CHALMERS

Linux in Automotive Environment

Master of Science Thesis at Computer Science and Engineering

ARON ULMESTRAND
DANIEL ÖKVIST

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, June 2010

ARON ULMESTRAND
DANIEL ÖKVIST

Examiner: Catarina Coquand

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2010

Preface

Linux in Automotive Environment is a Master Thesis at the department of Computer Science and Engineering at Chalmers University of Technology. It is written by Aron Ulmestrand and Daniel Ökvist. The work has been carried out at Mecel AB in Gothenburg. We would like to thank Mecel AB for giving us the opportunity to do this thesis work. We extend our thanks our mentors; Jerker Christoffersson, Lars Matsson, Lars-Henrik Lottberg and Anders Arnholm for all their help and support. Finally a big thank to Catarina Coquand mentor and examiner, who has been giving us generous support and creative criticism.

Abstract

A modern car contains many electronic devices communicating over a network in the car. Normally these devices run special real-time operating systems. If a general purpose OS like Linux could be used on these devices, several benefits are gained. First, many development tools are already available and it is more likely that developers have experience with known tools. Secondly, the source code of Linux is licensed under the GPL license which means that it is available to view and modify to the particular needs of the hardware being used. The purpose of this master thesis is to adapt Linux to run on a limited resource platform typical found in cars and optimize the boot time to meet the demands of the automotive industry. To meet the requirements, investigation of different techniques to reduce boot up time was explored and implemented. The most notable technique was to execute directly on the flash memory of the hardware that was used. Analysis of what times that was interesting to measure and how to reliably measure these times was performed. The result from this thesis was a kernel that in 190 ms was ready to execute this kind of communication, which is an improvement of the unoptimized kernel by 5.24 s but did not reach the goal of 100 ms.

Contents

1 Introduction **4**

2 Technical Background **6**
- 2.1 System Overview . . . 6
- 2.2 OMAP OSK5912 Board . . . 7
  - 2.2.1 ARM . . . 8
  - 2.2.2 Flash Memory . . . 9
- 2.3 GPL - GNU General Public License and Free Software . . . 9
- 2.4 Description of Software . . . 10
  - 2.4.1 Buildroot . . . 11
  - 2.4.2 Das U-Boot . . . 11
  - 2.4.3 GNU/Linux . . . 11
- 2.5 CAN - Controller Area Network . . . 12
- 2.6 Real-time System . . . 12
- 2.7 Different Filesystems . . . 13
  - 2.7.1 NFS - Network Filesystem . . . 13
  - 2.7.2 JFFS2 - Journaling Flash Filesystem . . . 13
  - 2.7.3 Cramfs - Compressed RAM Filesystem . . . 13
  - 2.7.4 SquashFS . . . 13
  - 2.7.5 AXFS - Advanced Execute In Place Filesystem . . . 13
- 2.8 XIP - Execute in Place . . . 14

3 Method **15**
- 3.1 System setup . . . 15
- 3.2 Choice of Software . . . 16
  - 3.2.1 GNU Compiler Collection . . . 16
  - 3.2.2 Linux Kernel . . . 16
  - 3.2.3 Using NFS . . . 16
- 3.3 Optimizing the kernel . . . 17
- 3.4 Implementation of Execute in Place . . . 17
  - 3.4.1 Filesystem for XIP . . . 17
- 3.5 Measuring Times . . . 18
- 3.6 Optimization After XIP . . . 19

4 Results **20**
- 4.1 Linux . . . 20
- 4.2 Measured Times . . . 20
  - 4.2.1 Test setup . . . 20
  - 4.2.2 Table with Results . . . 21

5 Conclusions and Discussion **22**

A Configuration of Software **23**
A.1 Software for the Host PC 23
A.1.1 Setup buildroot 23
A.1.2 Setup TFTP Server 23
A.2 Software on the OSK5912 24
A.2.1 Setup the Connection 24
A.2.2 Bootloader 25
A.2.3 Kernel 26
A.2.4 JFFS2 Filesystem 27
A.2.5 Cramfs Filesystem 28
A.2.6 Configuration of Linux for XIP 28
A.2.7 Creating a Filesystem for XIP 29
A.3 Optimizations after XIP 30

B Resetting the Board to Factory Bootloader **31**

C Memory configuration **33**

D Program to Measure Times **34**

E Boot Example for XIP **39**
E.1 Print with quiet 39
E.2 Print without quiet 39

List of Abbreviations

Term	Definition
ABS	Anti-lock Breaking System.
APT	Advanced Packaging Tool is the package manager of the Debian distribution.
ARM	Advanced Risc Machine.
Bluetooth	Is an open wireless protocol.
Bootloader	A program that loads an opererating system.
CAN	Controler Area Network.
CPU	Central Processing Unit.
DSP	Digital Signal Processor is used for A/D signal conversions.
ECU	Electronic Control Units.
ED	A differential that is controlled electronically to shift power between wheels.
EEPROM	Electrically Erasable Programmable Read-Only Memory.
Flash	EEPROM memory that is non-volatile.
FTL	Flash Tranlastion Layer.
GCC	GNU Compiler Collection.
GDB	GNU Debugger.
init	Program that spawns all other processes.
OS	Operating System.
OSK	OMAP Starter Kit.
QNX	A Unix-like real-time operating system, aimed at the embedded systems market.
RAM	Random Access Memory.
RISC	Reduced Instruction Set Computer.
ROM	Read Only Memory.
RTOS	Real-Time Operating System.
shell	Provides an interface for the user to access the OS.
TFTP	Trivial File Transfer Protocol.
TI OMAP	Texas instrument Open Multimedia Application Platform.
U-Boot	A flexible bootloader working on a variety of computer architectures.
USB	Universal Serial Bus.
Vanilla Linux	The mainline branch released by Linus Torvalds.
WinCE	Is an operating system for minimalistic computers and embedded systems.
XIP	Execute In Place.

1

Introduction

As the use of electronics in our everyday life has increased so has also the use of electronics increased in cars. Almost every technical part of the car has an ECU, electronic controlled unit, monitoring or controlling it. There can be as many as 50 to 70 ECUs in a modern car. They are spread out through the car forming a network of nodes each handling a particular task.

The network made of ECUs, has to be able to react to changes both inside and outside the car. It also has to react to the driver when he or she controls the car. This reaction time has to be without any delays. The term real-time is used to describe this behavior. As an example we take the ECU controlling the Anti-lock Brakes System. It measures, at all times the rotation of the wheels. During braking if one wheel lock, the ECU instantly signals to releases the brake pressure, to let the tire resume rotation and grip the surface before applying the brake pressure again. Apart from its main use it can also tell the engine when the wheel is spinning during acceleration. This will, on some cars cut down the engine power to reduce the spinning. This is done with communication between the ECUs of the ABS brakes and the engine. Another example could be that a signal is sent to the ECU controlling the electronic differential that distributes the power between the wheels, telling it to send less power to the wheel spinning. Often the correct decision is made only after interaction between several ECUs. To secure that this interaction is fast enough, these signals must be sent in real-time. This demand and the need to reduce the total amount of wiring led to the development of a new type of protocol called the CAN, which is now well established as an industry standard for the data bus in cars.

When the car is started all ECUs have to report in their status, and hopefully all the ECUs reported in as operational. If they fail to respond in 100 ms the other ECUs will recognize that this unit is broken. This is a long time for a booting a real-time system but in relation to booting Linux on personal computer, which takes tens of seconds it is short. This is why much emphasis is put on shortening the boot time to see if it is possible to reach this goal of 100 ms, a common requirement in the automotive industry.

The report mentions demands created by software and hardware in cars. These demands are many and sometimes strict which leads to complex solutions and long development cycles. Requirements such as, spare parts have to be available for many years after production has ceased of a model. This is not that difficult to meet for a plastic fender or a headlight, but for a complex component like an ECU it is harder. The fact that a CPU must be guaranteed to be available for production over many years with the same specification is a tough demand. Especially looking at Moore's Law which states that the long term trend where the number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years. This makes even the newest and most impressive hardware obsolete long before the car in which it is placed. Also development times in the car industry is generally longer than for other industries, most note worthy the mobile phone industry.

This Master Thesis is about Linux. More specifically how it is possible to adapt Linux to run on a limited resource platform typical found in cars and optimize the boot time to meet the demands of the automotive industry. That is, to make Linux the base OS for an ECU. Many ECUs today like the ones in the case with ABS brakes above run small special purpose real-time operating systems. In the case where the OS is bigger, more complex, additional hardware such as help processors are used to secure that the communication meet real-time constraints. A growing field in the industry is infotainment, a term combined from information and entertainment. These systems generally have a need for a more complex OS to be able to not only work in the background but also to interact with the user. Gauges and tachometers are being replaced by displays showing increasingly enhanced graphics, to aid the driver. Such systems may include, but not be limited to: audio systems, reverse cameras, video streaming devices. This thesis tries to answer if it is possible to use Linux, which is a relatively complex OS, as a base for an ECU on a low-resource platform.

In everyday speech the name Linux is sometimes carelessly used when referring to a entire Linux distribution. But a distribution is made of many programs where the Linux kernel is only one part, although a major one. Other major components are the bootloader and filesystem. There are several more programs needed before a complete Linux distribution is formed, but for the purpose of this thesis the kernel and filesystem is in focus.

Linux is chosen because it has certain benefits that are interesting for the automotive industry. One benefit is that it is a well known operating system. Writing a new operating system is generally considered bad practice since it is a rather complex task and all software written has to be modified or rewritten for the new OS. Another is that it is free to use, no license fees to pay for development or sale of a commercial product based on Linux. A third benefit would be that a great amount of tools are available to support further development. Also there is a wide range of programs written for Linux available under the same GNU GPL License. It is also possible to modify the source code to fit a special need or port it to other architectures. One of the strengths of Linux is the large community, made of private persons and big multinational companies that contribute to the development of both Linux kernel and userspace application.

The purpose of this master thesis is to analyze the possibility of installing and running Linux on an limited resource platform while still meeting the requirements without the use of extra hardware.

2

Technical Background

In the introduction a wider perspective was presented and this chapter describes more detailed explanations of the technical components used. As mentioned above the car contains numerous ECUs. In this system overview, explanations zoom in on just a part of the system given in the introduction. There is a section zooming in even more to explain the board where the processor is found. It is on this board and with this processor that Linux will be running. Sections give background information on bootloader, Linux and filesystem. Together they form a Linux system. Tools for compiling this system for our board are found in buildroot. All detailed installation instructions for software and setups are found in appendix A.

2.1 System Overview

A car radio that was once just a radio has evolved over time to become more and more complex. A radio in a modern car is today a small computer controlling and serving the audio and visual resources of the car. For instance it can integrate radio, iPod, mobile phone, graphical displays and many more gadgets and functions to be found in the car. The computer needs an operating system to run these applications. Usually this means small real-time OSes but it is possible to find WinCE or QNX, which are more advanced. Such a unit is no longer a simple radio but a complex ECU that is connected through other ECUs over the CAN bus.

The CAN network in a car is divided in different buses. For example, the system controlling the brakes has different requirements than the system controlling multimedia in the car. These two systems function independent of each other connected through gateways so that the non critical systems does not interfere with the critical ones.

A part of a more complex system with many ECUs could look like the figure below. This part describes how only one ARM processor 2.2.1 is connected to its environment. In this system it is possible to run Linux on the ARM processor but then another dedicated processor is needed to take care of the communication over the CAN bus in real-time. The ARM processor is also connected to a display in the car. The dedicated processor that handle the communication in real-time is connected directly to the CAN bus. Another component that is connected to the CAN bus could be some kind of diagnostics tool, that make sure that all ECUs are connected and operative. Buttons and sensors connected to other ECUs could also send messages over the CAN bus. This configuration is pictured in figure 2.1.

As mentioned before, the number of processors in a car can be rather large and if one of them can do the work of two, the cost is reduced. Usually the processors used are not that expensive but when a large amount of cars are produced the cost naturally gets high. In this case, if the work done by the dedicated processor used for real-time communication could be carried out by the ARM processor the need for the dedicated processor would vanish. This would require that

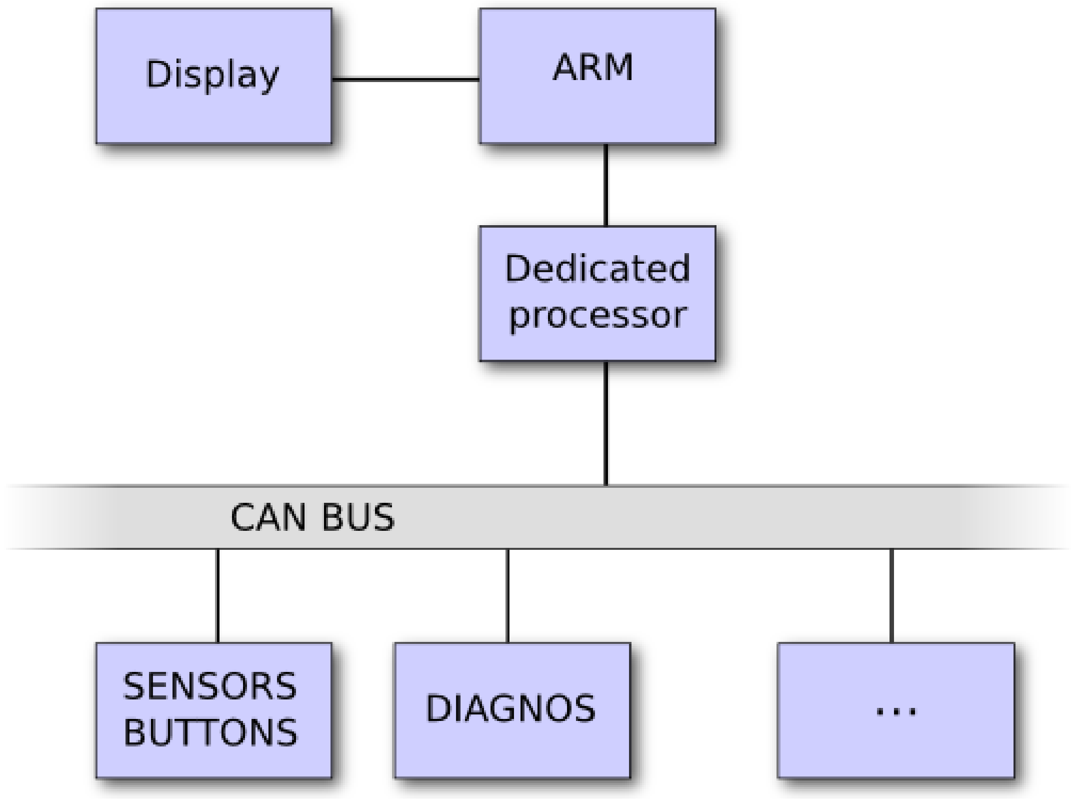

Figure 2.1: Overview of the system with the dedicated processor.

the ARM processor has finished booting or at least has reached a phase in the boot procedure, where it is possible to do the diagnostics communication within the allowed time frame given in the master thesis proposal. Also the need for real-time support in the Linux kernel would be necessary for further communication. The new setup is described in 2.1.

2.2 OMAP OSK5912 Board

This project uses a development board with an ARM processor. The board has three connectors, USB, serial and Ethernet. It also supports expandable modules that make it flexible as a base board for production development.

The communication with the board can be done in a number of different ways. The serial port is used to communicate and send commands. Data is sent through the Ethernet port using ordinary TP cable. Last for resetting the board to the factory settings a connection with an USB to USB cable has to be made.

Hardware:

- CPU 192 MHz, ARM family 9E
- 32 MB NOR Flash in two banks
- 32 MB SDRAM

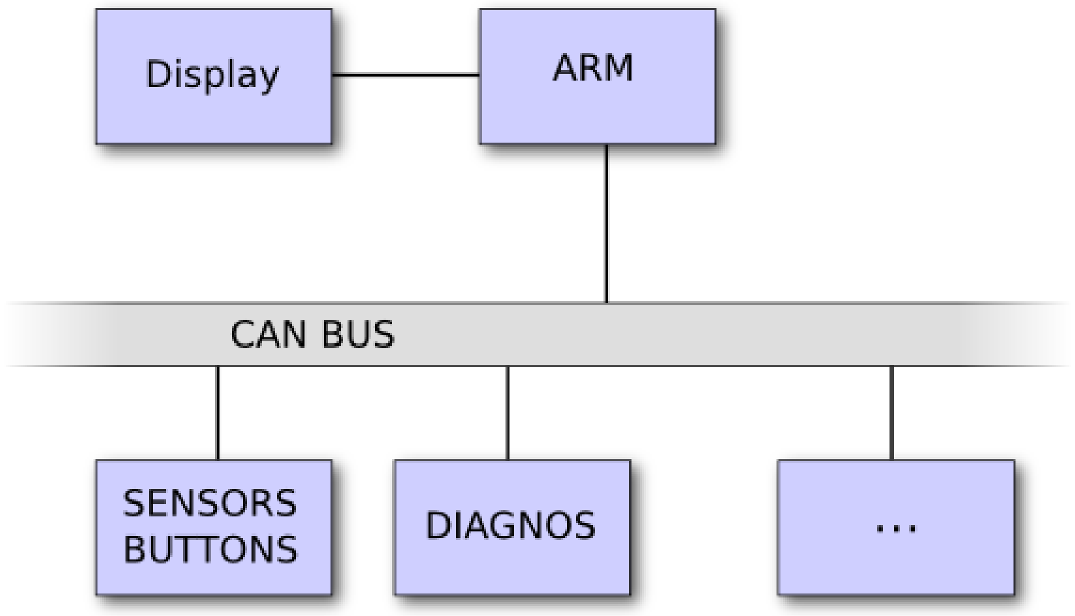

Figure 2.2: Overview of the system without the dedicated processor.

It is the ARM box in the figure 2.1 above that represents our board on which Linux is installed. To run Linux on our board some additional parts is required for Linux to start such as a bootloader and a filesystem. The bootloaders task during a boot is basically to jump to the address in the memory where the Linux kernel is located. It also sets up the CPU, registers and memory. One common bootloader that is often used on embedded hardware is U-Boot. This is what we used during development since it is rather simple but still has sufficient functionality to upload files and modify the flash memory of the board with the ARM processor. The extra functionality of U-Boot takes some time to load and in a production system, U-Boot is usually replaced with an automotive bootloader that is adapted for the hardware in use. This bootloader would not have all the functionality of U-Boot and adaptability but that is not needed when the system is ready.

When the kernel is jumped to and starts its execution it will at some point need a filesystem to function. The address in the flash memory of the filesystem can be specified in the bootloader and is then passed on to the kernel when the execution starts. A picture to illustrating the flash memory with the bootloader, kernel and filesystem is shown in the figure 2.4.

2.2.1 ARM

The ARM family of processor is found in many embedded system that can be found in everyday life. For one instance it can be in the ECU in your car controlling the engine response or in your mp3player controlling the user interface and playing the music. For the last ten years it has found its way in the mobile phone industry, and has with the expanding use of cell phones increased in popularity. In 2011 the expectancy is to sell 5 billion ARM processors.[8]

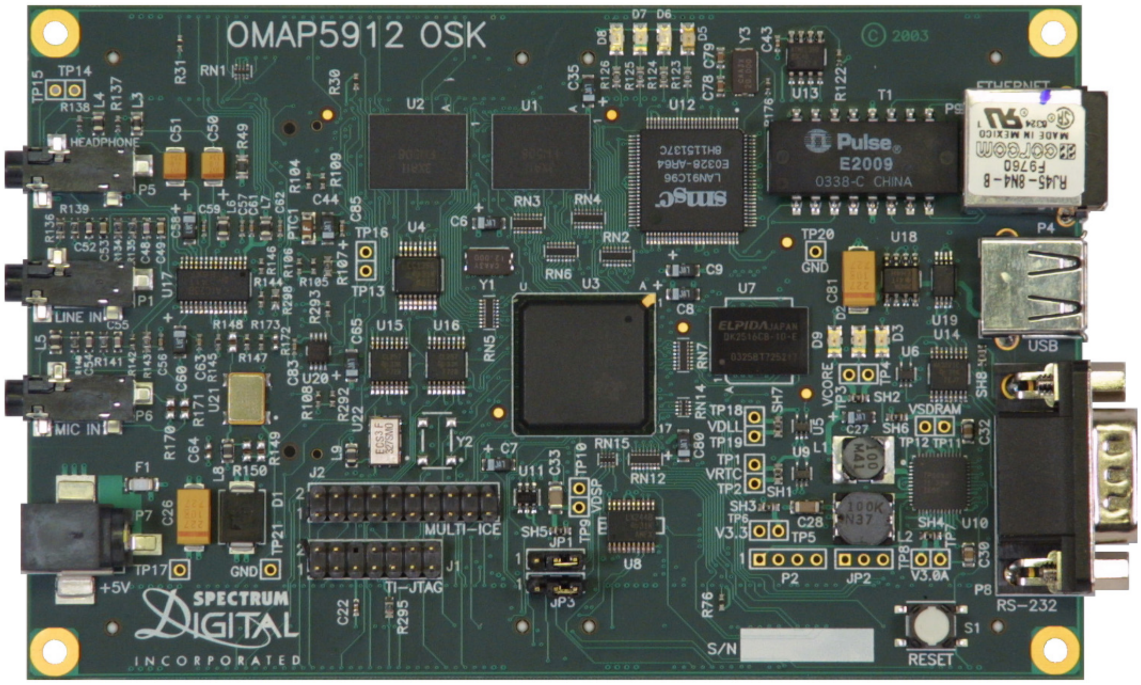

Figure 2.3: OMAP OSK5912 Board.

2.2.2 Flash Memory

Flash memory is used in many embedded devices since it provides solid state storage with high reliability at a low cost. There are two different kinds of flash memory NOR and NAND flash[6]. Here it is important for further reading on Execute In Place in section 2.8, to note that we have NOR flash memory on our board. NOR is the traditional flash memory. One advantage is that it is directly accessible but it is on the other hand more expensive. NAND has lower cost per megabyte but on the other hand is only accessible through a single 8-bit bus used for both data and addresses. They share the feature that each bit is set to a logic one when it is erased and can be set to zero with a write operation. But they are different in how the blocks are arranged. NOR flash has a block size of typically 64-128KB and NAND of 8-64KB. Resetting bits from zero to one cannot be done individually, but only by resetting (or “erasing”) a complete block. This applies to both NAND and NOR. The lifetime of a flash chip is measured in such erase cycles, with the typical lifetime being 100,000 erases per block for NOR flash memory. To ensure that no single erase block reaches this limit before the rest of the chip most users of flash chips attempt to ensure that erase cycles are evenly distributed around the flash memory a process known as “wear leveling”.

Some comparable facts about flash memory[7]:

- NOR fast and expensive
- NAND more dense, cheaper.
- NOR 100 000 write cycles guarantied for all blocks.
- NAND 1 000 000 write cycles guarantied for block zero.

2.3 GPL - GNU General Public License and Free Software

GNU GPL[14] stands for GNU General Public License and was written by Richard Stallman to give the users of software freedom. GPL is only one of many licenses acquiring this goal but since

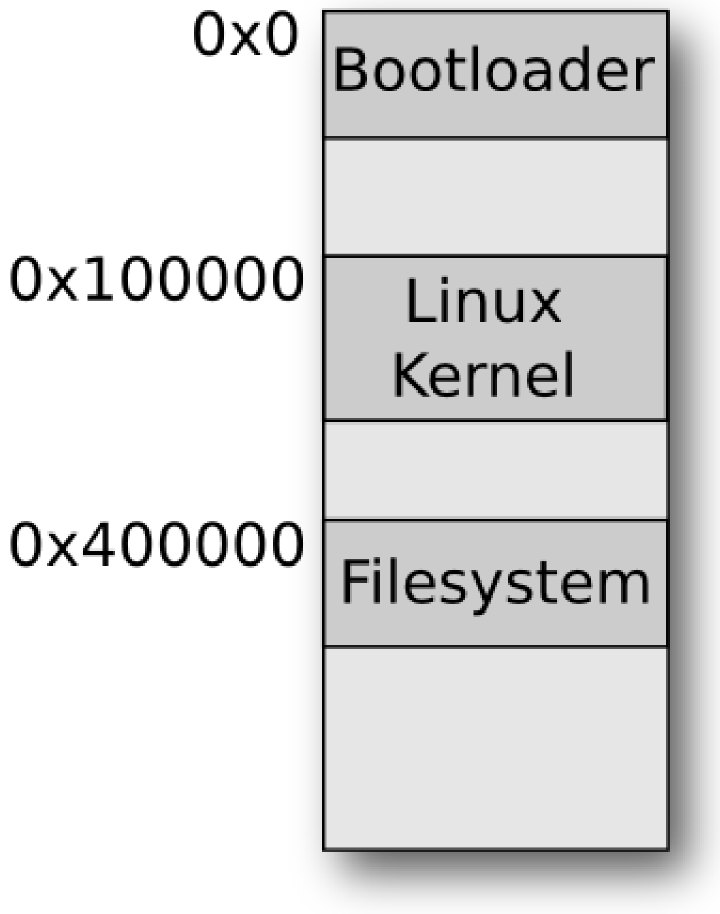

Figure 2.4: Illustrating where in flash memory binaries is placed

most of the software used in this thesis was released under this particular license it is mentioned and not the others. The freedom does not refer to price but simply stating that the following criteria should be met[9]:

- the freedom to use the software for any purpose
- the freedom to change the software to suit your needs
- the freedom to share the software with your friends and neighbors
- the freedom to share the changes you make

When software meets the above it can be called “free software”according to the free software foundation. When modifying source code released under this license the source code has to be released publicly under the same license.

2.4 Description of Software

This section describes the most important software that was used during this master thesis.

2.4.1 Buildroot

Buildroot[15] is a toolkit to make development for small or embedded systems easy. It is basically a set of make files used to compile a set of tools for development for a wide range of platforms. The tools are the same as those found in your regular Linux distribution, GCC and GNU debugger, but they are compiled for the target platform instead. To compile the kernel for target is a process called cross compiling. In addition to the tools mentioned above buildroot also helps with the compilation of the actual kernel and filesystem. Buildroot is used to configure the software that is required in a similar way the Linux kernel is configured by selecting what features are needed. To use it the target platform has to be specified. A filesystem structure can be generated with buildroot and made into an suitable filesystem image, for example JFFS2 of CRAMFS. The kernel can also be compiled by buildroot but in our case we did this separate to have more control over the process. When the configuration is done and accepted, buildroot downloads the specified code and compiles it incorporating the settings that have been selected. This is a simple way of getting a whole development environment for the target architecture that is used.

2.4.2 Das U-Boot

The name is a play on words, the German particle das implying that it is a “unterseeboot”and not a boot loader but the actual meaning is the Universal Boot loader. Das U-boot itself is not a play but a serious bootloader for booting a wide range of kernels on a wide range of systems including Linux on ARM hardware. It does more than simply booting Linux. In case one desires to write a stand-alone application, U-Boot has support for setting up vectors for interrupts, I/O and, serial communication. It also comes with a rather large list of commands to manipulating environment variables and communicates over both TFTP and NFS. It is also possible to manipulate Flash and RAM memory directly. For instance to replace U-boot or the kernel by erasing one of them and uploading new version.

2.4.3 GNU/Linux

Writing about GNU is impossible without mentioning Richard Stallman in the same way that it is impossible to mention Linux without including Linus Torvald. These two gentlemen represent two generations of hackers. Richard Stallman was running the GNU project with its attempt to create a free (as in the definition above) operating system. The final piece of software to complete the system would be the HURD kernel. Following the success of GNU Emacs and GCC, the GNU project ran into delays when developing the HURD kernel.

Linus Torvald was in the meantime writing on his hobby project, a kernel based on Minix a UNIX like microkernel. The kernel was first written for i386 and didn’t get much notice. Then Linus released his kernel under the GPL and it was ported to newer architecture[9].

The interest in Linux comes from if not only the operating system but also the added property the GPL license gives it. It states that you have full access to the source code if you in turn release your modifications to the code under the same GPL license. This has come to mean that most programs for Linux are also available under GPL which gives an unbelievable rich flora of programs and source code to aid further development. These are the two strengths of Linux. To expand more on these, the first gives the possibilities to modify the operating system to our every need. And the second removes the need to reinvent the wheel. When the kernel was released it attracted more developers. First it was ordinary users but later commercial companies joined in to support the development. Currently it is developed both commercially and non-commercially.

Writing complex software like an operating system from scratch is generally not a good idea if a well tested one already exists. Not only needs, the software for the operating system, to be rewritten or adapted to the new system but the staff also needs training to work with the new system. If a well known system is used chances are that the staff already has a good idea of how to operate the system and resources can be saved. One more important aspect is that it is actually free of charge to use.

When the kernel is compiled it can be specified to be compiled as an uImage. This means a header is added, after the compilation to make it compatible with U-Boot. When compiled for XIP a header must be added manually, making it a xipImage.

2.5 CAN - Controller Area Network

Controller Area Network or CAN as it is more commonly known is the leading serial bus system for embedded control. It was internationally standardized (ISO 11898-1) in 1993.

The need of a serial bus in cars came from the rapid increase in complexity by the end of the 1990s. The number of networked ECUs distributed in the car increased from 5 or less to around 40 [1]. This gave rise to many problems. One such was the sheer amount of electric cables needed to connect all components together with traditional point to point connectors. The weight of all that copper was one issue, not to mention the cost of manufacturing, installing and maintaining it. The solution the industry came up with was the CAN protocol.

One technical reason why it works well in this environment is that the worst case response time in CAN networks can be calculated. Based on this it became possible to engineer CAN based system for timing correctness, providing guaranties that all messages, and the signals they carry, would meet their deadlines. This is a quality highly searched for and one of the reasons why CAN-protocols work so well with real-time systems with demand for high and accurate responsiveness.

The technical specifics of the CAN protocol is that it is an asynchronous multi-master serial data bus that uses Carrier Sense Multiple Access Collision Resolution (CSMA/CR). This specifies how messages are propagated over the bus and how collisions are solved. All nodes have identification ID. The IDs form a hierarchy with no two alike. All nodes can send when the bus is free. A message sent is sent to all nodes. When a conflict is detected the message with the higher ranked id continues to be propagated over the network and the lower ranked message is killed. One important aspect of the CAN protocol is that it is deterministic. There are several CAN protocols and like ordinary networks the bandwidth has been upgraded allowing higher throughput. Any unit connected on the can bus can send a wake up message, then all the functional units on the CAN bus must respond in 100 ms or else they are considered to be inoperable and is subsequently removed from the list of available nodes.

2.6 Real-time System

We often mention the need of real-time systems in cars, but what is a real time system and why is it needed?

The term real-time system is used to describe software that must produce correct response to an event at the proper time [2]. There are two categorizations based on what occurs when this demand is not met, soft and hard real-time failure. The first, soft real-time, is when failure produces undesired result. The second hard real-time means that such a failure is catastrophic. For example we can take an electric speedometer or electric controlled anti-lock brake system (ABS). If the task of adjusting the speedometer is not performed in real time the driver feels the unresponsiveness and it may be inconvenient and dangerous. However if we would have unresponsiveness in the ABS and the brakes would fail to operate it would be catastrophic and might lead to a serious accident.

These examples above show why there is a need for a real-time system in cars. A system that runs in user time or schedules processes without giving real-time tasks real time access to CPU can give no guaranties that the same errors would not occur.

2.7 Different Filesystems

There are as many combinations of Linux kernels and filesystem that there are flavors of ice-cream. It is possible to choose filesystems not only on requirements, but also on taste and personal preference since many of them have similar features. Obviously we have not looked at them all, but choose to focus on the types of filessystem that is used in embedded Linux. They differ from regular filesystems in a number of ways. One of these differences is compression. Compression helps to relieve some of the cost and size constrains on embedded system. Some use MTD device API instead of block device API, enabling them to use special characteristic of FLASH memory, reducing the cost of overhead. A third difference is that, not only is read only system often used, it is preferred in some cases since it can offer higher performance and runtime reliability as well as more space-efficient design. It is also possible to use more than one filesystem. For example if some files are static it could be a good idea to use a read only filesystem but the changeable userdata can be stored in another filesystem that support both read and write.

2.7.1 NFS - Network Filesystem

NFS means network file system, and that is exactly what it is, a file system that is possible to mount over a network. It was developed by Sun Microsystems in 1984 and is designed to be independent of machine network and transport protocols[3]. It can be a useful tool in development. With the filessystem mounted in Linux over the network, it is possible to make changes in the filesystem on its remote location, i.e. typically the developer's computer. This is practical since there is no need to go through the steps of recreating the filesystem and downloaded to target board to test changes.

2.7.2 JFFS2 - Journaling Flash Filesystem

JFFS2 is the second version of Journaling Flash Filesystem [4]. The first JFFS on which the later is based on was among the first filesystem to operate directly on flash memory. Other filesystem works against the FTL layer that emulates a block device.

2.7.3 Cramfs - Compressed RAM Filesystem

Cramfs or Compressed ROM Filesystem[5] is a read only filesystem found mainly in embedded systems. Files are compressed one page at a time to allow random read access and it is read only to simplify design. It is limited to filesizes 16MB and total system size less than 272MB. Like the name indicates it is a compressed filesystem but unlike a conventional compressed filesystem it can be used without a decompression first.

2.7.4 SquashFS

SquashFS is a compressed read-only file system for archival use and embedded systems. SquashFS is already a stable file system integrated in Linux kernel 2.6.29.

2.7.5 AXFS - Advanced Execute In Place Filesystem

Cramfs saves a lot of RAM memory but requires extra Flash memory, SquashFS saves Flash but requires more RAM. AXFS[11] is an alternative to both of them combining their best attributes while adding original ideas. Along with the support for the filesystem in the kernel there is also profiling extension. In here is where some of the cleverness that makes AXFS excel. With profiling extension on, it identifies what pages in the filesystem image are actually accessed. Later this information is used to make optimum strategy on a page level regarding the choice of leaving files uncompressed to be able to executed in place in ROM or compress them.

2.8 XIP - Execute in Place

Execute in place is a common technique found in embedded systems. It is based around the idea that copies are costly, especially so in the boot up phase. By executing directly from read only memory instead of copying machine code to random access memory before executing there is time to be gained. Also the size of the OS or kernel does not any longer influence boot up time, since it does not have to be moved. Additional benefits from that are that it is possible to compile not for minimum size but rather for faster exceptional speed. The downside is that RAM can be much faster than flash so it depends on the hardware configuration if there are any gains in total performance.

3

Method

This chapter describes how the work was done. All setups and work were done in Linux except where we reset the bootloader to factory version that was done in Windows XP. We decided to use the Linux distribution Kubuntu 9.10 on the host machine because it was easy to install and have extensive documentation.

3.1 System setup

The physical setup of the system can generally be done in two different ways. One option is to use a serial cable for sending commands and USB to send binaries to the board as figure 3.1 describes. The other solution is to use an Ethernet connection to send binary data. The serial port is used

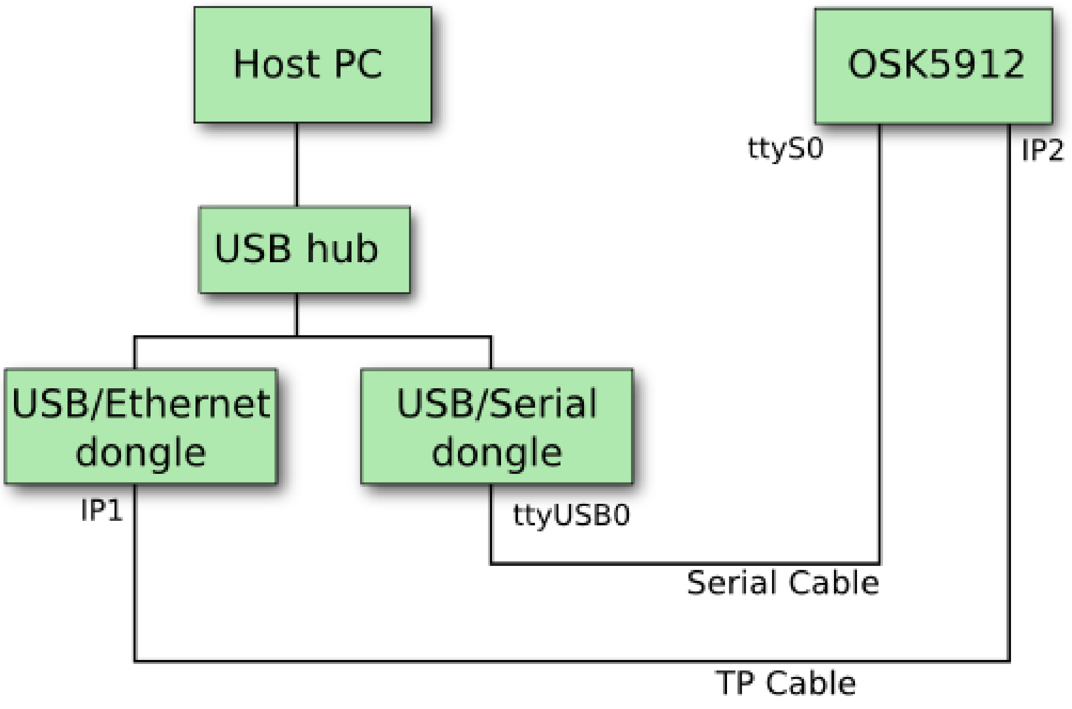

Figure 3.1: One possible configuration using USB.

to send commands like the previous setup. In our case we also used a switch between the board and host. This is described in figure 3.2.

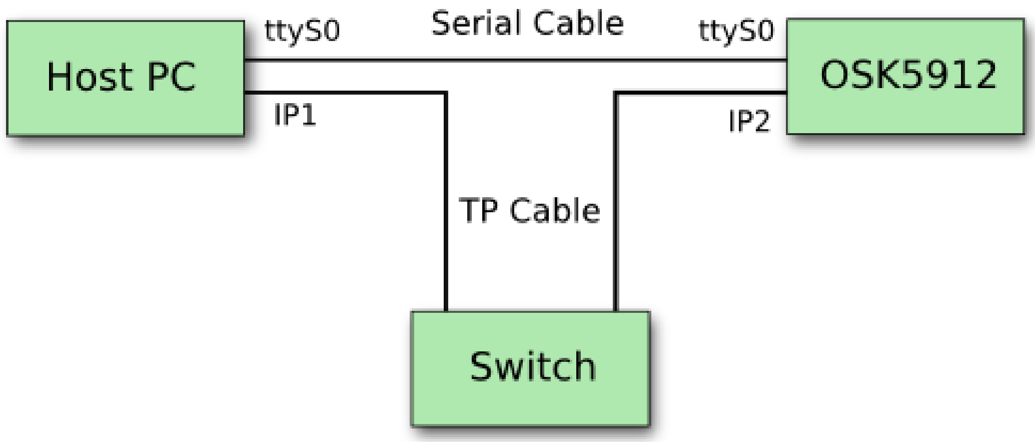

Figure 3.2: A second configuration using an Ethernet connection.

3.2 Choice of Software

3.2.1 GNU Compiler Collection

The software used for compiling on the host PC was the GNU Compiler Collection generated by buildroot. One solution would be to download GCC separately and compile it. Since we used buildroot for many other tools it was an easy choice to use it for the compiler as well.

3.2.2 Linux Kernel

The Linux kernel[10] has a fast developing pace and new versions show up approximately every 3 months. These kernels that are released by Linus Torvalds and the core developers are called the vanilla kernel or mainline kernel. Some Linux distributions use the vanilla kernel by default, while others apply patches to a chosen kernel, often of lower version number than the most current. The most recent kernel may include major changes in the code and many new features. All the new code may not be extensively tested and those who do not want to risk getting buggy code can stick to an older release that has been around for a while. Some distributions stick to an older kernel and apply security and important patches containing features necessary to get it to work at all on some hardware. For example drivers that is needed for the kernel to work on new hardware. Major refactoring of the kernel is avoided in these cases.

Since our hardware is known and will not change we decided to use one of these older kernels that has been around enduring more testing and patching of bugs. The choice fell on version 2.6.27. How the installations was done is found in appendix A.2.3

3.2.3 Using NFS

Using NFS as a tool when developing can be an efficient solution since that eliminates the need to upload files with TFTP to the board because the target has a filesystem on the host machine mounted. In our case we were not editing a lot of files in the filesystem after it was created. In the final solution the system does not use NFS and that is why it is not that interesting for this master thesis. Instead we use a filesystem optimized for our specific case. We did however setup NFS but rarely used it since the files we uploaded to the board was placed outside the filesystem directly on an address in the flash memory.

3.3 Optimizing the kernel

Optimizing the kernel is often done in an iterative manner, where one start with the unoptimized kernel and then remove one (or a few) options in the configuration, until all unnecessary features are disabled until the requirements are met. Some features might make the kernel larger in size but will make the boot faster. During a normal boot the kernel is decompressed into the RAM and then execution of the kernel code is started from there. The time for uncompressing and copying to RAM depends on the size of the kernel. Naturally this means, the more code that is excluded from the final kernel image the smaller it gets which can shorten the boot up time.

3.4 Implementation of Execute in Place

What happens during a general boot is that the bootloader decompresses the kernel and then writes it to RAM memory, in order to execute it there since RAM is faster than non-volatile flash memory. In relation to the total boot time the process of decompressing takes up quite a lot of time. If this can be avoided the boot process time could be reduced considerable. One option here is to use a kernel that is not compressed, to avoid the time it takes to decompress, before writing it to RAM. Time is gained removing the decompression stage, but now the kernel is larger and that in turn extends the time it takes to copy the kernel to RAM. An even better solution is to use Execute In Place for short XIP. This way the execution can be started directly of an uncompressed kernel from the non-volatile storage media in our case a NOR flash memory. Just like in the previous solution we will take up more space on our non-volatile storage memory but since we start the execution directly in flash the penalty will be the extra space that is used on the flash memory, but this is easily traded for the faster boot time. Another thing to have in mind is that execution from a non-volatile memory is not as fast as execution from RAM. This is also a trade off one has to live with since the execution from RAM involves first to copy the kernel from flash to RAM which is more time consuming than the slower execution time from flash.

3.4.1 Filesystem for XIP

There is a number of filesystems that is suited for use in embedded system with Linux. Some of them are Cramfs, JFFS2, SquashFS, YAFFS2, LOGFS, UBIFS and AXFS. There is also a patched version of Cramfs called Linear Cramfs. In most embedded systems the hardware is quite limited, in our case we only have 32 MB of flash memory and 32 MB of RAM memory to use. In the previous section about XIP, execution of the kernel from flash was mentioned but it is also possible to execute applications in the filesystem as XIP. To do this a filesystem is needed that is specially developed for this purpose and only Linear Cramfs and AXFS of the filesystems listed above have this feature built in. Neither Linear Cramfs nor AXFS are part of the Linux vanilla kernel so in order to use them the kernel has to be patched.

If the space usage of a 2 MB uncompressed frequently used file is compared between the cases of XIP and non-XIP. Assuming that the compression is 50 percent, then in the case of non-XIP we would have 1 MB compressed in flash and 2 MB uncompressed in RAM, in total 3 MB of memory. If XIP is used the file would take up 2 MB uncompressed in flash. In the non-XIP solution 2 MB is used instead of 3 MB. In total 1 MB more flash is used and 2 MB less RAM than the non-XIP case.[11] Since this only concerns programs running from the filesystem it will not affect the boot time which we are trying to optimize but for later use when applications are running from the filesystem this can be useful.

Memory Type	Non-XIP (MB)	XIP (MB)
RAM	2	0
Flash	1	2
Total	3	2

To save space the option to keep parts of the filesystem compressed is available. In Linear Cramfs it is possible to manually set what files to keep uncompressed for use with XIP and what parts to compress. AXFS offer the possibility to decide which pages to use XIP instead of whole files like in Linear CRAMFS which can make the system more effective. To find the pages that should be XIP, one uses a profiler to measure during runtime what is actually paged and then later takes this information and create a new filesystem image incorporated with the new information.

A read only filesystem is chosen since the data in the filesystem are static and not to be changed, the exceptions being upgrades of the system where a new filesystem will overwrite the old. The benefit of this is that the file structure can't be corrupted if the power is broken. Since the content of the root filesystem is not changed, this will offer increased stability. Both Linear Cramfs and AXFS is read only.

The features of AXFS described above in combination with that it mounted in negligible time guided us to decide on this filesystem.

3.5 Measuring Times

Since this master thesis revolve around boot times, this section describes what has been measured and how.

The interesting time to measure is the time until the communication over CAN could be carried out. Since the bootloader later can be replaced, the time the bootloader takes is not that relevant. But some of our measured times has to include the decompression and copying of the kernel since those steps always have to be done. To be consistent we decided to take a point in the bootloader before the decompression is carried out and measure from that point in all the cases.

The next question to answer is what criteria should be met before the kernel could be considered to have finished booting. The interesting time to stop measuring at is actually when the kernel is ready to communicate over the CAN bus, this is right before the shell or init is started. The shell is normally the last thing executed before the system can be used. But in this case it would be possible to start the CAN communication before the shell is started.

A few different methods for measuring is possible. One is to use a script that would print the uptime when the system has booted. Unfortunately this gives little insight in where the time is spent since it just gives the time at the end. This script also starts to measure a little too early because it also includes the time for the bootloader. The script also stopped later than where it is possible to start the communication so this method only gave wrong time, although an exact one.

Another possible method is to enable an option in the kernel that prints out exact times during the boot procedure. The downside with this is that all the prints are made to the screen this is something that takes up a fair amount of time. This method also measures the total running time from power on, but it does not start printing until the bootloader has finished. Preferably a method that gives exact times but is not dependent on prints from the kernel would be a better method.

To accomplish this, a program could listen to the data on the hosts serial port where output from the boot is sent. If this data is collected and before each of these collected lines a timestamp is added, the complete boot process would be covered. A short program was written for this purpose. This method would also work when kernel prints are turned off since it is possible to add a separate print, outside the ordinary prints from the kernel, just before the shell is started (or at any other place during the boot process where a measure point is needed).

To see an example of a XIP boot see appendix E.

3.6 Optimization After XIP

When using XIP a big reduction in time was made, though after this other optimizations were made. This section will describe these other major optimizations.

- Filesystem: Depending on what filesystem that is used time can vary considerably. The first configuration used the filesystem JFFS2. This filesystem takes more time to mount than for example Cramfs or AXFS.
- Quiet: During a normal boot the kernel usually prints out messages to show how the boot process progresses. These prints can be disabled by passing a flag in the arguments line that the bootloader passes to the kernel. Depending on how much output that will be printed the savings here vary but generally the time gain will be around 0.3 seconds.
- BogoMIPS: BogoMIPS is a hardware dependent value that is calculated during boot of the Linux kernel. It is possible to statically pass this value in the boot argument line to the kernel so that it is not calculated each time. This naturally gives some time savings.
- Disabling Legacy PTY: A PTY or pseudo terminal is a software device. There is an option in the kernel configuration for enabling these legacy devices but disabling them will not cause any problems on most systems including our system. If they are enabled they will significantly increase the boot time.
- Remove Hotplug: Hotplug lets the user plug in devices and use them immediately without manually configuring them. In this case nothing is plugged in and the hardware is specific and Hotplug is therefore not needed. This is an option that is disabled in the kernel before compilation.
- Remove MTD: Memory Technology Device is a type of device file used for interaction with flash memory. Removing this from the kernel and statically specify the address in flash will reduce the boot time.
- Replace Init: In the end of the boot the kernel will start a program named init which will spawn all other processes. This step takes some time and in a final system this would be adapted to the actual needs through a custom init program. In our case we just replaced it with a simple shell to make the boot progress, beyond this step.
- Removing RC-scripts: Services in the system are started during boot with simple scripts called RC-scripts. This could be networking or logging. These scripts are usually located in /etc/init.d/ in the filesystem and if they are removed from there, the services will not start. If RC-scripts are needed they would execute faster as a compiled program instead of a shell script and could be incorporated in the custom init program.
- Verifying Checksum: When a boot is started in U-Boot it calculates a checksum for the kernel to make sure it is not corrupt in any way. This step can be avoided by setting an environment variable in U-Boot. Since the bootloader will be replaced in a final solution this step is not that important.

4

Results

4.1 Linux

The work done in this master thesis resulted in a vanilla Linux kernel patched for OMAP and AXFS. It was also optimized and compiled to be run as execute in place with the AXFS filesystem on the OMAP OSK5912 board.

4.2 Measured Times

4.2.1 Test setup

The first four kernels are compiled, compressed and run as standard uImages 2.4.3 while the fifth and last is left uncompressed as a XIP image. The kernels for which boot times are presented are as follows.

First an unoptimized kernel that only has minor changes for mounting JFFS2, otherwise it is as near as possible to the kernel gotten by compiling default configuration. The second is a derivative of the first but now optimized as seen in section 3.3. The third one has been changed to support Cramfs instead of JFFS2 but otherwise unchanged from its predecessor. The last two setups are equal except for one major difference, the later is compiled as execute in place. To see all the optimizations for these two kernels read section 3.6.

Explanations of the times that are measured and why are found both in section 3.5 about time and in the discussion. Times are measured before and after the shell is started. This is because it is possible to start communicating over CAN before the shell is started, which is the purpose of this thesis. The time after the shell is started is interesting since either a shell or init is required for the kernel to complete the boot procedure.

To explain how the results were calculated from the boot logs an example is given below: This row can be found in appendix E.1 and the times here are in seconds:

```
[    0.359929] Kernel boot OK
```

From the print in appendix E.2 without quiet we get:

```
[    0.412834] Before running init shell.
[    0.583985] Kernel boot OK
```

To find how much time the prints take, subtract 0.359929 from 0.583985 and get 0.224055.

Finally to get the time before running the shell, subtract 0.224055 from 0.412834 and get 0.188778.

4.2.2 Table with Results

Configuration	Before shell is started (s)	After shell is started(s)
Unoptimized with JFFS2	5.428	6.207
Optimized with JFFS2	2.771	2.964
Optimized with Cramfs	2.627	2.814
Optimized with AXFS	1.451	1.576
Optimized XIP with AXFS	0.189	0.360

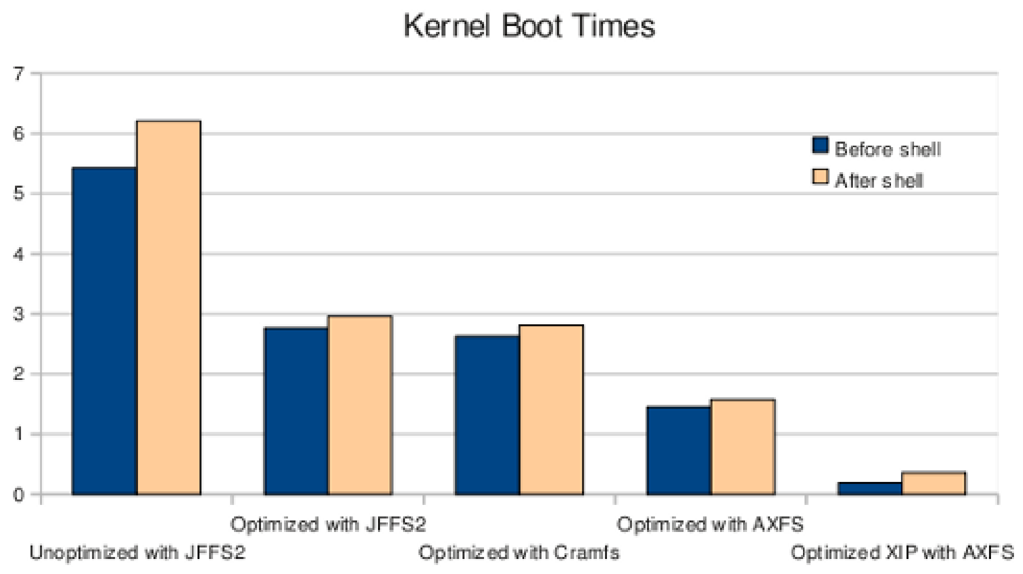

Figure 4.1: Boot times for different configurations.

5 Conclusions and Discussion

In this thesis we analyzed and optimized the boot time of a Linux kernel patched for the ARM processor. This was done as a theoretical and practical study to see if Linux can be used as OS on a limited resource platform. The platform for this thesis was the OMAP board. That Linux kernel 2.6.27.45 was chosen is explained under section 3.2.2. This was a good decision since the patches that needed to be applied were available for that version.

A setup of Linux as an base for an ECU might use combinations of the filesystems mentioned in technical background 2.7. There they are each tailored for one or several special purposes. The important thing is that the filesystem contributes to a fast boot time and this can be accomplished by a few different filesystems and we chose AXFS, since it worked well with XIP.

Another interesting aspect that came up is how the system would perform after boot, during runtime, with all the optimization that was applied. That is, how would optimization, in particular XIP affects runtime. This was not tested in this master thesis. We can only refer to other investigations in the field. [13]. In the case of application XIP the programs tend to start faster the first time but execution is a little slower. The second time the program is started no increase is seen.

In regard to boot times this thesis shows how effective Linux can be, with a boot time of only 190 ms. There are still some small uncertainties in our measurements, not the times but if we are measuring the right thing. There is a high possibility that another 20 ms can be removed. This is time spent in U-Boot before booting the kernel. But since swapping U-Boot for an automotive bootloader has not been tried, this has not been verified. Our worst case is then 190 ms and the best is 170 ms. This still puts the boot time 70 ms high of the target 100 ms in the best case.

We also mention in the report that real-time support was required during the CAN communication. Real-time execution has not been evaluated in this thesis. To further expand on this master thesis, the CAN protocol and communication in Linux, should be tried to reach a full conclusion.

There are two ways to go from our result of 190 ms to achieve 100 ms. The first is simple, but costly. Change the hardware from our board to state of the art hardware. But this solution kind of removes part of the purpose which was to boot Linux on a limited inexpensive hardware. The second solution is to build a program which schedules interrupts to send messages over CAN during the boot up phase in. This will make the boot take longer but secures the communication over CAN in real-time.

Appendix A

Configuration of Software

A.1 Software for the Host PC

A.1.1 Setup buildroot

We used version 2009.11 of buildroot which can be downloaded from http://buildroot.uclibc.org/. The setup is rather straight forward and only a few settings have to be changed. First unpack it:

```
$ tar xvfj buildroot-2009.11.tar.bz2
```

Then enter the configuration menu:

```
$ make menuconfig
```

The options we need to change or set are the following:

```
Target architecture -> arm
Target architecture variant -> arm926t
Toolchain -> GCC compiler version -> gcc 4.4.x
Toolchain -> Build gdb debugger for the target
Toolchain -> Build gdb debugger for the target -> GDB debugger version -> gdb 6.8
```

Exit and save the configuration file. The next step is to compile all we have selected in the configuration and to do this we will need to install some extra packages. The easiest way is to use the package manager and search for the required libs and programs when they are required and then install them. This can be an iterative process depending on what you already have installed. Their names and versions can vary depending on the distribution you use but generally the names are about the same. To start the compilation type:

```
$ make
```

Now you have a working toolchain so that you can compile and debug for your target architecture. The next thing at hand is to set the PATH variable to include the new compiler and debugger. The binaries that you want to add are located in 'buildroot-2009.11/output/staging/usr/bin' To do this add the following to your .bashrc file:

```
$ export PATH=${PATH_TO_BUILDROOT_FOLDER}/output/staging/usr/bin:$PATH
```

(The PATH_TO_BUILDROOT_FOLDER should be replaced by the path to the folder you unpacked in the beginning of this section.)

A.1.2 Setup TFTP Server

The installation and configuration of a TFTP server can differ a little between different Linux distributions but the principle is the same. This configuration is done on the distribution Kubuntu, but it should not be too hard to make it work with any other distribution. We use TFTP to upload files from the host that is running the server, to the board that is connecting as a client. First we need to install the required packages on the host. This can be done in a terminal with the package manager apt.

```
$ sudo apt-get install xinetd tftpd tftp
```

Secondly we need to create the file '/etc/xinetd.d/tftp' and input the following in it:

```
service tftp
{
protocol     = udp
port         = 69
socket_type  = dgram
wait         = yes
user         = nobody
server       = /usr/sbin/in.tftpd
server_args  = /tftpboot
disable      = no
}
```

Create a directory from where the OSK board will download. The parameter server_args in '/etc/xinetd.d/tftp' corresponds to this directory and in our case it is '/tftpboot'. We also need to set the correct permissions and owner on that directory:

```
$ sudo mkdir /tftpboot
$ sudo chmod -R 777 /tftpboot
$ sudo chown -R nobody /tftpboot
```

Now we are ready to start the server with the following command:

```
$ sudo /etc/init.d/xinetd start
```

The TFTP-server should now be up and running.

A.2 Software on the OSK5912

A.2.1 Setup the Connection

The commands to the board are sent from the host on a serial connection. One suitable application used on the host for this serial communication is minicom and the serial device that is used is ttyS0 or COM1. First start minicom. The flag -w turns line-wrap on.

```
$ minicom -w /dev/ttyS0
```

When minicom has started press 'CTRL-A O' to open the configuration menu. Enter 'Serial port setup' and set the serial port to /dev/ttyS0 and the settings for this port to 115200 baud, 8 Bps, no parity, 1 stop bit, and no flow control. Now check that the serial cable is connected from the host to the board and the power cable is plugged in the board.

When the board is started for the first time the U-Boot boot loader will start. When this happens a key has to be pressed to keep the board from automatically boot a preinstalled kernel. U-Boot has a variety of commands that can be given and to see them just type 'help'. For now we only need the commands to configure the network.

First the board needs to be connected to a network with a TP-cable. If there is a working DHCP server on the network this can be used to configure the network settings on the board. The configuration of U-Boot consists of setting environments variables. The first variable that has to be set is the ethaddr which is the hardware address. This address can be found beside the white reset button on the board and to set this type the following in the U-Boot prompt where the letters marked X is specific for your board and the 'OMAP5912 OSK #' is the console.

```
OMAP5912 OSK # setenv ethaddr 00:0E:99:XX:XX:XX
```

The other environment variables that have to be set correctly are ipaddr, netmask, gateway and serverip. The variable ipaddr is the IP of the board and serverip the IP of the host machine. To save your changes type:

```
OMAP5912 OSK # saveenv
```

Use the printenv command or its short hand pri to have a look at the environment variables.

```
OMAP5912 OSK # pri
```

Now a print out of the environment variables should look like this, where all 'X' is replaced by your own settings:

```
baudrate=115200
bootcmd=bootm 0x100000
bootdelay=10
ethaddr=00:0E:99:XX:XX:XX
bootargs=console=ttyS0,115200n8 noinitrd rw root=/dev/mtdblock3 rootfstype=jffs2 mem=32M
filesize=c0000
fileaddr=10000000
netmask=XXX.XXX.XXX.XXX
ipaddr=XXX.XXX.XXX.XXX
serverip=XXX.XXX.XXX.XXX
gateway=XXX.XXX.XXX.XXX
bootfile=/tftpboot/uImage
stdin=serial
stdout=serial
stderr=serial
```

A.2.2 Bootloader

The boot loader that is installed from factory is of version 1.1.1. U-Boot is used for a several important tasks when working with the OSK board. The first thing at hand is to upgrade to a more recent version. Download the version you want to run from U-Boots download page. Then unpack it and enter the folder.
Observe! Read this whole section before upgrading and dubble check everthing before erase. If the upgrade fails see appendix B for how to reset the board to factory settings.

```
$ tar xvfj u-boot-1.1.6.tar.bz2
$ cd u-boot-1.1.6/
```

Then we need to configure and finally compile it:

```
$ make distclean
$ make omap5912osk_config
$ make
```

Make sure you have the toolchain installed and added to the PATH variable, see section A.1.1. If all went well a binary named u-boot.bin is created in the U-Boot folder. Copy this file to the folder that is used by tftp, in our case /tftpboot.

```
$ cp u-boot.bin /tftpboot
```

Next we need to upload U-Boot to the board. To do this connect with minicom if you don't already have a minicom terminal. The Flash memory is divided into banks and sectors where U-Boot occupies the first sector, address 0x00000000. The second sector contains parameters and both the first and second sector are write protected at first. The print out of the first rows in the flash memory will look like this:

```
OMAP5912 OSK # flinfo

Bank # 1: INTEL FLASH 28F128J3A
  Size: 32 MB in 259 Sectors
  Sector Start Addresses:
    00000000 (RO) 00020000 (RO) 00040000      00060000      00080000
    000A0000      000C0000      000E0000      00100000      00120000
    00140000      00160000      00180000      001A0000      001C0000
```

First we need to upload U-Boot to the RAM of the board.

```
OMAP5912 OSK # tftpboot 0x10000000 /tftpboot/u-boot.bin
```

From the output we get the size in hexadecimal of the uploaded file. This size will be used when we copy U-Boot to the flash memory. But before that we need to turn off the write protection of the first sector where we are going to place U-Boot. To turn off write protection of the fist sector type:

```
OMAP5912 OSK # protect off 1:0
```

Upload the new version first to RAM. Then erase the first sector where the old U-Boot is located. Observe that if you failed to send U-Boot in the first step then it is unwise to erase the current U-Boot without any replacement ready.

```
OMAP5912 OSK # erase 1:0
```

Replace XXXX in the following command with the size you got from the output when we transfered U-Boot to the ram.

```
OMAP5912 OSK # cp.b 0x10000000 0x0 XXXX
```

That is all you need to do to update U-Boot. Now reset the board and if everything worked the new U-Boot will load. If this is not the case see Appendix B.

A.2.3 Kernel

To install the kernel we first need to download and uncompress the original vanilla source from http://www.kernel.org. We decided to use version 2.6.27.45. Next we need to patch the source to work with TI OMAP processors and this patch can be found at http://www.muru.com/linux/omap/. To download, uncompress and patch the kernel type the following while standing in the Linux source folder:

```
$ tar xvfj linux-2.6.27.45.tar.bz2
$ bunzip2 patch-2.6.27-omap1.bz2
$ cd linux-2.6.27.45/
$ patch -p1 <../patch-2.6.27-omap1
```

Next step is to configure the kernel. First lets set the configuration to the default that was supplied with the patch set and then enter the configuration menu.

```
$ make omap_osk_5912_defconfig
$ make menuconfig
```

Save the changes and exit the configuration. To compile the kernel and create an uImage type:

```
$ make uImage
```

The resulting uImage will be located in arch/arm/boot/uImage. Copy this file to you /tftpboot so that we can transfer it to the board. The transfer is done the same way as with U-Boot in the precious section.

```
$ cp arch/arm/boot/uImage /tftpboot
```

Enter the minicom terminal and do the following to upload the kernel to the RAM of the board.

```
OMAP5912 OSK # tftpboot 0x10000000 /tftpboot/uImage
OMAP5912 OSK # erase 0x100000 +XXXX
```

Where XXXX is the hexadecimal size printed by tftpboot. Copy the kernel to the erased area in the flash memory.

```
OMAP5912 OSK # cp.b 0x10000000 0x100000 XXXX
```

To successfully boot the kernel we also need a filesystem but it is possible to boot and see that it completes the booting process until it needs the filesystem.

To boot the system simply type:

```
OMAP5912 OSK # boot
```

A.2.4 JFFS2 Filesystem

The choice of filesystem can influence the boot time considerable. A more simple and fast filesystem does not always have all the required features. In our case the first filesystem used was JFFS2 since it is a read and write filesystem and has good support in the Linux kernel. To make things simple buildroot can be used as an aid in creating the filesystem. When the compilation of buildroot was finished a directory hierarchy was created that can be used to create a filesystem image.
Before we create the filesystem image we modify a file in the filesystem so that it starts getty which shows a login prompt when the user has booted the kernel. Open the file /etc/inittab in your directory structure (make sure you open the file in your filesystem for the OSK and not the file in your host filesystem) and check that the following line is present and uncommented:

```
ttyS0::respawn:/sbin/getty -L ttyS0 115200 vt100
```

Now create the JFFS2 filesystem from the directory structure and then copy it to /tftpboot with the following command:

```
$ mkfs.jffs2 -p -l -e 0x20000 -n -v -r PATH_TO_FS_STRUCTURE -o fs.jffs2
$ cp fs.jffs2 /tftpboot
```

Now you will have a filesystem image named fs.jffs2 to copy to your OSK board. Replace XXXX with the size as usual.

```
OMAP5912 OSK # tftpboot 0x10000000 /tftpboot/fs.jffs2
OMAP5912 OSK # erase 0x240000 +XXXX
OMAP5912 OSK # cp.b 0x10000000 0x240000 XXXX
```

Now the filesystem image is in place on the board and the next step is to do some minor reconfigurations in the kernel to make the filesystem work correctly. Follow the procedure how to configure the kernel described in the previous section and add the following configs.

```
CONFIG_MTD=y
CONFIG_MTD_CONCAT=y
CONFIG_MTD_PARTITIONS=y
CONFIG_MTD_REDBOOT_PARTS=y
CONFIG_MTD_REDBOOT_DIRECTORY_BLOCK=-1
CONFIG_MTD_CMDLINE_PARTS=y
CONFIG_MTD_CHAR=y
CONFIG_MTD_BLKDEVS=y
CONFIG_MTD_BLOCK=y
CONFIG_MTD_OMAP_NOR=y
```

They can be found under Device drivers.

```
Device drivers -> Memory Technology Device (MTD) support
```

Recompile and copy the kernel to the board. Now the kernel should have a working filesystem. If everything went well you should be able to boot the new kernel without any errors or kernel panic.

Also make sure that the bootargs variable in U-Boot has the flag rootfstype=jffs2.

```
OMAP5912 OSK # setenv bootargs 'console=ttyS0,115200n8 noinitrd ro
root=/dev/mtdblock3 rootfstype=jffs2 mem=32M'
```

A.2.5 Cramfs Filesystem

If cramfs is to be used some minor changes needs to be done. First to create the filesystem image type:

```
$ mkfs.cramfs -p -e 0x20000 -n -v PATH_TO_FS_STRUCTURE fs.cramfs
```

Upload it using the same procedure like when using JFFS2. Then change the bootargs variable in U-Boot using the following input:

```
OMAP5912 OSK # setenv bootargs 'console=ttyS0,115200n8 noinitrd ro
root=/dev/mtdblock3 rootfstype=cramfs mem=32M'
```

A.2.6 Configuration of Linux for XIP

When adapting Linux to use XIP there are a few extra things that needs to be done. First we need to enable XIP in the configuration file of the kernel and specify what address in flash where it will be placed. Open the configuration menu of the kernel:

```
make menuconfig
```

When running XIP we cant use MTD that we enabled when we configured the kernel for uImage, so we have to disable MTD here:

```
Device Drivers -> Memory Technology Device (MTD) support
```

Next under section 'Boot Options' one can find the XIP option that needs to be enabled.

```
Boot options -> Kernel Execute-In-Place from ROM
```

Also change:

```
Boot options -> XIP Kernel Physical Location
```

This location should be specified to '0x100040' since we have to add a header before the image we get after compilation. We will later upload the finished image to address 0x100000 but since we add this header, which has a size of 64 bytes, we need to specify the address to be 64 bytes or 40 in hexadecimal greater than the address we upload the kernel to. This header will contain some information about the image that U-Boot will use. But before we add the header we need to compile the kernel:

```
make xipImage
```

This will produce an image built for XIP without the header so now we need to add just that. The procedure to do this consists of concating the image with a binary file of 64 bytes containing ones. We then use a tool called mkimage to owerwrite the added ones with the correct header. This step is done so that we do not overwrite the actual image. The following simple python code produces a file named 'ones' of size 64 bytes containing just ones:

```
#! /usr/bin/python
# -*- coding: utf-8 -*-
import os

byte = int('11111111', 2) # 255, 0xFF
```

```
f = file('ones', 'wb')

for i in range(0, 64):
    f.write('%c' % byte)
f.close()
```

The xipImage that we got when we compiled will be located in arch/arm/boot/xipImage. To add the ones and produce a xipImage named xI.img type:

```
cat ones xipImage > xI.img
```

Now we have a XIP image with the added ones in the beginning of the file. To overwrite the ones with the acutal header:

```
mkimage -x -A arm -O linux -T kernel -C none -a 0x100000 -n "xip" -d
xI.img xipImage.img
```

Now we have a file named xipImage.img. The next section will describe how to create a filesystem that our new xipImage will need to boot. For now we can upload the file xipImage to the address 0x100000 the same way we did with the uImage in the previous section.

A.2.7 Creating a Filesystem for XIP

The installation of the filesystem for XIP has som extra steps that will be explained in the following section. First the kernel needs to be patched to work with the filesystem AXFS that we are going to use. Then we need to create a new filesystem image. Finally we have to modify the bootargs variable in U-Boot that will be used by the kernel during boot. Start by downloading and unpacking the latest AXFS tools. Then find the correct patch file in the axfs directory that we got when we unpacked it. It should be located in a directory with the same versionname as the kernel you are using, in our case 2.6.27. Change directory to the directory of your kernel, then just apply the patch with the following command:

```
patch -p 1 <PATH_TO_YOUR_AXFS_PATCH/000_xip_file_fault.patch
```

Then run the patchin.sh script found in the following directory axfs/driver/branches/linux/ while standing in your kernel directory.

```
./<path_to_patching_script>/patchin.sh
```

If all went well we now can run make menuconfig and see a new option there.

```
Advanced XIP File System (AXFS) support (EXPERIMENTAL)
```

Enable this and recompile the kernel, also don't forget to add the header like we did before.

Creating the new filesystem is done with the following command that is found in the directory axfs/mkfs.axfs/trunk/

```
mkfs.axfs PATH_TO_FS_STRUCTURE fs.axfs
```

When we uploaded the JFFS2 filesystem in the previous section we put it on address 0x240000 but since the kernel is not compressed now it will need more space and will overwrite this address. Therefore we have to specify that the filesystem is located at a higher address and then pass this on to the kernel in the bootargs variable. We chose the address 0x400000 instead of 0x240000. Upload the file fs.axfs to address 0x400000 in the same manner as when we uploaded our JFFS2 image.

Finally we need to change the bootargs variable in U-Boot so that the kernel knows that the filesystem is of type axfs. Set the bootargs variable in U-Boot to the following changing both the type of the filesystem and the address it is located on:

```
OMAP5912 OSK # setenv bootargs 'console=ttyS0,115200n8 noinitrd ro root=/dev/null
rootflags=physaddr=0x400000 rootfstype=axfs init=/bin/sh ip=off mem=32M'
```

A.3 Optimizations after XIP

Setting a few values in the bootargs variable in U-Boot specifying that we want to skip some steps can speed up the boot procedure: Disable the print to the screen by passing quiet to the bootargs. Setting the BogoMIPS statically can be done by passing lpj=373760. This value can be obtained by first booting one time and notice what lpj is set to in the line starting with “Calibrating delay loop...”. Finally it is possible to replace init with a shell. Note that this is just a temporary solution since in a real system init will be replaced by a customized init.

```
init=/bin/sh lpj=373760 quiet
```

Further removing or disabling a few options in the kernel can help us cut some time off. The most notable gains come from disabling Legacy (BSD) PTY support, MTD and hotplug. To disable hotplug just remove the 'path to uevent helper':

```
Device drivers -> Character devices -> Legacy (BSD) PTY support
Device Drivers -> Generic Driver Options -> path to uevent helper
```

MTD was already removed in the section where we created a filesystem for XIP.

All the RC-scripts can be found in /etc/init.d/ and by removing them from this location we disable them.

To disable the checksum check that U-Boot perform every time we boot, just set the environment variable verify in U-Boot to no:

```
OMAP5912 OSK # setenv verify no
```

Appendix B

Resetting the Board to Factory Bootloader

Guide in how to reset OMAP OSK 5912 by flashing it with OST-host software
You will need.

- OSK5912 board or similar
- USB male to male cable
- PC with Windows XP or newer

First download the file with the factory settings from below and change the file ending to .raw

```
http://linux.omap.com/pub/bootloader/osk/osk-u-boot_8_11_05_RevD.bin
```

Download the program Ost-host from address below. Unpack and install.

```
http://focus.ti.com/download/wtbu/OSTTools_NoSource_v2_5.zip
```

Make sure that the board is not connected to the power cable or anything else. Check to see if jumper Jp3 is in position 2-3 else move it there. Plug in the power and the male to male USB. In board and PC. Windows will say that it has found hardware and wants to search for drivers. Click on manually select drivers and select the ones found in the catalog where you installed OST-host. Start OST-host if you have not already. Go through the settings, it should look like below.

```
Platform: 16xx_17xx DDR
Interface: USB
Flashtype: NOR
Device: GP
Always use default path = check
```

Then you also need to set the correct hexadecimal target addresses as follows:

```
Flash loader address: 0x10000000
Variable address: 0x1000FFF0
Images address: 0x10010000
Write address: 0x0C000000
Read address: 0x0C000000
Read size: 0x00000000
```

Load the image devD.raw from where you downloaded it. Click on image download and flashing. Then on download to target. Break power and remove USB, now move Jp3 to position 1-2. Connect the power again and then the usb cable. Important: Connect in the right order. And this is the magic step. Wait untill the display reads waiting for connection and then press the reset button. On power on the program automaticly erases the corupt data in flash memory and installs uboot 1.1.1 in its place. Now give it a couple of seconds. If everything works ok it should read "Download and Flash Writing Successful" Now break the power again and reset Jp3 to position 1-2. Connect to the card like you usually do. For instance by teraterm in windows or minicom in Linux. Press reset and watch your board boot on its own.

Appendix C

Memory configuration

The high level memory map of the board that we started with:

adress (hex)	values
0x00000000	U-Boot
0x00020000	Parameter
0x00100000	Linux Kernel
0x00240000	Filesystem
0x10000000	SDRAM

The high level memory map of the board after we moved the filesystem:

adress (hex)	values
0x00000000	U-Boot
0x00020000	Parameter
0x00100000	Linux Kernel
0x00400000	Filesystem
0x10000000	SDRAM

Appendix D

Program to Measure Times

This program takes lines from the serial commection and adds a timestamp to it before printing it out to the screen [12]. To use it without the terminal with minicom stealing output add a small delay before the boot:

```
OMAP5912 OSK # sleep 7; bootm 0x100000
```

Then press enter and close the terminal with minicom followed by starting the script with the following command line before the delay has passed:

```
./grabserial.py -v -d "/dev/ttyS0" -b 115200 -w 8 -p N -s 1 -e 30 -t
```

To use the grabserial script you have to have pyserial installed. In kubuntu this package is named python-serial so to install this just type:

```
sudo aptitude install python-serial
```

The following is the script being used for measuring.

```
#!/usr/bin/env python
#
# grabserial - program to read a serial port and send the data to stdout

# Copyright 2006 Sony Corporation
#
# This program is provided under the Gnu General Public License (GPL)
# version 2 ONLY.
#
# 2006-09-07 by Tim Bird <tim.bird@am.sony.com>
#
# To do:
#  * buffer output chars??
#  * format the time output better
#

MAJOR_VERSION=1
MINOR_VERSION=0
REVISION=0

import os, sys
import getopt
import serial
import time
import re
```

```
cmd = os.path.basename(sys.argv[0])
verbose = 0

def vprint(message):
if verbose:
print message

def usage(rcode):
print """%s : Serial line reader
Usage: %s [options] <config_file>
options:
    -h, --help            Print this message
    -d, --device=<devpath> Set the device to read (default '/dev/ttyS0')
    -b, --baudrate=<val>   Set the baudrate (default 115200)
    -w, --width=<val>      Set the data bit width (default 8)
    -p, --parity=<val>     Set the parity (default N)
    -s, --stopbits=<val>   Set the stopbits (default 1)
    -x, --xonxoff          Enable software flow control (default off)
    -r, --rtscts           Enable RTC/CTS flow control (default off)
    -e, --endtime=<secs>   End the program after the specified seconds have
                           elapsed.
    -t, --time             Print time for each line received.  The time is
                           when the first character of each line is
                           received by %s
    -m, --match=<pat>      Specify a regular expression pattern to match to
                           set a base time.  Time values for lines after the
                           line matching the pattern will be relative to
                           this base time.
    -v, --verbose          Show verbose runtime messages
    -V, --version          Show version number and exit

Ex: %s -e 30 -t -m "^Linux version.*"
This will grab serial input for 30 seconds, displaying the time for
each line, and re-setting the base time when the line starting with
"Linux version" is seen.
""" % (cmd, cmd, cmd, cmd)
sys.exit(rcode)

def main():
global verbose

# parse the command line options
try:
opts, args = getopt.getopt(sys.argv[1:],
 "hd:b:w:p:s:xrtm:e:vV", ["help", "device=",
"baudrate=", "width=","parity=","stopbits=",
"xonxoff", "rtscts","time", "match=", "endtime=",
"verbose", "version"])
except:
# print help info and exit
print "Error parsing command line options"
usage(2)

device="/dev/ttyS0"
baudrate=115200
width=8
```

```
parity='N'
stopbits=1
xon=0
rtc=0
show_time = 0
basepat = ""
endtime = 0

# get a dummy instance for error checking
sd = serial.Serial()

for opt, arg in opts:
if opt in ["-h", "--help"]:
usage(0)
                if opt in ["-d", "--device"]:
                        device = arg
if not os.path.exists(device):
print "Error: serial device '%s' does not exist" % device
usage(2)
if opt in ["-b", "--baudrate"]:
baudrate = int(arg)
if baudrate not in sd.BAUDRATES:
print "Error: invalid baud rate '%d' specified" % baudrate
print "Valid baud rates are: %s" % str(sd.BAUDRATES)
sys.exit(3)
                if opt in ["-p", "--parity"]:
parity = arg.upper()
if parity not in sd.PARITIES:
print "Error: invalid parity '%s' specified" % parity
print "Valid parities are: %s" % str(sd.PARITIES)
sys.exit(3)
if opt in ["-w", "--width"]:
width = int(arg)
if width not in sd.BYTESIZES:
print "Error: invalid data bit width '%d' specified" % width
print "Valid data bit widths are: %s" % str(sd.BYTESIZES)
sys.exit(3)
if opt in ["-s", "--stopbits"]:
stopbits = int(arg)
if stopbits not in sd.STOPBITS:
print "Error: invalid stopbits '%d' specified" % stopbits
print "Valid stopbits are: %s" % str(sd.STOPBITS)
sys.exit(3)
if opt in ["-x", "--xonxoff"]:
xon = 1
if opt in ["-r", "--rtcdtc"]:
rtc = 1
if opt in ["-t", "--time"]:
show_time=1
if opt in ["-m", "--match"]:
basepat=arg
if opt in ["-e", "--endtime"]:
endstr=arg
try:
endtime = time.time()+float(endstr)
except:
```

```
print "Error: invalid endtime %s specified" % arg
sys.exit(3)
if opt in ["-v", "--verbose"]:
verbose=1
if opt in ["-V", "--version"]:
print "grabserial version %d.%d.%d" % (MAJOR_VERSION, MINOR_VERSION, REVISION)
sys.exit(0)

# if verbose, show what our settings are
vprint("Opening serial port %s" % device)
vprint("%d:%d%s%s:xonxoff=%d:rtcdtc=%d" % (baudrate, width,
 parity, stopbits, xon, rtc))
if endtime:
vprint("Program will end in %s seconds" % endstr)
if show_time:
vprint("Printing timing information for each line")
if basepat:
vprint("Matching pattern '%s' to set base time" % basepat)

# now actually open and configure the requested serial port
# specify a read timeout of 1 second
s = serial.Serial(device, baudrate, width, parity, stopbits, 1,
xon, rtc)
# read from the serial port until something stops the program
basetime = 0
linetime = 0
newline = 1
curline = ""
vprint("Use Control-C to stop...")
while(1):
try:
# read for up to 1 second
x = s.read()

# see if we're supposed to stop yet
if endtime and time.time()>endtime:
break

# if we didn't read anything, loop
if len(x)==0:
continue

# ignore carriage returns
if x=="\r":
continue

# set basetime, by default, to when first char
# is received
if not basetime:
basetime = time.time()

if show_time and newline:
linetime = time.time()
elapsed = linetime-basetime
sys.stdout.write("[% 12.6f] " % elapsed)
newline = 0
```

```
# FIXTHIS - should I buffer the output here??
sys.stdout.write(x)
curline += x

if x=="\n":
newline = 1
if basepat and re.match(basepat, curline):
basetime = linetime
curline = ""
sys.stdout.flush()
except:
break

s.close()

main()
```

Appendix E

Boot Example for XIP

E.1 Print with quiet

This is a print out of a XIP image with the filesystem AXFS. This print was performed with the option quiet passed to bootargs.

```
[    0.000002] ## Booting image at 00100000 ...
[    0.000820]    Image Name:   xip
[    0.001347]    Image Type:   ARM Linux Kernel Image (uncompressed)
[    0.008348]    Data Size:    1350520 Bytes =  1.3 MB
[    0.012152]    Load Address: 00100000
[    0.012762]    Entry Point:  00100040
[    0.016367]    XIP Kernel Image ... OK
[    0.020056]
[    0.020121] Starting kernel ...
[    0.021089]
[    0.359929] Kernel boot OK
[    0.360285] Mounting /proc
[    0.660019] Starting Shell
[    0.728030]
[    0.728091]
[    0.728124] BusyBox v1.2.2 (2007.04.27-09:40+0000) Built-in shell (ash)
[    0.732262] Enter 'help' for a list of built-in commands.
[    0.736149]
```

E.2 Print without quiet

Same as above except that this is with all prints. That is, quiet is not passed to bootargs.

```
[    0.000001] ## Booting image at 00100000 ...
[    0.000793]    Image Name:   xip
[    0.001292]    Image Type:   ARM Linux Kernel Image (uncompressed)
[    0.008123]    Data Size:    1350520 Bytes =  1.3 MB
[    0.011977]    Load Address: 00100000
[    0.012585]    Entry Point:  00100040
[    0.016219]    XIP Kernel Image ... OK
[    0.020019]
[    0.020067] Starting kernel ...
[    0.020534]
[    0.176003] Linux version 2.6.27.45-omap1 (student@MEG-882) (gcc
version 4.4.2 (GCC) ) #174 Wed May 26 14:47:33 CEST 2010
[    0.184158] CPU: ARM926EJ-S [41069263] revision 3 (ARMv5TEJ), cr=00053177
[    0.188645] Machine: TI-OSK
[    0.192025] Memory policy: ECC disabled, Data cache writeback
[    0.196179] OMAP162123 revision 2 handled as 16xx id: 0505940859952213
[    0.200528] SRAM: Mapped pa 0x20000000 to va 0xd8000000 size: 0x100000
[    0.204637] CPU0: D VIVT write-back cache
[    0.208311] CPU0: I cache: 16384 bytes, associativity 4, 32 byte lines,
128 sets
[    0.215978] CPU0: D cache: 8192 bytes, associativity 4, 32 byte lines,
64 sets
[    0.220511] Built 1 zonelists in Zone order, mobility grouping off.
Total pages: 2032
[    0.228104] Kernel command line: console=ttyS0,115200n8 noinitrd ro
root=/dev/null rootflags=physaddr=0x400000 rootfstype=axfs init=/bin/ainit
ip=off lpj=373760 mem=8M
[    0.240979] Clocks: ARM_SYSST: 0x1000 DPLL_CTL: 0x2833 ARM_CKCTL: 0x2000
[    0.245351] Clocking rate (xtal/DPLL1/MPU): 12.0/192.0/192.0 MHz
[    0.249594] Total of 128 interrupts in 4 interrupt banks
[    0.253680] OMAP GPIO hardware version 1.0
[    0.257310] PID hash table entries: 32 (order: 5, 128 bytes)
[    0.261446] Console: colour dummy device 80x30
[    0.265258] Dentry cache hash table entries: 1024 (order: 0, 4096 bytes)
[    0.269413] Inode-cache hash table entries: 1024 (order: 0, 4096 bytes)
[    0.276807] Memory: 8MB = 8MB total
[    0.277369] Memory: 7904KB available (1172K code, 161K data, 16K init)
[    0.284753] Calibrating delay loop (skipped) preset value.. 95.83
BogoMIPS (lpj=373760)
[    0.289440] Mount-cache hash table entries: 512
[    0.293287] CPU: Testing write buffer coherency: ok
[    0.297070] OMAP DMA hardware version 1
[    0.300721] DMA capabilities: 000c0000:00000000:01ff:003f:007f
[    0.304837] omap_dsp_init() done
[    0.305309] USB: hmc 16, usb2 alt 0 wires
[    0.308988] i2c_omap i2c_omap.1: bus 1 rev2.2 at 400 kHz
[    0.313034] tps65010: version 2 May 2005
[    0.316718] OMAP OCPI interconnect driver loaded
[    0.317567] NetWinder Floating Point Emulator V0.97 (double precision)
[    0.324823] msgmni has been set to 16
[    0.325402] io scheduler noop registered
[    0.329006] io scheduler anticipatory registered
[    0.332880] io scheduler deadline registered
[    0.336669] io scheduler cfq registered (default)
[    0.337542] omap_rng omap_rng: OMAP Random Number Generator ver. 20
[    0.344855] Serial: 8250/16550 driver2 ports, IRQ sharing disabled
[    0.348911] serial8250.0: ttyS0 at MMIO 0xfffb0000 (irq = 46) is a ST16654
[    0.353578] console [ttyS0] enabled
[    0.360672] i2c /dev entries driver
[    0.364842] mice: PS/2 mouse device common for all mice
[    0.373344] Disabling unused clock "usb_dc_ck"...  done
[    0.374348] FIXME: Clock "tc2_ck" seems unused
[    0.376926] Disabling unused clock "tc1_ck"...  done
[    0.381256] Skipping reset check for DSP domain clock "dsptim_ck"
[    0.385341] Skipping reset check for DSP domain clock "dspxor_ck"
[    0.392794] Skipping reset check for DSP domain clock "dspper_ck"
[    0.404679] VFS: Mounted root (axfs filesystem) readonly.
[    0.408721] Freeing init memory: 16K
[    0.409291] After freeing init mem.
[    0.412834] Before running init shell.
[    0.583985] Kernel boot OK
[    0.584356] Mounting /proc
[    0.643979] ATZ
[    0.883967] Starting Shell
[    0.951980]
[    0.952043]
[    0.952077] BusyBox v1.2.2 (2007.04.27-09:40+0000) Built-in shell (ash)
[    0.957006] Enter 'help' for a list of built-in commands.
[    0.960854]
```

Bibliography

[1] R.I. Davis, A. Burns, R.J. Brill, J.J. Lukkien:Controler Area Network Scheduability Analysis. Kluwer Academic Publishers (Januari,2004)

[2] R. Williams: Real-Time System Development. Butterworh Heinemann (2006)

[3] http://tools.ietf.org/html/rfc3010 (December, 2000)

[4] http://sourceware.org/jffs2/jffs2.pdf (2010)

[5] http://focus.ti.com/lit/an/spraat3/spraat3.pdf (February 2008)

[6] http://www.data-io.com/pdf/NAND/Toshiba/NandDesignGuide.pdf.pdf (April 2003)

[7] http://www.dataio.com/LiveImages/26/13/DocumentURL.pdf, (July 2003)

[8] http://www.arm.com/about/newsroom/19720.php, (22 januari 2008)

[9] S. Willians: Free as in Freedom. O'Reilly (2002)

[10] L. Walleij: Att använda Linux and GNU. Studentliteratur (2006)

[11] http://axfs.sourceforge.net/wordpress/, (1 June 2010)

[12] http://elinux.org/Grabserial, (2 June 2010)

[13] http://elinux.org/upload/7/78/ReducingStartupTime_v0.8.pdf, (2003)

[14] http://www.gnu.org/licenses/gpl.html, (29 June 2007)

[15] http://buildroot.uclibc.org/, (11 June 2010)