Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

A Cross-platform Picture Transfer Protocol for Linux-based Camera

by

Marcus Högberg

LIU-IDA/LITH-EX-A--15/003-SE

2015-01-27

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

A Cross-platform Picture Transfer Protocol for Linux-based Camera

by

Marcus Högberg

LIU-IDA/LITH-EX-A--15/003-SE

2015-01-27

Supervisor: Maria Vasilevskaya
IDA, Linköping University

Eric Hampusgård
Memoto

Examiner: Mikael Asplund
IDA, Linköping University

Abstract

The Universal Serial Bus, USB, is widely used for connecting peripheral devices to a computer. Through the years devices that use USB has evolved and more and more complicated communication protocols have been developed using the USB standard. There are many different ways to set up communication between a USB device and a host computer. The USB standard does not include any security and this poses risks when designing communication over such a connection.

This thesis investigates how a USB-based picture transfer protocol can be designed between a small camera running embedded Linux and a host computer. The USB functionality in Windows and Mac OS/X operating systems are investigated. Solutions to create a secure USB communication are also investigated. One of three the methods of creating a USB connection with a USB device running embedded Linux are chosen based on the investigations. A protocol is then designed and an implementation developed. The protocol designed in the thesis uses existing USB functionality in the host computer operating systems Windows and Mac OS/X.

The designed protocol is evaluated for performance and security. The evaluation is made on an evaluation platform for the camera. The transfer speed of the protocol is measured to around 18 MB/s in an ideal environment. The designed protocol could be improved by using one of the security methods found in the investigations.

Terms and Abbreviations

ACM	– Abstract control model
API	– Application programming interface
DWG	– Device working group
ECC	– Elliptic curve cryptography
ECM	– Ethernet control model
EEM	– Ethernet emulation model
FTP	– File transfer protocol
HID	– Human interface device
HTTP	– Hypertext transfer protocol
IP	– Internet protocol
MSC	– Mass storage class
NTP	– Network time protocol
RNDIS	– Remote Network Drive Interface Specification
RTC	– Real Time Clock
SCP	– Secure Copy
SCSI	– Small Computer System Interface
SSH	– Secure Shell
SSL	– Secure Sockets Layer
TCP	– Transmission control protocol
USB	– Universal serial bus
USB-IF	– USB implementers forum

Table of Contents

Chapter 1

Introduction

This chapter presents the motivation and purpose of this master thesis project (30 credit points) examined at the Department of Computer and Information Science (IDA) at Linköping University. The thesis work has been performed in collaboration with Memoto[1], a newly founded company in Linköping, which creates a small autonomous camera.

1.1 Background

Memoto is a company dedicated to lifelogging[2] with the vision “Remember every moment”. This is achieved by using a small (36x36x9 mm) camera which automatically takes pictures at a predefined time interval (default is 30 seconds), a cloud based storage and applications for Android and iPhone. An overview of the Memoto system is shown in Figure 1.1. The Memoto vision is supported by sophisticated algorithms for grouping the pictures into moments and sorting the pictures according to a quality measurement. The mobile applications are then used to present the moments to the user.

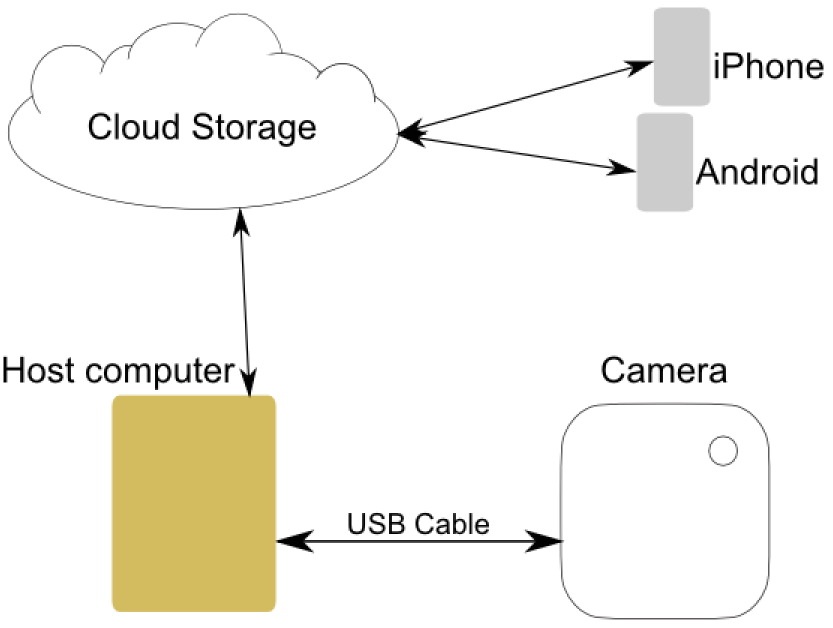

Figure 1.1 An overview of the Memoto system.

To get the pictures from the camera to the cloud storage the user connects the camera to a host computer with an internet connection. The camera connects to the computer using a *USB 2.0 High Speed* connection and an application on the computer transfers the pictures to the cloud

[1] http://www.memoto.com
[2] http://en.wikipedia.org/wiki/Lifelog

storage. The computer used for the transfer can have either Windows or Mac OS/X as the operating system. The supported version should be Windows XP and Mac OS/X 10.6 and newer.

The camera runs an embedded Linux operating system. According to Gatliff [1] there are three main ways of implementing a USB connection in an embedded Linux device. Including: making a custom proprietary solution, using the USB connection as a high speed serial link or emulating an Ethernet connection. There are no physical buttons on the camera and therefore the USB communication is the only way to change settings on the camera.

The camera can potentially store a large amount of pictures. It is important for Memoto that the pictures are not compromised and cannot be seen by other persons, at least until the owner whishes the pictures to be seen. The pictures should not become available just by connecting the camera to a computer. Compare to how a USB flash drive exposes a file system when you connect it to a computer. Then you can access files on the flash drive easily. The USB standard is inherently insecure since there is no security built-in to the standard [2].

1.2 Purpose and objectives

The purpose of this thesis is to investigate how a solution can be developed that supports picture transfer between a camera using embedded Linux and a cloud based storage. The camera will not have an internet connection of its own and need to be connected to a computer via USB 2.0 and use the computer as a gateway to get access to the Internet and the cloud storage.

The thesis investigates which of the three mentioned methods of implementing USB connectivity for embedded Linux [1] is most preferred according to the following defined properties.

- The protocol should comply with both Windows XP and Mac OS/X 10.6.
- The complexity of implementing the solution on the camera and computer.
- The possibility to make the connection secure.

Possible solutions to make a secure connection between the camera and the cloud storage are examined in the thesis. The focus in this thesis being how to make the USB connection secure.

1.3 Limitations

When the work observed in this thesis was done there were no cameras available for evaluating the USB communication protocol. The design and implementation has instead been evaluated on an evaluation kit with the same processor as the one used in the camera.

The security of the USB communication is only analyzed in theory. No solution to make the USB connection secure is implemented in this thesis.

1.4 Method

A requirements specification for the USB communication was created. The requirements were developed using use case diagrams. An investigation of methods to make a UBS connection secure was conducted. The support for the three USB communication methods in the Windows and Mac OS/X operating systems was also investigated. An analysis of the results from the two investigations was used to choose the method for implementing the USB communication.

A performance test was conducted to verify that the protocol has an acceptable transfer speed. The requirements were verified by using tests on an implementation of the protocol and analysis of the requirements that was not tested in the implementation.

1.5 Thesis overview

Chapter 2 explains the findings from the studies needed to understand the USB communication protocol. This includes the USB 2.0 standard, its security issues and the Linux-USB Gadget API Framework including the gadgetfs driver.

Chapter 3 presents the requirement specification for the USB communication protocol.

Chapter 4 describes the design of the USB communication protocol.

Chapter 5 describes the design and implementation of the protocol on the camera and the computer.

Chapter 6 presents an evaluation of the implementation and the USB communication protocol. The evaluation is done by testing the communication against the requirements.

Chapter 7 is a concluding chapter that describes the conclusions of the thesis and some further work that could be done.

Chapter 2

Background

This chapter presents the information needed to understand the rest of the thesis work. The first section presents security aspects of the architecture and design. After that information about the USB 2.0 specification is presented and then the USB functionality in the embedded Linux operating system is described. Finally three different ways to implement USB on a Linux device is presented.

2.1 Security

There are many ways of attacking the USB protocols, devices and drivers. One can install malware that snoops the bus, create malware USB devices or malware that attack specific USB protocols [2]. Another way of attacking USB drivers to get potentially access to the computer kernel is "fuzzing" [3]. Fuzzing means that an attacker randomly switches bits or bytes in the USB messages. This could be achieved using a man in the middle attack. By fuzzing the data an attacker could potentially get access to or crash USB drivers.

One way to secure the USB connection is to use authentication and encryption. This is hard to do since it is relatively easy to get access to the USB stack of a computer and snoop the data sent on the USB connection. Any passwords or user identifications sent on the USB connection can be intercepted in this way. Li and Lin [4] have designed an encryption filter driver that resides between the disk driver and the USB stack. The filter encrypts and decrypts data sent to a USB mass storage device thus making the data on the USB connection encrypted and no keys are sent on the connection. This solution is not suitable in the context of this work since it requires the data to be stored and encrypted from the USB host to the USB device. In this work the camera already have the data that need to be securely transferred thus we need another way to make the transfer secure.

Wang et al. [5] proposes a general USB stack extension that enables USB devices and hosts to authenticate each other using a public key based approach. This proposal comes as a result of their earlier research on using USB as the point to attack computers with mobile devices [6]. The solution to use a public key approach for authenticating the use of the camera with a computer is an interesting proposal. This could be adopted for the protocol in our thesis to lock the USB connection to the camera until a correct authentication is made.

Lee et al. [7] have researched the security level in three different kinds of authentication schemes used in secure USB flash drives. They have found that all three of these schemes are relatively easy to hack using replay attacks. In a replay attack the hacker intercepts user information and uses that information to pose as the real user. This could be done since the three methods send the user ID and password over the unsecure USB connection. Lee et al. [8] proposes a new authentication scheme that does not send any user information over the

USB connection and have a different encryption key every session. This authentication method places computation in the USB device. This is one of the methods to secure the USB connection that would be suitable for use with the protocol in this thesis. The camera has the necessary computational power needed and the method could be easily adapted to a proprietary USB protocol.

Gupta el al. [9] proposes an architecture for connecting to an embedded device using an internet connection (TCP/IP) to a gateway and then a more lightweight protocol between the gateway and embedded device. That architecture is very similar to the one in this thesis where a cloud storage is connected through the Internet to a computer (gateway) and then connected to an embedded device using USB. Gupta et al. [9] describes an end-to-end security solution based on the secure sockets layer, SSL. Their implementation uses less CPU and memory resources than regular public-key cryptography algorithms. They use elliptic curve cryptography (ECC) for key exchange and Wang et al. [10] have optimized the algorithm for platforms with small computational resources. This lightweight solution could be used in the proposed architecture in this thesis to enable a secure connection between the camera and the cloud storage.

2.2 Universal serial bus

USB is a polled protocol bus that was first developed in the mid 1990's. It has evolved since then and today the USB is the most widespread standard for connecting peripheral devices to a computer. USB is a master-slave protocol bus where the host has to initiate all transfers to and from a device. The USB 2.0 standard can support 127 devices on every host controller. The USB 2.0 specification [11] specifies three logical layers which are presented in the following section.

2.2.1 USB Protocol architecture

The USB 2.0 protocol consists of three logical layers. These layers are shown in Figure 2.1. The lowest layer is the *USB Bus Interface Layer*. This layer handles transfer of the signals on the USB cable. It is responsible for the physical and electrical aspects of the protocol.

The middle layer is the *USB device layer*. This layer structures the USB packets that are sent on the bus and packet the data in accordance to the USB specification. In this layer the communication is logically done through the *pipes*. The pipes are connections from the *USB System Software* to the *USB Logical Device*. The connection point on the device side of a pipe is called *endpoint*.

The topmost layer is the *Function Layer*. This is where the device uses the actual data transferred on the bus. The different endpoints in the *USB Device Layer* are grouped together to form logical *interfaces* through which applications on the host computer communicate with functions on the device.

The protocol in this thesis is designed to be a part of the *Function Layer* of the architecture. The following sections will introduce three parts of the USB standard that are used in this thesis. They are needed to understand the design.

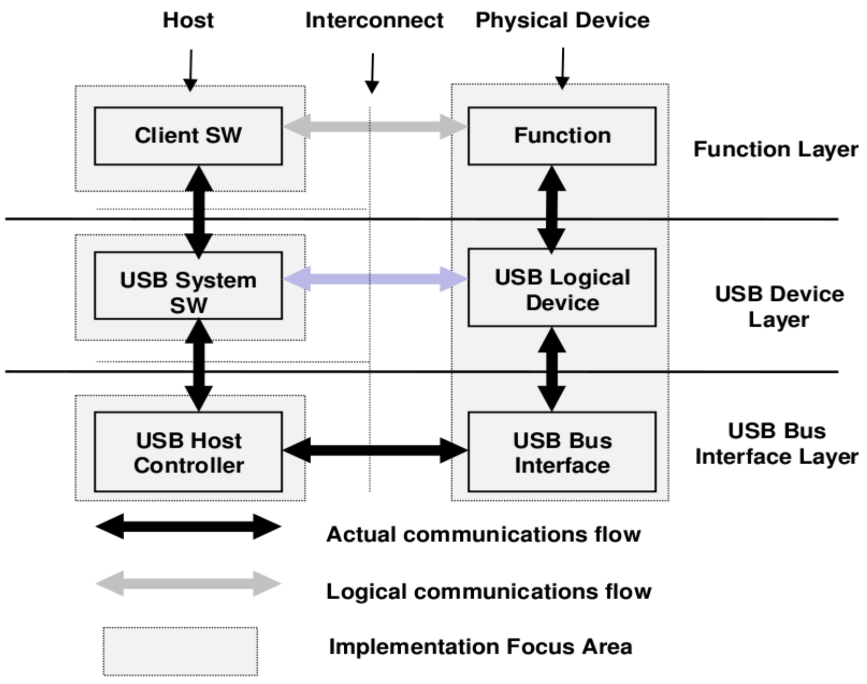

Figure 2.1 USB protocol architecture as described in USB specification [11].

2.2.2 Endpoint zero

The USB 2.0 specification states that every USB device need to listen to *endpoint zero*. This endpoint is used by the *USB Host Controller* to enumerate the USB devices and configure the devices to be ready for further communication. The setup is made using *control transfers* described in the next section of this chapter. The USB device also supplies the host computer with descriptors. The descriptors define the pipes and interface that the USB device uses.

2.2.3 Transfer types

There are four types of transfers in the USB specification:

- Interrupt transfer
- Isochronous transfer
- Bulk transfer
- Control transfer

Interrupt transfers are used to send interrupts from a device to a host. This is for example used by mice and keyboards. The interrupt transfer is not designed to send big payloads of data.

The *isochronous transfer* is used for devices that need a recurring access to the bus (e.g. web camera). The isochronous transfers do not have any mechanism for resending any corrupt packets, but they guarantee access to the bus at an interval specified by the device, thus making it a good choice for time critical systems.

Bulk transfers get transmitted when there is no other traffic on the bus. An advantage compared to isochronous transfers is that bulk transfers can get faster transfer speeds.

Isochronous transfers are limited by the bandwidth that they request in their descriptors. Bulk transfers on the other hand get the spare room that is on the bus, thus making it able to reach higher bandwidth. One downside is that when the bus has a high load, bulk transfers gets delayed.

Control transfers are mainly used to send commands to a device. They can also be used to get status updates from a device. They are not designed to handle larger amounts of data, but have a clear structure.

2.2.4 USB Setup Packet

The USB standard *control transfer* is a transfer with a predefined structure. The protocol designed in this thesis uses the *control transfers* to communicate with the camera. A *control transfer* consists of a *setup packet* and optionally a *data packet*. The *setup packet* declares whether a *data packet* should be sent. The *control transfers* are used to send USB device requests as specified in the USB specification [11]. The requests are used to setup a USB device correctly to enable communication. In Table 2.1 the fields in the *setup packet* are listed. The different fields of the USB *setup packet* will be described below.

bmRequestType

This bitmap field specifies the characteristics of the request. The field is divided into three parts including: direction, type and recipient. The first part is used to identify the direction of the optional data packet. The type can be *Standard*, *Class* or *Vendor*. This specifies if the control transfer is a standard USB request, part of a USB class or specified by a third party vendor. The last part, recipient, specifies if the control transfer is directed to the USB device as a whole, a specific interface or even to an endpoint.

bRequest

This field specifies the specific request. If the type in bmRequestType is set to *Standard* this field specifies the USB standard request (e.g. SET_CONFIGURATION which is a USB standard request that tell a USB device to switch to a specific configuration).

wValue and wIndex

The contents of these fields are dependent on the request. These fields are used to send the request's specific parameters to the device. To continue the example with SET_CONFIGURATION, the wValue field is used to send the specific configuration that should be used. Configurations are defined by a device and communicated to the host for selection.

wLength

The length of the data packet in the second part of the control transfer is specified in this field. If the wLenght field is set to zero the optional data packet is not sent.

Table 2.1 Format of the USB setup packet. The table is from the USB specification [11].

Offset	Field	Size (byte)	Description
0	bmRequestType	1	Characteristics of request: D[3]7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6..5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4..0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	bRequest	1	-
2	wValue	2	-
4	wIndex	2	-
6	wLength	2	-

2.2.5 USB classes

There are different classes of devices in the USB standard, that are defined by the USB Device Working Group, DWG [12]. The classes of USB devices that are defined are: Audio, Communication, Human Interface Device, Physical, Image, Printer, Mass Storage, Hub, Smart Card, Content Security, Video, Personal Healthcare, Audio/Video Devices, Billboard Device Class, Diagnostic Device, Wireless Controller, Miscellaneous, Application Specific and Vendor Specific. Many of these classes often have device drivers implemented in different operating systems. The classes defined by the DWG make it possible to have software that doesn’t need to implement proprietary protocols for different USB devices. If a device and an application both follow the class specification then they can communicate, without requiring proprietary drivers.

According to Kolokowsky and Davis [13] one big mistake people do when designing USB devices is not to use the standard classes. Standard classes often have driver implementations in different operating system. So by using these classes development is often easier, faster and potentially cheaper.

The Mass Storage Class is one of the classes specified by USB DWG. In this thesis a mass storage device is setup to compare transfer speeds with the developed protocol. The Mass Storage Class is used by storage devices such as USB flash memories and external USB hard drives to communicate with a computer [14].

[3] D is indicating a specific bit in the field bmRequestType.

2.3 Embedded Linux

The camera runs an embedded Linux kernel. In the Linux operating system there are two types of drivers. One is the ordinary drivers (e.g. printer drivers) and the second is *gadget drivers*. When an embedded Linux system is used as a USB device the USB drivers are called *gadget drivers*. The Linux operating system includes the Linux-USB gadget API framework [15] which includes *gadget drivers* on Linux embedded systems. The comprehensive explanation of the Linux USB stack can be found in Reguphaty [16]. This section will outline those parts required to understand the rest of this work.

2.3.1 Linux Gadget Framework

The Linux-USB Gadget API framework makes a connection between the *USB Bus Interface Layer* and the *USB Device Layer* (Figure 2.1). All of the gadget drivers in Gadget API framework are implemented in the kernel space of the Linux operating system. Many of the classes in the USB standard exists as *gadgets drivers* in the framework. There is one gadget driver that allows the use of USB standard transfers (section 2.2.3). That is the gadget *gadgetfs* described in the next section.

2.3.2 Linux gadgetfs

The gadget file system, *gadgetfs*, is a gadget driver in Linux. The gadget driver is designed to enable development of USB drivers in user space instead of in kernel space. All four transfer types of the USB specification, described in section 2.2.3, are supported by gadgetfs.

USB communication with gadgetfs is done through the file system. Each endpoint that is needed by the user space driver gets a unique file from gadgetfs. Gadgetfs supplies one file to start with. This is the file relating to endpoint zero in the USB specification, described in section 0. By writing USB descriptors to this file gadgetfs creates new files for the endpoints declared in the descriptors. To generate USB communication data is written to the files.

2.4 Linux USB device communication

According to Gatliff [1] there are three main ways to make communication with Linux-based USB devices.

- Proprietary USB solution
- Point-to-point serial connection
- Emulating Ethernet connection

These three ways of communication have been considered in this thesis. The following three sections describe them and their drawbacks and advantages in relation to the requirement specification.

2.4.1 Proprietary USB solution

Making a proprietary USB solution is the most ambitious and complicated way of these three. But it has the main advantage that it is a powerful solution with the capability to implement sophisticated high-level protocols. A drawback is that device specific drivers often need to be implemented for both the device and the host computer.

2.4.2 Point-to-point serial connection

The second of these, point-to-point serial connection is an easy way to do the communication. This method is a good choice for a peripheral device that does not need to respond to commands from the host computer (e.g. sensors). This variant of communication is not that suitable if a more complex interaction is needed between the host computer and the device.

2.4.3 Emulating Ethernet connection

The last one of emulating an Ethernet connection could be considered one of the most powerful ways of making USB connectivity. By using the USB as an emulated Ethernet connection, the USB device could make use of all protocols that use the TCP/IP stack. This includes protocols such as FTP and HTTP. The use of an emulated Ethernet could make the USB device able to connect directly to cloud based storage servers or use the Network Time Protocol (NTP) for synchronizing its internal clock.

There are four main protocols for emulating an Ethernet connection over USB as defined by USB DWG [12]. The first is Remote Network Driver Interface Specification (RNDIS) that is a Microsoft proprietary protocol. Then there are three protocols defined by the USB device working group (DWG). Those are the Ethernet emulation model (EEM), Ethernet Control Model (ECM) and the Network Control Model (NCM).

Of the aforementioned protocols RNDIS is the first developed that supported emulated Ethernet. The protocol is an integrated part of the Network Driver Interface Specification of the Windows operating system. ECM was the first protocol released by the USB DWG. As USB developed and the speed got faster the ECM protocol needed improvement. EEM and NCM protocols were developed to be efficient with the faster versions of the USB standard. According to K. Kim et al. [17] the speeds of these protocols can be disappointing. They propose three methods to increase the throughput when emulating Ethernet over a USB connection.

Chapter 3

Requirements

This chapter will present an overview of the system, including: the camera, the host computer, the cloud storage, smartphone app and he user. Then the parts that are in the scope of the thesis are identified and finally the requirements for the system are presented.

3.1 System overview

An overview of the Memoto system and how the different parts are connected is shown in Figure 3.1. In this thesis the USB connection between the computer and camera is addressed, marked with a rectangle in the figure. Some use-cases for the USB connection have been identified and will be described in the following section.

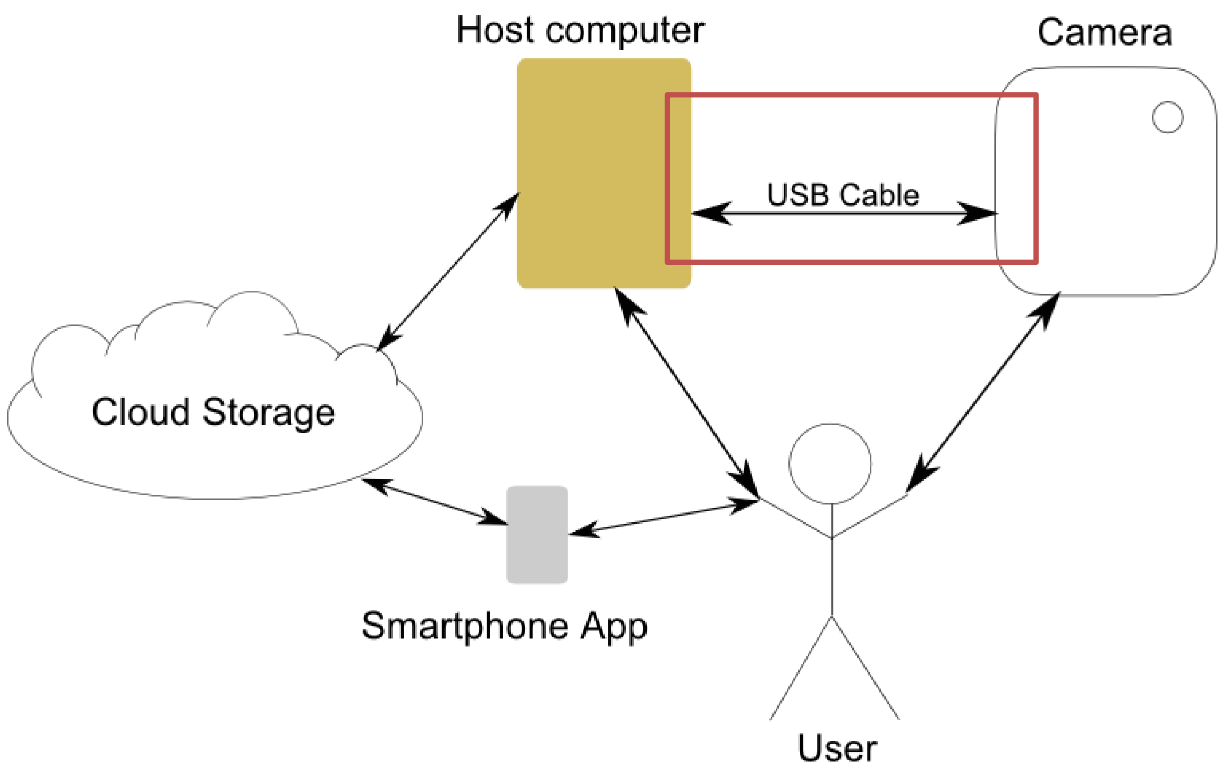

Figure 3.1 Overview of the system. The parts of the system that are in the scope of this thesis are marked with a rectangle.

3.2 Use-cases for USB connection

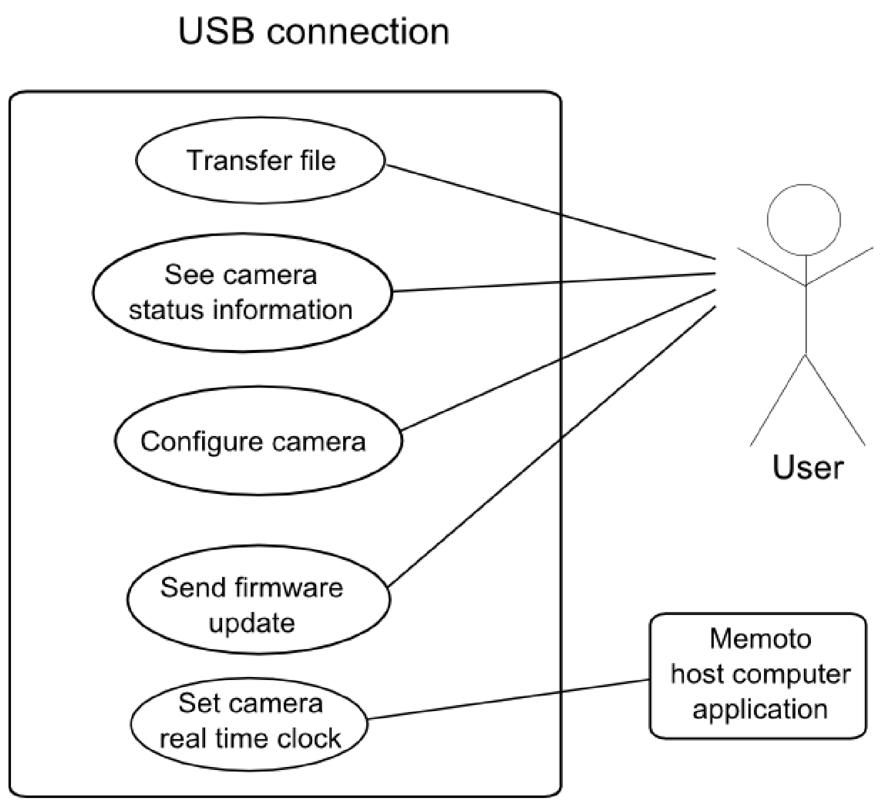
USB connection
Transfer file
See camera status information
Configure camera
Send firmware update
Set camera real time clock
User
Memoto host computer application

Figure 3.2 Use-cases for the USB connection in the system.

In Figure 3.2 five use-cases are identified for the USB connection.

Use Case: 1 Transfer picture
Primary actor: User
A user wants to transfer pictures from the camera to the host computer.

Use Case: 2 See camera status information
Primary actor: User
The user wants to see status information about the camera. The relevant information is battery status, configured time interval between photos, memory usage and remaining memory.

Use Case: 3 Configure camera
Primary actor: User
User wants to change the time interval that the camera waits between pictures.

Use Case: 4 Send firmware update
Primary actor: User
The camera firmware should be updated to a newer version.

Use Case: 5 Set camera real time clock
Primary actor: Memoto host computer application
The camera’s real time clock need to be corrected due to drift.

3.3 Requirements

Each requirement has a unique ID, a description and a priority. The functional requirements are presented in Table 3.1 and the non-functional requirements in Table 3.2.

The priority of a requirement has the following definition

1. Critical functionality that shall be done.
2. Shall be done after all priority 1 requirements are fulfilled.
3. Not in scope of this thesis, but further possibilities of the communication.

Table 3.1 The functional requirements for the USB communication.

ID	Description	Priority
1	The protocol shall provide a way to transfer files from the camera to the host computer.	1
2	The protocol shall provide means to detect corrupted files.	1
3	The protocol shall support the host to send delete file commands to the camera.	1
4	It shall be possible to send firmware updates to the camera.	1
5	The protocol shall provide the host with the camera's battery status.	1
6	The protocol shall provide the host with the real time clock (RTC) of the camera with an accuracy of 1 second.	1
7	The protocol shall provide the host with the camera's serial number.	1
8	The protocol shall provide the host with the camera firmware version.	1
9	The protocol shall provide the host with the camera's time interval between photos.	1
10	The protocol shall give the host a way to set the RTC of the camera.	1
11	The host shall be able to set the time interval between photos on the camera.	1
12	The protocol shall be locked and opened by user information.	2
13	The protocol shall provide the host with the memory remaining on the camera.	2

Table 3.2 The non-functional requirements for the USB communication.

ID	Description	Priority
14	The communication shall be based upon the USB 2.0 High Speed specification [11].	1
15	The communication shall work on Windows XP and newer versions.	1
16	The communication shall work on Mac OS X 10.6 and newer versions.	1
17	The communication shall work on a camera running embedded Linux.	1
18	The transfer speed shall be in the same range as the USB Mass Storage Class (section 2.2.5).	1
19	It should be possible to implement the host side of the protocol in user space. Without need to create kernel drivers.	2
20	Communication over USB should be encrypted	3
21	The camera should communicate directly with the cloud storage.	3

Chapter 4

USB communication protocol

This chapter will describe the design of the USB communication protocol. The first part of the chapter will describe the design decisions made. Then a description of the design of the USB protocol follows. Starting with an overview and then describing each request in the protocol. The final part presents the functions that use the request to realize the requirements described in section 3.2. Three of the requirements affect the final design: embedded Linux kernel for the camera (R17) and Windows and Mac operating systems for the host computer (R15 and R16).

4.1 Communication structure

In section 2.4 three methods of implementing USB communication with a Linux device was described. These three methods to communicate are analyzed in regard to the properties defined in section 1.2. The result of the analysis is summarized in Table 4.1.

Table 4.1 Evaluation of important properties (section 1.2) in the different implementation methods for the USB connection.

Method / **Properties**	Custom (Proprietary)	Serial	Ethernet RNDIS	Ethernet ECM, EEM
Support[4] in OS/X 10.6	√	√		√
Support[5] in Windows XP	√	√	√	
Implementation complexity	Medium	High	Medium	Medium
Complexity implementing the following security solutions:				
End-to-end security architecture. Gupta et al. [9]	High	High	Low	Low
Reverse-safe authentication for USB. Lee et al. [8]	Low	Low	Medium	Medium

To use an emulated Ethernet connection was the initial approach chosen. The thought was to use existing TCP/IP based protocols for the communication. When researching the ability to use emulated Ethernet USB functionality in the required operating systems it was found that

[4] Here support means native support in the operating system. There are some solutions on the market with license fees for some of the other alternatives.

the Windows and Mac OS/X do not have the same support for this functionality. It has been conclude that EEM and ECM both have native drivers in the Mac OS/X operating system from version 10.3 and newer versions. However there is no support for them in Windows XP. The other class RNDIS has a native driver in Windows XP and newer Windows versions but has no native support in Mac OS/X. However an open source RNDIS solution for MAC OS/X 10.6 and newer exists and could be used.

Both operating systems have existing native support to use the USB connection as a point-to-point serial connection. Using a serial connection is a simple solution but to use it for this USB communication protocol a full design of a request service would be needed. That would require a more complex implementation.

Designing a custom solution would allow the use of *USB Setup packages* (section 2.2.4) in control transfers to send request to and from the camera. Both Windows and Mac OS/X operating systems have native support to send the four transfer types defined in the USB standard (section 2.2.3).

The end-to-end security architecture described by Gupta et al. [9] is based upon the TCP/IP protocol. Using an Ethernet emulation approach for the USB connection would make the implementation of that architecture fairly easy. The architecture can be used even with a serial and custom USB connection but would pose the need to implement a method of sending the SSL handshakes over the USB connection. That gives this security architecture a higher implementation complexity for serial and custom USB connections.

The reverse-safe authentication method proposed by Lee et al. [8] would be easy to implement in the case of a custom or serial solution. With one of these solutions there would already be an infrastructure for sending small amount of requests to the camera. If an Ethernet emulation solution would be used no such infrastructure exists. To use this security method with emulated Ethernet would require an implementation of requests, thus making the implementation more complex.

Based on the analysis made in this section, Ethernet emulation would be the preferred approach. However, the different support for the Ethernet emulation in the target operating systems led to not choosing this method. Instead a custom solution is chosen for the USB protocol. The custom solution is chosen above a serial implementation for the lower complexity in the implementation. The rest of this chapter will describe the design of the transfer protocol. The following chapter will describe the implementation.

4.2 Design overview

The previous section described why a custom solution was chosen for this USB communication protocol. The rest of this chapter will describe the design of this protocol.

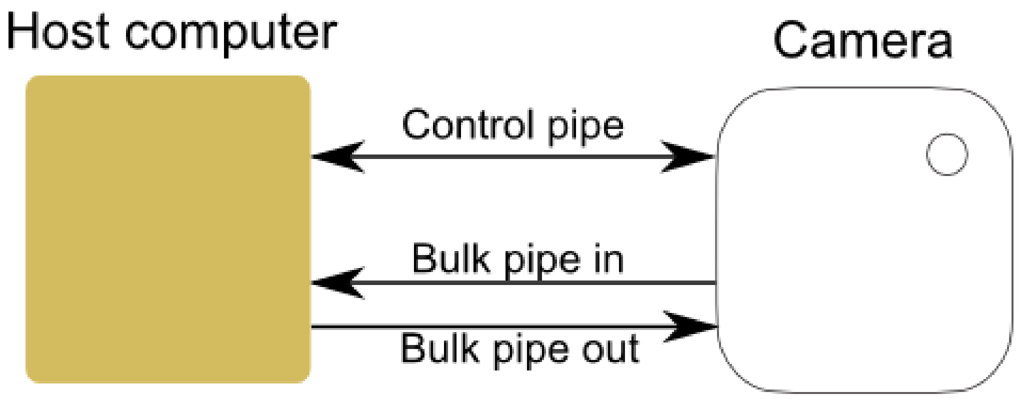

Figure 4.1 Overview of the pipes in the communication protocol. One bi-directional control pipe and two bulk pipes, one in each direction.

The protocol uses three pipes to supply the needed functionality of the protocol. One bi-directional *control pipe* and two bulk pipes, *bulk pipe in* and *bulk pipe out*, shown in Figure 4.1. The three pipes are grouped together in an interface and the communication between the host computer and the camera is controlled by the computer. The host computer sends requests via control transfers to the camera on the *control pipe*. Depending on the request the camera can answer the request either by the data packet in the control transfer or in a separate bulk transfer on one of the *bulk pipes*. The bulk pipes are needed due to the limited size of the data packet of the control transfer. The size of the data packet is specified by the *wLength* field of the setup packet (Table 2.1). This limits the data packet size to 64 KB. Thus it is possible to transfer small amounts of data in the control transfer but larger files such as pictures need to be sent with the bulk transfers.

4.3 Requests

The communication protocol designed in this thesis specifies six requests found in Table 4.2. The requests are sent on the control pipe as control transfers. The requests use the setup packet structure defined in the USB 2.0 Specification, described in section 2.1.

Table 4.2 Request specified in the protocol developed in this thesis.

bmRequestType (Direction bit)	bRequest	wValue	wIndex	wLength	Data Packet
1	GET_PARAMETER	0	Parameter	Size of Value	Value
0	SET_PARAMETER	0	Parameter	Size of Value	Value
0	GET_FILE	0	0	Size of Filename	Filename
0	DEL_FILE	0	0	Size of Filename	Filename
0	GET_FILE_LIST	0	0	0	0
0	SEND_FIRMWARE	0	0	0	0

The type and recipient of the *bmRequestType* field is the same for all the requests. The type is set to “vendor” and the recipient is set to “device”. The direction bit in *bmRequestType* is set to “1” if the data packet should be sent from the camera to the host and set to “0” for host to camera. The zeros in columns *wValue*, *wIndex*, *wLength* and *Data Packet* indicate that this part of the control transfer is not used by that particular request. All requests in Table 4.2 will be explained in the following sections.

4.3.1 GET_PARAMETER and SET_PARAMETER

The GET_PARAMETER and SET_PARAMETER are very similar in function. They are used to transfer parameters (e.g. battery status) between host and camera. The *wIndex* field is used to indicate which parameter is requested. The *wLength* field defines the size of the parameter that is sent in the *data packet*.

This request does not restrict the type of parameter that can be sent between the host and camera. However, the host and the camera need to be in agreement on what parameters are used and their respective size. The available parameters are described below.

Parameters

Requirements 5 to 11 specify information about the camera (e.g. Battery status and serial number) that should be available to the host computer through the USB protocol. To fulfill these requirements the SET_PARAMTER and GET_PARAMETER requests were designed. In Table 4.3 parameters are defined that realize the requirements. Each parameter is given a type, size and description. There is also a specification whether the camera should update a parameter when a SET_PARAMETER requests is sent for that parameter.

Table 4.3: The parameters with their respective type and size. There is also a specification if they should be changeable from the host.

Parameter	Size (bytes)	Type	Changeable from host	Description
SERIAL_NUMBER	4	Unsigned Integer	No	A unique identifier for the camera
RTC	4	Unsigned Integer	Yes	UNIX timestamp
FW_VERSION	4	Unsigned Integer	No	The firmware version is formatted as 01.01.12 represented as the integer number 10112.
BATTERY_STATUS	4	Unsigned Integer	No	The percent of remaining battery on the camera.
PHOTO_INTERVAL	4	Unsigned Integer	Yes	The interval between photos in seconds.

4.3.2 GET_FILE_LIST

This request is used to transfer a list of files on the camera to the host computer. The request does not use any additional fields in the setup packet. The list of files is sent from the camera to the host computer as bulk transfers on the *bulk pipe in*. Each item in the list is formatted according to Table 4.4. The end of the file list is noted by an item that consists of only zeros.

Table 4.4: Structure of an item in the file list.

Field	Size (bytes)	Description
Filename	100	This field is used to transfer the name of the file. The filename is formatted with ASCII characters ending with a null sign.
File size	4	Size of the file.
Hash value	16	The MD5[5] hash value of the file. Described by ASCII characters.
Inode	4	The inode[6] of the file on the camera.

4.3.3 GET_FILE

The GET_FILE request is used to transfer a file from the camera to the host. The request uses the data packet in the control transfer to send the name of the desired file. The length of the name is transferred in the field *wLength*. When the camera receives this request the requested file should be sent on *bulk pipe in* using bulk transfers. The host needs to receive the file by reading the bulk transfers on *bulk pipe in*. If a file that do not exist on the camera is requested the camera sends one byte with the value zero.

4.3.4 DEL_FILE

The request DEL_FILE is used to request a file to be deleted from the camera. The name of the file to be deleted is sent in the data packet to the camera. The size of the name is specified in the *wLength* field. There is no confirmation sent that the deletion has been successful. If a confirmation is required the file list could be checked for the file before and after the deletion using the GET_FILE_LIST request.

4.3.5 SEND_FIRMWARE

This request initiates a transfer of a file from the host to the camera. It is used to send a new firmware to the camera. The request does not use any additional fields in the setup packet. The host sends the firmware as bulk transfers on the *bulk pipe out*. The camera needs to receive the transfers on the bulk pipe.

[5] MD5 is a cryptographic hash function producing a 128-bit hash value.
[6] Inodes are used in UNIX file systems to identify files.

4.4 Functions

This section describes the functions in this protocol that use the requests specified above. Each function in the protocol is related to one or more of the functional requirements in Table 3.1.

4.4.1 Transfer files

Figure 4.2 depicts how a file is transferred using the protocol. This function is related to requirements 1 and 2. The file transfer is made in three steps. First the host computer uses the request GET_FLIE_LIST to get a list of the files on the camera. The list contains unique identifiers for the files. One of these identifiers is the file name which is used in the request GET_FILE to identify the file to send. When the file is transferred to the computer the MD5 hash value of the file is calculated and compared to the specified hash value in the file list. If the two sums match the file has been transferred correctly.

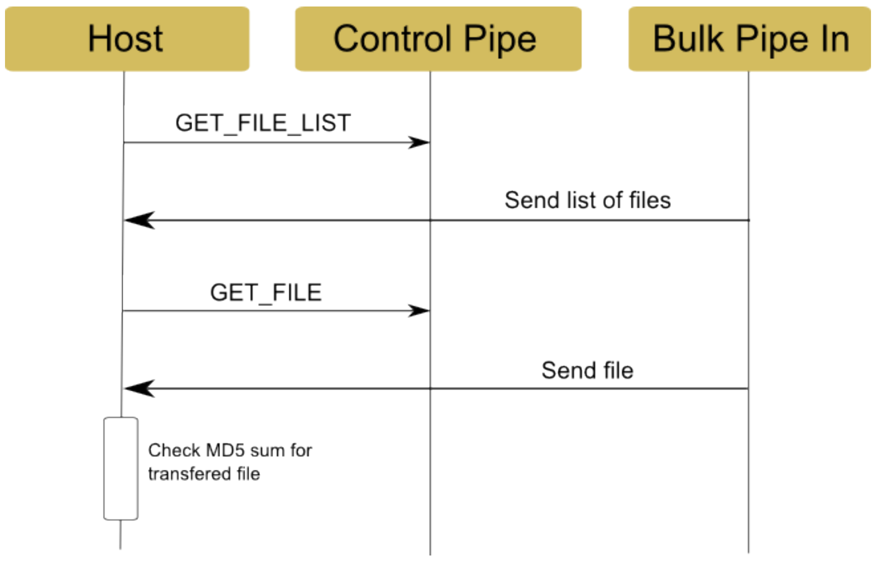

Figure 4.2: The flow to complete a transfer of a file from the camera to the host.

4.4.2 Get parameter

This function realizes requirements 5, 6, 7, 8 and 9, which specify information about the camera that should be available to the host. Figure 4.3 show how to get a parameter from the camera. The GET_PARAMETER request is sent specifying the parameter according to Table 4.3. The value of the parameter is sent to the host in the data packet.

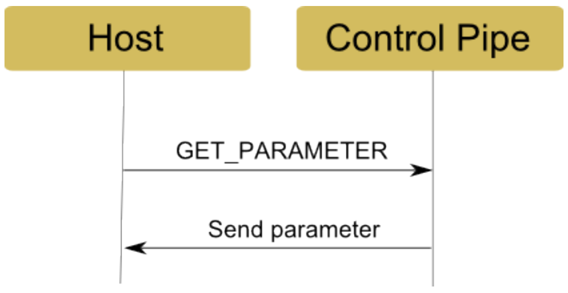

Figure 4.3 Shows the functions to get parameters from the camera.

4.4.3 Set parameter

The set parameter function realizes requirements 10 and 11, specifying two parts of the camera settings that should be configurable form the host. Table 4.3 defines the parameters that could be set using this function. The SET_PARAMETER request is used according to Figure 4.4 to request a change for the parameters in the camera. The new value of the parameter is sent in the data packet.

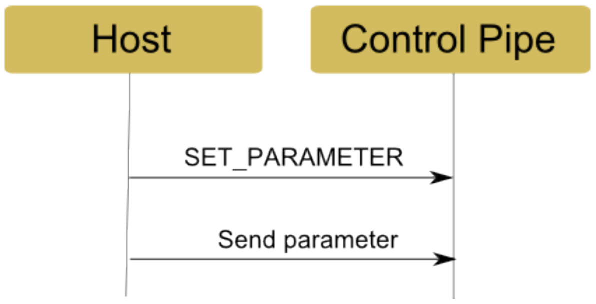

Figure 4.4 The request used to change parameters in the camera.

4.4.4 Delete files

The function to delete files in the camera realizes requirement 3. Figure 4.5 shows how to delete files in the camera. The GET_FILE_LIST request is sent to retrieve the available files on the camera. Then the DEL_FILE request is sent to the camera specifying the file that should be deleted. Optionally another GET_FILE_LIST request could be sent to verify that the deleted file is no longer on the camera.

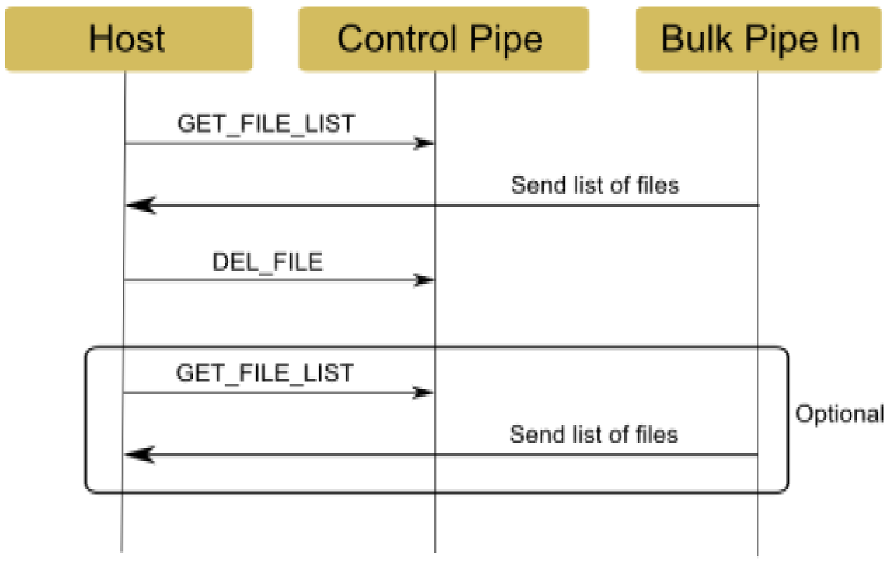

Figure 4.5 Requests used to delete files in the camera.

4.4.5 Send firmware updates

To realize requirement 4 the send firmware updates function is used. The SEND_FIRMWARE request is used according to Figure 4.6. The request is sent to the camera and then the firmware is transferred on the bulk out pipe.

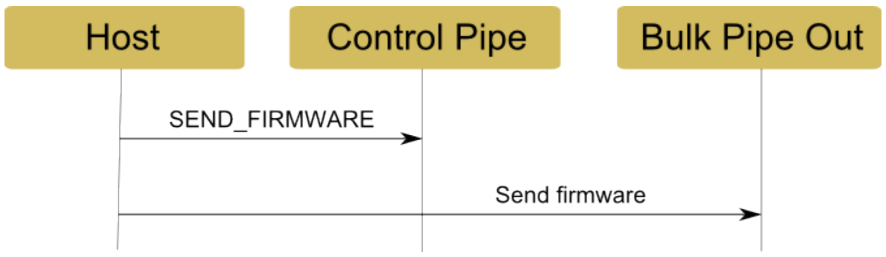

Figure 4.6 Shows how firmware updates are sent using the protocol.

Chapter 5

Protocol implementation

This chapter describes an implementation of the USB communication protocol. The protocol implementation is composed of two parts. One part runs on a camera and the other on a host computer. On the camera the protocol is implemented as an application that controls all the USB communication. The second part is an application for the host computer. The application communicates with a camera using the USB communication protocol developed in this thesis. It is outside of the scope of this thesis to implement a fully functional host application. However, some functionality is partly realized for testing purposes (see Chapter 6). The chapter is divided into two parts. The first describes the camera software and the second the host computer software.

5.1 Camera software

An overview of the software is shown in Figure 5.1. The software is based upon an embedded Linux kernel that runs on an Atmel Processor[7]. The USB protocols transfer types are accessed by using *gadgetfs*. Different processors have different hardware implementations of the USB protocol and *gadgetfs* needs a driver to communicate with the hardware. Atmel has a Linux driver for the USB hardware on the processor and this is used by *gadgetfs* to communicate with the hardware. *usbapplication* is the part of the software that is implemented in this thesis. The following sections will describe this part and the interaction with *gadgetfs*.

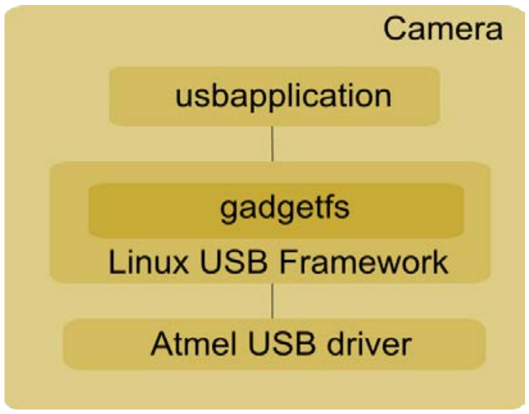

Figure 5.1 Overview of the software in the camera.

[7] The Atmel processor used is the AT91SAM9G25 (http://www.atmel.com/devices/SAM9G25.aspx).

The *usbapplication* is dived into three parts, *usbdriver*, *RequestHandler* and *ParameterHandler* (Figure 5.2). The main part is the *usbdriver* which sets up the device descriptors and controls the main flow of the USB communication. *RequestHandler* handles the requests specified in Chapter 4 and *ParameterHandler* handles the requests for parameters specified in the same chapter.

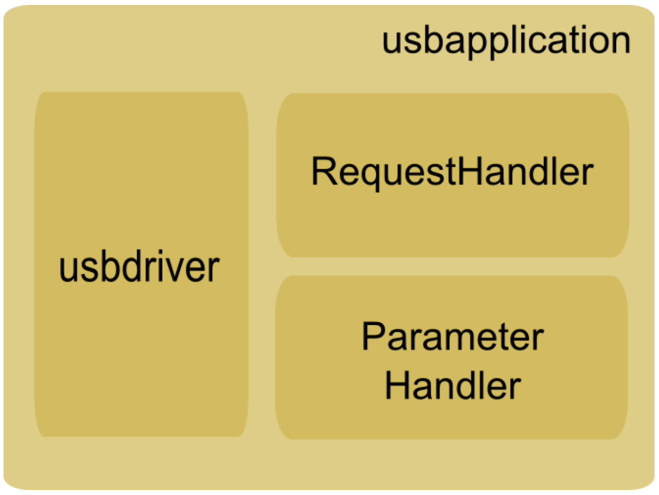

Figure 5.2 Overview of the usbapplication in the camera.

5.1.1 usbdriver

This is the main part of the *usbapplication* and controls the logic of the application. When the application is started the USB system is initiated. The system is initiated by opening the initial endpoint zero file of *gadgetfs*. This file is used to setup *gadgetfs* in accordance to the protocol description in Chapter 4. The protocol needs a control pipe and two bulk pipes. The configurations needed to setup these pipes are written to the endpoint zero file. This makes *gadgetfs* setup the USB descriptors so that a connection with a USB host is possible. *Gadgetfs* creates a file for each endpoint descriptor that is written during setup. The software uses these three files to communicate with the endpoints.

- ep0 – Endpoint zero and control pipe
- ep_in – Bulk pipe in
- ep_out – Bulk pipe out

The first, ep0 is used both for USB standard requests as well as for the control pipe used in the protocol developed in this thesis. The other two files are the connection points for the two bulk pipes that are used in the protocol. The files ep_in corresponds to the bulk pipe going to the computer and ep_out corresponds to the pipe from the computer.

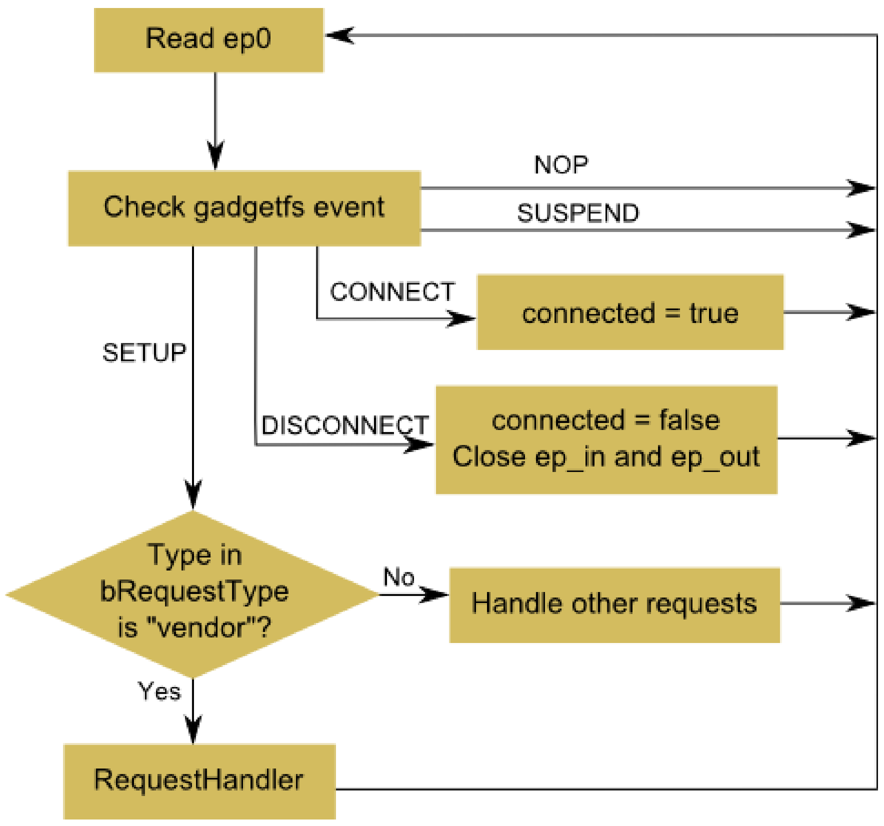

Figure 5.3 Program flow in *usbdriver*. Program always start at read ep0.

The program flow of *usbdriver* is described in Figure 5.4 as a flow chart. The *usbdriver* start by polling ep0 for updates. The following events are defined in *gadgetfs* and are read at ep0.

- GADGETFS_NOP
- GADGETFS_CONNECT
- GADGETFS_DISCONNECT
- GADGETFS_SUSPEND
- GADGETFS_SETUP

The first event GADGETFS_NOP is a no operation request specified in the USB standard. The next three events are states of the USB connection that the *usbdriver* responds to. GADGETFS_CONNECT is sent when the USB controller has got a connection with a host computer. When this event is received the connected state of *usbdriver* is set to true. When the camera is disconnected from the USB port the USB connection is lost and the GADGETFS_DISCONNECT event is written to ep0. In this case the *usbdriver* deactivates the protocol by closing the files for the endpoints and setting the connected state to false. The third event is GADGETFS_SUSPEND which is a response to the USB suspend request that a host can send to a device. There is no need for the *usbdriver* to do anything when this event is received.

The event GADGETFS_SETUP is written to ep0 when a control transfer has been received on endpoint zero. If the type in bmRequestType is “vendor” one of the requests that are part of the protocol designed in this thesis (section 4.3) is received. When one of these requests is received the *RequestHandler* will be called. It will be described in the next part of the chapter. The next section will describe the other request that are not of the type “vendor”.

For the camera to connect correctly to a host there are some USB standard requests that the *usbdriver* responds to. The *usbdriver* respond to the USB standard requests USB_REQ_SET_CONFIGURATION, USB_REQ_SET_INTERFACE and USB_REQ_GET_INTERFACE. The request USB_REQ_SET_CONFIGURATION is used to change the configuration of the camera. A configuration is a set of interfaces that is used to realize certain functionality. In this implementation there is only one configuration, the configuration to use the USB protocol designed in Chapter 4. When this configuration is sent to the camera the *usbdriver* will initiate the software by opening the files corresponding to the protocol endpoints so that communication can be done using the designed protocol.

In section 0 one interface for the protocol is defined with the setting id 0. When the request USB_REQ_GET_INTERFACE is received *usbdriver* send that the active interface setting has id 0. When USB_REQ_SET_INTERFACE is received *usbdriver* checks if the requested interface setting has id 0. If it is then the two bulk endpoints, *ep_in* and e*p_out* are reset and opened, but other interface settings will be ignored.

5.1.2 RequestHandler

The R*equestHandler* determines the type of the received request specified in section 4.3. *RequestHandler* defines the functions in Listing 5.1.

```
void request_handler_init(int __ctrl_fd, int __source_fd, int __sink_fd)
void handle_request(struct usb_ctrlrequest *setup)
```

Listing 5.1 Public functions in RequestHandler.

The function *request_handler_init()* initializes the *RequestHandler* by setting the file descriptors for the control and bulk endpoints. Figure 5.5 describes the function *handle_request()*. First the function determines the type of the request by the *bRequest* field of the setup packet. The *struct usb_ctrlrequest* is created by *gadgetfs* and stores the values of the USB setup packet described in section 2.2.4. When the request is determined the appropriate action should be taken to ensure the requests functionality. The action for each request is described below.

When the request GET_FILE_LIST is received the *RequestHandler* creates a list of all the files in the camera. The list is formatted according to Table 4.4. Then the list is sent to the host using bulk transfers written to the file *ep_in*. For the GET_FILE request, the file name is read from the data packet. To get the data packets from gadgetfs, the ep0 file is should be read again. When the file name is transferred it is used to locate the file. Then the file is sent on the bulk pipe using bulk transfers. If the requested file is not found in the file system one byte with the value zero will be sent on the bulk pipe. The DEL_FILE request is handled by reading the data packet from ep0. When the file is located the file will be deleted from the camera. If there is no file found with the supplied file name the *RequestHandler* will not do anything.

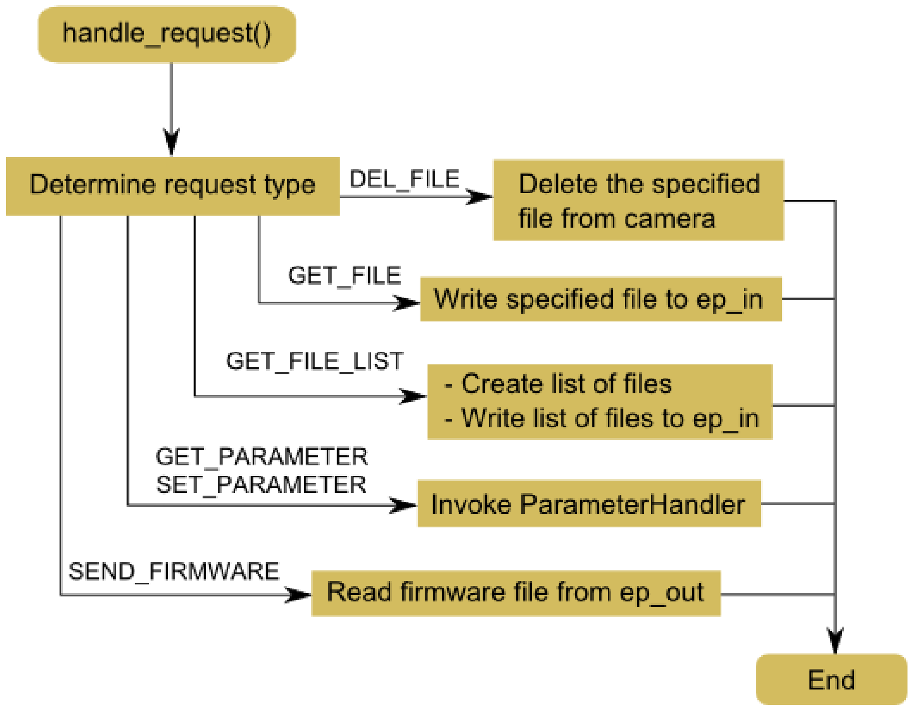

Figure 5.4 Program flow for handle_request() in RequestHandler.

The request SEND_FIRMWARE is handled by initiating bulk reads on the file *ep_out*. When the firmware file is received it will be stored on the camera memory. The actual update of the firmware is not in the scope of this thesis.

The last two requests are the SET_PARAMETER or GET_PARAMETER. When these are received the *ParameterHandler* will be invoked with its corresponding function get_parameter() or set_parameter(). If it is a get request the requested parameter will be identified and then its value will be written to the file ep0 to send the value in the data packet of the control transfer. When it is a set request the parameter value will instead be read from ep0.

5.1.3 ParameterHandler

ParameterHandler handles the parameters defined in section 4.3.1. The parameters are stored in the camera memory as binary file. When the application starts the parameters are loaded into the application for easier access. The standard c library for file access is used to read and write to the file. *ParameterHandler* defines the functions in Listing 5.2.

```
int get_parameter(int param, unsigned char *value, int *size)
int set_parameter(int param, unsigned char *value, int *size)
```

Listing 5.2 Public functions in ParameterHandler.

The get_parameter() function reads the *wIndex* of the setup packet and identifies the parameter that is requested. The parameter value is fetched and then returned to RequestHandler. The RTC is not stored in the file with the other parameters. When the parameter is identified to be RTC the current time is requested from the operating system and then returned to the RequestHandler.

The function set_parameter() identifies the parameter in the same way. The value of the parameter to be set is provided to the function by the RequestHandler. Only the parameters RTC and PHOTO_INTERVAL will be set by the function. The other parameters in Table 4.3 do not need to be set since they are status parameter of the camera. If the parameter is identified to be RTC the time is set in the operating system. Then the hardware clock in the processor is synchronized with the operating system clock. In the case when the parameter is PHOTO_INTERVAL the struct will be updated with the new value. The struct will also be written to memory at this point.

5.2 Computer software

Within this thesis a simple application was implemented for the host side of the USB communication protocol. The application was made for the purpose of verifying the USB communication protocol and the camera software against the requirements. The application implements the functions described in section 4.4. The tests that are created using these functions are described in the verification of the functional requirements (section 6.3).

The application runs on a Linux computer and uses *libusb1.0* [18] to create and use the USB connection to the camera. *Libusb* is a library that allows users to setup, control and use USB connection to USB devices. Communication with a USB device is done using a *libusb_device_handle*. The handle is the library's representation of a USB device. To setup the handle a request is sent to the library to open a specific USB device. If the specified device is connected to the computer the handle is setup and can be used to communicate with the USB device. Two functions from *libusb* were used to implement the USB requests designed in section 4.3. The functions are presented in Listing 5.3 and the rest of this section describes how they are used.

```
int libusb_control_transfer ( libusb_device_handle *      dev_handle,
                              uint8_t                     bmRequestType,
                              uint8_t                     bRequest,
                              uint16_t                    wValue,
                              uint16_t                    wIndex,
                              unsigned char *             data,
                              uint16_t                    wLength,
                              unsigned int                timeout
                            )
int libusb_bulk_transfer  ( struct libusb_device_handle *  dev_handle,
                            unsigned char                  endpoint,
                            unsigned char *                data,
                            int                            length,
                            int *                          transferred,
                            unsigned int                   timeout
                          )
```

Listing 5.3 Functions from *libusb* used in host application.

Both functions take a *libusb_device_handle* as argument. This is used to specify the device that should receive the transfer. Both functions have the parameter *timeout*. The *timeout* is set in milliseconds and the function will return after that time regardless of the transfer's success. If *timeout* is set to zero, the functions will only return on a successful transfer.

The function *libusb_control_transfer* creates a control transfer to the specified USB device. The parameters in the function are setup as the different parts of a USB setup packet (section 2.2.4). The parameter *data* is used to send the data packet of the USB setup packet.

The function *libusb_bulk_transfer* is used to communicate over the two bulk pipes specified in the protocol (section 0). The parameter *endpoint* is used to select which one of the bulk pipes the transfer should use. If *endpoint* is set to the value used for the *in bulk pipe* the function is used to receive data from the camera. The function is used to send data to the camera if *endpoint* is set to the *out bulk pipe*. The *length* parameter specifies different things if the transfer is used to send or receive data. For a send transfer, *length* specifies the number of bytes to be transferred. In a receive transfer it specifies the maximum number of byte that can be received. The parameter *transferred* specifies the actual number of bytes that were transferred. Finally the parameter *data* specifies the data to be transferred. Or in the case of receiving data specifying where the received data should be stored.

Chapter 6

Evaluation

This chapter will present an evaluation of the design and implementation of the USB communication protocol developed in this thesis. First the platform used in the evaluation is described. Secondly the test preparations and requirement verification are presented. The last part of the chapter presents the evaluation of the protocol performance.

6.1 Platform

Since there was no actual camera available when the thesis was conducted an evaluation board (see Figure 6.1) was used for the evaluation of the protocol. The evaluation board has the same processor as the one used in the camera, the Atmel processor AT91SAM9G25. The evaluation board is also equipped with a USB 2.0 connection that is the same as the one used in the camera.

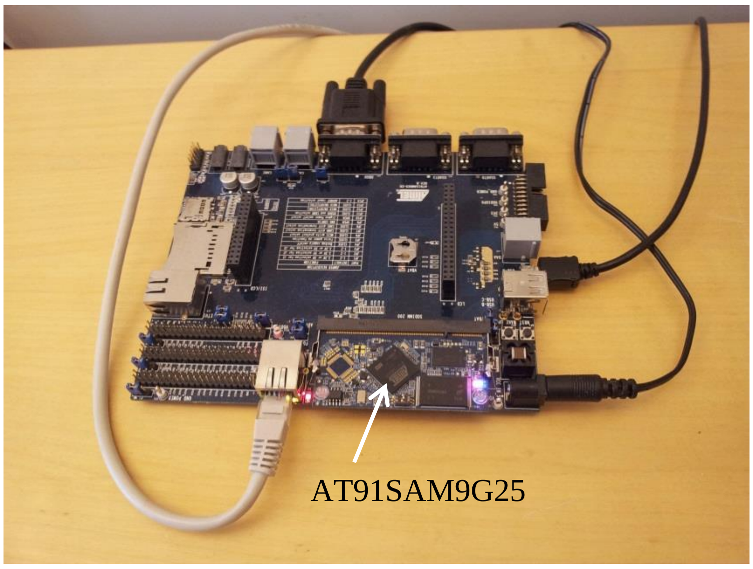

Figure 6.1 Evaluation kit for the Atmel processor AT91SAM9G25

The evaluation board is delivered with a simple Linux kernel. It also has a bootloader to allow installation of a new kernel. The initial kernel is not configured to allow the use of the USB gadget functionality in Linux. To enable this functionality the tool Buildroot[8] was used to create a new Linux kernel. Buildroot is a set of makefiles that creates a Linux kernel and a

[8] http://buildroot.uclibc.org

cross-compiler tool chain for the Linux kernel. A cross-compiler is used to create executable code for a platform other than the one on which the compiler is running.

6.2 Test preparation

The software developed for the camera was cross-compiled for the processor. Then the application was loaded into the evaluation board using the secure copy command (scp) over a secure shell (SSH) connection. Pictures and data files for the testing were sent to the evaluation board. Ten pictures with meta-files that have been gathered from a camera prototype have been used for the tests. After the preparations were made the software was started and the test procedures were invoked on the computer to evaluate the communication.

6.3 Requirement verification

6.3.1 Functional requirements tests

To verify the functional requirements a test application was developed to test the functions of the USB communication, see section 5.2. The software uses the developed USB communication protocol to communicate with the camera. The functional requirements are repeated in Table 3.1. The priority 2 functional requirements are not implemented and thus not tested. The functional requirements 1 to 11 with priority 1 are verified by executing the following tests. All the tests were passed and the functional requirements are considered to be satisfied.

Test 1: (requirement 1, 2)

The GET_FILE_LIST request is used to get a list of the files available on the camera. Then the GET_FILE request is used to transfer one of the files from the camera to the computer. The checksum for the transferred file is calculated and then compared to the MD5 sum in the file list. If they are equal the test is successful and the file has been transferred correctly.

Test 2: (requirement 1)

The computer sends the GET_FILE request with a file name that does not exist on the camera. The computer expects to get a zero byte.

Test 3: (requirement 3)

The computer requests the files on the camera and then request to delete one of the files with the DEL_FILE request. Verify that the file is deleted by requesting the file list again and checking that the file is removed.

Test 4: (requirement 4)

Send a file to the camera using the SEND_FIRMWARE request and verify that it is received.

Test 5: (requirement 5 to 9)

Send the GET_PARAMETER request to the camera for all parameters in Table 4.3. Verify that the parameters are received correctly. Also test to request parameters that are not defined and verify that that camera software is still running.

Test 6: (requirement 5, 7, 8)

Test to set the corresponding parameters using SET_PARAMETER. Verify that the parameters did not change by requesting the parameter with GET_PARAMETER. This tests that the camera software does not change the parameters that should only be read.

Test 7: (requirement 10)

To verify that the RTC is set correctly the SET_PARAMETER request is sent to the camera with the computers current RTC. Then the GET_PARAMETER request is used to get the cameras RTC and it is compared to the current time of the computer. Since the accuracy of the RTC is only in seconds the test is successful if the time difference is maximum one second.

Test 8: (requirement 11)

The SET_PARAMETER request is sent to the camera with wIndex to set PHOTO_INTERVAL and a value is sent in the data packet. Then the GET_PARAMETER request is used to verify that the camera has updated the value.

6.3.2 Non-functional requirements

This section will present the verification of the non-functional requirements mentioned in Table 3.2. The section describes the verification of the fulfillment of these requirements. The requirements 20 and 21 have priority 3 and have not been considered in this thesis and are not verified.

Requirement 14 specifies that the solution should be based on the USB 2.0 High Speed specification. The design of the USB communication protocol is based upon the USB 2.0 specification and the requirement is thus satisfied. The possibilities of the USB stack in the embedded Linux operating system have been accounted for in the design of the USB protocol. The camera software is implemented and compiled for the Linux operating system and validated for the functionality. Thus requirement 17 is satisfied. The USB functions in both Windows XP and Mac OS/X was researched for the development of the USB protocol in Chapter 4. The protocol has not been tested on Windows XP or Mac OS/X but the design support implementation on both operating systems. The functional verification of the USB protocol and the camera software has been made on a Linux computer. The functional verification combined with visual inspection of operating systems drivers concludes that an implement of this protocol on Windows XP and Mac OS/X 10.6 would be theoretically possible. This deems requirements 15 and 16 to be theoretically satisfied. Requirement 18 is satisfied and the motivation follows in the next section. Thus all priority 1 non-functional requirements are satisfied.

Requirement 19 regards the possibility to implement the protocol in user space, without creating proprietary kernel drivers. Both operating systems Windows and Mac OS/X contain libraries that allow use of the standard USB transfers (section 2.2.3) from software in user space. This will allow for the designed USB protocol in this thesis to be implemented in user space and the requirement is deemed satisfied.

6.4 Performance

According to the requirement 18, the transfer speed of the designed USB communication protocol should be close to the speed of the USB mass storage class. The performance of the USB communication is not critical. When the files have been transferred to the computer they are going to be sent to a cloud based server. One can assume that the internet connection is going to be slower than the USB. USB 2.0 has a theoretical throughput of 60 MB/s with an actual throughput of around 35 MB/s according to Alan Norton [19] and an internet connection speed of between 1-12 MB/s are common today.

To verify the requirement 18, a mass storage device was set up on the evaluation kit. The *Mass Storage Gadget* from the Linux-USB Gadget API framework was used. For the verification, an application on the host computer opens a picture from the mass storage device and then saves it to the computer hard drive. The same was done for the protocol in this thesis. The file list was requested and then a file was transferred and stored to the computer hard drive. The time of the transfers were measured by taking the difference of the time before the file transfer started and the time when the file was written to the hard drive. This was done for both the protocol developed in the thesis and the mass storage class.

Table 6.1 The files used to test the transfer speed of the protocol including: size of the files, the average speeds in MB/s and standard deviation in parenthesis. Transfer speeds are measured for the Mass Storage Class (MSC) and the USB protocol (USB) developed in this thesis.

Number	Name	Size (Byte)	MSC (MB/s)	USB (MB/s)
1	img_130313_144616.jpg	1180560	12,668 (1,496)	18,198 (0,396)
2	img_130313_144618.jpg	829840	9,399 (2,253)	17,746 (0,972)
3	img_130313_144620.jpg	829840	6,979 (2,397)	17,867 (0,231)
4	img_130313_144622.jpg	1329040	12,319 (2,837)	17,963 (0,971)
5	img_130313_144626.jpg	1239440	10,797 (2,961)	18,317 (0,208)
6	img_130313_144628.jpg	1293200	10,044 (2,828)	18,309 (0,443)
7	img_130313_144630.jpg	1183120	9,496 (3,111)	18,253 (0,269)
8	img_130313_144632.jpg	1318800	7,951 (2,387)	18,429 (0,161)
9	img_130313_144634.jpg	1175440	8,421 (2,117)	18,314 (0,140)
10	img_130313_144650.jpg	1334160	9,957 (1,180)	18,419 (0,198)

The first measurements of the mass storage class transfer speed showed some interesting results. The measured speed of the transfer was over 100 MB/s. This speed is beyond the theoretical limit of the protocol, which was surprising. When it was analyzed why this problem occurred it was found that the USB mass storage driver in the Linux operating system sets up the mass storage device as a Small Computer System Interface (SCSI) device. The SCSI devices in Linux uses cache memory and this explained the high transfer speed. The transfer speed was high because the computer only read the files from local cache memory. In the Linux operating system, the command “echo 3 > drop_caches” was used to clear the read cache between the transfers. This forced the file to be fetched from the USB mass storage device.

The pictures in Table 6.1 have been taken with a prototype of the actual camera and these have been used to test the transfer speed of the protocol. In Figure 6.2 the transfer speed of the mass storage transfers and the protocol transfers are visualized. The transfer speed is the average speed of each picture sent thirty times.

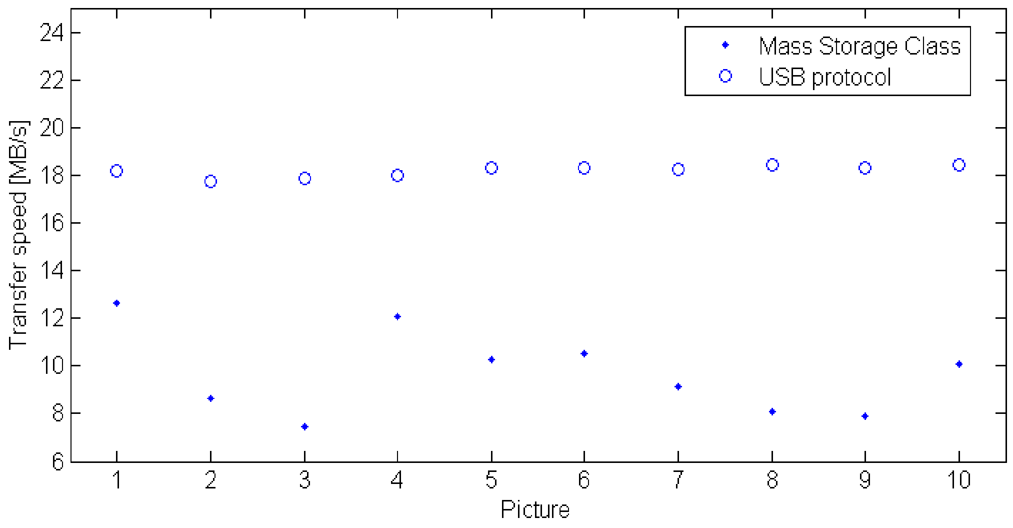

Figure 6.2: Average transfer speed for the files in Table 6.2 transferred 30 times.

The standard deviation for the transfer speeds with the Mass Storage Class is quite higher than that of the USB protocol in this thesis (Table 6.1). The cause of this variation has not been investigated as it is outside the scope of this thesis. The results are considered to be accurate enough to use in the verification of requirement 18. Since the result of the measurements show that the protocol developed in this thesis is faster than the Mass Storage Class, requirement 18 is fulfilled.

Here follows a possible clarification for why the standard deviation and result differ. Both the mass storage transfer and the protocol are based on the same USB transfer, the bulk transfer. By this fact alone the speeds should be fairly equal. The big speed difference could be due to some internal part of the mass storage driver in the operating system or the Linux driver on the evaluation board. The result is anyway deemed to be a fair measurement of the transfer speed.

6.5 Security analysis

This section makes a basal security analysis of the implemented USB communication protocol.

By using a bus listener on the USB connection the different requests of the protocol could be analyzed and identified. With that information the computer side of the implementation could be reverse engineered and malicious software could communicate with the camera.

The function to send firmware update to the camera is very vulnerable for attacks against the camera in the current version. For now there is no method for ensuring that the file sent is actually a valid upgrade of the firmware, thus making it possible to send any files to the camera. This could be a potential source for infecting the camera with malware.

Since the computer side of the protocol don't use any specially designed kernel drivers, our protocol implementation do not open up any new attack vectors to the host computer kernel. The software that implements the protocol on the computer could potentially be compromised by an infected camera since there is no verification of the validity of the sent files.

The easiest method of securing the protocol as it is designed in this thesis would be to implement the reverse-safe authentication, Lee et al. [8]. That would still allow for reverse engineering the protocol but any data sent over USB would be encrypted. To get access to the user's identification and password a brute force attack on the host computer would be needed.

Chapter 7

Concluding remarks

7.1 Conclusions

This thesis describes the investigation and design of a USB based picture transfer protocol has been developed. The communication protocol allows for transfer of pictures and meta-information from the camera to a host computer. The protocol also provides a means to read status information from the camera. Further provides the host with a way to set the cameras real time clock and other configuration parameters in the camera.

The designed protocol is based on studies of the USB 2.0 specification and articles regarding USB communication with Linux based USB devices. One big challenge in developing the protocol was to accommodate the different operating systems that were specified in the requirements. The native USB support in the two operating systems Windows and Mac OS/X differs quite a bit. Also the USB support differs greatly between versions of Windows as well. Where Windows XP that is required to be supported have more limited USB support than later editions of Windows. If the requirement had been on newer version the solution would probably be different.

When evaluating the USB communication protocol it has been concluded that the specified requirements are satisfied. The security aspects of the USB protocol and architecture were addressed. The scope of this thesis did not permit to include an implementation of a security solution. Although an analysis of the different security solutions described in section 2.1 suggests that the method for authentication and encryption proposed by Lee et al. [8] would be possible to adapt to the protocol developed in this thesis.

7.2 Further work

The current version of the USB protocol developed in this thesis relies on the assumption that the protocol is hidden. However, security aspects of the protocol could be further developed according to section 2.1. To implement the proposed authentication of Lee et al. [8] would be a way of introducing security the USB protocol by adding an authentication and encryption of the protocol. This addition to the protocol would not require a redesign of the base of the protocol.

The camera’s software today only uses one thread. When the camera is transferring pictures other functionality is unavailable. A possible improvement is to make the software threaded with one thread per pipe. This would allow a host computer to get status updates from the camera at the same time as transferring pictures. Given that the host software also is threaded.

A completely different solution to the communication could be considered. A solution could probably be made where the camera software communicates directly with the cloud servers. This solution would only use the host computer as a proxy for relaying the cameras request to the server.

Bibliography

[1] B. Gatliff, "Linux-based USB Devices," 17 Dec 2002. [Online]. Available: http://www.embedded.com. [Accessed March 2013].

[2] D. V. Pham, A. Syed and M. N. Halgamuge, "Universal serial bus based software attacks and protection solutions," *Digital Investigation,* vol. 7, no. 3-4, pp. 172-184, 2011.

[3] M. Jodeit and M. Johns, "USB Device Drivers: A Stepping Stone into your Kernel," in *2010 European Conference on Computer Network Defense,* Berlin, 2010.

[4] Y. Li and L. Shi, "Design and implementation of encryption filter driver," in *Fourth International Symposium on Computational Intelligence and Design,* Hangzhou, 2011.

[5] Z. Wang, R. Johnson and A. Stavrou, "Attestation & Authentication for USB," in *IEEE Sixth International Conference on Software Security and Reliability Companion,* Gaithersburg, 2012.

[6] Z. Wang and A. Stavrou, "Exploiting smart-phone USB connectivity for fun and profit," in *Proceedings of the 26th Annual Computer Security Applications Conference* , New York, 2010.

[7] K. Lee, H. Yeuk, Y. Choi, S. Pho, I. You and K. Yim, "Safe Authentication Protocol for Secure USB Memories," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications,* vol. 1, no. 1, pp. 46-55, 2010.

[8] K. Lee, K. Yim and E. H. Spafford, "Reverse-safe authentication protocol for secure USB memories," *Security and Communication Networks,* vol. 5, no. 8, pp. 834-845, 2012.

[9] V. Gupta, M. Wurm, M. Millard, S. Fung, N. Gura, H. Eberle and C. S. Shantz, "Sizzle: A standards-based end-to-end security architecture fir the embedded Internet," *Pervasive and Mobile Computing,* vol. 1, no. 4, pp. 425-445, 2005.

[10] H. Wang, B. Sheng and Q. Li, "Elliptic curve cryptography-based access," *International Journal of Security and Networks,* vol. 1, no. 3-4, pp. 127-137, 2006.

[11] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC and Philips, "USB 2.0 Specification," USB implementers forum, 2000.

[12] "USB Class Codes," 7 Dec 2011. [Online]. Available: http://www.usb.org/developers/defined_class. [Accessed 2012].

[13] S. Kolokowsky and T. Davis, "Common USB Development Mistakes - You Don't Have To Make Them All Yourself!," Cypress Semiconductor, 2006.

[14] "Unniversal Serial Bus Mass Storage Class Specification Overview," USB-IF, 2003.

[15] "Linux-USB Gadget API Framework," 2005. [Online]. Available: http://www.linux-usb.org/gadget. [Accessed November 2012].

[16] R. Reguphaty, Bootstrap Yourself with Linux-USB Stack: Design, Develop Debug and

Validate Embedded USB, Cengage Learning, 2012.

[17] K. Kim, J. Kim and A. Deep, "Throughput Improvement for Ethernet over USB," in *The 18th IEEE International Symposium on Consumer Electronics*, JeJu Island, 2014.

[18] "LibUSB," 20 April 2012. [Online]. Available: http://www.libusb.org/. [Accessed March 2013].

[19] A. Norton, "10 things you should know about USB 2.0 and 3.0," 2009. [Online]. Available: http://www.techrepublic.com/blog/10things/10-things-you-should-know-about-usb-20-and-30/1265. [Accessed 5 Apr 2013].

På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Marcus Högberg