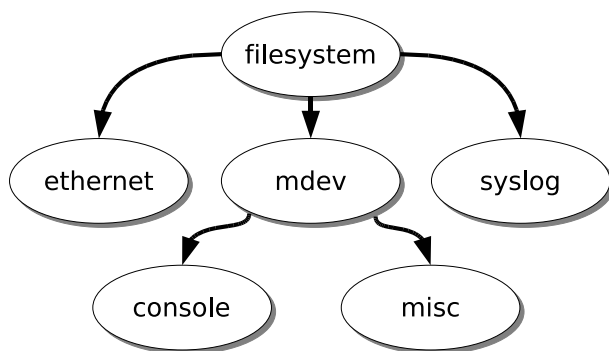


Figure 6: *myinit* script dependency tree

other kernel internals (like for instance scheduling policy or CPU frequency scaling information). It is also possible to interact with drivers through this interface, if they support this feature.

In common desktop systems, nearly every distribution installs *udev* as the standard hot-plugging daemon. This is a feature-rich but quite bulky application. In many embedded environments, *udev* can be replaced with *mdev*. This is a lightweight *busybox* applet which implements the core features of *udev*. However, the initial device node creation after kernel execution might still take a relatively long time. This is because the hot-plug daemon has to iterate through the complete *sysfs* tree at startup. In the field of embedded system, the system configuration is not likely to change. Therefore, it is feasible to completely eliminate the device node probing. Of course, one option is to fall back to static devices nodes and fully avoid *mdev* or *udev*. But this also implies the loss of comfort and system quality, that dynamically created device nodes are able to achieve. The following proposal tries to meld the advantages of both worlds. At first, we started the system with fully enabled *mdev* in order to create a device node tree which is considered to be representative for the setup. After that, we copied these nodes into a backup folder. As a next step, we modified the startup script to avoid running the *mdev* device node probing after mounting the `/dev` filesystem. In fact, we replaced this command with a routine which copies the backed-up devices nodes into the empty virtual `/dev` filesystem tree. This yields the same results as if we ran the unmodified *mdev* script. In our case, the suggested method reduced the runtime of the *mdev* script from 620 *ms* to 220 *ms*.

5.4.4 System libraries

System libraries are useful to swap out functionality and services that are used by more than just one program. They are a collection of subroutines used to develop individual software packages. A detailed description about libraries and dynamic linking is out the scope of this work and can be found in [18, 19]. If a shared library is dynamically linked against an application, which is quite common in Linux systems, the process of linking in fact does not occur at compile but on runtime. That means, if a user starts an application, the loader of the underlying operating system tries to resolve the function names that are referenced by the calling process. This is

a time consuming process as the loader has to find the appropriate libraries prior loading the code into the address space of the process. Of course, this can be avoided, if all binaries are compiled statically. But this increases the required disk space in most cases and is only suitable in a few applications. An interesting approach is linking programs to shared libraries ahead of time. This is also called prelinking [20]. A special tool accomplishes the symbol-name resolution and call-address resolution for a certain application and its shared libraries. Another proposed method, studied in [21], is to reorder the function inside a library as the "relocation time and excessive number of page faults are the key factors affecting application startup time".

6 Guidelines to speed up the boot process

Considered as a whole, this chapter gives an overview about what embedded system engineers might look at in order to reduce their system start up time. We found the following points very useful during research, whereby this list neither claims to be exhaustive nor does it claim to be in the correct order according to the potential for optimization.

- Reduce kernel, library and application size by using a thin configuration.
- Review and optimize memory configuration and enable caches as soon as possible.
- Disable console output during start up.
- Avoid *udev* or *mdev* populating the device tree.
- Disable the *lpj* calculation at system start up.
- Always evaluate the impact of compression before using it.
- Use parallel init script execution if possible.

7 Conclusion and outlook

In the present work, the author addressed the reduction of start up time in embedded Linux devices. This has always been a discussed topic but became even more relevant since Linux began to move into the market of consumer electronic devices, like mobile phones, PDA's and GPS navigation systems.

Bootstrapping such complex and multilayered systems is a challenging task and was therefore divided into three levels. For each individual phase, we identified the most critical and time consuming workflows, we proposed techniques and demonstrated solutions to reduce the consumed time. All suggestions have been verified on a *Gumstix verdex XM* embedded mainboard. As a result, we created guidelines which help embedded system engineers to review their system setup in order to reduce the overall system start up time.

The final boot time, from power-up to a working bash console, could be reduced dramatically. As you can see in table 3, we demonstrated that a system start up time under 1.5 seconds is achievable.

	partly optimized setup	optimized setup
bootloader phase	152 <i>ms</i>	135 <i>ms</i>
kernel copy/decompression	814 <i>ms</i>	600 <i>ms</i>
kernel initialization	1,343 <i>ms</i>	247 <i>ms</i>
application phase	1,047 <i>ms</i>	488 <i>ms</i>
total	3,356 <i>ms</i>	1,470 <i>ms</i>

Table 3: Final boot time

We strongly encourage further research on that topic. Unfortunately, we haven't had the time to investigate application layer techniques like prelinking [20] or library optimization[22]. We would expect further performance improvement, if flash is used in synchronous operation mode. *AXFS* is a promising filesystem which couldn't be evaluated with execution in place.

References

- [1] Linus Torvalds post at comp.os.linux (1991-10-05). Available online at <http://groups.google.com/group/comp.os.minix/msg/2194d253268b0a1b>; visited on 2008-12-08.
- [2] Matthew Klahn. Visualizing resource usage during initialization of embedded systems. Embedded Linux Conference, 2006. Available online at <http://tree.celinuxforum.org/CelfPubWiki/ELC2006Presentations?action=AttachFile&do=get&target=VisualizingResUsageDuringBoot.pdf>; visited on 2008-12-03.
- [3] Intel Corporation. *Intel PXA27x Processor Family Developer's Manual*, April 2004. Order Number: 280000-001
<http://pubs.gumstix.org/documents/PXA%20Documentation/PXA270/PXA270%20Processor%20Developer%27s%20Manual%20%5b280000-002%5d.pdf>, last visited at 2008-12-03.
- [4] Intel Corporation. *Intel StrataFlash Embedded Memory (P30)*, April 2005. Order Number: 306666.
- [5] Inc Gumstix. Gumstix buildroot svn repository. Available online at <http://docwiki.gumstix.org/index.php/Buildroot>; visited on 2008-12-03.
- [6] Erik Andersen. Buildroot. Available online at <http://buildroot.uclibc.org/>; visited on 2008-12-03.
- [7] Wikipedia, The Free Encyclopedia. Bootstrapping (computing). Webpage. Available online at [http://en.wikipedia.org/wiki/Bootstrapping_\(computing\)](http://en.wikipedia.org/wiki/Bootstrapping_(computing)); visited on 2008-11-25.
- [8] Wikipedia, The Free Encyclopedia. Kernel (computer science). Webpage. Available online at [http://en.wikipedia.org/wiki/Kernel_\(computer_science\)](http://en.wikipedia.org/wiki/Kernel_(computer_science)); visited on 2008-11-25.
- [9] Owen Taylors post at Fedora developers mailing list (2004-10-13). Available online at <http://www.redhat.com/archives/fedora-devel-list/2004-November/msg00447.html>; visited on 2008-12-18.
- [10] CE Linux Forum. embedded linux wiki. Webpage. Available online at <http://www.elinux.org>; visited on 2009-01-14.
- [11] Daniel Parthey. Analyzing real-time behavior of flash memories. Specialeafhandling, Chemnitz University of Technology, 2007. Available online at <http://rtg.informatik.tu-chemnitz.de/docs/da-sa-txt/da-pada.pdf>; visited on 2009-01-05.

- [12] Vitaly Wool. Xip: the past, the present... the future? FOSDEM, 2007. Available online at http://archive.fosdem.org/2007/slides/devrooms/embedded/Vitaly_Wool_XIP.pdf; visited on 2008-12-03.
- [13] Tim Bird. Reducing startup time in embedded linux systems. CE Linux Forum, 2003. Available online at http://elinux.org/upload/7/78/ReducingStartupTime_v0.8.pdf; visited on 2008-12-09.
- [14] Jonathan Corbet, Alessandro Rubini og Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.
- [15] William Marone Sujaya Srinivasan Justin Treon Eric Anderson Jared Hulbert og Cheng Zheng. Advanced xip filesystem for linux. Webpage. Available online at <http://axfs.sourceforge.net>; visited on 2009-01-08.
- [16] Ziga Mahkovec. Bootchart - boot process performance visualization. Available online at <http://www.bootchart.org>; visited on 2008-12-18.
- [17] Stamatis Mitrofanis. myinit. Webpage. Available online at <http://sourceforge.net/projects/myinit/>; visited on 2008-12-18.
- [18] M. Tim Jones. Anatomy of linux dynamic libraries - process and api. Webpage, 2008. Available online at <http://www.ibm.com/developerworks/linux/library/l-dynamic-libraries/>; visited on 2008-12-03.
- [19] Wikipedia, The Free Encyclopedia. Library (computer science). Webpage. Available online at [http://en.wikipedia.org/wiki/Library_\(computer_science\)](http://en.wikipedia.org/wiki/Library_(computer_science)); visited on 2008-11-25.
- [20] Jakub Jelinek. Prelink. Webpage, 2003. Available online at <http://people.redhat.com/jakub/prelink/prelink.pdf>; visited on 2009-01-14.
- [21] Oleksiy Kokachev Minchan Kim. Instant startup for application using reduced relocation time and rearranged functions. Embedded Linux Conference, 2008. Available online at http://www.celinux.org/elc08_presentations/DDLink%20FunctionReorder%2008%2004.pdf; visited on 2009-01-09.
- [22] Joe Green. Library optimizer tool. Webpage. Available online at <http://libraryopt.sourceforge.net/>; visited on 2009-01-14.

List of Symbols

AXFS	Advanced XIP filesystem
BSP	board support package
cramfs	compressed ROM filesystem
DMA	direct memory access
GPIO	general purpose input/output
GPS	global positioning system
JFFS2	Journaling flash filesystem 2
JTAG	joint test action group
KiB	Kibibyte, 2^{10} Byte
MiB	Mebibyte 2^{20} Byte
mtd	memory technology devices
OS	operating system
RTC	real-time clock
SoC	system-on-chip
SPI	Serial Peripheral Interface
UART	universal asynchronous receiver/transmitter
USB	universal serial bus
XIP	execution in place

A Root filesystem listing

```
/
|-- bin
|   |-- busybox
|   |-- mount -> busybox
|   |-- wget -> busybox
|   `-- ..
|-- dev
|   |-- console
|   |-- log -> ../tmp/log
|   `-- null
|-- dev.startup
|   |-- console
|   |-- loop0
|   |-- mtd0
|   |-- ttyS0
|   |-- zero
|   `-- ..
|-- etc
|   |-- init.d
|   |   |-- boot
|   |   |   |-- S10filesystem -> ../filesystem
|   |   |   |-- S20mdev -> ../mdev
|   |   |   |-- S30log -> ../log
|   |   |   `-- S80misc -> ../misc
|   |   |-- filesystem
|   |   |-- log
|   |   |-- mdev
|   |   |-- misc
|   |   |-- rcBoot
|   |   |-- rcShutdown -> rcBoot
|   |   `-- shutdown
|   |       |-- K60log -> ../log
|   |       `-- K80filesystem -> ../filesystem
|   |-- myinit
|   |   |-- console
|   |   |-- ethernet
|   |   |-- fs
|   |   |-- mdev
|   |   |-- misc
|   |   `-- syslog
|   |-- fstab
|   |-- hostname
```

```

| |-- init.in
| |-- inittab
| |-- mdev.conf
| |-- mtab -> ../proc/mounts
| |-- myinit.boot
| |-- myinit.in
| |-- passwd
| |-- profile
| |-- resolv.conf -> ../tmp/resolv.conf
| `-- ..
|-- flnato0
| `-- ..
|-- home
| `-- ..
|-- lib
| |-- modules
| | `-- 2.6.21-imms
| |     |-- gumstix-smc911x.ko
| |     |-- modules.dep
| |     `-- smc911x.ko
| |-- ld-uClibc-0.9.29.so
| |-- ld-uClibc.so.0 -> ld-uClibc-0.9.29.so
| |-- libc.so.0 -> libuClibc-0.9.29.so

| |-- libpthread-0.9.29.so
| |-- libstdc++.so.6.0.8
| `-- ..
|-- mnt
|-- proc
|-- root
|-- sbin
| |-- modprobe -> ../bin/busybox
| |-- myinit
| `..
|-- sys
|-- tmp
| `-- hosts
|-- usr
| `..
|-- var
| |-- log -> ../tmp
| `..
'-- linuxrc -> bin/busybox

```

B DVD content listing

```
/
|-- configs
|   |-- busybox_1.12.1_config
|   `-- kernel_config_1
|-- flاناتoo_plots
|   |-- optimized.png
|   `-- original.png
|-- howtos
|   `-- qt-embedded-howto
|-- images
|   `-- ..
|-- nfsroot
|   `-- ..
|-- patches
|   |-- flاناتoo
|   |   `-- flاناتoo_pxa270.patch
|   |-- linux-2.6.21gum
|   |   |-- add_build_kernel_script.patch
|   |   |-- add_gpio_measurement_points.patch
|   |   |-- add_linear_cramfs_support.patch
|   |   |-- add_zboot_rom_make_target.patch
|   |   |-- gumstix_flash_cached_mapping.patch
|   |   |-- gumstix_flash_imms_layout.patch
|   |   `-- xipuImage_target_support.patch
|   `-- u-boot
|       |-- add_gpio_command.patch
|       |-- boot_gpio.patch
|       |-- move_flash_sector_protection.patch
|       |-- optimized_memcpy.patch
|       |-- optimized_mementimings.patch
|       `-- update_environment_address.patch
|-- src
|   |-- busybox-1.12.1.tar.bz2
|   |-- flاناتoo-1.0.tar.bz2
|   |-- gumstix-buildroot_with_download_and_config.tar.bz2
|   |-- mkcramfs_1.1.tar.bz2
|   |-- mkcramfs_xip_support.tar.bz2
|   |-- myinit.0.4.tar.gz
|   `-- qt-embedded-linux-opensource-src-4.4.3.tar.gz
|-- studienarbeit
|   |-- paper.pdf
|   `-- slides.pdf
```

```
|-- toolchain  
|   |-- arm_toolchain_20081010.tar.bz2  
|-- xip_imms_all_files_20090121.tar.bz2
```