Practical Analysis of Gadget Framework on Android OS

Studienarbeit
im Rahmen des Diplomstudiengangs Informatik

**Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik**

eingereicht von: Taras Iks
geboren am: 02.02.1986
in: Frunse

Gutachter(innen): Dr. ret. nat. Wolf Müller

eingereicht am: ……

Contents

1 Introduction

The usage of electronic devices in daily life increases constantly so that a lot of people nowadays are owners of smartphones. A smartphone, is a mobile phone with the functionality of a personal digital assistant. Thus, a smartphone can be used for electronic identification and authentication. In this context, security of personal data plays an important role. It can be assumed that a smartphone is reliable due to the fact that the user takes care of the device and is familiar with its usage. A touchscreen is one of the important aspects of mobile security of a smartphone as it provides a visualization of transactions.Thus, the two factor authentication could be implemented: possession factor in form of a smartcard and knowledge factor in form of a PIN. Such device could act as a replacement for card readers. Considering the prevalence of Android smartphones with NFC modules an assumption was made of its public necessity.

This work has a purpose to set a cornerstone in building up a communication between the host computer and the Android smartphone over a USB interface. The host should enumerate an Android smart phone as a smartcard reader. A noteworthy issue is to prevent a necessity to install any additional drivers on the host because the integrated standard CCID-driver is present on recent operating systems. As a result a smartphone can be used immediately as a smartcard reader on operating systems like Linux or Windows without any additional configurations.

The research will cover analysis of the USB gadget functionality on the Android operating system and its practical usage. The results of the research will provide opportunities for a further analysis on this subject as well will yield recommendations for additional studies.

2 Analysis of USB Host and Gadget on Android

The chapter will provide information about USB system architecture which is divided into USB host and USB gadget. The important characteristics of the architecture will be systematized and analyzed continuously covering the topics needed to proceed in this work.

The starting point will be an overview of the USB technology. The physical and logical characteristics of this technology will be discussed in details. Besides, the possibility of a device to play a host or gadget role, the so called On-The-Go technology will be introduced. This technology is needed because on the one hand the smartphone device should bring card reader interface and on the other hand it should possess minimalistic host capabilities so that it can be extended to plug in some other peripheral devices like keyboards or mass storage devices. The different transfer types of data will be presented to cover the needs to transport big or small amount of data depending also on timing requirements. As mentioned above, the goal is to provide the communication possibilities between the host and the gadget so that exchange of information becomes possible between them. To sum it up, the gadget framework will be introduced.

In contrast to the host part the different modules with special features will be described. An introduction of the use cases for appliance of modules will provide intuitive understanding of the features of gadget framework. Furthermore, this research will provide information about Gadgetfs, a module which brings the configuration of the communication from kernel space into user space. Therefore, the Gadgetfs was chosen as a module to implement the communication with Android smartphone. The Gadgetfs module will be discussed further in a separate chapter.

As a gadget part in the communication model, the phone with Android embedded operating system was selected. Therefore, the embedded Android will be the further subject of the discussion. Android operating system brings a different way of initialization of hardware during the boot process.

The hardware plays crucial role because of the device drivers which each developer of embedded devices provides for its own hardware. This brings the discussion to the last point in this chapter where the concept and physical characteristics of System-on-Chip will be discussed and presented. The embedded device selected for the programming and testing purposes is Samsung Nexus S.

2.1 USB Overview

The starting point in this section is the introduction of the USB technology. Universal serial bus is one of the most used components to connect user's computer and peripherals. Moors Low is still valid over the years and describes the phenomena that computation power of computes doubles every year. As a consequence there is a need to move an increasing amount of data between peripheral devices and host computer. Historical fact is that the USB was originally designed to overcome the shortcomings of various input and output interfaces found on computer architecture. Nowadays it is difficult to find an electronic device that does not have a USB port. Digital cameras, printers, keyboards and card readers are typical examples of devices that have USB interfaces.

Different devices use different data rates to exchange the information. So here are some of them:

Low speed was introduced in USB 1.0, the first generation of USB technology and supports data rates of 1.5 Mbit/s. It is used in devices like keyboards because there is no need to transport big data to or from device.

Full speed was introduced in USB 1.1 generation and brought data rates of 12 Mbit/s. The latency and the bandwidth is guaranteed for full speed devices. It means that the amount of data that can be transferred during one second and the amount of time it takes a packet to travel from source to destination is fixed. This property is used in microphones so that a good sound quality can be achieved.

The hard drive devices easily become a bottleneck with full speed standard, whereas with high speed standard it becomes more comfortable and usable. High speed supports data rates of 480 Mbit/s and is available in the USB 2.0 generation. Over the time the demand for higher performance in connection between the computer and the miscellaneous peripherals grows constantly. The super speed USB 3.0 standard was introduced in 2008 and adds an even higher transfer rate of 5 Gbit/s to match those needs. In 2013 the USB 3.1 technology was introduced, the super speed plus brings a data transfer rate of 10 Gbit/s.

2.1.1 Topology

The discussion will proceed with the USB topology. This will bring more precisely understanding of information covered in the following chapters, in particular the endpoint configuration. The physical USB topology presents devices that are physically connected by a tiered star connection model. The hub acts as an attachment point which is called port. The host communicates with each physical device as if it was directly connected to a root hub. The host stays always aware of the physical topology to support connections and disconnections of devices.

The master-slave communication model defines the principle how USB technology works. The USB host plays a master role and is responsible for initiation of communication. Device responds to the requests from the host and plays a slave role. Host is capable to have up to 127 devices connected simultaneously. Each USB bus can have only one master which is called host controller. The host controller in association with a root hub is the low level piece of hardware that operates the USB master-slave bus protocol.

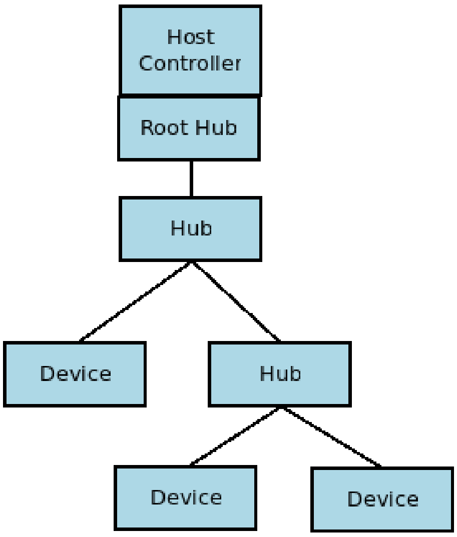

Figure 1: USB Physical Topology

As it is shown in Figure 1: USB Physical Topology, the root hub acts as an interface to the host controller. The host controller controls the devices on the bus in a polled manner. Only one device at a certain moment can exchange massages on the bus. The other end of a USB network is the device end also called gadget. The gadget functionality within Linux kernel simply refers to an ability to operate as a device in the slave mode.

On the other hand the logical topology describes how USB subsystems like host, hub and devices which make up the physical topology are interconnected to each other and communicate controlled by the host software. The information about each USB device is described by a number of descriptors which are data structures with a defined format. Every device must have a single device descriptor because the host starts the enumeration process by polling it. Device descriptor allows the host to differentiate what kind of device is plugged in at a moment. There also must be at least one configuration, but if there are many then the device can play different roles defined in each configuration. When a device is plugged into the host, the host asks for information from the device and assigned an address that will be used for further communication. The unique address is assigned when device is attached and gets power. The enumeration process is performed automatically and indicates therefore that the operating system gets specific information from a newly connected hardware in form of descriptors. Descriptors are data blocks of few bytes. A special role in an enumeration process plays a device descriptor. Among many specifications of a device, it also contains three IDs which help to find an appropriate driver. The host searches for an appropriate driver with assistance of information retrieved from the descriptor. Each manufacturer has its own VendorID and ProductID. If a device belongs to a certain class then the device descriptor contains a special ClassID. When a device cannot be assigned to a particular class then the driver will be found depending on VendorID and ProductID. If the host identifies the class then a proper class driver will be loaded, this functionality is usually adopted in operating system so that an extra driver installation is not needed. This feature represents the Plug and Play mechanism for the new connected devices. Many drivers are already in the system and are found automatically, however, under certain circumstances it may happen that during the initial connection of a device, it would be required to install an appropriate driver manually. The host driver with the host controller is responsible for configuring USB devices when they appear in the topology. Multiple configurations are sometimes packaged together in what looks like a single physical device. Such a device configuration it is called compound device. For example, an Android smartphone plugged into a host has a configuration as data storage and a configuration which enables the communication over Android Debugging Bridge.

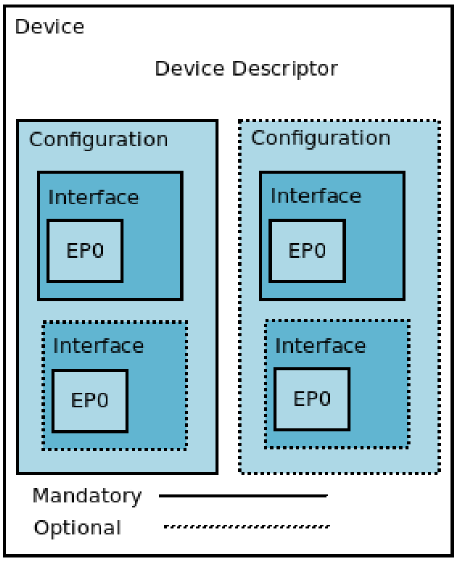

Figure 2: USB Device Configuration

This section illustrates in textual form what can be seen on the Figure 2: USB Device Configuration. Every configuration defined in the device descriptor has a configuration descriptor. A configuration descriptor contains the number of interfaces available for each configuration. There must be at least one interface implemented. Each interface described by a configuration descriptor has an interface descriptor. An interface descriptor contains one field specifying how many endpoints are defined. To make it more clear, an endpoint defines a unique location of a USB device and provides source or sink points for communication. There is one endpoint which has always to be defined, it's called endpoint 0. An endpoint is the logical element that software communicates with during USB device operation. Each endpoint specified by an interface descriptor contains an endpoint descriptor, endpoint 0 excluded. The endpoint descriptor declares parameters needed for communication such as endpoint address and also various attributes describing the features of data transfer for each endpoint.

The endpoint 0 has the address 0 and is a special endpoint that all devices must implement. The host uses it to initialize and gather information from the connected device. The endpoint 0 is not included in an interface descriptor.

The information about interface class, subclass and protocol is also included in an interface descriptor. Except of endpoint 0 which is used to control the transfer, the USB gadget can have up to 31 endpoints, 15 IN endpoints and 15 OUT endpoints. This will be an important issue in the next chapter where the practical implementation will be discussed.

The endpoints can be configured depending of the transfer type. There are different transfer types used by USB protocol. The USB specification, a document which describes all levels of the USB software stack, defines different transfer data types. Depending on kind of communication it is possible to use the type which best meets the requirements. Below different transfer types are covered, this information is needed while configuring the endpoints of Gadgetfs module from the user space program.

Control transfer is intended to support status commands and configuration between the host and device, so it is used for non-periodic communication. Each USB device has at least one control pipe which provides access to the status and control information. The control transfer is bidirectional and requests the configuration information to and from the device using the bidirectional endpoint 0. The endpoint 0 is used always in control transfer mode.

Isochronous transfer is used for a time dependent data transfer like video streams and telephony. This type of transfer allocates a portion of bandwidth to ensure that the data can be delivered to the receiver at a desired rate. The isochronous transfer is used by full speed and high speed devices, low speed devices are not included. The isochronous transfer is periodic and unidirectional.

Interrupt Transfer is used for devices to send and receive non-periodical, small data packages. An example is a computer mouse, the data transfer rate is guaranteed but what cannot be guaranteed is that the transfer will occur a defined moment, but rather it will occur within the defined period of time. This transfer type will reattempt the transmission of data in the next period if there was an error on the bus.

Bulk transfer is typically used transfer type for large, non-time-sensitive data. It takes up all the bandwidth that is available after the other transfers have finished so that if the bus is not available at the moment then the transfer may be delayed. The bulk transfer is unidirectional, it uses any available bandwidth and provides error check mechanism with retry of attempts.

2.1.2 Host Functionality

In the preceding sections the general information about USB was covered. Here the host side of the communication process will be presented.

In the description of computer hardware and mobile devices there is often a mark that convey that the current hardware supports USB. This is not quite true. As a matter of fact a USB controller in computers and a USB controller in mobile devices are different objects. In the first case there is a USB host so that the other peripheral devices can be attached to it. In the second case it's a USB device which can be attached to a USB host. Additionally, a USB hub can be attached to the host so that a number of USB ports can be increased which makes it possible to attach multiple devices.

Linux provides reliable drivers for all host controller standards and drivers for almost all device classes. The different device classes are shown in the Table 1: Driver Classes, below. For all USB device classes there are drivers already present on the host side and only in the rare cases it is necessary to implement a device driver himself.

Class	Example	Host Driver
Audio	USB Sound	yes
CDC	network	yes
HID	keyboard, mouse	yes
Mass Storage	USB flash drive	yes
RNDIS	network	yes
Serial	RS-232 to USB	yes

Table 1: Driver Classes [29],[24]

Gadget Framework has no built-in CCID class driver. The open source project libccid provides source code for a generic USB CCID driver which can be used together with the Gadgetfs to provide CCID interface to the host.

A minimalistic host functionality can also be implemented by device. On-the-Go(OTG) technology was introduced to enable support of USB devices for minimalistic USB host capability to allow a point to point communication. Such function allows embedded devices like digital cameras, mobile phones and printers to be connected to each other directly. So for example, a digital camera acting as a host can be connected to a printer so that it allows to print images from the camera. Otherwise if a camera is connected to the host computer the camera acts as a device so that it is possible to edit, delete and copy pictures. There is no need of coping pictures from a digital camera to the computer and then from the computer to the smartphone. With the help of OTG it can be done without an intermediate step. Another useful case is to connect the USB flash drive directly to a smartphone. For that reason two factors needed to be considered. First, at least one of the gadgets has to support the OTG standard that is not the case by many commercially available devices, and the second, there has to be an USB OTG adapter cable so that both gadgets can be connected together. Whether the device supports OTG or not it can be usually identified by a USB Logo with a green arrow with "OTG" on it. At the beginning the purpose of USB interface was used only to connect peripheral devices to computers or notebooks, whereas a USB host controlles the whole transport on the bus. Obviously the described technology has no options to connect two devices directly via USB interface, consequently, it was not possible to connect, for example, a digital camera with a printer. So no one will be impressed that such a technology as OTG was soon developed. This allowed to implement all the connections described earlier. The OTG specifications allows each of the connected devices to become master on the bus. It is also worth mentioning that the USB OTG controllers consume less power compared to the USB 2.0 and upper standards. The OTG controller has also a mini USB port for connections, this makes them a perfect solution for usage in embedded devices such as smartphones. The OTG technology can be considered as a rival product for technologies like Bluetooth and FireWire. At the moment of connection the host is called A-Device and the gadget part is called B-Device. Moreover, the roles can change on the fly. For example, when a printer and a camera are connected in wrong direction so that the printer acts as a host and the camera as a client then the roles will switch what is done by Host Negotiation Protocol. The OTG framework is responsible for handling and responding to the OTG protocol negotiations. The OTG driver consists of both the device driver framework and the host framework so that OTG framework change roles depending on the situation.

2.2 Gadget Modules

The device side is unfortunately not so accurately standardized in accessing the controller mechanism as the host side. The gadget framework gives the device drivers a certain amount of modularity. It should be considered that not all Linux system are capable to use the device configuration. Simply because computers and notebooks have only the USB host hardware chip. On the other hand many embedded systems include a USB device controller besides a USB host controller, an example of such system is a development board BeagleBone Black. It makes it possible to use BeagleBone Black as a host with a keyboard connected to its USB host port and at the same time to use it as a gadget connected to the host computer. A lot of peripherals like smartphones and printers include a USB device controller hardware by default. Some of such devices provide its own power supply other rely on the host.

The USB gadget framework is used to implement the peripherals and can be divided into three layers:

1. USB device controller (UDC) driver is a layer that talks directly to the hardware. The implication of this statement is that different developers of device controllers need to provide their own device controller drivers. The UDC driver communicates directly with the USB controller chip on the device. Depending on the type of the controller many tasks are done directly in hardware. In the case of Samsung Nexus S which is used in this research, it is Exynos device controller with s3c_udc_otg device controller driver. The UDC driver provides a certain number of endpoints which are associated with a queue to send and receive data.

2. Gadget drivers implement USB functions and are hardware independent because they rely on UDC driver. This research will introduce gadget drivers in details in a separate chapter below.

3. Layers, such as file systems and network. Those systems work with data provided by gadget drivers which is received or sent to the host via UDC driver. In most situations there are several layers which are interconnected.

The control of communication is spread between the USB device controller driver and gadget driver. The UDC driver is responsible for those functions which are frequently used by the hardware. The purpose of a device controller driver is to transfer the data between the gadget driver and the controller hardware, to manage the input and the output queues of various endpoints. The main part of the functionality is implemented in a gadget driver. It covers the management of device configurations, changes of device states and the configuration of device descriptors.

The Table 2: Device Controller Types, shows the different types of full speed USB device controllers supported at the moment by gadget framework:

Module	**Vendor**
S3C2410 ARM	Samsung
OMAP USB Device Controller	Texas Instruments
net2280	NetChip
at32ap7000	Atmel
TC86C001 "Goku-S"	Toshiba

Table 2: Device Controller Types

The gadget driver is logically arranged above the UDC driver and picks up on its API. This work uses the gadget driver Gadgetfs with the source code file inode.c. The Gadgetfs module uses the API of the lower level UDC driver s3c_udc_otg.c. There are gadget drivers for almost any device classes which are defined by the USB standard. If the gadget driver for USB mass storage is loaded then the device can be used as a storage device such as a flash drive. If the network gadget driver is loaded then it is possible to establish a network connection via a USB protocol. The described gadget drivers are mutually exclusive therefore only one gadget driver may be loaded at a moment. However, it is readily possible to remove the driver and then to load another. The modularity of the kernel allows the replacement of the device drivers. Depending on kind of loaded gadget driver further layers on top of it are used. For a mass storage it is a file system, for a network gadget driver there is a network stack and for a serial gadget driver there is a serial subsystem.

The gadget drivers which are described below implement a single USB functionality.

2.2.1 USB Gadget Zero

With the help of this module the functions of the USB subsystem on both the client and host sides can be fully tested. Therefore, primarily the gadget driver g_zero is used with its counterpart the usbtest driver on the host side. The Linux machine is required as a host which contains the USB driver usbtest.ko. Once the system is connected to the host nothing happens at first yet. For as long as no unique USB function is determined by the gadget driver the USB device must be invisible to the host. This is achieved by switching off the pull up resistor. When a gadget driver, for example, g_zero is loaded then the pull up resistor is set by the UDC driver to 3.3V. So the host can start the enumeration. During the enumeration process first the device descriptor is picked up by the host. The host then performs a reset. Immediately after a reset the USB address is assigned to the device. Only now all descriptors are polled from the device. Among other things the device descriptor is transmitted one more time. The g_zero defines a device descriptor whose IDs make usbtest module to be loaded on the host side automatically.

2.2.2 Serial USB Gadget

Another way of communication via USB offers the gadget driver g_serial. Serial gadget brings tty interface so that a serial communication between the host and the gadget becomes possible. After enumeration and loading of an appropriate driver on the host, the virtual serial ports are set up on the host and client. This functionality becomes very useful when a developer has software which runs on top of serial interface with serial protocol. The serial gadget provides the solution to use USB protocol with serial interface. With the help of this module the software stack which is build up depending on serial interface will function further.

2.2.3 USB Network Gadget

A USB device with the network gadget module g_ether loaded will be configured as an ethernet device. A network connection between the USB host and device becomes enabled. Since g_ether contains a device descriptor with appropriate ClassID the host loads the usbnet driver on its part. On both devices additional network interfaces have been created. When TCP/IP properties are set up on both sides the data can be exchanged between them. This driver can operate as a counterpart with Linux and Windows operating systems.

2.2.4 Storage USB Gadget

The storage gadget module g_file_storage uses a file to store the information. The mass storage class is generally not designed for multiple and simultaneous accesses. It has no coherence protocol to inform the communication partners about the changed files. The USB host can therefore have a very different picture of the contents of the memory on the client. It should be avoided to access the data from the host and the client simultaneously. The host mass storage driver can be found on both Windows and Linux via the provided ClassID. An extra driver installation on the host is just not necessary as it is usual with a USB flash drive. Once the enumeration is complete, the host usually binds the content in its directory hierarchy, on Windows it appears as an additional drive and on Linux as a directory in /media directory. Now, any data can be added or deleted.

2.2.5 Gadgetfs

The gadget driver framework extends the control over the virtual file system into the user space. It becomes possible to access the gadget from it. The Gadgetfs is not available by default and must be configured during the kernel configuration.

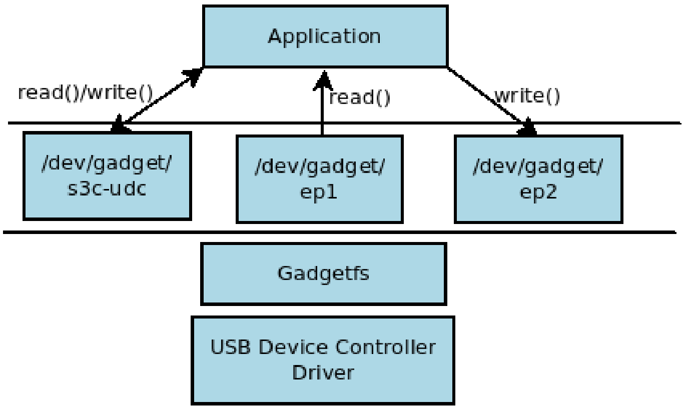

Figure 3: Gadgetfs read/write

The Figure 3: Gadgetfs read/write, shows schematically the ability of the program to use bulk sink-source functionality by reading the data from the endpoint 1(ep1) and by writing the data to the host using endpoint 2(ep2). The ep* files are created for each endpoint during the driver binding. If an endpoint is configured with an OUT transfer direction, so it possible only to read it. If an endpoint is configured with an IN transfer direction, so it is possible only to write into it.

The s3c-udc file serves to access the endpoint 0 where the device configurations are written. This enables the the binding of module with USB controller. In particular, by writing the descriptors on Samsung Nexus S in the file /dev/gadget/s3c-udc the device gets configured. The device stays active and ready for enumeration until file descriptor is closed. It is possible to obtain the events by polling endpoint 0, see the Table 3, Gadgetfs Events. The Gadgetfs provide a file system based access to the device controller driver which in its turn controls the device controller hardware.

Event	**Description**
GADGETFS_CONNECT	The event is generated when the device setup is successful.
GADGETFS_DISCONNECT	The event indicates disconnection of the device from the gadget driver.
GADGETFS_SETUP	The event is generated when the gadget driver requests a device setup.
GADGETFS_SUSPEND	The event is generated when the gadget driver requests a device suspend.

Table 3: Gadgetfs Events

The Table 3: Gadgetfs Events, shows the events which the user space program polls to obtain the driver status. These events are used in the user space program to check the current state and enable the right control flow of Gadgetfs. It becomes possible to handle different situation like errors or unexpected states. They are very useful by debugging Gadgetfs, it became possible to see in the log files the states of the Gadgetfs module.

2.3 Embedded Android

The chosen platform for this work is Samsung smartphone with Android operating system. Android presents a operating system for mobile devices based on Linux. Given OS was developed by Android Inc. which was bought by Google afterwards. Although this OS is based on Linux, it is not possible to use all Linux applications because of the absence of various standard libraries and also because some libraries were developed entirely by Google.

In 2013 estimated Android platform market share claimed nearly 79 percent. Today it can be seen that Android has quickly reached the top of the smarphone selling, but there are signs that the growth is cooling off. In embedded world it became a de facto standard for a vast majority of embedded devices. There are signs that it might displace the classic embedded Linux. An entire ecosystem therefore rapidly grows around Android. System-on-Chip (SoC) manufacturers such as Qualcomm, Freescale, Nvidia and ARM have added Android support for their products. On the other hand, phone and tablet manufacturers such as Samsung, HTC, Sony, LG are constantly increasing number of Android devices. Many of those projects are done by forking the official Android source code release with the purpose to create an own Android distribution with custom features. CyanogenMod and Linaro are typical examples of Android projects enhancements which provide own Android custom images.

Android was made in fact to run on all architectures supported by Linux like ARM, MIPS, PowerPC and x86 that means on any hardware that runs Linux. But beyond being able to run Linux, there are few other hardware requirements for running Android. Apart from the logical requirements of having some kind of interaction mechanism to allow users to use the input interface and some kind of display, there are also needs to provide enough memory to store Android image file and a sufficiently powerful CPU to give the user a decent experience with the device. By implementing the Gadgetfs module on the phone there has to be a possibility to bring back the standard configuration and to allow the user to make use of the predefined settings.

Hardware support in Android is significantly different from the approach found in Linux kernel. The usual way in Linux to provide support for new hardware is to create a device driver. The driver can be built as a module and be loaded at a runtime or as an built-in kernel module so that the corresponding hardware is generally accessible in the user space through files in /dev. In Linux there are three types of devices: character devices also known as stream devices, block devices and the network devices. This allows various software stacks to be built on top of files in /dev to interact with the hardware. Android approach is very different. Android software stack relies on shared libraries provided by manufacturers to interact with hardware, instead of standard /dev entries. Android uses on what is called a Hardware Abstraction Layer. Generally speaking, a Hardware Abstraction Layer can be considered as the hardware library loader along with the header files defining the various hardware types. Android does not specify how the shared library and the driver should interact. The Hardware Abstraction Layer defines only the API provided by the shared library to the upper layers. Android has no libc library it uses reduced version named bionic. The USB controller driver acts as a hardware abstraction layer for the

USB device controller. It exports the hardware to the layers above. The Linux USB controller driver implements hardware specific routines that allows access to the memory space and registers.

2.3.1 System on Chip

To be able to compile the right kernel configuration it was necessary to identify the special features of the hardware of the given smartphone. In order to proceed in the research it was important to analyze Samsung Nexus S hardware, its specification and configuration. This chapter will cover the common features of a System-on-Chip so that it will be possible afterwards to turn to the special Exynos platform designed by Samsung.

A System-on-Chip (SoC) is an integrated circuit that includes various parts. Single monolithic systems include a processor, a bus and other elements. Integrated circuits are used in a wide range of electronic equipment like portable handheld devices. In general, System-on-Chip technology is the ability to place multiple subsystems on a single semiconductor chip. It typically uses a powerful processor and is capable of running software such as the desktop versions of operating systems. The SoC design usually consumes less power, has lower costs and higher reliability then the multichip systems that they replace. The SoC typically consists of a 32-bit CPU cores with a separate core for USB. The SoC are optimized for efficient power consumption because in the most cases SoC has separate power supply. The typical components are microcontrollers or microprozessors, memory blocks which include ROM, RAM, EEPROM and flash memory. Another elements of SoC are peripherals like real timer controller and external interfaces including industry standard such as USB, FireWire or Ethernet.

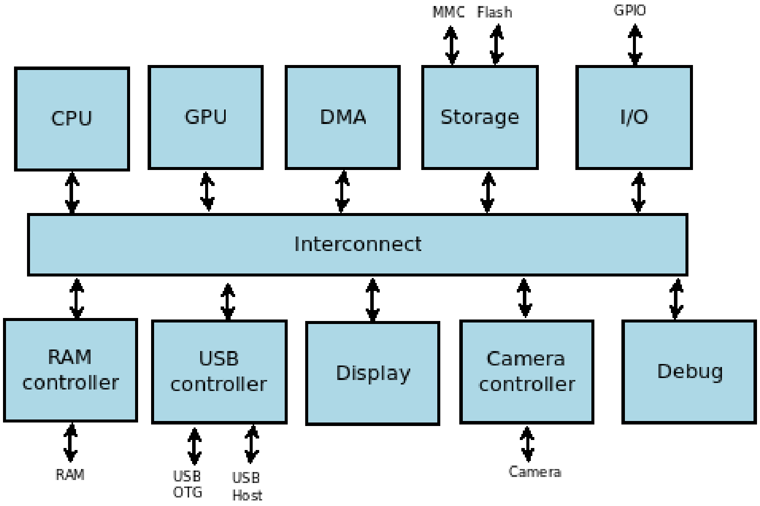

Figure 4: SoC Structure

The Figure 4: SoC Structure, shows that SoC is much more then only CPU cores. A SoC is to some extent a circuit board with a bus interconnecting a variety of different components. Because there is no standard, the manufacturer decides over the number and complexity of the components. Although most SoCs include a similar set of basic components, the SoCs are produced by different manufacturers. It is to mention that not all components within a SoC operate at the same clock speed, so CPU may work in gigahertz frequency and GPU with several hundred megahertz only. The GPU is responsible for accelerating the rendering of graphics to the device display.

A big advantage of SoC is its tiny and universal structure. A tiny structure of the system is due to placing the components on the same circuit. The universal feature is based on the possibility of SoC to be used on various devices with the minimalistic changes.

2.3.2 Samsung Nexus S

One of the overriding aims of this work was to analyze the USB controller on SoC. Samsung Crespo, also known as Google Nexus S is one model out of the production line of Nexus devices. The full line is shown below:

Version	**Type**
Google Nexus One	smartphone
Google Nexus S	smartphone
Google Galaxy Nexus	smartphone
Google Nexus 7	tablet
Google Nexus 4	smartphone
Google Nexus 10	tablet
Google Nexus 5	smartphone

Table 4: Nexus Production Line

The table shows generations of Nexus devices. It is a well-known fact that this line was developed by Google developers therefore the devices have no predefined mobile network operator and no additional software packages preinstalled. Maybe the most advanced issue in this production line is that the updates are coming very fast. In other devices the vendor includes custom changes so that an update to a newer Android version takes much longer.

As an example, Android 4.1 Jelly Bean was introduced in June 2012 and the very same day the Galaxy Nexus users could update to a newer version. Samsung Nexus S got an update to Jelly Bean one month later in the July 2012. Another positive aspect of Nexus S is its medium size which allows good control of the whole display surface with one hand. A noteworthy issue is a good price performance ratio in the mid-range price category. The display of the phone is bright, it offers comfortable usage experience also outdoors. The negative sides of the phone are 16GB of internal storage without a possibility to extend it by using SD card. The battery usage leaves something to be desired so that about 6 hours of hearing podcasts or music can be achieved. Nexus S uses a Exynos 3 platform. Exynos is a series of ARM based System-on-Chip developed and manufactured by Samsung Electronics and is a continuation of Samsung's earlier S3C and S5P production line.

In the following table there is a listing of Nexus S hardware features with more technical details.

Samsung Nexus S	
Model	Exynos 3 (previously S5PC110, Hummingbird)
Platform	Samsung S5PC110
Release Date	2010 December
Vendor	Google
Manufacturer	Samsung
Codename	crespo
Semiconductor technology	45 nm
RAM	512 MB
Internal Storage	16 GB
CPU instruction set	ARMv7
CPU	1.0 GHz, single-core ARM, Cortex-A8 based CPU Subsystem with NEON
GPU	IT PowerVR SGX540 @ 200 MHz; 3.2 GFLOPS
Memory Technology	32-bit dual-channel 200 MHz LPDDR, LPDDR2, or DDR2
Camera	8 M pixel for scaled and 16 M pixel for unscaled resolution
USB	On-chip USB 2.0 OTG supporting high speed (480Mbps, on-chip transceiver) On-chip USB 1.1 Host supporting full speed (12Mbps, on-chip transceiver)

Table 5: Hardware Specification

The Table 5: Hardware Specification, shows the hardware details of Nexus S device. The most important features which are needed for the configuration of the Android kernel are the following: at first, to be able to compile kernel the device platform should be known; secondly, it is also significant to know the USB hardware characteristics of the device. The Nexus S has OTG capabilities that allow to set up device as a host or a gadget. The platform used by Nexus S is Samsung S5PC110. Knowing this fact and also knowing that the codename of the device is crespo, allows to choose the right kernel configuration file automatically which includes a working kernel configuration. It is also necessary to know that the model name changed during the time so that Exynos 3, S5PC110 and Hummingbird are names of the same platform.

3 Practical Approach

This chapter will build upon the knowledge discussed in the Chapter 2, providing practical implementation and evaluation of USB functionality with Gadgetfs module. The introductory part will provide a configuration of Gadgetfs module from the user space using program usb.c. Depending on the device controller some changes are needed in configuration of Gadgetfs module to make it work with various SoCs. The configuration will provide the settings for different descriptors. Next step will handle the execution of Gadgetfs on Linux system to test its functionality. In almost the same manner the research will proceed in the analysis of communication on Android, tested directly on the host computer by using an Android emulator. The problems which arise by using an emulator for testing the gadget functionality will be presented. In addition the OTG configuration of different modes on Samsung Nexus S will be introduced. A sequence of measures is required to set up the Gadgetfs as a loadable kernel module on Nexus S device, to enable the right kernel configuration. In other case the device just won't boot. Also the ways to deal with the bricked device will be mentioned. It is necessary to compile kernel and user space programs for ARM architecture with a cross compiler. Additionally, the results of flashing different kernel versions, tested for the purpose of this inspection will be presented. The examination shows the problems and achieved solutions using Gadgetfs on Exynos SoC. The work will proceed with the evaluation of debugging information resulted in different approaches tried out during an embedded kernel debugging process. The short overview about the same functionality of Gadgetfs module will be investigated with the help of BeagleBone Black development kit.

3.1 Gadgetfs Usage

The right point to start to unfold the accomplished work is the usb.c program which is the official user space test program for Gadgetfs module. This program decides dynamically which device controller is present on current hardware. Depending on this information it configures the device by writing the correct configuration descriptors and device descriptor to the endpoint 0. When the endpoints for IN and OUT transfer are configured then the two threads are provided to read and write the data to or from the endpoints. One extra thread controls the endpoint 0. Some gadget framework structures became initialized in the user space and therefore allow a dynamic configuration. It requires the cross compilation of the user space program and its execution on the device. In contrast to the preceding methods, a modification of the kernel module would bring the disadvantage of the kind that it would need to be recompiled against the kernel source tree. The possible bugs in kernel module could bring the system to a crash.

The first step was to change the user space program and to test its functionality. Following options were in assortment. The fist one, to set up the dummy_hcd module which simulates the USB gadget functionality on Linux. The second approach was to get it working on the Android emulator. The last alternative was to work directly with the Nexus S hardware. It is obvious that the first two alternatives were favored because of the possibility to work directly on the host computer.

The dummy_hcd allowed to test module on already approved to work system like Linux to be able to understand the functionality and to test the configuration and communication. The module dummy_hcd exposes a device side of USB gadget API and simulates requests to a Linux host controller driver. Thus, it allows testing of the gadget modules directly on the host by loading the dummy_hcd and the gadget driver. To provide the gadget modules along with dummy_hcd module on Linux, it was necessary to recompile Linux kernel and to enable them during the kernel configuration. This was successfully managed therefore it was possible to load Gadgetfs module and simulate the communication via dummy_hcd. The data could be exchanged by writing into the endpoints on one side and by reading the data on the other side. To be able to access the USB interface on the host side the approach was chosen with the usage of usb-skeleton host module. This module is a part of the kernel source tree so that by modifying this module a device file on the host side could be created. The device file allowed reading and writing into it to exchange the data. It was possible to test successfully all gadget modules described in Chapter 2.2.

This approach turned out to be successful and next step was to test it on Android. In order not to forget the main goal to enable communication between the host and Android, the second step was to try it out on the Android Emulator which is provided with Android SDK. The special kernel for an emulator with the codename "goldfish" was compiled with enabled gadget drivers support. The procedure of testing gadget modules on emulator was proved to fail because Android emulator is build on the basis of out-dated version of Qemu emulator and has no support for USB device controller. Accordingly, it was not possible to test gadget module on host via an Android emulator. This brought this research to the necessity to start to work with the real hardware based on Exynos SoC, see Chapter 2.3.3.

At that time the research had to deal with real hardware. The following changes were done to the user space program to make it run with an Exynos SoC. The code is part of the program where the initial configuration of gadget is specified. There are data structures to be set to configure the device. One of the fields is device name which depends on UDC driver. The s3c_udc_otg.c is the source code file for the USB device driver for Nexus S. Correspondingly, the created device file in directory /dev/gadget has a name s3c-udc. Depending on device controller there are various aspects to set up. It has to be known in advance whether the device controller supports the high speed and what are the data transfer types of the endpoints. The configurations of endpoints which are defined in user space have to be equal to predefined endpoints in UDC driver.

The changes which are needed in usb.c user space program to work on Exynos SoC are shown below:

```
} else if
(stat (DEVNAME = "s3c-udc", & statb) == 0) {
HIGHSPEED = 1;
device_desc.bcdDevice = __constant_cpu_to_le16 (0x0100);
fs_source_desc.bEndpointAddress
= hs_source_desc.bEndpointAddress
= USB_DIR_IN | 2;
EP_IN_NAME = "ep2-bulk";
fs_sink_desc.bEndpointAddress
= hs_sink_desc.bEndpointAddress
= USB_DIR_OUT | 1;
EP_OUT_NAME = "ep1-bulk";

source_sink_intf.bNumEndpoints = 3;
fs_status_desc.bEndpointAddress
= hs_status_desc.bEndpointAddress
= USB_DIR_IN | 3;
EP_STATUS_NAME = "ep3-int";
}
```

The listing shows that Exynos SoC needs high speed to be activated. In addition three endpoints are created. Two of them the bulk endpoints for IN and OUT communication and the third is the control endpoint.

3.2 Cross Compilation for ARM

After the configuration of the user space program has been accomplished, the next step was to test the functionality on the Samsung Nexus S smartphone. It was needed to set up the cross compiler environment to be able to compile on host x86 architecture for an ARM architecture. There is still a possibility to compile its own cross compiler, but a precompiled one was sufficient for the purpose of this work. It was necessary to cross compile the Android kernel and the also user space program for ARM architecture. This part of the work was already handled during the IT Security Workshop 2012. The unsolved problem in the workshop was that it was not possible to compile the user space program for an ARM architecture because of non-present header files. The right solution was to use a different cross compiler for ARM architecture and static or dynamic linking. Noteworthy issue was the option to link program dynamically against Android system libraries by setting the right paths. Another option was to use static flag during the compilation process. The static option links the program statically so that it does not require dependencies on dynamic libraries at runtime in order to run.

After a successful cross compilation of the user space program and the kernel it was necessary to flash the kernel image to the smartphone to be able to test it. The important issue is to use the right version of kernel and ROM which is a firmware for Android. They need to match because otherwise the smartphone just decline to boot properly. When the ROM is flashed to incompatible device type the smartphone gets bricked. It is possible to recover the bricked smartphone to its original stated by doing factory reset using ClockworkMode recovery image or by reflashing the stock or custom ROM.

As already discussed it is advisable to start the usage of the gadget framework by testing it with g_zero module. This module supports only basic functionality so that the standard test cases can be automatically tested. When device is configured with g_zero module the host enumerates the gadget and it is possible to read and write to it. The module g_zero writes back automatically every data which host wrote to the device. The tests with this module on Nexus S phone ended successfully.

It was also possible to load and positively test the following kernel modules on Nexus S: serial gadget, network gadget and mass storage gadget.

3.3 Gadgetfs Debugging

The execution of user space program on Nexus S in connection with Gadgetfs resulted in a system crash so that smartphone rebooted. Further explanation will describe the steps which led to a bug fix.

The UDC driver for Exynos SoC defines endpoints which differ in direction and transfer mode. The endpoints which are defined and configured in user space have to correlate with those endpoints. If the certain endpoint in UDC driver is defined as an IN endpoint with bulk transfer mode then this endpoint cannot be redefined in user space as an OUT endpoint or an endpoint with different transfer mode such as interrupt or isochronous. The various configuration of endpoints in user space were tried out but without any success to prevent the system crash. The usual way to configure Gadgetfs module is to mount Gadgetfs in /dev/gadget and then to execute the user space program which configures the USB interface. As it is shown in the listing below, after mounting the Gadgetfs in kernel log files, the message appears: "s3c-udc: bind to driver nop -> error -120". After the execution of user space program the Android phone got kernel panic and rebooted. The first idea was that the preceding error message is the cause of the trouble because as the massage says the USB device controller driver could not be bound to device controller.

```
mkdir /dev/gadget
insmod gadgetfs.ko
mount -t gadgetfs gadgetfs /dev/gadget
ls /dev/gadget
s3c-udc
dmesg
[299.539226] Gadgetfs: USB Gadget filesystem, version 24 Aug 2004
[353.895836] s3c-udc: bind to driver nop -> error -120
```

After debugging it became clear that the driver was developed in such a way that it started binding process only after execution of user space program. The error message -120 is just a reminding that the Gadgetfs needs the execution of the user space program to start the binding process. That was obviously not the root of the problem with kernel crash and the rebooting was still unsolved.

To be able to work with gadget modules the device needs to be configured in the device mode. The configured kernel used OTG functionality. It was important to be able to switch between the host and the device functionality on Android. By configuring the kernel with support for OTG functionality it was possible to switch the modes. When OTG functionality is configured on Samsung Nexus S then the directory

```
/sys/bus/platform/drivers
```

should contain the following entry: dwc_otg. The dwc_otg is a sign that OTG support on the Android is enabled. To change modes it is necessary to use root privileges. The description of configuration of host, gadget and OTG modes on Nexus S device are listed below.

Host mode:

```
write 0 > /sys/devices/platform/dwc_otg/setmode
```

Device mode:

```
write 2 > /sys/devices/platform/dwc_otg/setmode
```

OTG mode:

```
write 1 > /sys/devices/platform/dwc_otg/setmode
```

The modification is also possible by installation of a Google Market Application which controls the different modes.

Due to the fact that the device with Gadgetfs module loaded has no additional interfaces to debug it remotely, so it requires to work with the terminal directly on Samsung Nexus S touchscreen. It was soon clear that the set of commands offered by the ToolBox is insufficient. The decision was made to cross compile BusyBox for ARM architecture to supplement the command set and to use it instead of ToolBox.

As already mentioned, the consequence of user space program execution was the restart of the smartphone with a kernel panic error message. The analysis of the kernel dump was the starting point in the necessity of debugging the kernel panic on embedded device. Additional obstacle in the debugging was the issue that by loading the Gadgetfs module it was not more possible to use ADB or serial connection to be able to access the smartphone. The way to examine the kernel log messages was the option to do it only on the smartphone display or to reload the phone without Gadgetfs module to be able to copy the debug files to the host computer over ADB interface.

The decision was made first to try out another Android kernel version because of the bug fixes which are included in the newer versions. The following kernel versions were tested on a stock ROM which is a firmware for Android and on a custom CyanogenMod ROM:

- Android Gingerbread 2.3 with the kernel version 2.6 was tested.
- Android Ice Cream Sandwich 4.0 with kernel version 3.0 was tested.
- Android Jelly Bean 4.2.2 with kernel version 3.0.50 was tested.

Unfortunately the usage of another kernel version did not brought better results because the device controller driver for Exynos SoC did not changed from version to version and was not affected by a bug fix. The Exynos SoC can be considered as already pretty old. It is over five years on the market so that the bug fixes on the linux-usb mailing list concern more recent systems.

To be able to proceed with this research it was necessary to find the cause of the buggy behavior. The most useful technique to debug the kernel on an embedded device turned out to use printk() commands. This command allows to log the kernel function messages with parameters to a log file. The log file can be analyzed after the kernel crash with the aim of finding the last called function. As described in previous chapters the gadget framework consists of 3 layers. The starting point for debugging was the user space program usb.c. It could be simply modified by debug information and executed on embedded device. The program uses the API of Gadgetfs and it turned out that the function which caused the the bug was placed in the lower level. It was necessary to include debug information to the inode.c file which represents the Gadgetfs module. As already discussed the Gadgetfs uses the API from USB device controller driver so that it was also necessary to add the debug information the device controller driver which is represented by the file s3c_udc_otg.c. To be able to analyze the kernel modules and its interdependence, it was necessary to become acquainted with data structures used by gadget framework in special in the gadget and in the UDC driver. At the end, it was possible to allocate the function. But in order to proceed further, it is worth to remember that the S5PC110 chip which is used in Nexus S is a subsequent model after S3C2410 chip. The device controller driver which is used for S5PC110 is s3c_udc_otg and has to be compatible with it. So it happened that that the number of declared endpoints in header file and the number of initialized endpoints in USB device controller driver was different. The consequence was that some endpoints were not initialized. An attempt to dereference the uninitialized pointer led to kernel panic. The patch of the misbehavior was to reduce the number of available endpoints in the header file from 15 to 14 or to add the non-present endpoint 15 into the device controller driver.

Here comes the kernel patch for USB device controller driver s3c_udc_otg.c. It brings the initialization of endpoint 15 which has to be defined.

```
1201,1216d1179
<  .ep[15] = {
<    .ep = {
<   .name = "ep15-bulk",
<   .ops = &s3c_ep_ops,
<   .maxpacket = EP_FIFO_SIZE,
<   },
<    .dev = &memory,
<
<    .bEndpointAddress = USB_DIR_OUT | 0xf,
<    .bmAttributes = USB_ENDPOINT_XFER_BULK,
<
<    .ep_type = ep_bulk_out,
<    .fifo = (unsigned int) S3C_UDC_OTG_EP15_FIFO,
<    }
```

The user space program uses only the first three endpoints to exchange data. The endpoint 15 is never used by the user space program. The endpoint 15 defines a bulk endpoint with the OUT direction from the host's point of view.

After applying the patch it was possible to execute the user space program. The user space program created 15 endpoints excluded endpoint 0 in /dev/gadget directory without causing the kernel panic. The following step was to plug the device into the host so that the host can start enumeration process. The result was another bug. This bug disrupts the enumeration so that the device is not recognized by the host.

3.4 Host Enumeration

Starting the user space program on Samsung Nexus S with verbose output resulted in the following message.

```
#> ./usb
/dev/gadget/s3c-udc ep0 configured

** Wed Feb 26 10:02:02 2014
SUSPEND
CONNECT high speed
DISCONNECT
CONNECT high speed
DISCONNECT
CONNECT high speed
DISCONNECT
CONNECT high speed
```

The host kernel log file below shows the attempts of enumeration. The enumeration report error messages in endless loop without any success.

```
[ 5453.889566] usb 2-1.5: device descriptor read/64, error 18
[ 5454.065627] usb 2-1.5: device descriptor read/64, error 18
[ 5454.241502] usb 2-1.5: new high-speed USB device number 4 using ehci-pci
[ 5454.313667] usb 2-1.5: device descriptor read/64, error 18
[ 5454.489593] usb 2-1.5: device descriptor read/64, error 18
[ 5454.665655] usb 2-1.5: new high-speed USB device number 5 using ehci-pci
[ 5454.686276] usb 2-1.5: device descriptor read/8, error -61
[ 5454.806488] usb 2-1.5: device descriptor read/8, error -61
[ 5454.981641] usb 2-1.5: new high-speed USB device number 6 using ehci-pci
[ 5455.014346] usb 2-1.5: device descriptor read/8, error -71
[ 5455.146450] usb 2-1.5: device descriptor read/8, error -71
```

The log file messages show that the host proceeds with the enumeration process by reading the device descriptor from the gadget. The device side reports the connected status, see Table 3: Gadgetfs Events. The next message on the device shows the disconnect status and the host reports an error.

By analyzing the USB transfer it was possible to verify the data transfer via Wireshark USB sniffer tool. It requires the loading of the usbmon module on the Linux host. The usbmon module allows to monitor the USB transfer. The usbmon extends kernel details to the user space through file system. An analysis of the captured packages revealed that the cause of an error message were the malformed USB packages. The error code 71 in the listing above means protocol error and the error code 61 signifies that no data is available. The result is that the host gets an error while reading and parsing endpoints so that the host tries again and again to enumerate the device. The possible cause of this behavior is the corrupt transfer of the device descriptor. It is also possible that the host parses the device descriptor in a wrong way.

To be able to use another USB device controller to test Gadgetfs module on embedded device this research used an embedded system with a different device controller chip. The suitable device was BeagleBone Black with OMAP3530 USB device controller so that in this case the special UDC driver for OMAP chip was provided in kernel. The BeagleBone Black brings an additional host USB port so that the board could be accessed and configured directly from the host computer. The usage of BeagleBone Black with Gadgetfs module was successfully tested. It was possible to exchange data between the host and gadget. This issue strengthens the assumption of the buggy UDC driver of Nexus S device.

An interesting fact is that Galaxy Nexus the next generation of Nexus production line uses OMAP chip instead of Exynos. As already mentioned, the OMAP SoC is also used on the BeagleBone Black which was positively tested. It could be useful to work with Galaxy Nexus device for testing the Gadgetfs functionality and if it proves to work then to port the realization to Samsung Nexus S.

4 Conclusion

The purpose of the research was to analyze the concept of the gadget framework on Android and to provide a practical approach for host-device communication via Gadgetfs module. The Gadgetfs allows to configure the USB interface from the user space. In particular it is possible to define VendorID and ProductID so that the host enumerates the smartphone as a card reader without any additional software needed to be installed on the host. As a conclusion a smartphone can be used as a smartcard reader on operating systems like Linux or Windows. For that purpose the device descriptor, configuration descriptor, interface descriptor and the endpoint descriptors are to be configured. This allows the host to identify the device and to create endpoints which are logical elements for communication. The way how to exchange the data with the help of device files on the host side was also mentioned. It can be done only after successful enumeration of the device. In order to do so, the device should be capable to switch between gadget and host modes using On-The-Go technology. The host mode allows plugging of various peripheral devices into smartphone. The gadget mode makes it possible to use the smartphone with card reader functionality.

The research is based on Exynos 3 an ARM based System-on-Chip developed by Samsung. The Google Nexus S is one model out of production line of Nexus devices with Android operating system. To proceed with the task of configuring and debugging, the gadget framework was discussed which contains three layers: the user space program, gadget driver and USB device controller driver. Thus, the right configuration of USB device controller for Samsung Nexus S is necessary.

The research has to deal with ARM target architecture therefore the cross compiler environment was necessary to be able to compile Android kernel and user space program on the x86 host. To be able to use Gadgetfs on Samsung Nexus S the USB device controller driver for that platform needs to work properly. The assumption at the beginning of this work was that the gadget driver was functioning and that configuration in the user space has to be modified. This assumption was false. This research covered the problem of buggy driver and successful attempts which have been made to fix the bug. The user space configuration of Gadgetfs on Samsung Nexus S with Exynos System-on-Chip led to a system crash. The kernel panic was successfully debugged and patched. The endpoints on the device side were created. This issue led to next bug where the host could not enumerate the device because some packages were malformed. The cause could be that the device descriptor it transmitted incorrectly to host or the host parses the descriptors in the wrong way due to the change in the USB specification. The detailed analysis of this issue would exceed the scope of this work but can be investigated in a further research.

Moreover, the following gadget modules were introduced: gadget zero, serial gadget, network gadget and storage gadget.

Another embedded system the BeagleBoard Black which is based on OMAP System-on-Chip was presented. The OMAP platform uses a different USB device controller driver so that Gadgetfs and also other gadget modules could be tested with positive results.

Thus, this research suspects the buggy implementation of UDC driver for Nexus S as a root cause of the malfunctioning. The possible solution is to use Galaxy Nexus as it uses OMAP SoC made by Texas Instruments.

Similar research and implementation was done by Frank Morgner with OpenMoko phone. The OpenMoko is based on Samsung S3C2410 System-on-Chip, the Nexus S uses S5PC110 System-on-Chip which is the subsequent model.

The research analyzed the USB gadget framework on Android, showed possible shortcomings of driver implementation and introduced the bug fixes in respect to the device configuration.

5 References

[1] *Bootstrap Yourself with Linux-USB Stack*, Rajaram Regupathy (2012)

[2] *Building Embedded Linux Systems*, Karim Yaghmour, Jon Masters, Gilad Ben,Yossef, Philippe Gerum (2008)

[3] *Embedded Linux Primer: A Practical Real Word Approach*, Christopher Hallinan

[4] *Embedded Android Porting, Extending and Customizing*, Karim Yaghmour (2013)

[5] *Essential Linux Device Drivers*, Sreekrishnan Venkateswaran (2008)

[6] *Linux Kernel Development*, Robert Love (2003)

[7] *Linux-Treiber entwickeln*, Jürgen Quade, Eva-KatharinaKunst (2012)

[8] *Linux Device Drivers*, Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini, Third Edition (2005)

[9] *Linux Device Drivers*, Jonathan Corbet, 3rd Edition (2005)

[10] *Mobile smart card reader using NFC-enabled smartphones*, Frank Morgner, Dominik Oepen, Wolf Müller, Jens-Peter Redlich (2012)

[11] *Mobiler Chipkartenleser für den neuen Personalausweis*, Diplomarbeit, Frank Morgner (2012)

[12] *Pro Linux Embedded Systems*, Gene Sally (2009)

[13] *The Linux Kernel Module Programming Guide*, Peter Jay Salzman, Michael Burian, Ori Pomerantz

[14] *USB complete: everything you need to develop custom USB peripherals*, Jan Axelson (2005)

[15] *USB Complete The Developer's Guide*, Jan Axelson (2009)

[16] *Talking to Device Files*, available at 7.5.2013
`http://tldp.org/LDP/lkmpg/2.6/html/x892.html`

[17] *Linux-USB Gadget API Framework*, available at 4.2.2014
`http://www.linux-usb.org/gadget/`

[18] *USB-Interface*, available at 4.2.2014
`http://www.sprut.de/electronic/interfaces/usb/usb.htm`

[19] *Building a Kernel from source*, available at 23.03.2013
`http://xda-university.com/as-a-developer/getting-started-building-a-kernel-from-source`.

[20] *Building for devices*, available at 23.03.2013
`http://source.android.com/source/building-devices.html`

[21] *CCID free software driver*, available at 12.6.2014
`http://pcsclite.alioth.debian.org/ccid.html`

[22] *Connect USB peripherals to your Nexus One*, available at 28.9.2013
`http://sven.killig.de/android/N1/2.2/usb_host/`

[23] *Data Transfer to and from USB Devices*, available at 7.6.2013
`http://www.opensourceforu.com/2011/12/data-transfers-to-from-usb-devices/`

[24] *Device Driver Support*, available at 12.6 2014
`http://www.linux-usb.org/devices.html`

[25] *HOW-TO Compile ICS AOSP*, availible at 2.4.2013
`http://forums.androidcentral.com/general-help-how/144804-how-compile-ics-aosp-4-0-3-xoom-gnex-nexus-s.html`

[26] *Programming Guide for Linux USB Device Drivers*, available at 3.4.2013
`http://www.lrr.in.tum.de/Par/arch/usb/usbdoc/usbdoc.html`

[27] *Samsung Nexus S*, available at 13.04.2014
`http://de.wikipedia.org/wiki/Nexus_S`

[28] *USB Gadget*, available at 8.6.2013
`http://www.armadeus.com/wiki/index.php?title=USB_Gadget`

[29] *USB Gadget*, available at 12.06.2014
`www.emlix.com/fileadmin/emlix/dokumente/FA_USB.pdf`

[30] *USB Gadgetfs*, available at 2.5.2013
`http://docs.blackfin.uclinux.org/doku.php?id=linux-kernel:usb-gadget:fs`