UPPSALA
UNIVERSITET

IT 14 049
Examensarbete 30 hp
September 2014

Implementation and Evaluation of Android on an ARM-based Vehicle Computer

Christoffer Klarin
Dennis Rosén

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Implementation and Evaluation of Android on an ARM-based Vehicle Computer

Christoffer Klarin and Dennis Rosén

The public transport system today has a growing number of information sources that need to be presented to the driver. Most of these information sources are displayed on separate screens - viewing and interacting with all of them at once is not beneficial from a safety and economical point of view. A solution to this problem would be to create a common platform where the driver can interact with all the information sources on a single display. This thesis aim has been to research suitable operating systems for this target platform through a survey, which has led to the implementation and evaluation of Android on the platform. The target hardware has limited resources such as low memory and no GPU. Therefore two versions of Android have been implemented and evaluated to find out which one is the most suitable for the system. The implementations of the two Android versions have support for an external touch screen. The evaluation of the final systems shows that the target hardware is not suitable to run Android with acceptable performance. Expansion of RAM and the integration of a GPU is suggested to improve the performance of the system.

Handledare: Anders Florén
Ämnesgranskare: Thiemo Voigt
Examinator: Philipp Rümmer
IT 14 049
Tryckt av: Reprocentralen ITC

Acknowledgements

We would like to offer our deepest thanks to our supervisor Anders Florén at Sylog for entrusting us to with this thesis. Without our discussions with Anders and his different proposals and solutions to the aim of this thesis we would have had a great deal more difficulty finalising this project.

Further earnest thanks to Thiemo Voigt, our reviewer, for helping us with how to proceed during the course of this project, and for giving insight and pointers on how to structure this report.

Lastly we would like to thank Sylog Sverige AB and Data Respons AB as a whole for giving us this excellent thesis opportunity, and for providing us with a locale and equipment to work with.

Contents

List of Tables

List of Figures

Acronyms

AOSP Android Open Source Project.

CPSW Common Platform Ethernet Switch.

DDR Double Data Rate.

FPS Frames Per Second.

GPL General Public License.

GPU Graphics Processing Unit.

HID Human Interface Device.

IDC Input Device Configuration.

LXDE Lightweight X11 Desktop Environment.

MCU Memory Control Unit.

MUSB Mentor Graphics Inventra USB.

OHCI Open Host Controller Interface.

OS Operating System.

OTG On-The-Go.

PSS Proportional Set Size.

RSS Resident Set Size.

SDHC Secure Digital High Capacity.

SDK Software Development Kit.

SoC System-on-Chip.

SPL Second Program Loader.

TI Texas Instruments.

UART Universal Asynchronous Receiver/Transmitter.

UHCI Universal Host Controller Interface.

1 Introduction

1.1 Motivation

The public transport system today has a growing number of information sources that need to be presented to the driver. This information could be views of cameras, the distance to the next station, the current timetable or position of the vehicle. Most of these information sources today presents their information on different screens, which is not beneficial from a safety and economical point of view. Instead it would be useful to create a common platform where all this information is presented on a single interface (figure 1). The customer of the public transport system wants the well-known Operating System (OS) Android to be implemented on the platform, due to its popularity as an operating system. This provides opportunities to recruit available developers to facilitate fast application development for expanding the system, and may contribute to increased sales of the finished product. However, due to the limited resources of the target platform, Android may not be entirely suitable to run with satisfactory performance. Therefore a survey has been conducted to find alternative operating systems to be considered for the target platform should Android not perform satisfactorily.

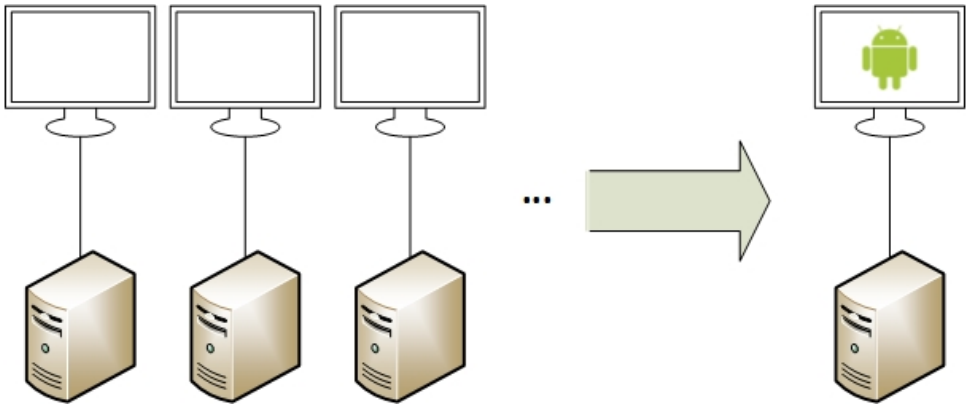

Figure 1: *Presenting information to the driver with a single interface is beneficial from a safety and economical point of view.*

1.1.1 The Target Hardware

The intended hardware for the implementation of Android is an ARM-based vehicle computer which is designed to operate in tough environments [1]. Currently the vendor of the hardware does not provide any Android support. The system runs on an ARM System-on-Chip (SoC) and has a low amount on RAM as well as a low frequency CPU, at least when compared to other Android based products on the market. The SoC does not contain a Graphics Processing Unit (GPU), which is recommended by the Android development team [2]. These limited resources are a drawback of the system and may lead to worse performance during resource-intensive tasks. It is therefore interesting to investigate if it is possible to make Android run with acceptable performance on the target hardware without compromising the functionality of the system.

1.1.2 Support from Vendors

The vendor of the target platform provides support for Linux [1]. This support is in the form of board-specific patches that can be applied to the Linux kernel. They also provide patches to U-boot, which is a bootloader that is able to load a Linux kernel [3].

The producer of the SoC that the target platform is based upon, Texas Instruments (TI), provides support for various Android versions [4]. They provide support from Android 2.3 up to Android 4.2 through various SDK's (Software Development Kits).

1.2 Problem Description

The intended target platform is an ARM-based vehicle computer. This hardware has limited resources which constrains the implementation of Android, such as low memory, low CPU frequency, and no GPU. Recent versions of Android has increased functionality but also increased memory requirements. According to the Android Compatibility Definition document newer versions of Android has a higher minimum memory requirement than what is available on the target platform [5]. However, later versions of Android are less likely to become obsolete in the near future. Because of this, developers of Android applications are more inclined to produce applications for these newer versions as opposed to older versions. Since the customer wants Android implemented on the system, it is therefore interesting because of the aforementioned reasons to implement and evaluate different versions of Android to be able to select the most suitable version to run on the target platform. This thesis aims to answer the following questions:

- **Other suitable operating systems for the target platform**
  Android is requested by the customer, but might not be the best choice for the target platform. This thesis looks into other alternatives that may be more suitable as OS for the target platform.
- **Performance impacts**
  Which performance impacts does newer versions of Android have running on the target platform? Can these versions even be run on such a low-resource system?
- **Bottlenecks**
  Identify and answer which bottlenecks that impacts the performance the most, be it the amount of RAM, the low-frequency CPU, or the lack of a GPU.

1.3 Aim

The first goal of the project is to make a survey in order to find the most suitable OS for the target platform. The second goal is to implement and evaluate two different versions of Android on the target platform, since the customer has specifically asked for Android to be implemented. Both implementations should have touch screen and

Ethernet capabilities. These two implementations should be benchmarked in order to find the most suitable candidate for the target platform.

1.4 Scope

The project is limited to evaluating which of a few select operating systems would be most suitable to implement on the target platform, with regards to performance and usability, as well as implementing and evaluating Android on the platform. The thesis does not include development of any applications that runs on top of Android. The only requested additional functionalities from the customer are touchsceen communication over USB and Ethernet functionality. Other interfaces such as WiFi or Bluetooth are not required. Due to the hardware requirements of Android, an older and a newer version have been implemented to evaluate which one is the most suitable to run on the target platform. Additionally this evaluation has been limited to performance and usability experimentation.

Since Android is based on the Linux kernel, some focus has been put into how to port it successfully. However, the topic of the Linux kernel and its many features is much too large to fit into the scope of this project. Only certain parts of the kernel that are normally required to be modified when porting the kernel to new hardware have been investigated.

1.5 Result

In this thesis we have analyzed different operating systems to be ported to the target platform. Out of these operating systems we have selected the most suitable one with regards to performance, usability, and time to market. Due to the customers request however the selected operating system to implement was Android, which we have implemented and evaluated two different versions of on the target platform. We have also researched the possibility of scaling down Android in order to improve its performance on resource constrained hardware. Our evaluation of the Android versions have been conducted using a variety of tools, which have mainly been selected due to already existing benchmarking conducted by these tools on similar hardware as the target platform. Finally, our evaluation shows the memory consumption and the CPU utilization of Android on the target platform and compares these results with other similar Android-based platforms. We conclude that the main resources acting as bottlenecks for the system are the lack of a GPU in both versions of Android and in the case of Android 4.2 the low memory as well.

1.6 Distribution of Work

Large parts of the practical parts in this project have been performed in a teamwork manner. Both members of the project have learned how to setup the various toolchains and SDKs, as well as how to properly setup and communicate with the hardware. However most of the work has been divided to create greater efficiency. Christoffer has looked into

the Android related parts in this project, such as the Android software stack and its input subsystem. Dennis has been responsible for the parts related to the Linux kernel and bootloader.

1.7 Thesis Disposition

Chapter 2 - A Survey of a Suitable Operating System for VTC 100
Presents previous research done in the topic of this thesis. Also contains a background study of different light-weight Linux distributions. Concludes with the choice of Android for the target platform.

Chapter 3 - Bootloaders in Embedded Systems
Describes the necessary functionality required by bootloaders to be considered for the target platform. Concludes with the choice of U-Boot as the bootloader of choice.

Chapter 4 - Android Internals
Describes the Android software stack and its key components. It also examines how applications are able to run within an Android system and what happens during initialization of an Android system.

Chapter 5 - Android External Device Support
Describes how to add external device support in Android and the Linux kernel.

Chapter 6 - The Target Platform
Describes the target hardware and its internal components. Compares the Android system requirements with the available hardware. Also presents a few similar platforms to the target platform and their specifications.

Chapter 7 - Implementation
Presents the tools and methods that have been used during the implementation. Also describes the support contributed from the hardware vendor of the target platform and from the Android SDKs used. Mainly this section describes how the two Android versions were implemented on the target platform, how U-Boot was customized, and how the Linux kernel was configured.

Chapter 8 - Evaluation
Presents the results achieved from the tests. Presents data for CPU utilization, RAM usage and boot time.

Chapter 9 - Conclusion
Discusses the results achieved from the experiments and provides answers for the questions detailed in the problem statement. Also presents possible future platform improvements.

Chapter 10 - Future Work
Gives recommendations and pointers about extensions to this project.

2 A Survey of a Suitable Operating System for VTC 100

As mentioned in section 1.1.2, the target platform comes with support for Linux. To keep the time to market short as requested by the customer, the OS of choice to be implemented should be Linux based. In particular, Android is requested by the customer, however it might not be the best choice for the target platform with regards to performance and usability. This chapter presents previous work related to Android in embedded systems. It also gives suggestions of different operating systems and out of these select the most suitable OS for the target platform.

2.1 Related Work

Previous research in this area has focused on the differences between Linux and Android systems and the suitability of Android as an operating system for vehicles. Frank Maker and Yu-Hsuan Chan examines in their paper A Survey on Android vs. Linux the differences between Android and standard Linux operating systems [6]. Tarik Al-Ani evaluates the usability of Android in vehicles in his paper Android In-Vehicle Infotainment System (AIVI) [7].

Additional research has been done regarding Android in embedded systems in the form of the thesis An Evaluation of the PandaBoard as a Set-Top-Box in an Android Environment, by the author Torbjorn Svangard [8].

Linux vs. Android
As the authors describes in [6], Android is an OS based on the Linux kernel that supports the x86 and ARM architecture. They detail the architecture of Android and its internal workings in a bottom-up fashion in their paper, however they do not conclude whether Linux or Android would be more suitable for embedded systems. Furthermore the paper does not examine any time to market aspects of porting Android, and does not contain any evaluation of the performance of Android systems. Thus we consider the title of the article to be misleading, since there are no comparisons between Android and different Linux distributions.

Android In Vehicles
In [7], the author presents an Android Infotainment System to be used as a standard platform in vehicles for the passengers. The author's thesis consisted of issuing a user survey to obtain the requirements of the system, implementing the Android-based system, and lastly conducting a user opinion survey to evaluate the system.

The paper contains a comparison to other Infotainment Vehicle Systems (IVI), such as MeeGo, Microsoft Auto, and Pioneer, where aspects like music and video playback file format support, GPS, and voice recognition were examined.

The Android version implemented in the Vehicle Infotainment paper is an earlier version

of Android with lower system requirements than the versions we have implemented in this thesis. Furthermore the author's version of Android does not have Ethernet support. The hardware the author ported his implementation to contains a GPU compatible and optimized for use with the standard OpenGL ES libraries used in Android. Disregarding the GPU the hardware is similar to the target platform used in this thesis.

The Vehicle Infotainment paper does not contain any direct performance evaluation of Android. However the paper contains user surveys where users have input their satisfaction upon testing the different services of the platform, such as video stream playback. The conclusion of the paper is that the surveys showed a generally positive outcome, which we interpret as the performance of the system was acceptable.

Android in PandaBoard
In [8], the author examines the performance of video-playback in Android version 2.2 running on the PandaBoard. The PandaBoard is a platform containing the OMAP4430 SoC, which comes with a dual-core Cortex A9 processor with each core running at 1GHz [9]. Additionally the PandaBoard contains the PowerVR SGX540 GPU and an Image Video Accelerator High Definition subsystem. The goal of the authors thesis was to add hardware accelerated video playback to an already existing Android port for the PandaBoard. This goal was not reached and the author instead used OpenGL ES 2.0 shaders and evaluated the performance of the board. The author evaluated the performance of the system by measuring the Frames Per Second (FPS) achieved when streaming video playback from two different external channels. One of the channels streamed a Standard Definition video and the system output 70 FPS, more than sufficient for the playback. However when streaming from the other channel a High definition video was used and the output was 14.8 FPS.

Compared to our project the hardware used in [8] has greater resources in the form of a dual-core processor with higher CPU frequency and a GPU. The Android version used is also older and less resource intensive than the implemented versions in our project.

2.2 Operating System Requirements

The selection of the operating systems to be considered is based upon several requirements stated by the customer and the limitations that comes with the target platform. The following requirements are considered in the choice of OS:

Short Time to Market
A request from the customer is to keep the time to market short. Since the target hardware comes with Linux support the operating systems to be considered should be based upon Linux.

Hardware Requirements
Two important parameters of the target hardware is the lack of a GPU and the small amount of RAM available. These limitations puts demands on the OSes to be considered.

They should not be graphically intensive and should require small amounts of RAM.

Open source and support for the ARM-architecture
The target platform intended for this project is based on the ARM-architecture. Any OS to be considered should therefore support this architecture. Another requirement in the selection of an OS is that it needs to be open source in order to customize it for the target platform.

Ethernet Support
The final system is supposed to be able to communicate with other sources on the vehicle to receive necessary information through Ethernet. The selected operating systems should therefore have support for Ethernet.

Touch Interface
Since the customer requires the system to handle touch events, any already existing touchscreen support reduces the time to market of the OS, making it another aspect to be considered.

2.3 Linux

The main operating system candidate to port to the target platform is the Linux OS, also known as the Linux kernel, and a few of its many distributions. This is due to several factors: the open source nature of Linux, its support for the ARM architecture, its abundance of device drivers, and the fact that the hardware vendor of the target platform provides patches for the Linux kernel to facilitate porting the OS to the target platform (see section 1.1.2). Linux was created in 1991 by the Finnish computer scientist Linus Torvalds [10]. It is responsible for communicating with hardware devices and allows processes to communicate with eachother using inter-process communication. The Linux kernel is programmed in the C programming language, and supports all major computer architectures such as ARM. Since its creation, Linux with its open source nature has expanded from contributions made from thousands of independent developers. Running under the General Public License (GPL) license and leveraging a vast amount of free to use system and application software, projects using Linux can decrease licensing costs and development time substantially. It is now the leading operating system of choice for embedded developers, thanks to the aforementioned features.

2.4 Lightweight Linux Distributions

Since the target hardware is severely resource constrained, any distributions of Linux to be considered for porting must be light-weight, meaning that they require low amounts of resources. Today there exists a significant amount of Linux distributions described as light-weight. However, "light-weight" is subjective and different light-weight Linux distributions require varying amounts of memory and CPU processing power. At the low end of the spectrum is for example BasicLinux, a distribution optimized for old PCs and laptops [11]. Zipped the DOS version of BasicLinux has a 2.8 MB footprint, and it

requires 3 MB of RAM and a 386 CPU. On the other end is CrunchBang Linux, a Debian based distribution that comes with several built-in applications [12]. On Crunchbang the footprint is about 775 MB, with a minimum required RAM of 128 MB. However many of these distributions do not support the ARM architecture. Therefore studies of two of the more popular light-weight distributions of Linux according to statistics from the website distrowatch.com, namely LUbuntu and Arch Linux ARM, that support the ARM architecture have been conducted.

Most Linux distributions are designed for use in general purpose computers, such as desktops and laptops. Since ethernet is widely used in such systems, all Linux distributions discussed here supports ethernet.

2.4.1 LUbuntu

LUbuntu is a variant of Ubuntu described as "fast, lightweight and energy-saving" [13]. The main difference to Ubuntu is that LUbuntu uses the Lightweight X11 Desktop Environment (LXDE) as its desktop environment instead of Unity, and comes with a selection of light applications. While testing has shown that LUbuntu works on an extremely constrained system with 64 MB of RAM running a pentium 2 CPU, the minimum required RAM for a usable LUbuntu system is 128 MB [14]. The later versions of LUbuntu supports a limited amount of ARM SoC platforms, such as LUbuntu 14.04 LTS that includes the 3.13 version Linux kernel. However unofficial versions of LUbuntu can be found that supports the AM335x SoC. The desktop environment in LUbuntu is as mentioned a light-weight version of X11, which comes with some touchscreen support [15].

2.4.2 Arch Linux ARM

Designed for the ARM architecture, Arch Linux ARM is a port of Arch Linux, another Linux distribution [16]. Arch Linux is described as "a lightweight and flexible Linux distribution that tries to Keep It Simple" [17]. The ARM port is optimized to run on low-end ARM architectures such as the ARMv5 as well as higher-end architectures such as ARMv7. There are existing ports of Arch Linux ARM for platforms similar to the VTC 100, like the BeagleBone (see section 6.5.1). The port for the BeagleBone includes Linux kernel version 3.14. Arch Linux uses the X11 desktop environment which comes with some touchscreen support [15].

2.5 Android

Since the customer has requested Android to be ported to the target platform it is the main candidate OS to be considered. Android is a Linux-based OS that was released in 2008 (Android 1.0). It is today the most popular operating system for smartphones on the market [18] and is also starting to become popular on other platforms [19]. The success of Android is partly because of its easy to use touch interface, and that its applications are written in Java, a language many developers are familiar with. After each release of a new Android version, Google makes the Android Open Source Project

(AOSP) available[1] for anyone who wants to access its source code. The combination of open source code and readily accessible development tools has led to a large amount of developers for Android. Android also comes with additional support for features such as GPS functionality using Google maps which may be useful for the intended uses of the final system to be implemented.

2.5.1 System Requirements

Google releases new versions of Android every sixth months [19]. As more features are introduced in the later Android versions, they also put higher demands on the hardware. Table 2 presents the minimum hardware requirements for the various Android versions.

Table 1: Minimum hardware recommendations for various Android versions.

Version	**RAM**	**GPU**
Android 2.1	92 MB	Recommended
Android 2.2	128 MB	Recommended
Android 2.3	256 MB	Recommended
Android 4.0	340 MB	Recommended
Android 4.1	340 MB	Recommended
Android 4.2	340 MB	Recommended
Android 4.3	340 MB	Recommended
Android 4.4	340 MB	Recommended

Note that all versions strongly recommend a hardware GPU. If there is no GPU present, the AOSP provides a software implementation of OpenGL (OpenGL ES 1.0) [2]. All versions of Android utilizes the GPU if there is one present. In older versions of Android, such as Android 2.3, the GPU is used when resource intensive events are fired that redraw the whole display. These are for instance window transition events such as menu "swiping". In newer versions of Android the GPU is available for any application that wishes to make use of it [20].

[1]Except from Android Honeycomb

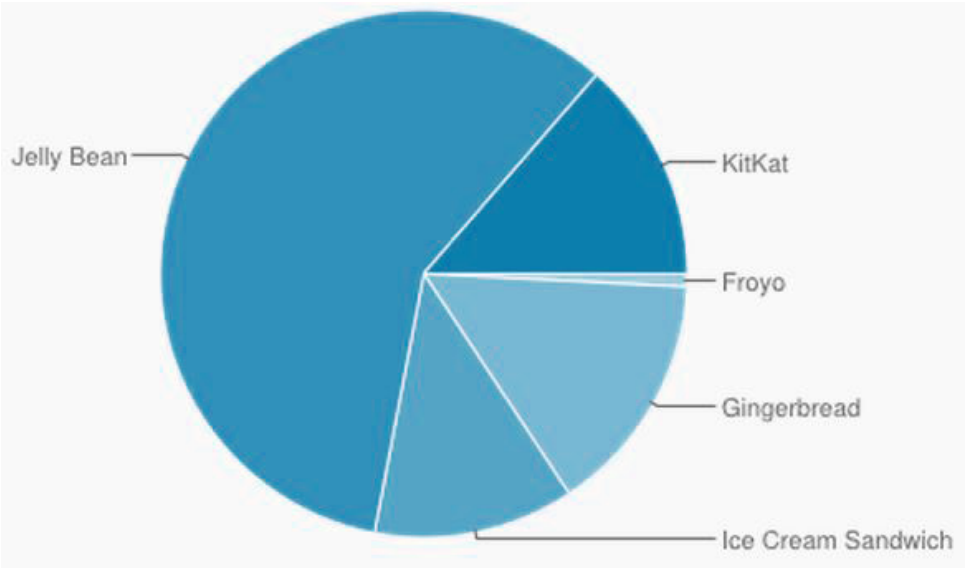

Figure 2: *Android version distribution.*

As discussed in section 1.1.2, Texas Instruments (TI), the vendor for the SoC in the target platform, provides support only for a limited number of Android versions. Therefore preferable Android versions to be considered are the ones supported by TI. The customer also wants a recent version of Android to be able to more easily find developers for the final system, and to make sure the selected version does not become obsolete in the near future. Figure 2 depicts the distributions of the various Android versions [21]. Most popular is Jelly Bean (Android 4.1.x - Android 4.3), followed by Gingerbread (Android 2.3.3 - Android 2.3.7) and KitKat (Android 4.4). This shows that later versions tends to have a greater user-base than the older versions. However, as mentioned above, newer versions comes with higher resource requirements. These resources include processing power, amount of RAM and battery power.

2.5.2 Ethernet

Ethernet is supported by the drivers in the Linux kernel. Android consists of a number of layers on top of a Linux kernel. In the top layer exists the Android applications. Between the top (Android applications) and the lowest (the Linux kernel) layers are the Java Framework services (see section 4.6) that applications utilizes to communicate with the Linux kernel. Android does not support Ethernet and is therefore missing services that handles Ethernet communication. Since Android does not support Ethernet, there is no documentation in this area. However there exists unofficial patches for Ethernet on the internet, although these are available only for a few Android versions [19, 22].

2.6 Conclusions on Suitable OS for VTC 100

The most important criterion in our selection of an OS is that porting it should hold a short time to market. All of the above OS / Linux distributions are based on the Linux kernel, which facilitates the implementation. LUbuntu has limited ARM architecture support, however there exists unofficial versions of LUbuntu which may ease porting to AM335x SoC based boards. Arch Linux comes with direct support for boards similar to the VTC 100 reducing the amount of work needed for a port. However both versions of the two operating systems run on newer Linux kernel versions than the version supported by Nexcom, the manufacturer of the target platform. This adds difficulty to porting.

Android as presented above is an operating system that is based on the Linux kernel. Texas Instruments, which is the manufacturer of the SoC that VTC 100 is based on, provides support for a number of Android versions for several platforms that are based on the this SoC. These Android versions are based on different versions of the Linux kernel, which impacts the amount of work required to port Android to the target platform.

Both LUbuntu and Arch Linux ARM are lightweight Linux distributions and have been proven to run successfully on older systems, or on systems with low resources. Android is available in several versions where each version has different hardware requirements. The later versions have more features and a more developed user interface, but also comes with higher demands on the hardware compared to the older versions. Of the Android versions supported by TI, Android 2.3 is the latest version that does not require more RAM than the amount available on the target platform. The latest Android version that TI offers support for is Android 4.2. Android 4.2 comes with more features than Android 2.3, but its hardware requirements are beyond what the target platform has to offer.

Other requirements for the OS are support for Ethernet and touchscreen over USB. These are requirements that both LUbuntu and Arch Linux ARM fulfill. Android however, lacks Ethernet support. There exists unofficial patches for some Android versions that are undocumented. This lack of support impacts the time to market criterion for porting Android. Android provides support for internal touchscreens, and some support for external touchscreens over USB.

With the above in mind, we conclude that the best choice of OS for the target platform is one of the two featured Linux distributions. There already exists ports of these systems to platforms similar to the target platform. They also support Ethernet and touchscreen over USB. The manufacturer of the target platform and the manufacturer of the components contained in it offers support for both Linux and Android.

The lack of any existing support of Ethernet in Android affects the time to market for a port negatively. However, the customer believes that products running Android are easier to sell compared to other systems. They therefore prefer an implementation of Android to be created for the target platform. For the same reason, they would like a newer version of Android to be implemented. Because of the associated increased hardware requirements of newer versions of Android, we conclude that the implementation of two

ports of two different versions of Android on the target platform has been the most suitable course of action. These implementations have been evaluated against each other regarding performance. One of the versions are an older version that fits within the hardware requirements of the target platform, and the other a newer version with more market appeal. Because of the aforementioned limited support from TI for their SoC, the versions we have selected are Android 2.3 Gingerbread and Android 4.2 Jelly Bean. The target platform meets the system requirements of Android 2.3, with the exception of it containing no GPU. However, the platform does not meet the minimum required RAM for Android 4.2. Newer versions, such as Android 4.4 KitKat, have not been chosen due to no support provided by TI.

3 Bootloaders in Android-based Systems

As concluded in section 2.6, Android was chosen as the OS ported to the target platform. In order to boot up an Android-based system a mechanism is required to perform machine specific initialization. This initialization is done by a bootloader [10]. This chapter presents suggestions of different bootloaders and from these select the most suitable bootloader according to the listed requirements below.

3.1 Bootloader Requirements

The selection of a bootloader is based upon several requirements that are stated by the customer and the limitations that follows with the target platform and the selected OS. Since bootloaders are such small parts of an embedded system, they typically only require a few seconds to initialize. The performance of a bootloader is insignificant compared to that of the Linux kernel or the Android layers. Therefore only the following requirements have been considered in the choice of a bootloader:

Short time to market
As stated in the requirements for selection of an OS, the customer wants to keep the time to market short. This means that the amount of existing support for any bootloader to be considered is important.

Able to load the Linux kernel
Since Android, which is based upon the Linux kernel, has been chosen as the OS for the target platform, another requirement is that any bootloader to be considered should be able to load the Linux kernel.

Open source and support for the ARM-architecture
The bootloader is also required to support the architecture of the target platform, which is the ARM-architecture. Another requirement important in the selection of a bootloader is that it needs to be open source in order to customize it for the target platform.

3.2 U-Boot

The target hardware comes with support for the bootloader U-boot [10]. This substantially reduces the time to market should U-Boot be selected as the bootloader. U-Boot has become a popular bootloader for Linux-based systems partly because of its open source nature and support for many different architectures and processors. Another fact that may have contributed to the success of U-Boot is the wide support for many standard development boards. With a large set of supported boards there is a high chance that custom boards have somewhat similar hardware to any of these boards. This opens up the opportunity for re-usage of a large portion of code during the process of porting U-Boot.

3.3 Conclusions on a Suitable Bootloader for VTC 100

Bootloaders are a small but important part of an embedded system. The main requirements for the bootloader in a platform running Android is that it should have a short implementation time and be able to load the Linux kernel. Since the target hardware comes with support for the open source bootloader U-boot, which is able to load the Linux kernel, we select it as the top and only candidate for the target system.

Any other bootloader would require a full port to the hardware. Each pin of the SoC that the target is based upon has up to eight different functionalities. The only way to obtain the exact hardware setup of these pins is through access to the circuitry schematics of the hardware, which is unavailable to us. With this in mind, we select U-Boot as the bootloader for the target platform.

4 Android Internals

Android is becoming more popular as an OS on embedded platforms. This is partly because of its easy to use touch interface and that its applications are written in Java, which has contributed to a large amount of developers for the platform. Android is a software stack that is built up by several layers. The lowest layer is comprised by a Linux kernel which has been customized for embedded platforms. The Android software stack is also built up by middle layers that helps the Android applications to utilize low level functionality.

The target platform where Android is to be implemented contains a limited amount of RAM. In order to achieve satisfactory performance with later versions of Android a system scale-down may be necessary. As concluded in section 2.6, Android 4.2 requires 340 MB of RAM while there is only 256 MB of RAM available in the target platform. This chapter looks into the internals of Android in order to reveal which parts, if any, that can be removed in order to reduce the requirements on system resources. The theory presented in this chapter is mainly targeted for the Android versions implemented in this project, which is Android 2.3 and Android 4.2.

4.1 Fundamental Android Concepts

Before delving deep into the internals of Android it is important to understand a few fundamental concepts in Android. This includes understanding the different building blocks of an Android application, how applications communicate with each other and how applications run in an Android based system.

Every Android application runs as separate processes within instances of a virtual machine, namely the *Dalvik Virtual Machine* (DVM). Android applications are build up by *components*, which are able to communicate with other components within the same application or outside of it [19].

What an Android device user sees at any moment on his screen comes from an *Activity* component. This component is normally shown in full-screen. The Activity component is responsible for creating a window where the *user interface* (UI) can be implemented.

An application component that does not provide any UI is the *service* component. This component has instead the ability to perform long-running operations in the background, such as music playback and network transactions. Other components can bind to a service to interact with it, and by utilizing Androids own *RPC/IPC* (Remote Procedure Call/Inter-Process Communication) mechanism it is possible for components to interact with services located in other applications.

There exists more application components than presented in this section. However, the presented ones should be enough to be able to follow the rest of this section.

4.2 Architecture

Android is built up by several layers that together form a software stack based on a Linux kernel. Many of the components in the Android software stack have been designed to run on resource constrained platforms. Android runs a modified version of the Linux kernel, its own VM and its own implementation of the libc library. All these parts are customized to perform better on resource constrained platforms [19].

Figure 4 depicts the different layers in the Android software stack. This section examines the different components that resides in the layers below the Android API libraries (`android.*`) that many Android application developers are familiar with. It presents how they are customized for embedded platforms and how they interact with each other. It starts with a description of the key modifications of the Linux kernel and goes up the through Android software stack.

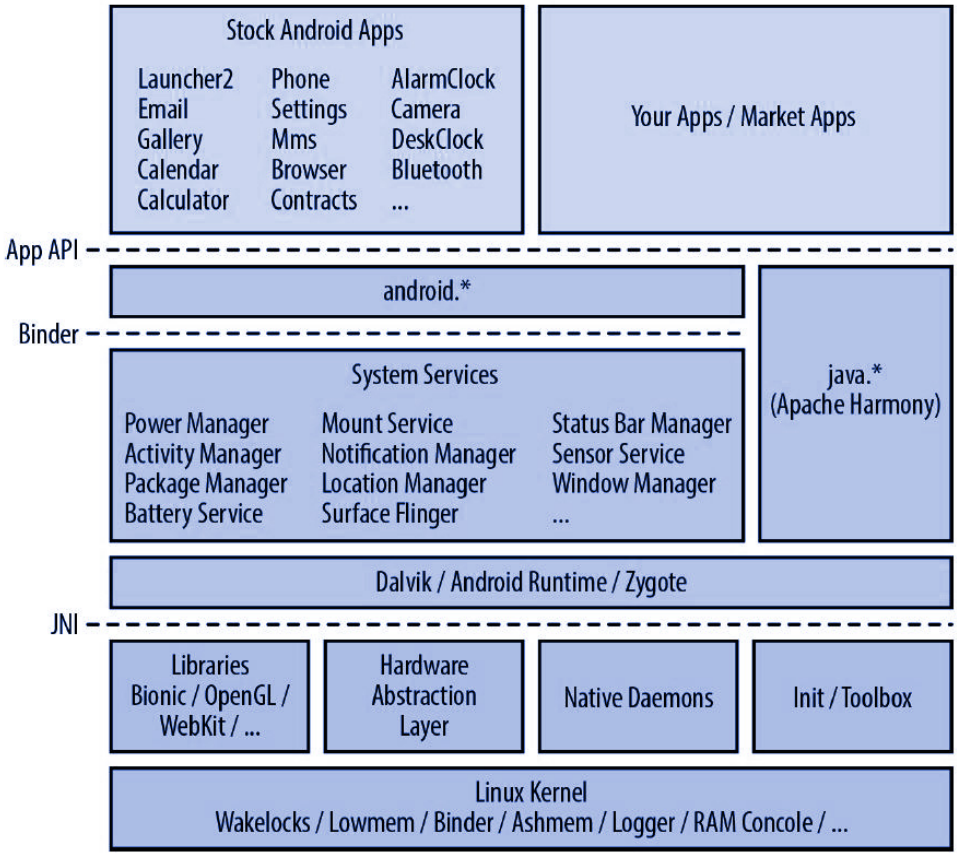

Figure 3: *The Android software stack.*

4.3 Modifications to the Linux Kernel

At the very bottom of the Android software stack is a Linux kernel. The Linux kernel provides functionality such as memory and process management, a permission based security model and a proven driver model that is already integrated in the core that is

necessary for the functionality of Android. To make Android more suitable for small hand held devices modifications to the Linux kernel have been introduced. Small hand held devices usually have limited resources such as limited processing power and memory, and are normally battery driven. The additional modifications that are introduced in the Linux kernel concerns these limitations and improves the performance for these types of devices, [19]. Some of the most significant modifications are presented in the next coming sections.

4.3.1 Power Management

Since Android is mainly targeted for handheld devices, which typically lacks effective battery packages, Android has been supplied with an aggressive power manager. The power manager turns off the screen and puts the CPU into a sleep power state shortly after use. Compare this to a normal Linux system on a desktop or laptop computer where the system would go to sleep when the user tells the system to sleep or when the lid of the laptop is closed. To prevent Android devices going into power saving mode during critical tasks, Google has introduced *wakelocks* in the Android API library. When an application wants to prevent the system from going into sleep, it can grab a wakelock. The device will not go into sleep as long as there is an active wakelock in the system [23].

4.3.2 Memory Management and Inter Process Communication

Another modification that can be found in an Android kernel is the Low-Memory Killer. The Low-Memory Killer is based on the Out Of Memory (OOM) killer that exists in the Linux kernel. The OOM killer in the Linux kernel starts to kill processes when the system runs out of memory. The Low-Memory Killer in Android prevents the system falling into this situation by cleaning up the memory at an earlier stage. The OOM killer in the Linux kernel has an OOM adjustment mechanism that is used to set OOM kill priorities for processes. The priorities indicates in which order the processes should be killed if a system runs out of memory. The Low-Memory Killer utilizes this mechanism by assigning OOM kill priorities to processes according to which components they are running. In addition to this, different "low memory" threshold configurations exist to make the low-memory killer take action before the OOM killer [19].

Each application in Android runs in a separate process. In order to make it possible for components in different applications to interact with each other the *Binder* mechanism has been introduced into the Linux kernel. Binder is an implementation of an RPC/IPC mechanism that provides the benefit of remote services being able to be treated as objects. The only things required to invoke a method of a remote object is its interface definition and a reference to it.

Another IPC mechanism that can be found in the modified Linux kernel is *Anonymous Shared Memory* (ashmem). It implements the IPC mechanism by shared memory. This feature is already implemented in the Linux kernel by POSIX SHM, but the development team of Android has decided to replace this functionality with their own. This is because

they consider the original implementation could cause resource leakage within the kernel, opening doors for malicious software. The ashmem implementation has been designed to be used for resource limited embedded systems. ashmem has many similarities with POSIX SHM, though ashmem has the capability to shrink mapped regions if the system is in need of memory.

4.4 Native User-space

The *native user-space* is located on top of the Linux kernel and includes all components in the Android user-space that runs outside the DVM. These components are comprised of a couple of libraries, a toolbox, and daemons. Also parts of the hardware support in Android are implemented here. These components provides a base for the higher layers in the Android software stack. The following sections describe what exists in this layer and how it can be utilized by the other components in the system.

4.4.1 Native libraries

In Android there exists native libraries on top of the Linux kernel. These libraries are written in C and C++ and provides low-level functionality and computationally intensive services to the platform.

One of the libraries found here is *Bionic*. Bionic is Googles own implementation of *libc* in Android. Like with many other components in Android, this library has been customized for resource restricted systems. This has been done by scaling down the library, thereby reducing its footprint [19].

4.4.2 Native Daemons

Native daemons are started during the boot of an Android system. These daemons run from boot until the device is powered off. Many of them provide functionality that is necessary for the system to work properly. Examples of functionality provided by the native daemons are the handling of mounting volumes and images, installing and uninstalling *.apk* files, and management of Bluetooth devices [19]. The following is a presentation of some of the key native daemons in Android.

Zygote
Every time a new application is launched in Android, a new instance of DVM is created. To keep loading times of applications short it is important to handle this process in an effective way. *Zygote* is a system service that is responsible for this task. Zygote provides fast application start up through pre-loading all Java classes and resources that may be needed by an application into RAM. When an application sends a request to Zygote to be launched, Zygote will clone itself and launch the application. Since this clone is a new instance of the DVM, which is preloaded with all necessary Java classes and resources that will be needed by an application, it will reduce the loading time of the new application [19].

System server
The System server is one of the key Native daemons that runs in the system. It is responsible for running a number of System services. Services that runs within the System server are different from the services that runs within an Android application. The services that runs within the System server lives until system shut down, unlike services within applications that are life-cycle managed. Another difference is that services that are hosted by the System server runs with system privileges, while service components within an application runs with the same privileges as the hosting application.

The System services that runs within the System server communicates with the lower layers in Android through the *Java Native Interface* (JNI), that acts as a "bridge" between the Java written services and the lower parts of the Android stack that are written in C/C++. Examples of System services that are hosted by the System server are the *Activity manager*, the *Package manager* and the *Power manager* [19].

Service manager
Remote object communication in Android is based on the Binder mechanism. Before an application can utilize a System service it requires a handle to it [19]. The *Service manager* keeps track of all System services that runs in the system. When an application wants to utilize a System service, it sends a request to the Service manager for a handle to the service. Due to its importance the Service manager is the first process that is started by the init process. Every System service instantiated by the System server is registered with the Service Manager.

4.5 Dalvik Virtual Machine

Applications that runs in Android are normally written in Java. Code that is written in Java is compiled by a Java compiler into `.class` files which are normally executed by a *Java Virtual Machine* (JVM). Since Android is targeted to run on resource limited embedded systems this has affected the design of the system. To reduce the usage of resources for Android based systems, Google has introduced a new type of virtual machine that replaces the JVM, namely the *Dalvik Virtual Machine* (DVM), [19].

Every application in an Android system runs in its own Linux process. Each of these processes is an instance of the DVM. The fact that every time a new application is started a new instance of the DVM is created, efficient creation of new instances and the memory usage of these are especially important in Android. As described in section 4.4.2, Zygote provides help to speed up the start up of new applications.

As mentioned earlier, Android applications are normally written in Java and compiled by a Java compiler into `.class` files. Since DVM is supposed to run on embedded devices with limited memory resources, it introduces a new type of file format. Instead of `.jar` files[2] which is used by the JVM, the DVM uses a file format called Dalvik Executable (`.dex`), which is about half the size of a `.jar` file. These `.dex` files are generated out

[2]A collection of `.class` files

from `.class` files by a tool named *dx*. In addition to this, Android replaces the standard Java libraries with *Apache Harmony*, which is an open source re-implementation of the Java libraries. Hence, Android avoids any licensing issues (for the moment) with Oracle.

Another feature that Dalvik has been provided with is a *Just-In-Time* (JIT) compiler. With a JIT compiler, the DVM can read applications bytecode in many sections and compile them into the host CPU's native instruction set. The output from this compilation is stored for later use, which contributes to faster application loading times and performance. JIT support in DVM is only available for a few architectures, where the ARM architecture is one of those who are supported.

4.6 Android System Services

The *System services* in Android are what enables high-level applications to communicate with the lower layers in the Android software stack [19]. All System services are built upon the Binder mechanism and provides the base for applications in Android. The System services are located just above the native libraries, which acts as a support environment for these services. The Android System services are provided by three processes, the *System Server*, the *Media Service* and the *Phone Application* (figure 4). These processes provide services that lives from boot until system shut-down.

The System Server provides a number of System services within the same process. It contains both Java-built and C-built services, where the Java-built services interacts with the lower layers through JNI. The System Server starts services such as the *Power Manager*, the *Activity Manager* and the *Package Manager*. The initialization of these services and all other System services that are started by the System Server are hard-coded into the System Server (`SystemServer.java`). In order to disable any of these services, specific parts of this file has to be removed or commented out. In the System server there also exists C-built services. One of these services is the *Surface Flinger*. This service plays an important role in an Android system since it is responsible for compositing the drawing surfaces into complete images.

The Media Service provides media related services to the system, such as the *Audio Flinger*, the *Media Player Service* and the *Camera Service*. The Media Service and its provided System services are launched from the `init.rc` script and is non configurable. The whole system will hang during boot up if any of the belonging System services are removed from the init script.

The third process that provides a System service is the phone application which is slightly different from the other two processes, since it provides its service through an application. This method is not optimal in the way that applications in an Android system are life-cycle managed and can therefore cause the System service to stop if the application is killed. As mentioned earlier, System services are supposed to run as long as the device is powered on. To fulfil this requirement of lifetime management the phone service application has a property set in the manifest file that makes the application not life-cycle

managed.

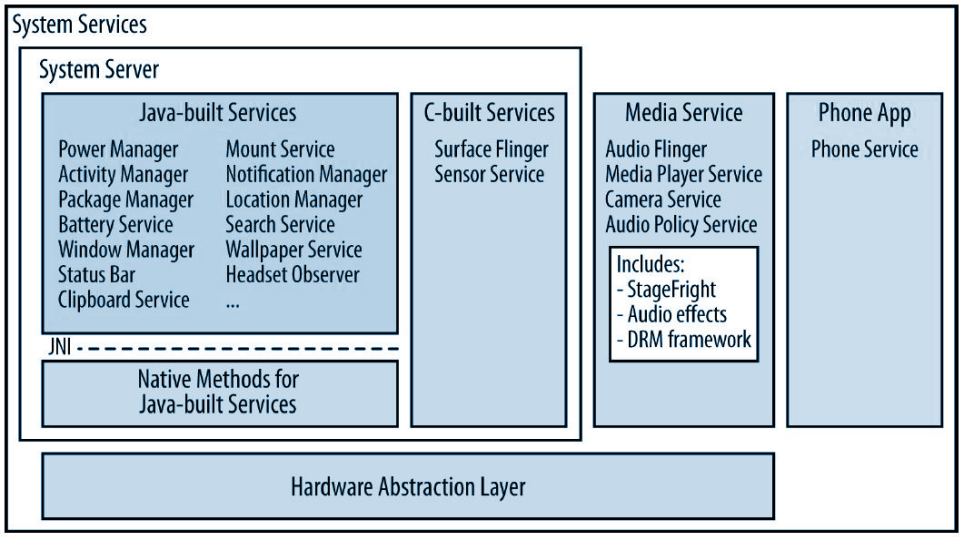

Figure 4: *Systems services in Android runs within native daemons.*

Another System service that can be found within the System server is the Activity manager. The Activity Manager is one of the most important System services in Android and is responsible for, among other tasks, starting up new application components, maintenance of OOM adjustments and task management.

4.7 Memory and Process Management in Android

Android is targeted to run on embedded systems which may have limited resources. One of these limitations may be the amount of RAM available in the system. Android has therefore been provided with a run time that handles the application memory and process lifetimes. As a user of an Android system it is not necessary to keep track of how many applications that have been started to save RAM.

Every application started in an Android system is launched in its own instance of a DVM. As the user launches more applications, more RAM will be consumed, and eventually the system will run out of RAM. When this occurs, the system will start to reclaim resources according to a priority-based schedule. Each process running in the system is placed into an "importance hierarchy". The placement into the hierarchy is determined by the status of the process' components. The lower priority a process has, the greater candidate it is for being killed [24].

The importance hierarchy divides the processes in the system into five categories: foreground process, visible process, service process, background process and empty process (figure 5). The processes at the highest level of the importance hierarchy are the foreground processes. The process that the user currently is interacting with and other processes that have components that contributes to this process are located at this level. At the second level are the visible processes, which means processes that affects what is

displayed on the screen, but does not have any foreground components. This happens when a dialogue box pops up on top of an activity, where it still is possible to see the previous activity in the background.

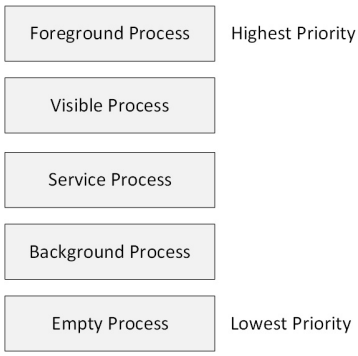

Figure 5: *Android process priority hierarchy.*

The next two levels are service processes and background processes. Basically, processes that are running a service falls into the service category. Processes that would be located at the background levels are processes that are holding an activity that is not currently visible to the user. Processes at this low level are good candidates for being killed. They are killed according to which process was used furthest ago. The lowest level in the process hierarchy are the empty processes level that holds processes that does not have any active components.

4.8 Conclusion on System Scale down Possibilities

As concluded in section 2.6, Android 4.2 is one of the Android versions that have been implemented on the target platform. Since the target platform has less available RAM than what is recommended for systems to run Android 4.2 we have scaled down the OS in order to run with acceptable performance. This has been done by removing RAM-intensive parts of the OS that are unnecessary for the final system[3]. Other candidates for being removed from Android are the processes that runs during the whole system lifetime. As presented in section 4.6 and 4.4.2, this includes all the System services and the native daemons.

The first candidates, the System services, have internal dependencies among each other. This means that if one System service is disabled, the whole system may lock up [19]. All the System services are also expected by the Android API to exist. An application will not work on an Android system if certain System services that the application relies on are disabled. The second candidates, the Native deamons, handles critical system functionality and are therefore necessary for the system to work properly.

There exists examples where the Android OS has successfully been scaled down by removing System services and Native daemons. One example is presented in [25], where the

[3]Such as support for Bluetooth.

author describes how he successfully removed several System services in order to create a headless[4] Android system.

Since it is not known which applications will run on top of the system, only guess-work can decide which System services might be unwanted and therefore taken into consideration of being disabled. If services are removed the system may not be able to execute the intended applications for the system. We have therefore decided not to remove any System services or Native daemons.

Another technique that could be used in order to save memory is to introduce virtual memory in Android. However, as described in section 4.7, Android has its own process management mechanism that starts to kill applications if the system runs out of memory. Introducing virtual memory would therefore most likely not contribute to any performance improvements of the system.

[4]A system that operates without a monitor.

5 Android External Device Support

When connecting a peripheral device to a platform running Android there may or may not exist support for the device in the system. If not this support will have to be created by the developers implementing the port. This is especially true for external touchscreen devices, where support is low since most Android devices on the market comes with their own internal touchscreens. This chapter goes through the lower level Linux input subsystem which Android relies upon, as well as the higher Android layers that handles and translates the events received from the kernel, all of which are necessary systems to understand to be able to perform a port of Android with functioning input.

5.1 The Linux Input Subsystem

While Linux has support for a wide variety of peripheral input devices, there are still those on the market with no support. It is therefore important to be familiar with the Linux input subsystem and how it works. In Linux the input event subsystem is divided into several parts [26]:

- **Event Handler Interface** – Module layer that passes events from the kernel to an application. Here the generic input event interface `evdev` is the most important, since it handles most device types. When debugging input events it is useful to read any events found in these files.
- **Input Module** – The main part of the Linux input system, the input module serves as communication between the event handlers and the device drivers. Typically when performing a port this layer should not need to be touched.
- **Drivers** – These are the modules that communicate with input devices and generate events for the input module. Examples are the USB Core driver, USB Controller drivers, and USB Device drivers, all detailed below. This is where an embedded developer might want to modify or create new files.

Input events in Linux contain a time stamp, the type of event, an additional event code, and a value. For a mouse device the type can be for example a button event, the code which button, and the value indicating a press or release (figure 6).

```
christoffer@christoffer-exjobb:/dev/input$ sudo evtest event13
Input driver version is 1.0.0
Input device ID: bus 0x11 vendor 0x2 product 0x1 version 0x0
Input device name: "PS/2 Generic Mouse"
Supported events:
  Event type 0 (Sync)
  Event type 1 (Key)
    Event code 272 (LeftBtn)
    Event code 273 (RightBtn)
    Event code 274 (MiddleBtn)
  Event type 2 (Relative)
    Event code 0 (X)
    Event code 1 (Y)
Testing ... (interrupt to exit)
Event: time 1401028553.644940, type 2 (Relative), code 0 (X), value -1
Event: time 1401028553.644950, type 2 (Relative), code 1 (Y), value -1
Event: time 1401028553.644951, -------------- Report Sync ------------
Event: time 1401028553.682510, type 2 (Relative), code 1 (Y), value -1
Event: time 1401028553.682526, -------------- Report Sync ------------
Event: time 1401028553.694428, type 2 (Relative), code 0 (X), value 1
Event: time 1401028553.694438, -------------- Report Sync ------------
Event: time 1401028553.755678, type 2 (Relative), code 0 (X), value 1
Event: time 1401028553.755689, -------------- Report Sync ------------
Event: time 1401028553.791277, type 2 (Relative), code 0 (X), value 2
Event: time 1401028553.791293, type 2 (Relative), code 1 (Y), value 3
```

Figure 6: *Example of input events using evtest in Ubuntu 10.04.*

5.2 General USB

To properly understand the USB subsystem in Linux, some basic knowledge of USB communication is necessary. A USB device communicates with a USB driver through the use of a *device descriptor* and additional data [27]. A device descriptor represents the entire device with information such as maximum packet size and its vendor and product ids. These are ids uniquely identifying the device, allowing the USB subsystem to correctly load which driver that should handle the device. A USB device can have several *configuration* descriptors that represent the state of the device, for example if it is in active or standby mode. For most devices one configuration descriptor is enough. A configuration has one or several *interface* descriptors that each represent a logical function of the device. For instance a loud speaker can have one interface for the speaker and one for the volume buttons. Finally at the bottom of USB communication is something called *endpoint* descriptors. These describe the endpoints for the device, unidirectional pipes that carry data one way either from the host to a device or from device to host. There are four types of USB endpoints:

- **Control endpoints** – These are small endpoints commonly used for configuring the device or retrieving information from it. Every USB device comes with a control endpoint called endpoint 0 that is used by the USB core in Linux to configure the device when plugged in. The USB protocol guarantees the transfers used by control endpoints always have enough bandwidth to successfully go through to the other end.
- **Interrupt endpoints** – Interrupt endpoints transfer small amounts of data every

time the host polls the device. They are also used to send configuration data to the device. Commonly used in keyboards and mice. Also comes with guaranteed data transfer bandwidth.

- **Bulk endpoints** – As the name implies, these endpoints transfer large amounts of data. This data is guaranteed to get through with no data loss, but there is no guarantee for how long it will take. Common in for example storage devices.
- **Isochronous endpoints** – Transfers large amounts of data similar to bulk endpoints, but with no guarantee regarding data loss. Used by real-time streaming such as in video and audio devices.

5.2.1 The Linux USB Subsystem

USB device functionality in Linux is split into three parts [28, 29]. The *USB Core Driver* is an architecture independent USB subsystem in Linux which implements the USB bus specification and handles most of its complexity. This system should not be modified when porting Linux. *USB Host Drivers* are hardware dependent drivers that work to communicate with the specific USB controller hardware on an SoC. When porting Linux a USB Host driver should generally come with the boards' support package and should not need to be modified. Examples of USB Host drivers are the Open Host Controller Interface (OHCI), Universal Host Controller Interface (UHCI), and Mentor Graphics Inventra USB (MUSB) controllers [30, 31]. Lastly there are specialized *USB Device Drivers* that are the "actual" USB drivers that needs to be created for new devices, which this section details.

When writing a new USB driver in Linux a structure called `usb_driver` with 5 essential fields needs to be defined:

- `owner` – Owner of the driver, used by USB core for reference counting so that the driver is not mistakenly unloaded.
- `name` – Unique name of the driver.
- `id_table` – When a USB device is connected to a host computer running the Linux Kernel, the USB core driver loads the correct driver for the device by receiving two identification strings from the device. These strings, called the vendor id and the product id, are matched with a special list of declared ids in the Kernel. When writing a device driver these ids are declared in the structure `usb_device_id`, which `id_table` is a pointer to. To populate the structure the macro `USB_DEVICE(vendor, product)` macro is generally used. These ids are important, without them the device will not be recognized by the system.
- `probe` – After the correct driver has been loaded, the USB core calls the drivers' `probe()` function to properly initialize the device. Any local structures and control threads needed to manage the USB device should be initialized here. The `probe`

function should also detect the endpoint addresses and buffer sizes of the device to be able to properly communicate with it.

- `disconnect` – Called when the driver should no longer control the device. Performs cleanup.

If these variables and functions are written correctly the USB core should be able to load the driver. However to send and receive data from the device the driver needs additional functionality. USB communication in Linux is done asynchronously mainly through a basic data structure called a USB Request Block (URB). These URBs are similar to networking packets and comes with some basic information and configurations, such as which device to send the URB to and what endpoint should be used (eg control, bulk, interrupt or isochronous).

To send an URB the `usb_submit_urb()` function should be called whereupon the URB is assigned to an endpoint. Each endpoint can handle a queue of URBs. An URB can be marked as reusable which means that upon being sent the URB can be reused in a different endpoint.

Each URB has a completion handler interrupt function that is called in 3 situations:

- The URBs data transferred successfully.
- An error occurred during the data transfer.
- The URB was unlinked by the USB core.

In the completion handler the status variable is checked to determine what caused the interrupt and execute actions accordingly.

5.2.2 The USB HID Layer

Most USB input devices on the market today complies with the Human Interface Device (HID) specification standard [32]. With such a well-defined standard many operating systems comes with an abundance of device drivers designed to handle such devices using a USB HID layer, such as Linux. Simple devices such as generic keyboards and mice that comply with HID usually function immediately when plugged into a system having HID support without having to write any specific new drivers. More advanced devices that do not already have device driver support needs some additional work though. Luckily in Linux the HID layer is quite extensive and it may be enough to modify an already existing similar driver.

To enable the dynamic functionality of HID devices a few additional descriptors are used [33]. The important ones are the HID descriptor and the report descriptors that are sent when the device is detected. The HID descriptor details how many different

data packets, called *reports* in HID, the device can send, and their types and sizes. Each report has in turn its own report descriptor that describes the structure of the report. For instance, a report descriptor for a USB HID keyboard device typically contains the bit location of a particular key press/release state. When receiving a report descriptor the host device driver has to parse it to be able to understand any incoming report data. The host then periodically polls the device's control endpoint - if the device has data to send it prepares a report and replies. The device can also use an additional interrupt endpoint to transmit asynchronous data to the host. USB HID devices do not use bulk or isochronous endpoints.

5.3 Hardware Support in Android

Since Android is based on a Linux kernel one might assume that it communicates with hardware devices in a similar way to Linux. However, Android uses a fairly different implementation compared to what can be found in many Linux distributions. Android has defined APIs to handle the communication with the hardware in a determined way. These APIs are also known as the *Hardware Abstraction Layer* (HAL) in Android, [19].

HAL defines the communication interface between the upper layers of the Android software stack and HAL modules. These HAL modules are shared libraries that are responsible for interfacing with both the underlying kernel driver and the corresponding HAL API (figure 7). The HAL module is normally provided by the manufacturer of the hardware or exists already in the AOSP. This design makes it possible for the manufacturer to choose whether they want to release their HAL module under GPL or not.

Android applications communicate with hardware through hardware related services. These services are responsible for loading the appropriate HAL module and interfacing with it.

5.3.1 The Android input subsystem

When a Linux system receives input from an external device, the input signal is decoded by a device driver in the Linux kernel and translated into an input event. These events are collected by Android by reading the *evdev* (generic input event interface in Linux) drivers associated with each input device. This task is managed by *EventHub* component. EventHub keeps track of which devices that have been connected or removed from the system and also which capabilities the connected devices have (what kind of events they generate). The events collected by EventHub uses Linux input protocol event codes. Since Android uses a different set of event codes than those used in Linux, the events are translated into Android event codes [34].

The raw Linux events are read from EventHub and translated into Android event codes by the *InputReader* component. InputReader places the translated events in a queue, to let them be fetched by the *InputDispatcher* component. InputDispatcher waits for incoming events on the queue, and dispatches them asynchronously to the appropriate

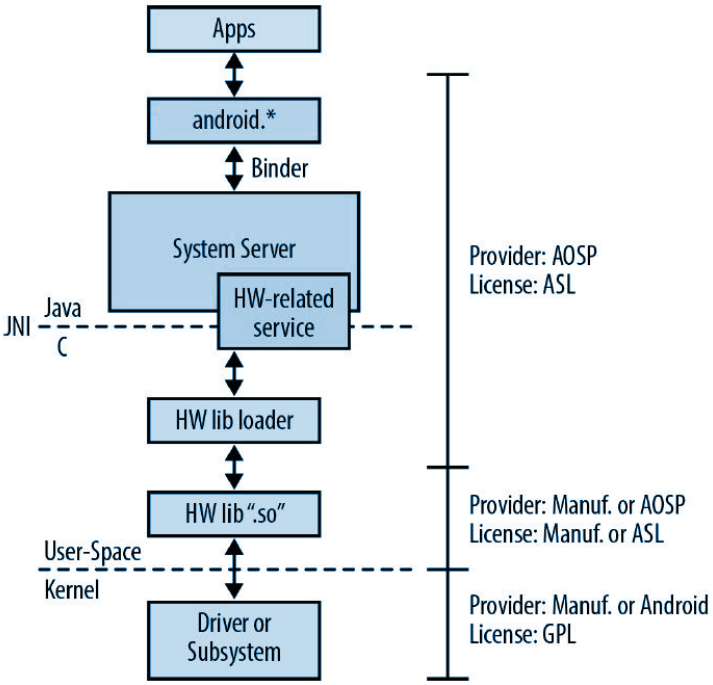

Figure 7: *HAL*.

applications.

A device that is connected to an Android system needs to be classified to be treated in an appropriate manner. For touch devices, there exists a wide variety of different kinds. They can be single or multi-touch, and the device could for example be a touch screen or a touch pad. The device is classified as a specific device depending on which events it generates. A device is identified as a single-touch device if it generates events of type ABS_X, ABS_Y and BTN_TOUCH. After this classification, it will load the Input Device Configuration (IDC) file. This file is used for device-specific configuration properties. While the system default configuration is enough for most standard devices or peripherals such as USB keyboard and mouse, these IDC files are almost always required by built-in embedded devices [35]. Touchscreens in particular are devices requiring IDC files that are relevant for this project.

6 The Target Platform

The target platform used in this project is the vehicle computer VTC 100 from Nexcom [1]. It is connected with the LCD touchscreen VMD 1001, also from Nexcom [36].

6.1 Vehicle Computer - VTC 100

The VTC 100 comes with the SoC AM3352 [37]. The system has 256 MB on-board DDR2 (Double Data Rate) RAM, and comes with several communication peripherals such as USB, VGA, Ethernet and Serial ports. It uses a Secure Digital High Capacity (SDHC) card as non-volatile memory. The VTC 100 has no GPU.

The choice of this platform has largely been due to how the VTC 100 is designed with robustness in mind. In the datasheet it is stated that the VTC 100 is able to run in a wide temperature range, and comes with some resistance to shock impacts. It also features a compact design. All of these qualities are important considering the fairly rough environment that it will be operating in.

6.2 Touchscreen - VMD 1001

The VMD 1001 comes with a VGA port for display and provides touchscreen capabilities over USB [36]. The 7" TFT LCD monitor resolution is 640x480 pixels. The touchscreen supports single-touch capabilities.

6.3 System on Chip - AM3352

VTC 100 is based on AM3352 which is an ARMv7 SoC produced by Texas Instruments that runs at 720 MHz. This SoC has a microprocessor unit subsystem that is based on the ARM Cortex-A8 microprocessor. It has peripheral controllers for Universal Asynchronous Receiver/Transmitter (UART), SPI, I2C and USB. To keep the area footprint low, each pin of the SoC has been supplied with multiple functions. To modify the functionality of each pin, pin muxing needs to be done. The AM3352 lacks a GPU.

6.3.1 Booting Process

When performing a port to an embedded system, it is important to be able to debug the system when it boots. When the VTC 100 starts, the ROM code inside the AM3352 is loaded and executed [38]. The next step, called the boot mode, is to locate and load the bootloader. The boot mode of the AM3352 is determined via its boot configuration pins, also known as SYSBOOT. These can be changed by pin-muxing. Different configurations of the SYSBOOT pins leads to different booting device sequences - one configuration might put peripheral booting via UART0 as top priority, while another may select memory card booting first. These booting devices are looped through until one of them succeeds in finding and executing an image. The exact setup of the VTC 100 SYSBOOT pins

is unknown, however it boots successfully with the bootloader residing on the SD card. This means that the SYSBOOT pins are setup in such a way that memory card booting is somewhere in the booting device sequence.

Traditionally a bootloader is loaded into the internal memory on a board and executed. However, newer bootloaders almost always require more space than is available internally on a board. Therefore when compiling U-Boot it is actually compiled into two different images, called `MLO` and `u-boot.img` [39]. The MLO, also called a Second Program Loader (SPL), is a smaller file (On AM3352 it is not allowed to be more than 109kB to fit in the internal memory [38]) that initializes a few key areas such as external memory and serial functionality. It then loads `u-boot.img`, the core of U-Boot, into external RAM and transfers control to it. U-Boot then initializes non-critical functionality such as peripheral ports other than serial ports before loading the Linux Kernel. This is called a two-stage bootloader.

6.4 System Requirements

Recent versions of Android have increased functionality but also increased memory requirements. According to the Android Compatibility Definition document [40], Android 4.2 has a higher minimum memory requirement than what is available on the target platform.

The minimum recommendation for Android 2.3.4 fits within the available resources on the target hardware. Note that both versions strongly recommend a hardware GPU.

Table 2: Minimum hardware recommendations for Android 2.3 and Android 4.2.

Version	**RAM**	**GPU**
Android 2.3	256 MB	Recommended
Android 4.2	340 MB	Recommended

6.5 Other Android Platforms

There exists many devices with all kinds of specifications on the market containing the Android OS today. For older mobile devices the HTC Desire S, an Android phone released on the market in 2010, has a 1GHz Snapdragon processor in the ARMv7 architecture and comes with 768 Mb DDR2 memory. It also features an Adreno 205 GPU and runs Android 2.2. However compared to high end phones on the market today, the HTC Desire S has poor specifications regarding processor power and amount of memory. One phone recently released is the Samsung Galaxy S5. It has a 2.5 GHz Qualcomm Snapdragon 801 Quad-core, 2 GB of RAM and runs Android 4.4.

6.5.1 AM335x-based Boards

Table 3 lists other boards that are based on SoC's from the AM335x-family. The boards presented in this table, excepting the VTC 100, are based on AM3358. The difference between AM3352 and AM3358 is mainly that AM3358 contains a GPU compared to AM3352 that lacks this part.

Of all boards presented in table 3, the VTC 100 has the weakest hardware configuration, since the CPU is the same as the other boards, however the memory is among the lowest and it contains no GPU. The most similar board that can be found in this table is the *BeagleBone*. It has the same amount of RAM as the VTC 100, but has a GPU which is missing in the VTC 100. These boards will be used as a reference in the evaluation part in order to compare the performance of the VTC 100 to the performance of other AM335x-based boards when they run Android. The similarity between the VTC 100 and the BeagleBone may reveal the effect of a system that runs Android without a GPU. TI provides benchmarking results for all these boards running Android 4.2.

Table 3: VTC 100 compared to other AM335x based platforms.

Platform	**SoC**	**RAM**	**GPU**
VTC 100	AM3352	256 MB DDR2	No
AM335xEVM	AM3358	512 MB DDR2	Yes
AM335x SK	AM3358	2 GB DDR3	Yes
BeagleBone	AM3358	256 MB DDR2	Yes
BeagleBone Black	AM3358	512 MB DDR3	Yes

7 Implementation

This Chapter describes the practical work that has been done to get Android 2.3 and Android 4.2 to run on the VTC 100. It describes the existing support from hardware vendors for the VTC 100 and the development platform environment, and details how the source files for the operating systems have been customized for the target platform.

As described earlier, the first step when implementing Android on a new platform is to get the Linux kernel running on the target hardware. During this process, we utilized tools provided from the different vendors. The Linux Kernel versions used by TI in their Android SDK's differ which required the kernel patches from Nexcom to be applied manually once per Android version. However since U-Boot is not dependent upon which kernel version that it will load, it is not affected by this kernel mismatch and it was therefore straightforward to apply U-Boot related patches.

7.1 Support from hardware vendors

The target hardware used for this project comes with Linux support from the vendor. This support is provided by patches for U-Boot and the Linux Kernel that contains board specific code for the VTC 100. The patches are supposed to be applied to an SDK provided by the hardware vendor of the ARM-SoC, AM3352, that the VTC 100 is based upon. TI is the vendor of AM3352. They provide SDK's for various Linux versions and support for a limited amount of Android versions.

The selected SDK's used in the project are targeted for Android versions 2.3 and 4.2. These SDK's contains the Android source code, including source code for standard reference boards for both the Linux kernel and U-Boot. They also contain cross-compilation toolchains and helper scripts to prepare an SD card with Android.

7.2 Equipment

We used the recommended development platform setup from TI, which is Ubuntu 10.04 64 bit as operating system. To prepare U-Boot and the Linux kernel for compilation a menu configuration application was used to give an overview of enabled and disabled settings. A simple serial terminal emulation program was used on the host platform to debug the VTC 100 over UART. Over this serial connection we used several debugging programs to output memory and CPU usage of the target platform, and to monitor touchscreen events. To apply the Nexcom patches we used several tools to find the location of specific files and to output differences between two or more files.

7.3 Customization of U-Boot

To customize U-Boot for a custom board, we modified the source code of U-Boot related to the board. In particular each board has its own configuration file that contains two types

of configuration variables, *configuration options* and *configuration settings*. According to the U-Boot readme file, configuration options are supposed to be general options, while configuration settings handles hardware specific configurations.

In U-Boot each board has its own board file that is set apart from the configuration file. Here all initialization logic for U-Boot occurs, such as voltage and power setup, pin muxing, and clock initialization. Detailed knowledge of the board such as register addresses and clock delays are needed to be able to create a board file.

The provided patches from Nexcom contains board specific source code to customize U-Boot to run on the VTC 100. We applied these patches to the Android SDK from TI that already contains U-Boot source code for other similar custom boards. The patches provided by Nexcom were enough to initialize the board and to load a Linux kernel, however it did not provide any debugging functionality.

We modified the source code of U-Boot in order to get interaction and debugging information during the development process. These modifications were mainly done in the configuration file. To receive console printouts over a UART connection the console output configuration option was enabled and the UART controller was switched to UART1 by specifying a special registry address. In addition to this, the Linux kernel boot parameters were modified to also enable the serial console in the kernel and in the Android layers, as well as to specify the location of the Android init file.

7.4 Configuration of the Linux Kernel

Nexcom bases their board port of the VTC 100 on the AM335x EVM, which is a reference board from TI. Large parts of the patch files from Nexcom include modifications to the board file for AM335x EVM, as well as other files. As mentioned earlier, the patches from Nexcom are targeted for Linux kernel version 3.2.

We modified the pin configuration in the board file for the AM335x EVM to match the circuit layout for the VTC 100. These modifications included enabling USB and UART peripheral functionality. The operating mode of the USB pin was also changed from On-The-Go (OTG) to HOST, since the VTC 100 does not have support for USB OTG. We modified the default kernel configuration file for the AM335x EVM by applying the patches provided by Nexcom to customize it for the VTC 100. Through the patches we also created a new driver for VTC 100 Memory Control Unit (MCU) power interrupt for board-specific power management. For Ethernet functionality the patches modified the initialization of Common Platform Ethernet Switch (CPSW). The rest of the changes made by the Nexcom patches were unrelated to this project, such as modifications to different LED direction settings or TI's own Touchscreen Controller software.

Once we had applied the patches and further modifications, the U-Boot images and the Linux kernel were compiled using the AM335x EVM configuration file. Any components marked as modules in the configuration file were changed to be built into the kernel to

ease debugging.

7.4.1 USB Touchscreen

The Linux kernel detected the VMD 1001 touchscreen over a USB connection using the generic USB HID driver. The events produced by the touchscreen were also detected successfully. However, the device was recognized as a mouse device instead of as a touchscreen. Since the touchscreen is a single touch device this did not have any impact on the functionality. Moreover the X- and Y-axes appeared to be swapped. Since the device was handled by the complex generic USB HID driver we determined that this problem would be easier to fix in the Android layer.

7.5 Implementation of Android

To properly compile for Android we had to modify the kernel configuration file once again. The recommended configuration options from the Android homepage [41] were applied and tweaked, such as configurations enabling Android Binder and the Low Memory Killer. Since the VTC 100 does not have USB OTG capability we removed the modules associated with USB Gadget functionality. Otherwise the newly added configuration properties were largely untouched.

Changes were required within the AOSP in order to include support for the touchscreen in Android. The Linux kernel was able to detect the touchscreen but the Android layer did not receive any input events from the kernel. We added an IDC file, specifying that the device should be considered a touchscreen. However, since the Linux Kernel considered the device a mouse device, two components of the Android layer had to be modified to properly receive the touch events. These components were the EventHub and the InputReader. We modified them to listen to mouse events, as well as to flip the X and Y coordinates received.

7.6 Implementation differences between Android 2.3 and 4.2

Setting Android 4.2 up for successful compilation required about the same workload as for 2.3. The same U-Boot version could be used since it does not need to change between versions. However, the Linux kernel provided by TI for Android 4.2 is a different version than the one provided for 2.3. This required us to manually apply and tweak the patches from Nexcom once more. No additional customization of U-Boot or the Linux kernel were required to get a working system.

7.6.1 System tuning

From the description given in section 4.8, other than disabling some unnecessary services in the Android `init.rc` file, we decided to not perform any substantial system downscaling. As mentioned earlier, disabling system components that Android expects to exist in the system might cause applications to fail to function correctly, since Android

system services are not designed with modularity in mind. For instance, disabling the phone system service caused other programs to continually crash, all while Android kept displaying a reoccurring obnoxious error message.

8 Evaluation

This chapter presents all results we have achieved from the experiments conducted. These results are contained in graphical charts presenting CPU performance, RAM usage and boot-up time for the two versions of Android running on the target platform. The charts additionally contains results from TI's testing of Android 4.2 on their own hardware [42], as well as results from other known Android devices on the market, to provide performance comparisons with the VTC 100.

8.1 Setup and Tools

We conducted all benchmarking using a variety of tools. The tools *RowboPERF* from TI, *Quadrant Advanced* from Aurora Softworks [43], *vmstat*, *procrank*, *dumpsys*, and *Bootchart* were used during evaluation. The main parameters that have been considered are memory usage, CPU utilization and boot-up time.

RowboPERF is a tool that contains a collection of benchmarking programs. It covers tests for the C library and system calls with a variety of mathematical and floating point operations, OpenGL—ES, 2D/3D drawing performance, Garbage collection in Dalvik and the JavaScript engine. The tool has been used to evaluate the performance of the CPU during resource-intensive tasks. The results are presented either in operations per second when performing mathematical tests, or in Frames Per Second (FPS) when evaluating drawing performance. The following tools from RowboPERF have been used for benchmarking the VTC 100:

- *Linpack* measures the floating point performance on a system. The results are presented in MFlops (Millions of floating point instructions per second).
- *Dhrystone* does not performs any floating point operations. Instead it mainly measures string operations and caching efficiency. Dhrystone manipulates a very small amount of global data, instead concentrating on local variables. Output is presented in acrfulldps.
- *Whetstone* measures floating point numerical operations. As opposed to Dhrystone, Whetstone manipulates mainly global data.
- *Circle, Canvas, and Teapot* are all drawing applications that draws a large amount of figures on the screen. Each program presents their results in FPS.

Quadrant is a commercial benchmarking application for Android. This tool has been used to evaluate CPU, memory accesses, I/O and 2D/3D graphics performance. Quadrant presents its results in "points", a unit-less measurement which is really only useful for comparisons with other Quadrant benchmarks.

We selected both RowboPERF and Quadrant as benchmarking tools because TI has already used them for performance benchmarking in Android 4.2 on their own boards.

Therefore we can draw comparison charts between the VTC 100 and TI's boards to be able to make conclusions regarding the relative performance of the VTC 100.

vmstat is a tool used to output total CPU usage of the system. procrank is another simple tool that outputs per process Resident Set Size (RSS) memory usage. RSS is the amount of memory of a process that is held in RAM. These values are generally not useful when determining how much memory would be freed if a process is killed, since a large amount of memory is shared between processes in Linux and Android [44]. However it can give an idea of the size of the processes. More useful values are Proportional Set Size (PSS) and Private Dirty Memory that is output by the program dumpsys. PSS is defined as a process' RSS divided by the number of processes that are using that mapping [10]. Private dirty memory is memory allocated as private to a process that the process has modified. These metrics together gives the closest approximation to how much memory a process uses, and how much would be freed upon killing the process.

Bootchart is a tool that is able to generate a graphical chart of the activity of a system during boot. We have used Bootchart to measure the required time for booting the system.

8.2 Memory Usage

Figure 8 shows the memory available after a complete system boot for the two Android versions running on the VTC 100. We performed this test by using the command `cat /proc/meminfo` after a complete system boot for each Android version. On a typical boot Android 2.3 has 126.6 MB of free memory, whereas Android 4.2 only has about 19 MB free.

As can be seen in the total memory graph, Android 4.2 requires substantially more memory at boot-time than Android 2.3. While there is still an adequate amount of RAM left for the system to continue, for general use it is not enough. Output from logcat during testing showed that the Garbage Collector in Android 4.2 starts using system resources to operate more and more while using applications in the System. Furthermore the Android Low-Memory Killer starts killing processes as soon as the user interacts with applications. Both of these types of events slows down the system by using CPU resources.

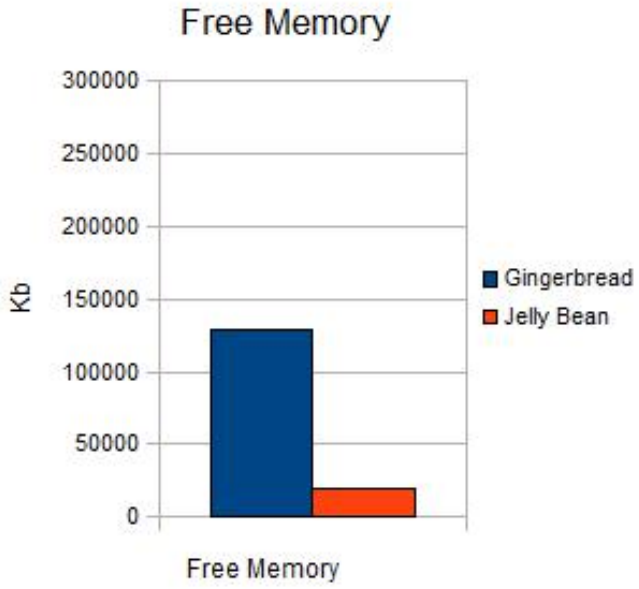

Figure 8: *Amount of free memory in Android on the VTC 100.*

The memory values from figure 9 and figure 10 are RSS values. These give an idea of the sizes of the processes that live after booting the two Android versions on the VTC 100. As seen in the charts these sizes in Android 4.2 are generally larger than in Android 2.3.

```
# ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      1     0     556    188   c00c4030 0000877c S /init
root      528   1     564    188   c00c4030 0000877c S /sbin/ueventd
root      742   1     748    336   c00407ac afd0c3ac S /system/bin/sh
system    744   1     820    248   c030c3b0 afd0b6fc S /system/bin/servicemanager
root      745   1     3880   572   ffffffff afd0bdac S /system/bin/vold
root      746   1     4252   836   ffffffff afd0bdac S /system/bin/netd
root      747   1     680    252   c0325e48 afd0c0cc S /system/bin/debuggerd
root      748   1     1284   540   c005e7a0 afd0bdac S /system/bin/rild
root      749   1     61604  26716 c00c4030 afd0b844 S zygote
media     750   1     21092  4212  ffffffff afd0b6fc S /system/bin/mediaserver
root      751   1     828    304   c039ffc0 afd0b45c S /system/bin/installd
keystore  752   1     1760   420   c0325e48 afd0c0cc S /system/bin/keystore
root      753   1     1352   140   ffffffff 00008294 S /sbin/adbd
system    779   749   131816 34828 ffffffff afd0b6fc S system_server
system    840   749   74840  22172 ffffffff afd0c51c S com.android.systemui
app_20    848   749   78692  23916 ffffffff afd0c51c S com.android.launcher
app_3     889   749   72400  16616 ffffffff afd0c51c S com.android.deskclock
app_0     898   749   74444  17728 ffffffff afd0c51c S android.process.media
root      916   742   908    300   00000000 afd0b45c R ps
#
```

Figure 9: *Memory intensive processes in Android 2.3.*

```
root@android:/ # ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      1     0     592    236   c00c7f48 0000f068 S /init
root      570   1     608    216   c00c7f48 0000f068 S /sbin/ueventd
system    811   1     892    348   c031670c 40118fc0 S /system/bin/servicemanager
root      812   1     4076   868   ffffffff 4014e76c S /system/bin/vold
root      813   1     9564   1096  ffffffff ffff0520 S /system/bin/netd
root      814   1     936    400   c032d758 401c8a70 S /system/bin/debuggerd
system    815   1     29420  10000 ffffffff 400e9fc0 S /system/bin/surfaceflinger
root      816   1     458276 35832 ffffffff 401390e4 S zygote
media     817   1     25564  5712  ffffffff 40184fc0 S /system/bin/mediaserver
install   818   1     904    444   c03a4f84 401ebd50 S /system/bin/installd
keystore  819   1     1832   892   c032d758 40146a70 S /system/bin/keystore
radio     822   1     4520   840   ffffffff 4017976c S /system/bin/rild
root      823   1     824    476   c0017658 400a894c S /system/bin/sh
system    1393  816   555124 44476 ffffffff 40138fc0 S system_server
u0_a4     1474  816   473676 27040 ffffffff 40139ebc S com.android.systemui
u0_a22    1491  816   472732 21964 ffffffff 40139ebc S android.process.media
u0_a17    1535  816   477956 23808 ffffffff 40139ebc S android.process.acore
u0_a0     1553  816   475488 31220 ffffffff 40139ebc S com.android.wallpaper
u0_a27    1571  816   469656 21632 ffffffff 40139ebc S com.android.inputmethod.latin
radio     1581  816   475476 22788 ffffffff 40139ebc S com.android.phone
u0_a3     1599  816   489912 38432 ffffffff 40139ebc S com.android.launcher
u0_a2     1642  816   467652 17864 ffffffff 40139ebc S com.android.location.fused
u0_a11    1664  816   466808 17636 ffffffff 40139ebc S com.android.smspush
system    1679  816   477152 20400 ffffffff 40139ebc S com.android.settings
u0_a10    1707  816   468340 19140 ffffffff 40139ebc S com.android.music
u0_a7     1749  816   474944 19188 ffffffff 40139ebc S com.android.exchange
u0_a15    1762  816   479604 21996 ffffffff 40139ebc S com.android.email
u0_a21    1796  816   470164 20608 ffffffff 40139ebc S com.android.providers.calendar
u0_a33    1821  816   470424 20180 ffffffff 40139ebc S com.android.gallery3d
u0_a41    1839  816   470480 20316 ffffffff 40139ebc S com.android.deskclock
root      1864  823   1152   428   00000000 401e3d50 R ps
root@android:/ #
```

Figure 10: *Memory intensive processes in Android 4.2.*

The values presented in figure 11 and figure 12 are in PSS and Private Dirty Memory format. These values give a closer approximation to how much memory can be freed if a process is killed, as mentioned in section 8.1. Private Dirty Memory in particular shows how much private memory a process has modified. The system process in Android 4.2 has a substantially larger Private Dirty Memory allocated to it than the same process in Android 2.3, which means that it uses and has modified a larger amount of memory.

Most of the processes at boot-time for Android are critical components of the system. A few services can be removed from boot, such as the Android deskclock. However, according to the results from figures 11 and 12, the memory required by for example the deskclock process is substantially lower than the memory needed by critical Android processes such as the system process and the launcher process. This is true for many of the optional processes that live at Android boot-time. Killing these only saves a few megabytes of RAM.

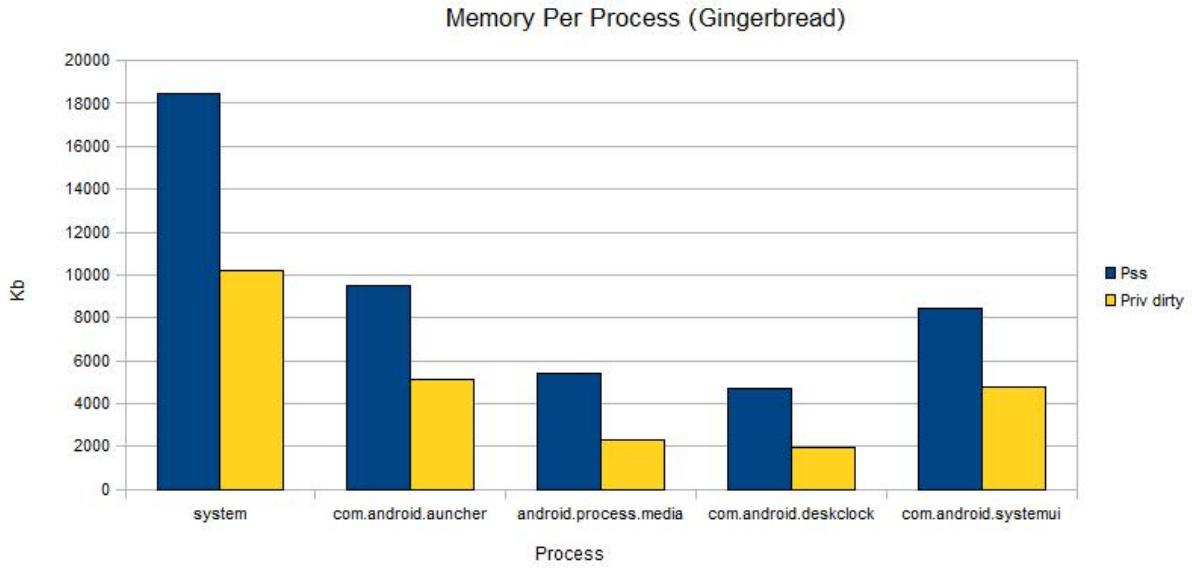

Figure 11: *Procrank output in Android 2.3.*

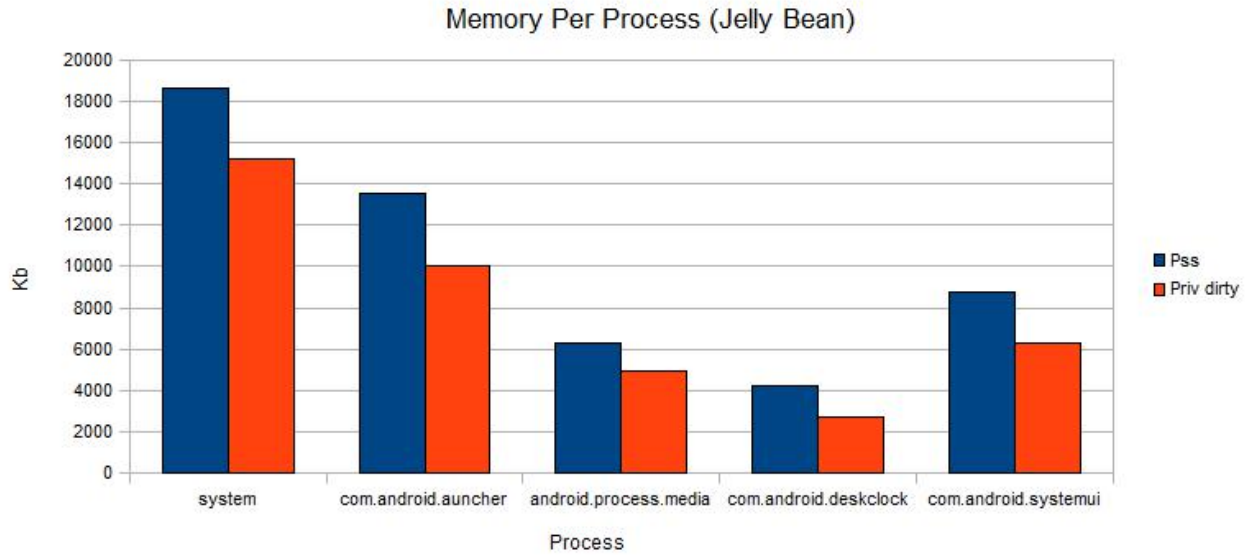

Figure 12: *Procrank output in Android 4.2.*

8.3 CPU Performance

We conducted several tests to measure CPU performance in Android. These are compared to the same tests executed by TI on their own boards, as well as testing conducted on the HTC Desire S. Most of the tests are from the RowboPERF benchmarking package

for Android [45], except the tests from Quadrant. The tests from TI were conducted in Android 4.2, running on four different boards: AM335xEVM, AM335x SK, BeagleBone, and BeagleBone Black. All the TI boards are based on the AM3358 SoC, which is similar to AM3352 with the important distinction that the AM3358 contains a graphic accelerator. For the Quadrant tests the HTC Desire S has also been benchmarked while running Android 2.2.

Figure 13 shows the results of the floating point performance benchmarking program Linpack run on the two Android versions.

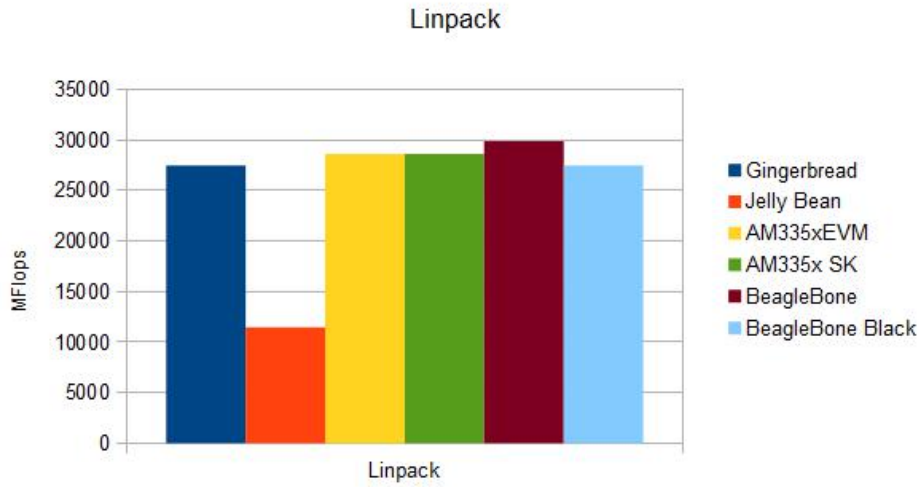

Figure 13: *Linpack.*

Figure 14 shows the results of the Dhrystone benchmarking program over 10 000 000 repetitions, in DPS. Dhrystone measures mainly local variables, string operations, and caching performance.

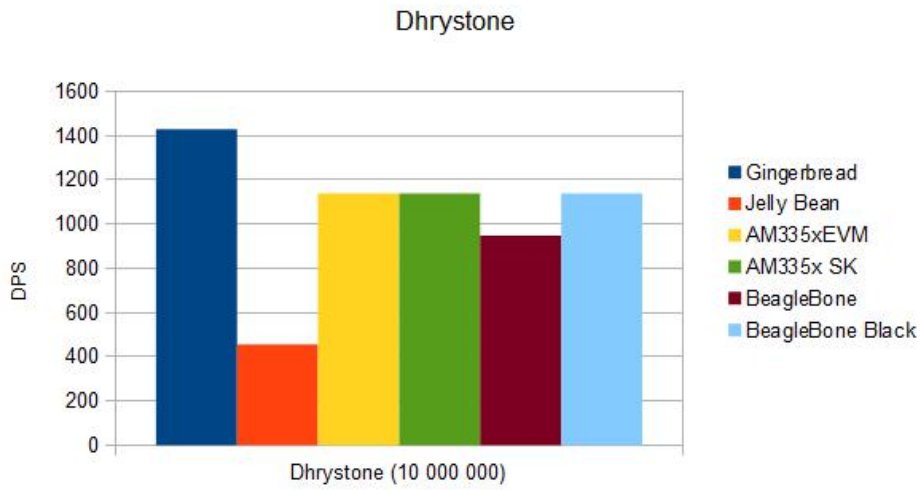

Figure 14: *Dhrystone.*

Figure 15 shows the results of the Whetstone benchmarking program over 20 000 repetitions. Whetstone measures mainly global data with floating point numerical operations.

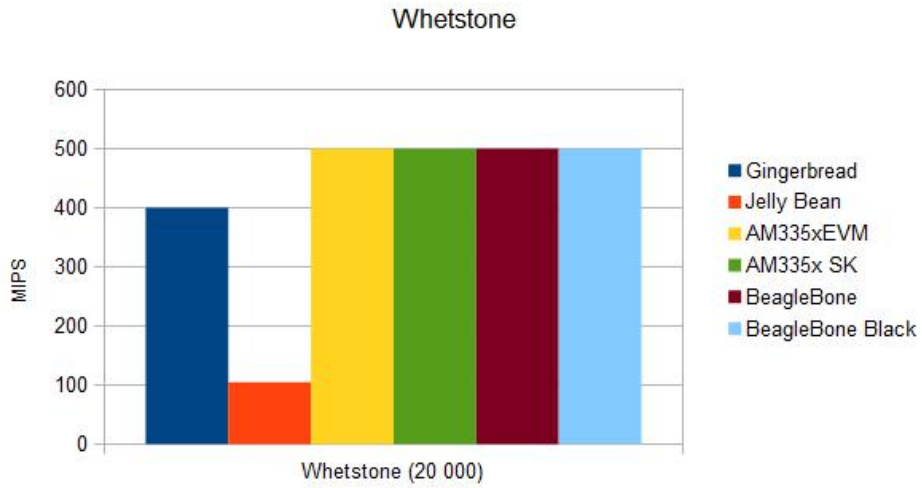

Figure 15: *Whetstone.*

For Android 2.3, according to figures 13, 14, and 15, the CPU in the VTC 100 performs similar to the CPU in TIs boards. This is expected as both the AM3352 and the AM3358 contains the Cortex A8 processor.

Another result in Android 2.3 produced by experimenting with the platform is that the FPS drops substantially when any kind of window transition or window scrolling event

occurs (see section 2.2). For instance, pulling down the top menu bar in any of the Android versions slows down the system to single-digit FPS while the menu is animating. The same happens when pressing the menu button and the menu overlay is created. This is due to that drawing or animating these events, called window compositing, requires more processing power than other simpler drawing events in Android since they update the whole screen [46]. In "vanilla" Android versions without modifications, these events are handled by the GPU, which the VTC 100 lacks. The VTC 100 is instead forced to use the CPU to process these events, slowing down the system significantly.

In Android 4.2, the FPS drops are even more pronounced. Starting from Android 3.0, most drawing events in Android are handled in the GPU [46], not only window transitions. Figures 13, 14, and 15 which were all obtained while the screen was newly booted into the home screen of Android, all give significantly poorer results in Android 4.2 than Android 2.3. Both versions are on the same board, so the only explanations are either that the memory runs out, or that drawing the home screen in Android 4.2 is significantly more expensive in terms of processing power than in Android 2.3. Indeed, measuring output from vmstat shows about 95% CPU usage while stationed in the home screen for Android 4.2. Moving into a menu drops the CPU usage to about 20%. This may be due to the standard deskclock being drawn on the home screen. The same measurements done in vmstat in Android 2.3 outputs only a few percentages of CPU usage both in the home screen and inside a menu. Moreover the boards from TI all have a GPU, and the Beaglebone has the same low amount of RAM as the VTC 100 but performs much better in the RowboPERF tests running Android 4.2 than the VTC 100. The only conclusion is that the lack of a GPU is the main bottleneck for performance in Android 4.2.

Figure 16 shows the results of running the 2D and 3D drawing applications canvas, circle, and teapot from RowboPERF. The results are in average FPS over three runs of the applications. The FPS values achieved are rather low, about 14-22 FPS throughout the tests. Normal Android framerate is usually around 60 FPS. However these are benchmarking applications that draw a significant amount of objects at the same time, except for the Teapot application which draws a single polygon with many vertices. Any application built for the platform would most likely require less performance output than the benchmarking applications.

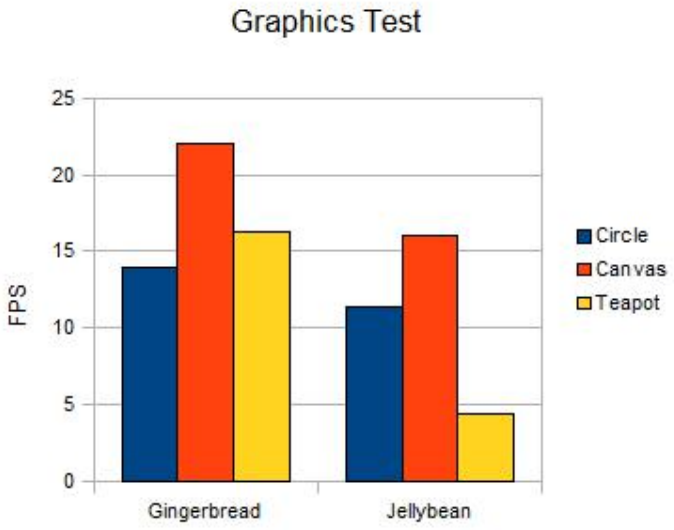

Figure 16: *FPS output from drawing applications.*

Figure 17 shows the results of running Quadrant, a commercial benchmarking application for Android (see section 8.1). TI has provided tests running Quadrant on their boards, which can be compared against the Quadrant results of the VTC 100. Both Android 2.3 and Android 4.2 have received a low CPU score compared to the boards from TI and the HTC Desire S, except the BeagleBone which also received a low CPU score.

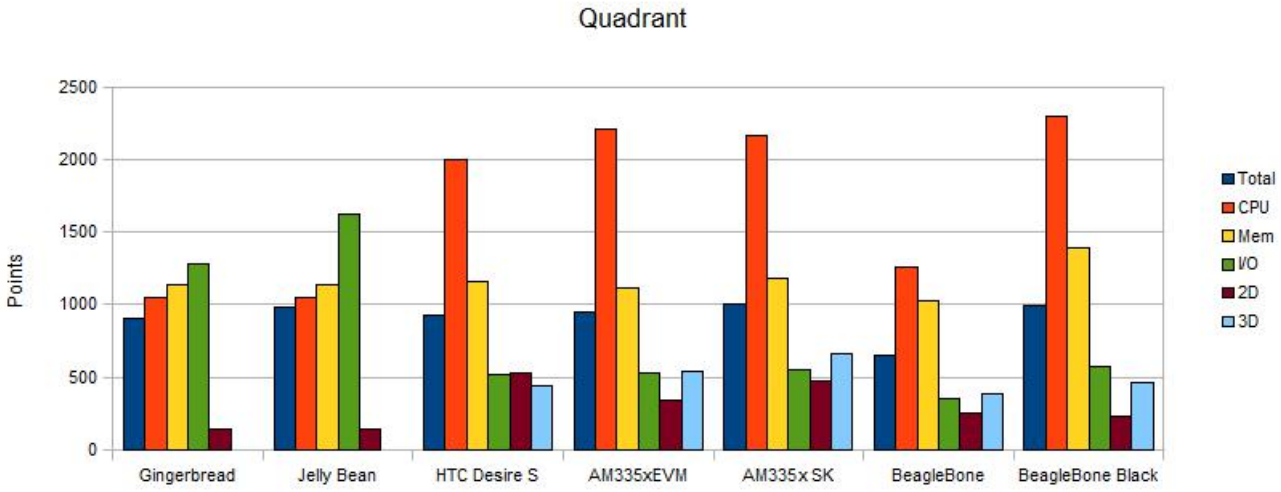

Figure 17: *Quadrant output.*

8.4 Boot up Time

Figures 18 and 19 found in appendix A shows which processes are running and for how long when the two Android versions boots, obtained using Bootchart. The Android system starts to respond to regular user input (eg. touchscreen events) a few seconds after the launcher application has started. In Android 2.3 the launcher application, according to figure 18, starts around 40 seconds into the boot-up sequence. In Android 4.2 the boot-up time is slower, with the launcher application starting around 70 seconds into boot-up according to figure 19. Furthermore as seen in the graph there are more system services left at this point in time for Android 4.2 to start, which slows down the time until practical usage of Android is possible. Both Bootchart graphs were created during a second boot-up sequence on the VTC 100, since the first boot-up sequence of Android is always slower than subsequent boot-ups, as seen in section 4.5.

9 Conclusion

To conclude the implementation and evaluation of Android on the VTC 100 we detail and answer the questions from the problem description below. Finally a few platform improvements are suggested to allow Android to run with acceptable performance on the VTC 100.

9.1 Perfomance Impacts and Bottlenecks

In this thesis we have looked into the run-time performance of an old Android version versus a newer version of Android running on the VTC 100. We have evaluated the individual impacts of different types of constrained resources to be able to determine resource bottlenecks in the two systems.

The implemented versions of Android runs fairly stable on the target platform. It is possible to navigate through the operating systems using the supplied touchscreen and to open applications successfully. However we conclude that the performances of both systems are too poor for their intended uses, especially Android 4.2. In particular the response times of the systems are unacceptable when utilizing features of Android that are recommended to be powered by a GPU. Therefore our conclusion is that the main bottleneck for both Android versions is the lack of a GPU on the VTC 100. This conclusion is further strengthened by the existence of other boards running Android that contain a GPU, such as TI's own implementations, and the implementation in the paper mentioned in section 2.1. These boards have been shown to run with higher performance than the VTC 100 running Android. The conclusions drawn for the performance impacts of Android and its bottlenecks are motivated in more detail below.

9.1.1 Android 2.3

Memory
Our system testing and performance benchmarking have produced results showing that 256 MB of RAM are more than enough for the target platform to run Android 2.3. When Android 2.3 has finished booting up half of the total RAM on the system is still available (see section 8.2). Applications requiring additional RAM, such as drawing performance benchmarks, have shown that the amount of available RAM at any time is adequate for the systems needs.

CPU and GPU
By comparing Android 2.3 with other systems containing similar CPUs we have shown that the ARM Cortex A8 microprocessor is slow but adequate for the current version of Android. Tests for CPU speed using large numbers of intensive math and logical operations have shown that the VTC 100 performs about as well as other boards from TI (see section 8.3). The main bottleneck for system performance seems to be the lack of a GPU, due to drawing events normally handled by the GPU in Android being particularly slow on the VTC 100. The testing results from Quadrant on the VTC 100 compared

with Quadrant results from TI show similar hardware to the VTC 100, with the addition of a GPU, having substantial increases in the amount of points scored in the CPU and 2D fields. Since the CPUs are similar if not the same, this increase can be attributed to the GPU in TIs boards. Another factor that gives more weight to the lack of a GPU is the low FPS shown in the drawing benchmarks from RowboPERF. The native FPS in Android 2.3, when not displaying any drawing animation, is substantially higher than the results from these tests.

9.1.2 Android 4.2

Memory
Unlike the earlier version, Android 4.2 is also limited by the low amount of memory present in the target hardware. After a clean boot-up, Android 4.2 only has about 19 MB of memory left (see section 8.2). This seems to result from additional services added to Android 4.2, compared to Android 2.3, as well as core system services requiring more memory. We have removed some of the optional services, but the system is still extremely slow. The low memory is even more evident when looking at output from logcat in Android. Logcat shows that the Android Low-Memory killer and garbage collector are continually invoked, trying to free as much memory as possible. Each invocation of these services slow down the system.

This being said, without first-hand testing on another device containing a GPU it is difficult to determine if the amount of memory is enough for the intended uses of the system.

CPU and GPU
Our results from the intensive math and logical operations benchmarks have indicated that Android 4.2 either utilizes the CPU substantially worse than Android 2.3, or require much more processing power (see section 8.3). However, since these tests were conducted on the VTC 100 while displaying the Android home screen, the fact that both Android versions had to draw some graphical components while running the tests may have influenced the results. These tests would most likely indirectly produce better results with the addition of a GPU. All drawing tests from RowboPERF have shown less FPS in Android 4.2 than in Android 2.3.

9.1.3 Booting Android

Android 2.3 and Android 4.2 had boot-up times of slightly above 40 and 70 seconds respectively (see section 8.4). For comparison, measurement of one of the authors phones, the HTC Desire S, revealed a boot time of about 33 seconds. However, for the intended use of the ported systems which is as a vehicle computer, we conclude that the boot up time is more than sufficiently short.

9.2 Other Suitable Operating Systems for the Target Platform

As we concluded in chapter 2.6 there exists more suitable operating systems than Android for the VTC 100. To maintain a short time to market, we have considered other light-weight Linux distributions for implementation, due to the existing Linux support from the hardware vendor of the VTC 100. Both distributions considered, LUbuntu and Arch Linux ARM, would most likely perform better on the VTC 100 than Android. We motivate this by their system requirements specifications, and the existence of implementations of these operating systems on platforms with similar resources to the VTC 100 running with acceptable performance.

9.3 Ethernet

Ethernet support is not present in either Android version on the VTC 100. The existing support and documentation for Ethernet in any Android version is either minimal or non-existent (see section 2.5.2). The time allocated to the project has not been enough to conduct a deeper study into the AOSP to gather the knowledge required to implement Ethernet without prior support.

9.4 Suggested Platform Improvements

The main bottlenecks in the hardware design of the VTC 100 are the small amounts of RAM and the lack of a GPU. Right now the VTC 100 only has 256 MB of RAM and runs on a Cortex A8 processor with no graphics accelerator. To make the system more suitable to run Android 4.2 we recommend to use at least the minimum recommended requirements from Google, which is 350 Mb of RAM. However, this parameter is dependent on the type of applications that are going to run on top of the system, making it difficult to determine if the minimum requirements would be enough.

Both the Android 2.3 and Android 4.2 implementations would perform substantially better with a GPU. As seen in section 2.1, Android performs satisfactorily during hardware accelerated video playback on a platform containing a GPU. An option we suggest would be to switch the AM3352 SoC for the AM3354 that comes from the same chip family. AM3354 has the advantage over AM3352 that it is supplied with a GPU. These two SoCs are pin compatible, which would reduce the amount of work required to change the SoC for the VTC 100.

10 Future Work

The current implementation runs Android 2.3 and Android 4.2 on the VTC 100 with support for an external touch screen. The system performance is poor for both versions however, and as an addition to this project an evaluation of the improved hardware suggestions in the previous chapter could be conducted. We also suggest to perform an evaluation of the performance of any applications desired on the target platform, instead of generic benchmarking programs. The suggested SoC, AM3354, comes from the same chip family as the current SoC and is pin compatible. This new SoC would eliminate the problem of the lack of a GPU. More RAM would be beneficial as well, since it is a cheap component today and would contribute to better performance for at least Android 4.2, if not both versions.

The final system was supposed to have support for communication over Ethernet, however the current implementations does not this capability. Android has no official support for this and available information on the web on how to implement Ethernet in Android comes in the form of unofficial patches for Android versions other than the ones implemented in this project. Time constraints, together with the lack of any documentation for these patches, have led to insufficient time to apply these patches correctly. A deeper investigation into how Ethernet support could be integrated into the system is therefore suggested.

11 References

[1] Nexcom. VTC 100. http://www.nexcom.com/Products/mobile-computing-solutions/in-vehicle-pc/in-vehicle-pc/car-pc-vtc-100.

[2] Android Compatibility Downloads. http://source.android.com/compatibility/downloads.html, 2/7-2014.

[3] U-Boot. http://www.denx.de/wiki/U-Boot, 1/6-2014.

[4] TI Android Development Kit for Sitara Microprocessors. http://www.ti.com/tool/androidsdk-sitara, 19/8-2014.

[5] Android Gingerbread Compatibility Document. http://static.googleusercontent.com/media/source.android.com/en//compatibility/2.3/android-2.3.3-cdd.pdf, 1/6-2014.

[6] Yu-Hsuan Chan Frank Maker. *A Survey on Android vs. Linux.*

[7] Tarik Al-Ani. *Android In-Vehicle Infotainment System (AIVI).* 2011.

[8] Torbjorn Svangard. *An Evaluation of the PandaBoard as a Set-Top-Box in an Android Environment.* 2011.

[9] OMAP4 PandaBoard - System Reference Manual, 2010.

[10] Christopher Hallinan. *Embedded Linux primer : a practical real-world approach*, volume 2. Pearson Education, Inc, 2011.

[11] Basic Linux. http://distro.ibiblio.org/baslinux/, 2/7-2014.

[12] CrunchBang. http://www.distrogeeks.com/distributions/crunchbang/, 2/7-2014.

[13] LUbuntu. http://distrowatch.com/table.php?distribution=lubuntu, 2/7-2014.

[14] LUbuntu Requirements. https://help.ubuntu.com/, 2/7-2014.

[15] Arch Linux Touch. https://wiki.archlinux.org/index.php/touchscreen, 2/7-2014.

[16] Arch Linux ARM. http://archlinuxarm.org/, 2/7-2014.

[17] Arch Linux. https://www.archlinux.org/, 2/7-2014.

[18] Smartphone OS Market Share, Q1 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2/7-2014.

[19] Karim Yaghmour. *Embedded Android*, volume 1. O´Reilly Media, Inc, 2013.

[20] Hardware Acceleration. http://developer.android.com/guide/topics/graphics/hardware-accel.html, 2/7-2014.

[21] Dashboards. https://developer.android.com/about/dashboards/index.html, 23/6-2014.

[22] Benjamin Zores. Dive Into Android Networking: Adding Ethernet Connectivity. Alcatel-Lucent SA, 2012.

[23] Intel Corporation Christopher Bird. Wakelocks for Android. https://software.intel.com/en-us/articles/wakelocks-for-android, 13/4-2014.

[24] Processes and Threads. http://developer.android.com/guide/components/processes-and-threads.html, 27/5-2014.

[25] Karim Yaghmour. Headless android. Opersys inc, 2012.

[26] Linux Input Subsystem. https://www.kernel.org/doc/Documentation/input/input.txt, 25/5-2014.

[27] USB in a Nutshell. http://www.beyondlogic.org/usbnutshell/usb1.shtml, 1/6-2014.

[28] Linux USB Drivers. http://free-electrons.com/doc/linux-usb.pdf, 1/6-2014.

[29] The Linux USB Subsystem. http://linuxusbguide.sourceforge.net/USB-guide-1.0.9/book1.html, 1/6-2014.

[30] USB Host Controller Interfaces. http://en.wikipedia.org/wiki/Host_controller_interface_(USB,_Firewire), 1/6-2014.

[31] USB MUSB. http://elinux.org/MUSB, 1/6-2014.

[32] USB Human Interface Device Class. http://en.wikipedia.org/wiki/USB_human_interface_device_class, 1/6-2014.

[33] USB HID Drivers on Solaris and linux. http://www.cs.dartmouth.edu/~sergey/cs258/2010/KCA3_Bruning_USB.pdf, 1/6-2014.

[34] Input - Overview. http://s.android.com/devices/tech/input/overview.html, 13/5-2014.

[35] Android IDC Files. https://source.android.com/devices/tech/input/input-device-configuration-files.html, 13/5-2014.

[36] Nexcom. VMD 1001 - Datasheet, 2013.

[37] ARM Cortex-A8 Processor. http://www.ti.com/product/am3352.

[38] Texas Instruments. AM3352 - Technical Reference Manual, 2013.

[39] Two Stage Bootloader. http://omappedia.org/wiki/Bootloader_Project, 1/6-2014.

[40] Google Inc. *Android 4.2 Compatibility Definition*, 2013.

[41] Android Kernel Configuration. http://source.android.com/devices/tech/kernel.html, 4/6-2014.

[42] TI Android Jelly Bean Performance Benchmark. http://processors.wiki.ti.com/index.php/TI-Android-JB-4.2.2-DevKit-4.1.1_PerformanceBenchmark, 1/6-2014.

[43] Quadrant. http://www.aurorasoftworks.com/products/quadrant, 31/5-2014.

[44] Marco Cesati Daniel P. Bovet. *UnderStanding The Linux Kernel*, volume 3. O´Reilly Media, Inc, 2005.

[45] RowboPERF. http://processors.wiki.ti.com/index.php/RowboPERF_User_Guide, 31/5-2014.

[46] Android Graphics. https://source.android.com/devices/graphics.html, 4/6-2014.

A Boot charts

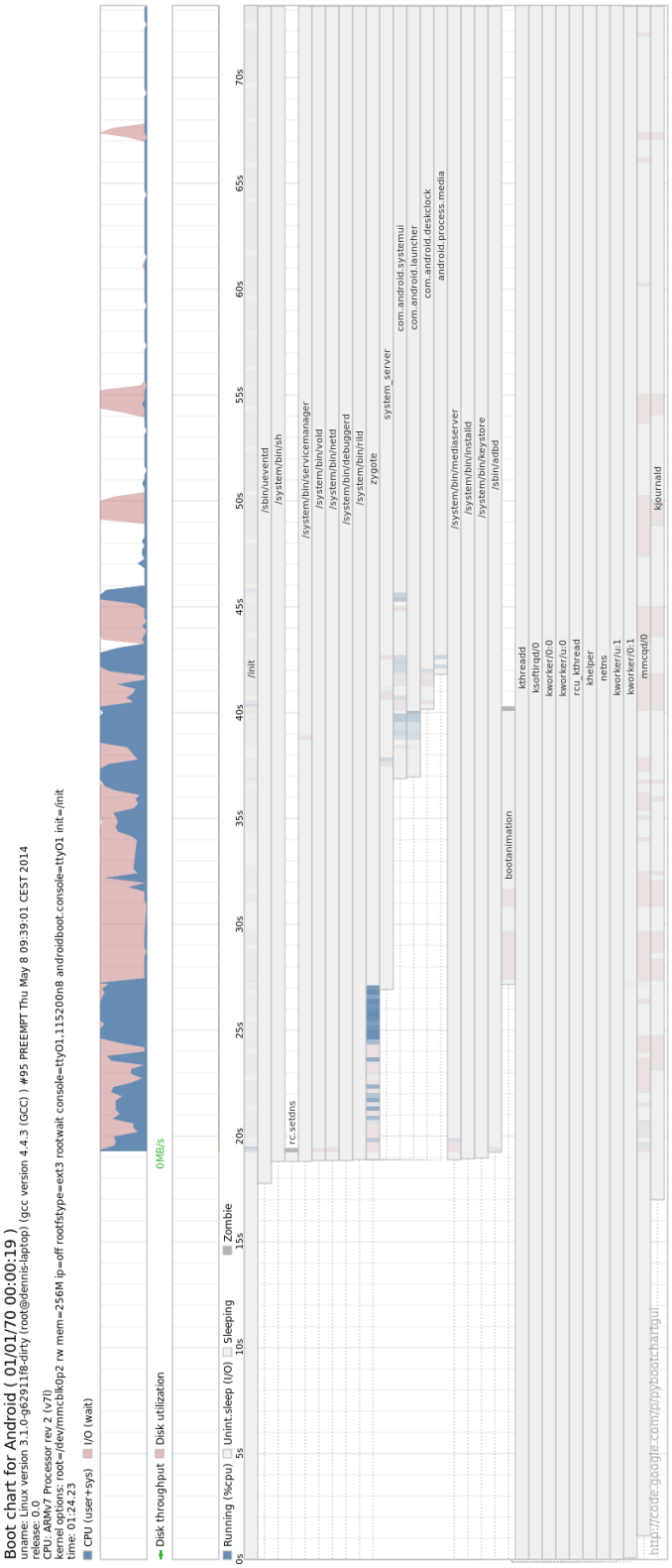

Figure 18: *Boot-up Android 2.3.*

Figure 19: *Boot-up Android 4.2.*