Firmware loading using standardized protocols

LUND UNIVERSITY

Linus Walleij

Supervisors: Per Andersson, Embedded Systems Design Laboratory, Department of Computer Science, Lunds Tekniska Högskola

Per Ståhl, Ericsson Mobile Platforms, Lund

2004-11-15

Abstract

Most embedded systems manufactured today will involve a microprocessor with some associated computer program known as its *firmware*. To this date, most firmwares have been deployed using custom protocols. This thesis seeks to investigate a number of *de facto* and *de jure* standard protocols that can be used for firmware loading across the Unified Serial Bus (USB). It is found through prototyping that a small, embedded network is a viable and scaleable solution for loading firmware files into embedded systems. Using this network, firmware files may be distributed across subsystems in a secure, fault-tolerant manner. The solution is compatible with current software signing schemes but must be scrutinized for security vulnerabilities.

Abstract

Flertalet av de inbyggda system som tillverkas idag innehåller en mikroprocessor och ett tillhörande datorprogram som brukar kallas dess *firmware* (ung. "stabilmjukvara"). Hitintills har sådana mjukvaror mestadels lagrats i systemen med hjälp av olika specialskrivna, skräddarsydda protokoll. Detta examensarbete går ut på att undersöka ett antal *de facto*- och *de jure*-standarder för protokoll som kan användas för att lagra sådan "firmware" med hjälp av Unified Serial Bus (USB). Genom utveckling av en prototyp visar det sig att ett litet, inbyggt nätverk är en framkomlig väg och ett skalbart alternativ för att lagra "firmware" i inbyggda system. Genom att använda detta nätverk kan "firmware" på ett felsäkert vis skickas till olika delsystem med bibehållen integritet. Denna lösning är kompatibel med vanliga mjukvarusigneringsmetodiker men måste hårdgranskas för att finna eventuella säkerhetsluckor.

Contents

1 Acknowledgements

I want take this opportunity to thank to thank my supervisors Per Andersson and Per Ståhl for their support and practical guidance. I also want to thank my employer Ericsson Mobile Platforms and especially Marcus Bodensjö for coming up with the idea. And to everybody at my department: Kennet, Erik and Lina for helping me out when I had problems with the tools or concepts. Thanks to the development tools organization for answering all my amateurish questions and why-did-you-shoot-yourself-in-the-foot styled problems.

A special thank you to Spjalle and Zingo for advice on the ARM 9E processor and other ASIC pecularities. Thanks to Mats and Mikael for USB driver advice and answering weird questions. Thanks to Morten Christiansen for extensive and generous expert USB advice and interest in the project, and for tirelessly answering my sometimes not so clever questions.

Thanks to my family, Eva and Ivar for supporting me, from a certain point of view you are all that I have.

2 Introduction

> It's not that we *use* technology, we *live* technology. Technology has become as ubiquitous as the air we breathe, so we are no longer conscious of its presence.
>
> – Godfrey Reggio[1]

The world has changed. The natural habitat for our species is no longer nature: it is technology. I have come with great regret to this conclusion. *Regret* is indeed the word; other people view technology as outright evil, often seeing it as serving the same corruptive influence over the 'noble savage' as Rousseau saw in the 'barbaric civilization' of the 18th century. (But such a view is still too narrow: technology is not transforming the life of the individual human or even man the species, it is transforming the entire ecosystem of this planet.) Others yet, will claim that technology is the source of all good, and those are known as extropists and transhumanists. I still cannot decide for which camp breeds the worst crackpots, and will stay with my regret.

[1]Godfrey Reggio is the director of *Koyaanisqatsi* (1983) and recently *Naqoyqatsi* (2002).

The *embedded system* lies at the heart of this change. When we say *system* we mean an artificial system of digital electronics. We know it is a digital automata, most typically containing a microprocessor. It is a special-purpose digital machine. The general trend of embedded systems is that they are decreasing in size and increasing in complexity and functionality. The *embedded* system is built into something else, typically an artifact serving a special purpose.

The first embedded systems in a more general sense may very well have been mechanical regulators and other such gimmicks studied by the scholars of automatic control. The first embedded *computer* system was, as far as I can tell, the Apollo Guidance Computer deployed within the Apollo spacecrafts during the 1960s.[2] It was developed by Charles Stark Draper, MIT Instrumentation Laboratory[24]. Sweden was not far behind, embedding the CK37 computer in the *AJS Saab37 Viggen* fighter airplane, developed by Datasaab only a few years or even months after the Apollo Computer[5].

The ghost in the machine of an embedded system is its *firmware* – a piece of software tailored for running the system. (Later in this Section we will make a more formal definition of what we mean by firmware.) Getting the firmware into the system was once a problem for a few select software engineers, working closely with the system. It would be installed at fabrication or even hard-coded into custom circuits (today known as ASIC:s, Application-Specific Integrated Circuits). It was small, non-replaceable, written once and running in the system until its retirement.

Recently, the nature of firmwares have changed: they are no longer fixed at fabrication: instead they are loaded over and over again: they are large and bug-ridden so end-users often have to upgrade the firmware of their products in the field. Developers at companies producing embedded systems repeatedly reprogram the firmware in iterative cycles during product development, even to the point of wearing out the semiconductor electronics storing the program.

As a result, the process of loading firmware has become an increasingly important issue. This thesis is about firmware loading, and how we shall be able to standardize protocols used in this process and lift firmware loading to a higher level.

[2]Though some would point out certain World War II-artifacts as embedded systems, such as digital machines as Bletchley Park, e.g. the Polish "Bombes" used by Alan Turing, the Colossus special-purpose computer etc. The analog computer used in the German V-2 rockets may also be considered as part of an analog embedded system and so on.

2.1 Reading advice

The structure of this thesis is as follows: after this introduction follows a definition of *firmware*. After this a separate Section, 3, deals with firmware in the context of embedded systems, and subcomponents such as *Boot ROM*, *firmware loader* etc. If the reader feels confident in these areas, the inaugural Sections may be skipped.

The next Section, 4, will introduce the goal of this thesis work, and must be read in order to understand the general structure and meaning of the thesis. It also defines the frame for the prototype described in secion 9.

Section 5 will describe the Universal Serial Bus (USB) as used in embedded systems. Knowledge in this subject is not yet commonplace among electronics and computer engineers, but a reader who already knows the subject well may just as well skip this Section and proceed to Section 6.

Section 6 will discuss the Device Firmware Upgrade Class, and also why it has not been used in the present thesis. This class has its virtues and should be understood by anyone seriously interested in firmware loading. This Section is important to read in order to understand the rest of the thesis.

Section 7 will present the low-level and mid-level infrastructure that was finally chosen for the prototype implementation. This Section is a large and necessary read.

No computerized systems today are created without careful security considerations. For this reason Section 8 briefly describes the security issues identified within embedded systems in general and networked embedded systems in particular. The lessons learned from this Section directly applies to the next Section, 9, which describes the prototype implementation of a networked firmware loader.

In Section 10 we finally conclude the thesis and findings.

2.2 Firmware

In order to understand what is at the heart of this thesis, we have to define what we mean by *firmware*. In the context of this document, *firmware* will refer to:

> A software program for an computerized embedded system, consisting of executable code in a format suitable for the CPU (or multiple CPU:s and accelerators[3]) of a certain computerized

[3]By *accelerator* we mean a small custom silicon device or IP-component external to the CPU(s) and dedicated to a certain function such as, for example, iDCT decoding.

> embedded system, activated at normal start-up (bootstrapping) and executed by the CPU.

Other, broader defitions are possible, for example this one from the USA Federal Standard FS-1037C[15]:

> Firmware: Software that is embedded in a hardware device that allows reading and executing the software, but does not allow modification, e.g., writing or deleting data by an end user.

The firmware may be a single-threaded[4] program running in an execution loop, interspersed by hardware-triggered interrupt handling routines (IRQ handlers). A more complex embedded system will typically have a firmware containing a real-time operating system facilitating preemtible task-switching and priority scheduling of different tasks. Such systems have names like OSE, VxWorks or eCos. It may just as well even be a fully POSIX-compliant operating system such as μCLinux.

The firmware will *always* contain device drivers facilitating the peripheral devices connected to the embedded system through hardware ports and visible to the software developer as memory-mapped or software port oriented registers.[5]

The firmware may or may not include FPGA configuration streams for configuring silicon at start-up, such as is used in abundance for prototyping and research, and getting more and more common also in deployed embedded systems. This case could be called firmware-in-firmware and also blurs the line between hardware and software.

3 Firmware in Embedded Systems

The firmware in a modern embedded system is currently just a simple binary file[6] that is either hardcoded into the system ASICs or programmed into a system PROM, EPROM or E^2PROM (Flash ROM) at manufacturing time.[7] Some devices will even place the firmware on an intrinsic hard disk and have it loaded into RAM memory at boot time.

[4]"Single-threaded" is of course a retronym: in the beginnings of Computer Science, nobody knew about threads.

[5]Most modern I/O is memory-mapped, only certain legacy architectures still rely on software ports.

[6]With the word *file* we understand an ordered sequence (stream) of bytes (octets) typically running from low to high addresses.

[7]PROM is *Programmable Read-Only Memory* whereas the prefixes *Erasable* and *Electrically Erasable* make up for the rest.

All firmware that is not hard-coded into ASICs and reside in discrete components is possible to replace after manufacturing, using more or less sophisticated methods.[8]

If the firmware resides in an electrically reprogrammable memory type (E^2PROM or hard disk) and the hardware has connected the apropriate *erase* and *write* signals[9] to the hardware/software interface (memory-mapped registers and the like) it can be reprogrammed by a computer program running in the system.

If either the Boot ROM (see following Section) or the firmware itself supports such programmatic control of the *erase* and *write* signals, and additionally opens up an external I/O channel for incoming files adhering to some specified protocol, the firmware can be reprogrammed using another system such as a PC workstation (a desktop computer) with the apropriate cabling. Such a program, or component in the firmware or Boot ROM, is known as a *firmware loader* (see 3.2 on page 9).

The current firmware upgrade or install scenario, recognized from industry, consists of an embedded system with a single CPU downloading a file with firmware from a host system over some simple communication link such as a serial port.

However, many of the embedded systems created today have multiple CPU:s. Sometimes the other CPU:s are slave processors, like a Digital Signals Processor (DSP), or a specialized controller. These are often configured at Boot time, using some specialized hardware mechanisms which make it possible for the main CPU to halt the slave CPU, send a small program to the memory used by this CPU, and reset the slave CPU. The firmware for the slave CPU:s are thus effectively a part of the firmware for the main CPU. Such slave processors thus take the role of a peripheral that can be controlled by the master CPU.

Several systems have however already emerged that feature multiple master CPU:s, which may share a common bus or communication link, but which boot independently and work autonomously. In this scenario, it is not so certain which CPU should take care of firmware loading for the other, or if the firmware should be upgraded independently for each CPU. Also, in the near future, most of the electronics and computer industry recognize a scenario where embedded systems have several CPU:s on a single die, forming a multiple-processor System on Chip (SoC).

With the number of interdependent processors in an embedded system

[8]A hard-coded ASIC could of course also be replaced, though the cost for doing such operations usually exceeds that of building an entire new system.

[9]These signals are used here to denote the process of *replacing memory or hard disk contents*.

ever increasing, the need for a clear standard on how firmware is to be loaded into the different CPU:s arises. If the systems are to be hindered from growing into the ever more complex cobweb of a Rube Goldberg-machine,[10] abstraction levels and standard protocols desperately need to be built.

Whereas the present thesis mainly deals with the scenario of a desktop computer connected to a single-CPU embedded system for firmware loading, the concepts developed aspires to be scaleable to embedded systems with several independent CPU:s. See further Section 6 on page 20.

3.1 Boot ROM

In practice, the firmware of a complex embedded system is not a single file in some storage, but two files in two different storage areas. The first part is always available when the system is powered up, and typically involves setting the program counter of the main system CPU to a hard-coded address in a small ROM which resides either on a printed circuit board (PCB) or inside an ASIC. This part is generally referred to as a *Boot ROM* or *BIOS* an acronym for *Basic Input-Output System*.[11]

The second part constitutes the main firmware and may need to be loaded into main memory by the Boot ROM.

An alternative approach is to construct an ASIC with some specialized state-machine, which halts the CPU until this specialized machine has had the opportunity to initialize some memory with a firmware from some external source, and then sets the program counter of the CPU to this firmware block and unhalts it.

As most engineers do not want to rely on executable firmware being readily available in the systems solid state memories or hard disks, such Boot ROMs almost always contain a simple firmware loader, i.e. a means to insert code into the main memory from some kind of serial port or similar, and execute it. A common approach is to use a JTAG interface for such firmware insertion,[12] so that the Boot ROM is instructed to listen to the

[10]Rube Goldberg was a cartoonist, famous for his machines, often a complex pataphysical device with a myriad interconnected components[25].

[11]The term BIOS actually refer to a ROM which, apart from Boot-up functionality, also contain some basic Input-Output routines in machine code for e.g. writing text to I/O ports or reading hard disk sectors from a certain interface.

[12]JTAG (which is actually the standard IEEE 1149.1) is an acronym for *Joint Action Test Group*, a group at IEEE that standardized this port type. JTAG is essentially a 4-pin serial port, originally intended for boundary scan of daisy-chained ASICs, but which has since been used for all kinds of things in embedded systems, notably debugging.

JTAG interface for incoming firmware at Boot time, and if such firmware arrives, download[13] it into the solid state memory or hard disk holding the main firmware.

3.2 Firmware Loader

By a firmware loader we mean a small program for an embedded system, loaded from the Boot ROM or from the operating system, running in a single thread, and whose only purpose is to serve firmware installation and troubleshooting capabilities to the embedded system.

The firmware loader is in most cases a part of the common Boot ROM or the operating system, but it may just as well in turn be loaded on demand from an external entity in order to keep this functionality from taking up valuable memory. This is especially the case when the loader has large and diverse functionality; in such cases a simple loader in the Boot ROM or operating system is employed to download the firmware loader through e.g. JTAG into the RAM of the system.

Once there, the firmware loader can help out in reprogramming solid state or disk storage with new contents, but also to do so selectively, and provide data dumps and debug information from inside the system. Another area of concern for the firmware loader is to customize certain features of the hardware or software by changing the contents of singular memory locations scattered around the system. It may also contain diagnostic tools of the kind that need to be run outside the operating system.

4 The Task

When I was appointed this thesis work, the starting point was:

- I was given an embedded system with a single-threaded execution ARM 9E[14] CPU unit, a serial port and a USB port. (The peripheral controllers had all been integrated into the same ASIC.)

[13]The words *download* and *upload* have an ambigous meaning. In the context of firmwares, *downloading* is the process of installing firmware onto an embedded system (so that the embedded system is “down the stream”) whereas *uploading* firmware means copy the firmware that exists in an embedded system to the host system.

[14]ARM reads out “Acorn RISC Machine” and RISC reads out “Reduced Instruction Set Computer”, 9E is simply the version. The instruction set of this processor is purposedly similar to that of the 8-bit MOS Technology 6502 processor.

- I was to locate standardized protocols to be employed for firmware loading.
- I was to implement a prototype firmware loader using the existing codebase and development tools for the embedded system, utilizing the standard protocols that seemed fit.
- It was desirable that the firmware loader should be usable from a host system without installing any extraneous drivers into it, e.g. standards all-the-way in practice, not just in theory.[15]

On this system, firmware could be loaded using custom protocols over both the serial and USB ports. The Section on implementation (Section 9 on page 46) will go into details about the actual implementation, whereas the following Sections will deal mainly with theoretical aspects of the goal set by this starting point.

I quite quickly decided to focus on solutions built on the USB bus (see following Sections), since there has been a lot of productive standardization work surrounding USB, and these standards had also been implemented in several operating systems available off the shelf.

5 USB for Embedded Systems

USB has been developed partly as an attempt to create what is called a "legacy-free PC". This means that old, large and troublesome ports on computers, like RS232 (serial), PS/2 (serial) or IEEE 1284 (Centronics, parallell) are to be phased out and replaced solely by USB. The numerous USB-featured peripherals for the PC architecture introduced lately gives at hand that this will also succeed.

Before we dig deeper into how the USB bus can be applied in firmware loading, we must make a brief overview of the standard as such. When you set out to use USB in a project, you quickly get overwhelmed by the sheer size of this standard.[16] The reference document specifying USB[20]

[15]Computer Scientific standardization suffers from a lot of unused (or even *useless*) standards that have not been implemented anywhere else than in research labs. a bunch of "standards" written by ITU-T (such as the Open Document Architecture) especially come to mind.

[16]USB is a *de facto* standard, not blessed by any standards body such as the IEEE. The USB Implementers Forum which supply the standard also pool and guide a number of patents relating to this standard, so the standard is effectively controlled by the member companies of USB IF.

deals with all details of the USB bus at great length, and as such varies from easy reading to unreadable. In particular, it spans at least three engineering disciplines: mechanics (connector sizes and similar), electronics (how signals are modulated on a media) and computer science (abstraction and protocol issues).

As my target system was given, the mechanical and electrical properties of the USB bus was secondary: what I needed to get at was the communication channel abstraction and protocol issues. For this purpose, Craig Peacock's *USB in a NutShell*[16] serves the purpose very well, as it gives the basic details and reading instructions to help out in comprehending the standard document.

Another excellent source of information for implementers is to try to communicate with some USB devices using the *libusb* software library[12], available for most operating systems. Looking into code written for *libusb* will help implementers to grasp the high-level software structure in all systems using USB. That said, we will try to pry out the basic knowledge of USB needed to understand this thesis.

5.1 Low-level USB

The USB bus is a tree-like master-slave bus. The master of the bus is called a *host controller* and is typically built into a desktop PC or similar. The host controller is responsible for controlling all traffic on the bus, and slave devices cannot initiate traffic on the bus themselves: instead the bus is polled, one device at a time, and the devices (also known as *functions*) will then have an opportunity to talk to the host controller.

The mechanical connectors of the USB cabling have been devised so that they have an upstream end and a downstream end. *Upstream* means in the direction towards the host controller, whereas *downstream* is in the direction towards the attached devices. The path from host controller to device can pass through several *hubs*, which fork the connections so that up to 127 devices can eventually be connected to one and the same host controller. A *root hub* always exist in direct connection with the host controller.

The USB bus cable consists of four wires, two of these are for +5 V power and GND, and the remaining two are a twisted pair[17] named D- and D+ that carry a NRZI[18] signal. The power lines may be used for moderately powering a device attached to a USB host controller.

[17]*twisted pair* simply means that the two wires are twisted around each other as to cancel out any electromagnetic interference, i.e. crosstalk.

[18]Non-Return to Zero Inverted, this means that zeroes and ones are encoded in the

The signal sent across the wires is divided into packets and prepended with a synchronization sequence named SYNC, in order to synchronize the phase-locked loop (PLL) at the other end of the cable. The SYNC sequence is followed by a packet ID signifying the type of packet, an address for the device to be contacted, an endpoint number (we will describe what this is later), a CRC[19] and finally an end-of-packet (EOP) that terminates the packet. The actual contents of the packet vary. The packets are addressed to a certain device (an address which is assigned to each device upon being attached to the bus). Erroneous packets are neither acknowledged (ACK) or non-acknowledged (NAK); instead they are silently dropped.

These packets can be sent at different data rates, whereof the *full speed* rate (which was the highest available rate in the USB 1.1 specification) is at 12 Mbit/s, and the *high speed* rate, available from the USB 2.0 specification, is at 480 Mbit/s.[20]

When a piece of data is to be moved across the bus, atleast three packets are needed: TOKEN, DATA and HANDSHAKE (status). Each of these packets have the features described in the previous paragraph. A train of TOKEN, DATA, HANDSHAKE packages is called a *transaction*. (There also exist transactions without any DATA packets, so this part is optional.)

The data transferred across the wires is split in maximum sized packets, which are bit-stuffed[21] so that the bit-clocks do not loose track of the binary stream. The packets are then transmitted in the DATA portion of a TOKEN DATA HANDSHAKE packet train.

We do not need to go further into the details of these packages, it is enough to conclude that they are closely related to the physical data carrying mechanism below it and already totally encapsulated on the next, more abstract level. All USB traffic up to this level is typically handled by electronics, not computer programs.

transitions on these wires, not in the signal levels themselves.

[19]CRC, Cyclic Rudundancy Check, is a method of verifying that the contents of a package was not damaged.

[20]USB 2.0 is backwards compatible with USB 1.1 so a USB 1.1 certified device is by definition also a USB 2.0 certified device. What the average consumer recognize by "USB 2.0" is a device that supports the new *high speed* data rate, though this is not, techically speaking, the definition of USB 2.0 — USB 2.0 is only a specification.

[21]Bit-stuffing means that the signal is encoded so that sufficiently many transitions occur.

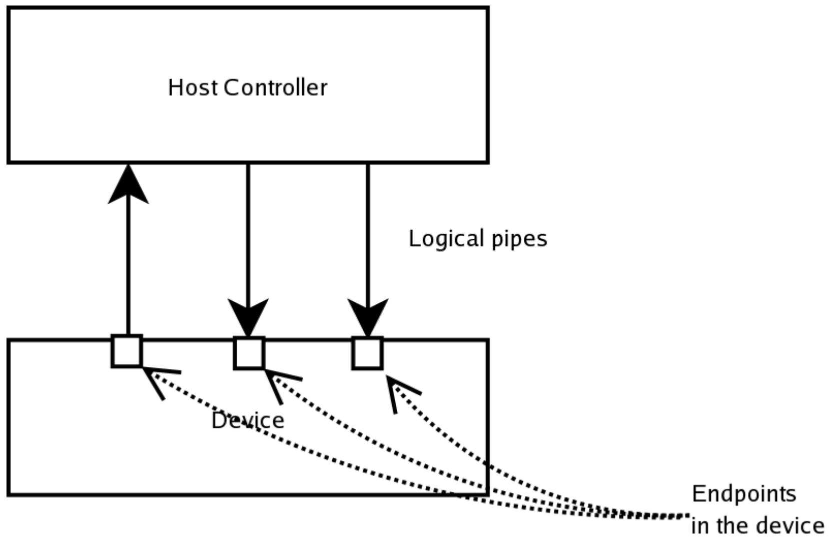


Figure 1: This Figure illustrates that endpoints exist in the device, whereas the directed byte-stream pipe will start or end in either the host controller or the device.

5.2 Endpoints

The programmer utilizing the USB bus will almost exclusively deal with software/hardware interfaces relating to so-called *endpoints*. Endpoints are endpoints of *pipes*, and pipes are channels that can send directed byte streams, just as the pipes known from Douglas McIlroys famous implementation of his pipes-and-filters framework employed in all POSIX systems.[22] Bytes that are pushed into one end of a pipe will appear undamaged on the other end. Pipes are unidirectional, which means they are either IN-type or OUT-type, where IN means *IN*to the host controller (device → host controller) and OUT means *OUT* of the host controller (host controller → device).

One end of an active pipe will always reside in the host controller, whereas the other end will appear in the hardware registers of an attached downstream device. Transmissions on a pipe will typically cause software interrupts to occur in the device when data is appearing in an OUT endpoint or when an IN endpoint is polled for new data. (Remember that all USB transfers are conducted by the host controller!)

Endpoints will *only* be defined by hubs and devices; the host controller cannot define any endpoints. Thus you may say that endpoints only *ex-*

[22]Named pipes, RPC sockets, or TCP sockets share much the same characteristics.

ist in devices and hubs, though host controller software will casually refer to endpoints as if they existed at the other end of the pipe as well. (Perhaps these should be called *startpoints* or something similar.) When some host software instructs a host controller to read or write to an endpoint, it means that it should communicate with the device having that endpoint, not that the endpoint exists in the host controller. The situation is somewhat illustrated in Figure 1.[23]

USB can support 32 different pipes for each attached device: 16 inbound *IN* pipes numbered 0 to 15 and 16 outbound *OUT* pipes numbered 0 to 15. There may be up to 127 devices attached.

Several silicon vendors have created IP-blocks for USB controllers that implement all functionality up to endpoint abstraction. These IP-blocks will differ mainly in the number of endpoints supported and the number of higher-level functions supported by the block. USB IP-blocks sometimes come with a *PHY* i.e. a physical interface to the wire, and sometimes not. The high-speed data rate of 480 Mbit/s supported by the USB 2.0 specification typically requires an external (partly analog) PHY, and a standard known as USB Transceiver Macrocell Interface (UTMI) has been developed for these PHY components. For the older, backwards-compatible USB 1.1 specification, all of the PHY can be incorporated into silicon and only an external mechanical connector is needed.[24]

There are also IP-blocks for the even more complex host controller part of the bus, which is responsible for scheduling transactions on the bus and polling each device. These are known as host controller devices and only a few standard discrete components and chipsets have appeared on the market, bearing such names as OHCI, UHCI and EHCI.[25] Embedded host controllers also exist. These are not very interesting in this context, and will not be treated further.

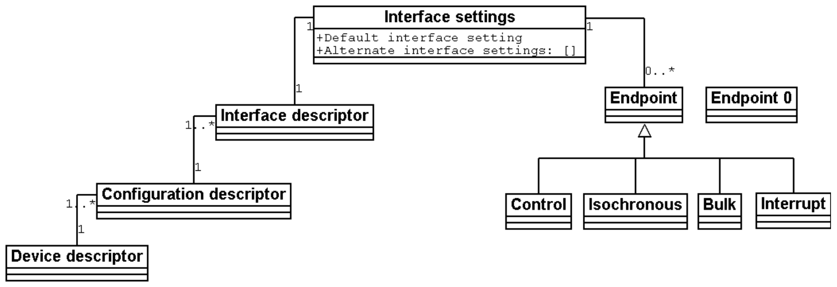


Figure 2: A UML diagram describing the basic relation between different USB entities.

5.3 Endpoint Abstractions

USB implementers could have stopped at unidirected pipes and endpoints, but have (luckily) chosen to abstract pipes into a higher level framework of device objects, which have the basic structure found in Figure 2.

What we can see from the Figure is that the pipe endpoints, which are the only communication channels that actually exist, are grouped into a hierarchical scheme, with the exception of one endpoint, *endpoint 0*. Endpoint 0 OUT (host → device) and endpoint 0 IN (device → host) *must* be implemented by all USB-compliant devices. Devices will also have to react to a set of standardized messages sent on this endpoint, known as SETUP commands, which are essentially an 8-byte sequence optionally followed by more data. The handling of these commands is sometimes carried out by hardware, and sometimes by software.

The remaining (up to 15) optional endpoints belong to one of four categories: CONTROL, ISOCHRONOUS, BULK and INTERRUPT. CONTROL endpoints are of the same type as endpoint 0 and typically endpoint 0 is the only CONTROL endpoint on a device. ISOCHRONOUS endpoints are intended for high speed non-failsafe transfers, e.g. soft real-time applications such as streaming video. BULK endpoints are for huge ("bulky"), non-time critical data transfers, and INTERRUPT endpoints are for spo-

[23]Sadly, documentation and source code relating to USB often confuse pipes for endpoints. In USB lingo, casually talking about an *endpoint* may refer to a pipe. (Ceci n'est pas un pipe.)

[24]OPENCORES have an open source USB 2.0 high speed controller and an older USB 1.1 full speed controller that can be investigated by anyone interested in learning more about USB IP-blocks, see `http://www.opencores.org/`

[25]OHCI is the *Open Host Controller Interface* from Compaq, UHCI is the *Universal Host Controller Interface* from Intel, and EHCI, the only of the three to support high speed transactions, is *Extended Host Controller Interface*, again from Intel.

radic, high-priority transfers.[26]

Needless to say, data pushed or polled from endpoints will have to be packetized and multiplexed by the low-level USB functions, typically by hardware. The different types enable the host controller to schedule endpoint data depending on their respective type.

As can be seen in Figure 2, endpoints are grouped into *interface settings*, one of which is called the *default interface setting*, belonging to a certain *interface descriptor*, and which must exist on every device implementing USB functionality. A device may have additional interfaces (which may in turn have several interface settings), described by their respective interface descriptors.

Additionally, a USB device must support at least one *configuration*, described by a *configuration descriptor*. In order for an interface descriptor and its settings to be valid, they must have been described in a configuration descriptor.

The configuration descriptor is in turn referenced from the top-level *device descriptor*, which apart from defining available configurations, also describes general properties for the device. There is ever only one device descriptor for a certain device. Two unsigned 16-bit fields given in the device descriptor are especially important: *idVendor* and *idProduct*.[27] The Vendor ID is assigned to each vendor producing USB devices by the USB Implementers Forum, and the Product ID can be used any way the device manufacturer seems fit. Some operating systems will use the combination of Vendor ID and Product ID as a key for locating a suitable device driver whenever the device is attached to the USB.

The recommended approach in the USB spec is that the operating system shall examine the *bDeviceClass* (class code) field to see if there is a standardized way to communicate with the device (see next paragraph). If this again fails, the operating system shall examine the respective interfaces to see if any interface is of a standardized interface class, because *parts* of the device *may* be using standardized protocols — this is illustrated in the Device Firmware Upgrade class, which will be introduced in the next Section. If this fails, a custom driver may be loaded with the *idVendor* and *idProduct* numbers as a key. If this also fails, the system will typically report an error to the user who just attached the device or just silently ignore it. (Note that Microsoft Windows takes a different approach: it will test a few supported device classes first, then immediately fall back on *idVendor+idProduct* keys.

[26]Note that INTERRUPT endpoints will in no way interrupt the traffic on the USB bus, it simply means that the host controller will prioritize polling these endpoints with regular intervals. Each INTERRUPT endpoint can also declare its desired poll interval.

[27]See the USB 2.0 specification[20] and Section 9.6.1.

Windows will not look for interface classes at all.)

By presenting special values in the device descriptors' one-byte *bDeviceClass* Class Code field (and additionally the *bDeviceSubClass* and *bDeviceProtocol* fields),[28] the device may declare that it belongs to a special *class* of USB devices. This means that its functionality and all related protocols are specified in a standardized class document from the USB Implementers Forum (who create all USB standards). In an ideal world, all such standard classes will be supported by all operating systems driving host controllers without requiring additional system drivers or configuration[21]. We will look into some such class specifications later in Sections 6 and 7.1. More often than not, you will find devices presenting the value 0xFF in the *bDeviceClass* Class Code field, signalling that this is a vendor-specific protocol and special software drivers are needed for the device to work. The common device classes are devised to avoid this situation.

On top of the descriptors mentioned hithereto, there are *functional descriptors* which describe certain functionality of the device, and *string descriptors* which may be referenced by other descriptors to e.g. give an interface a meaningful, human-readable name.

The rather complex object structure presented hithereto may seem to the reader like a case of creeping featurism; a typical USB device will have *one* configuration, which in turn has *one* interface descriptor with *one* interface setting — the default setting. This interface setting will typically have two endpoints: one inbound and one outbound of some type, so that the device can send and receive data. However in the case of more complex devices, e.g. a memory card reader combined with an ethernet firewall, supporting normal and low-power mode, these descriptors and interfaces have their use.

It should be mentioned that all the descriptors are in practice just a stream of bytes, retrieved from the device by the host controller via special SETUP messages on endpoint 0. The descriptors will then be interpreted by the operating system running the host controller, which may in turn load apropriate drivers.[29] The different settings presented by the descriptors, i.e. different configurations, interfaces and interface settings are also selected by the host controller by sending special SETUP messages.

For the case where two embedded systems are to communicate with each other, and no clear definition of which system will act as the host controller, a special specification named USB On-The-Go has been created,

[28]Ibid.

[29]I have yet to see a host controller that is manouvered by something else than an operating system.

but in this thesis we will only deal with "common" USB traffic.

As you have probably realized by now, it is almost implied by the USB specification that the devices (functions) connected to the host controller have to be some kind of embedded systems. All the complexity of the higher levels in the USB protocol are ill adapted for implementation in pure hardware state machines. Indeed, most if not all things you find equipped with a downstream USB port will be embedded systems of some kind.

6 The USB Device Firmware Upgrade Class

The most obvious standard choice for a firmware installation solution is the USB Device Firmware Upgrade Class[23], also known by its acronym *DFU*. This device class is devised to handle firmware upgrades for any USB device via a standardized procedure.

The principal workings of this class are the following: during normal operations, the device presents an additional interface descriptor beside those that are used during the common operation of the device. It also presents a functional descriptor, which gives some details about maximum transfer lengths and other DFU parameters. Apart from these DFU-specific descriptors, the device may be of any desired class, or even vendor-specific if need be.

The DFU interface descriptor has the interface class 0xFE (meaning "application specific class") and subclass 0x01[30] and is to be selected when the firmware is about to be upgraded. So during normal operations, the DFU run-time interface is *not* selected. When the firmware of the device is to be upgraded, the following happens:

- The DFU interface is selected by the host controller using a SETUP command on endpoint 0
- The host controller issues a USB reset by way of a command, causing the device to be given a USB address anew.
- When the device has been reset and addressed, the host controller will request its descriptors again, a process known as enumeration.

[30]The subclass 0x01 really only tell us that DFU is the first time anyone has ever defined an application specific subclass interface, which also partly explains why this standard has not yet caught on, as nobody wants to risk being first.

The device now presents a whole new device descriptor, using an *idProduct* number different from that used during normal operations.[31] It also presents a new configuration descriptor with the sharp DFU interfaces.

- The "new" interfaces are used to download a new firmware to the device.

- Once the new firmware is downloaded and verified, the host controller issues another USB reset, causing the device to reenumerate once again, assume its old *idProduct* and resume its normal operations.

This scheme has some obvious advantages: for example, the fact that the host controller resets the device gives the device an opportunity to not only change it's device descriptor and interfaces, but also provides a natural point to switch from the normal, multi-tasking operating system to a single-threaded firmware upgrade mode inside the system, initiating the program that was defined as *firmware loader* in Section 3.2 on page 9.

However, the Device Firmware Upgrade Class also has several downsides which make it apply only to a narrow class of quite simple devices.

The first downside is that it only use CONTROL transfers on endpoint 0. As most operating systems see this as low-priority traffic, only one transaction will be scheduled to this endpoint in each USB timeframe, resulting in a transfer capacity of just 64 kbps.[32] With solid state memories becoming faster and larger, the DFU threatens to become an irritating bottleneck.

The second, big downside is that it is not inherently supported by Microsoft Windows[19].[33] (Both GNU/Linux and Apple MacOS X support it natively.) This could theoretically be easily overcome by simply implementing a DFU kernel driver for Microsoft Windows outside Microsoft's domain, and supporting it separately. However the needed kernel interaction is quite massive, and it cannot be taken for granted that all necessary kernel interfaces actually exists. Perhaps an abstraction layer such as libusb could be employed to simplify the process.

[31]This is done to avoid conflicts with operating systems that use the combination of the *idVendor* and *idProduct* fields to identify system drivers for a certain device.

[32]A newer version of the DFU is in the works, which will support BULK endpoints and speed up transfers. Information from Morten Christiansen.

[33]Which is rather surprising given that one of the contributors to this standard, Tom Green, comes from Microsoft.

The third, more severe downside of DFU is that anything more complicated than simply downloading or uploading a single firmware file to/from a device is out of the question. No diagnostics, debugging, select uploading, configuration menus or other custom interfaces to the device firmware loader are possible to achieve using the DFU class only. It is obviously intended only for end-users upgrading an opaque embedded system.

These shortcomings could of course be solved by reworking and amending the DFU class, possibly defining an application layer on top of a more generalized DFU interface. This would mean quite a lot of paper-work and redoing a simple standard into something it was not intended for, so another (more realistic) possibility would be to define an all-new firmware interaction device class through the USB Implementers Forum. However, the task posed for this thesis did not involve writing new standards or custom protocols: the explicit goal of the assignment was to use present *de jure* or *de facto* standards.

The fourth and final downside of the DFU when applied to complex embedded systems is that it does not account for the development of externally and internally networked embedded systems. This fact is a turning point for the following discussion and will therefore be treated in detail.

The need for networks *between* embedded systems, and between embedded systems and host computers, is nothing new. This kind of functionality is nowadays proliferating in common household objects such as web cameras and broadband routers.

A more challenging scenario within the immediate future is the presence of networks *within* embedded systems, so as to communicate information between different functional units.

Whereas traditionally, this communication has been sent across custom buses such as an I^2C or CAN bus,[34] the recent rise in complexity of embedded systems make a case for deploying higher-level networking protocols as well, as a means to abstract away even more of these interconnections. We are thus moving from ad hoc-programmed hardware-centered buses towards a combination of hardware buses and higher level protocols that help out in structuring the information flow. What we want to achieve can be seen in Figure 3.

Apart from sheer complexity, other factors opt for constructing networks inside systems, and some peers have even started writing articles about Networks on Chips, NoC:s[4]. The reasoning for switching from

[34]I^2C is short for *Inter-Integrated Circuit* and CAN stand for *Controller Area Network*. For a more in-depth introduction to these buses, see Wolf[26].

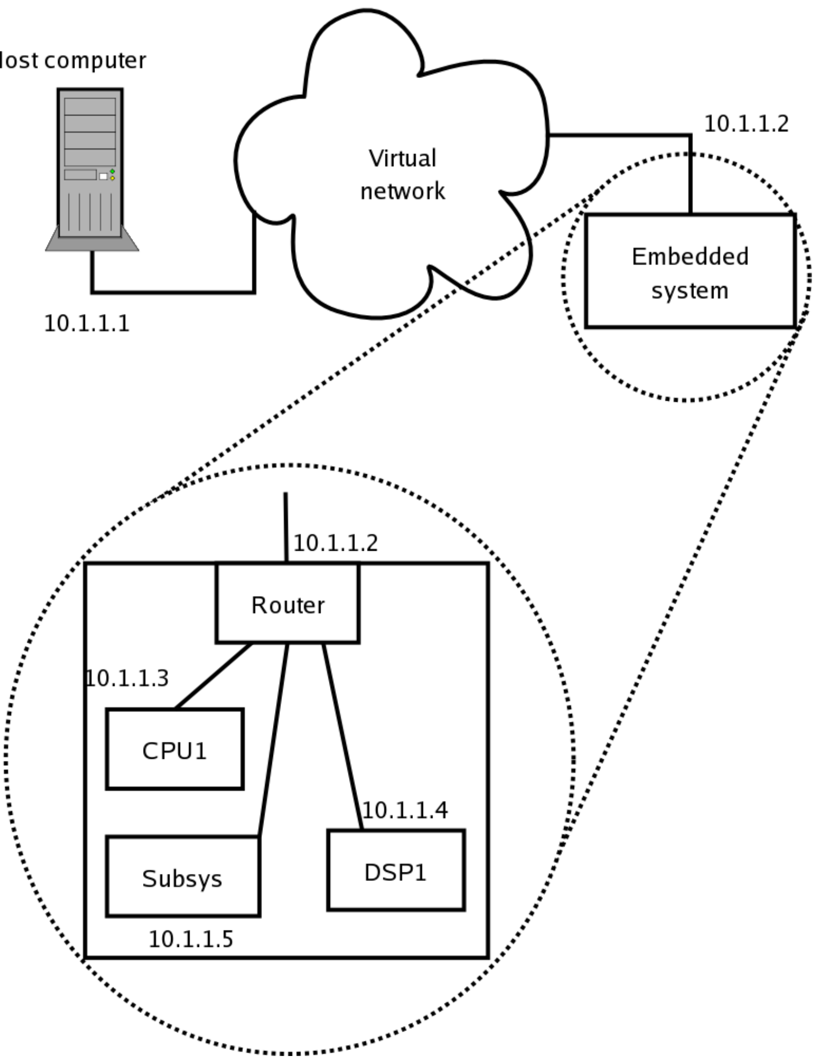


Figure 3: The external view of a network named 10.1.1.0 between a host computer and an embedded system, with a magnified system exposing its networked subsystems. The implemented prototype delivers part of this Figure: the host (10.1.1.1) and the system (10.1.1.2) but can quite easily be expanded to maintain the full scenario of interconnected subsystems.

common bus types to true networking both within systems and on highly complex chips can usually be boiled down to the following points:

- Networking increase componentization (abstraction) in designs, which is regarded as a good design pattern.
- Synchronization of buses between or within integrated circuits using a single clock become increasingly troublesome because of skew at high frequencies. Asynchronous networking can remove this obstacle.
- Determinism in large systems is increasingly hard to maintain. Unforeseen glitches, errors and sync problems necessitate a fault-tolerant transport mechanism, such as a network.[35]

Concerning the loading of firmware into embedded systems, addressing an ever increasing number of processor cores and memories will increasingly trouble developers of firmware loaders. Networking the different components of a system at a higher level and assigning each subsystem an IP-address[36] handily resolves this issue, as any high-level protocol may be built on top of the TCP/IP stack,[37] and distribution of firmware across network peers may occur in parallell using routing techniques well established and known from the field of computer communications theory.

The componentization provided by autonomous networked subsystems is also good if the components are coming from different vendors, as the network layers present a natural abstract high-level component interface, for example remote procedure call protocols (RPC) may be used. Figure 3 also demonstrates one of the subsystems as routing traffic to the other subsystems. This construction only serves to route network packets: from a network point of view, all subsystems are accessible on their own, and you may immediately address a certain subsystem. This is good especially when subsystems are added late in development: they may then be addressed without prior notification to other subsystems.

The DFU is not intended for this kind of scenario. The only way of using it in a networked embedded system would be to equip the device with a firmware loader that sets up an *ad hoc* firmware loading network inside the device before receiving or delivering a firmware file over DFU,

[35]The interested reader can gain an insight into the kind of physical and analytical phenomena presented by sub-micron design in Argonès et al[2].

[36]Internet protocols and IP-address will be discussed in the following Section.

[37]A *stack* is a colloquial term for the hierarchy of protocols present in a network protocol suite.

a file which must then be demultiplexed/multiplexed by a second portion of Boot ROM code and then finally distributed across the system. This will cut off the host computer from any possibility of acting as a network peer and effectively employ the simple DFU protocol as a transport mechanism for something it was never intended for. It is by no means certain that the extended firmware functionality (configuration, selective upgrade etc.) could be properly shoehorned into this transport container.

These limitation of the DFU class are so severe that other solutions had to be sought when trying to solve the assignment given for this thesis. One *could* imagine modifying or re-drafting the DFU to become a network transport mechanism, but to no avail: a device class for communications and networking has already been specified by the USB Implementers Forum, and the following Sections will deal with this mechanism in detail. The task was also again to use *existing* standards, not write new ones, and we thus switch our reasoning to a networked device context.

We shall briefly consider other alternatives to the DFU:

- One could possibly install a USB stack on each subsystem and let each system be an autonomous USB function, addressed in turn by the host controller. This would however require routing of the USB bus signal layer across silicon, which is a subject of analog electronics and not the easiest thing. It might be feasible by routing signals at a higher level in the USB stack, but this may in turn violate the idea of USB.

- One could imagine the firmware interface to be a file system and present it using the USB Mass Storage Device Class. This approach is similar to the configuration file system found in Linux' */proc* and */sys* file trees, and may be a viable alternative.

Having to focus on one solution for my prototype, I chose networking. The main reason behind this choice was that TCP/IP networking opens the possibility to use a web server in the firmware loader, so that a simple graphical user interface can be built using accepted Internet standards only. File systems and USB buses do not inherently support user interfaces. We will illustrate this possibility towards the end of this report.

7 Networking over USB

Wise from the lessons of DFU shortcomings, we turn to the task of externally and internally networking embedded systems from the ground up.

The most successful standard protocol for exchanging information over a network is without doubt the Transmission Control Protocol / Internet Protocol — TCP/IP for short. The merits and success of this protocol has been well documented elsewhere[1]. When we talk about "networking" in the following Section, TCP/IP networking is implied at some level.

Like all Internet standards, the TCP/IP protocol is documented through a series of Request For Comments (RFC) documents. RFC 791 (IP), RFC 792 (ICMP), RFC 826 and 903 (ARP and RARP), RFC 793 (TCP) and RFC 768 (UDP) make up most of the protocol specification that will be considered in this thesis. All these RFC:s are available from the Internet Engineering Task Force homepage[38] and the reader is assumed to be fairly acquainted with these protocols. A textbook such as Forouzan[9] may help, but in general the RFC documents are quite readable by themselves.

The general idea here is to make the firmware loader program from Section 3.2 on page 9 fully TCP/IP-enabled and recognized as a common network device. A problem with this approach is that we need to devise some model for switching the device into firmware-loading-and-networking-mode, and there does not exist a standard for this transition like what we had for the Device Firmware Upgrade class. However, before the DFU existed and in common practice, a user that wanted to upgrade firmware on an embedded system would either activate a menu choice to enter firmware loader mode, or enter this mode when the system is powered on, for example by holding down some special key available in the user interface.[39]

While casually adding TCP/IP to embedded systems has been regarded as bloating the system (taking up too much resources) in the past, recent research has shown that this is not necessarily the case. This question will be dealt with at length in Section 7.4 on page 35.

As was outlined for this thesis, a USB bus was to be used for building a network of two peers: a host computer and the embedded system. This network corresponds to the upper part of Figure 3, a network named 10.1.1.0 with two IP enabled hosts: 10.1.1.1 (the host) and 10.1.1.2 (the embedded system). This network is to be established over the USB bus.

To build a network over the USB bus may seem counter-intuitive for a reader who is familiar with network concepts such as common Ethernet, WLAN or token ring. USB is however a common means for peripheral interconnection, and as such has served as a channel for network traffic

[38]See `http://www.ietf.org/rfc.html`

[39]For the device used in prototyping the solution as described in Section 9 on page 46, an solution close to this was already available.

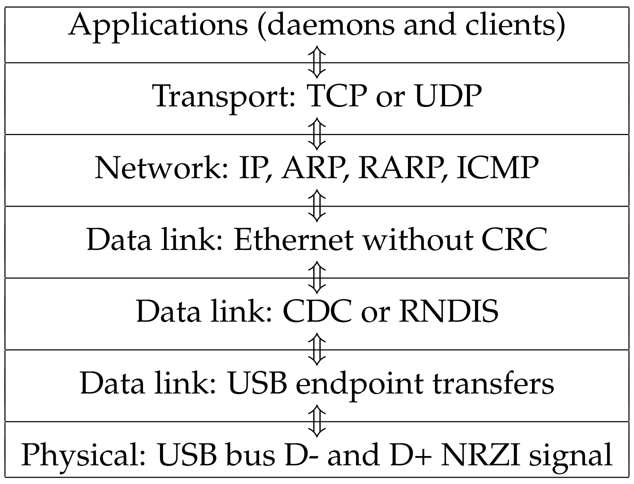

Applications (daemons and clients)
Transport: TCP or UDP
Network: IP, ARP, RARP, ICMP
Data link: Ethernet without CRC
Data link: CDC or RNDIS
Data link: USB endpoint transfers
Physical: USB bus D- and D+ NRZI signal

Figure 4: The network stack employed for networking systems over the USB bus.

not only to modems (serving as a simple serial line), but to complex Ethernet controllers and the like. Part of the research for this thesis included some examination of a USB-based Ethernet controller, in order to see how different operating systems would handle this device. The network stack employed in practice resembles Figure 4.

As can be seen, the concept of USB networking invariably involves emulating Ethernet. This emulation is not only used when the attached device is *per se* an Ethernet II controller,[40] it is used even for such things as USB cross-over cables i.e. a simple host-to-host network.[41] It is also used when connecting high-speed linked peripherals of almost any kind. At some level, almost anything that is networked over the USB bus is an Ethernet device, transmitting Ethernet frames. Ethernet frames, however, have a CRC16 checksum, and this is not used when networking across the

[40]The frame format used for the Ethernet transfers are Ethernet II, not X.802.3 which is almost invariably confusedly taken for being Ethernet II. Very few network adapters (if any) use X.802.3 in practice.

[41]This is necessary for any host-to-host connection over USB: as you know every USB cable has an *upstream* and a *downstream* connector, so it is in practice (and even more so in theory) of course impossible to connect two hosts, having two distinctive host controllers to each other, as there can be only one host controller for a USB bus. The solution is an embedded system which acts as a slave device on two USB buses, bridging the gap by making Ethernet frames transmitted on one bus appear as Ethernet frames received on the other bus. The system is in practice molded into plastic with two upstream USB connectors, powered by the USB bus and thus suspiciously similar to an illegal host-to-host connector[3].

USB bus, as the USB bus employs its own checksum and fault-recovery protocols. Presumably, the Ethernet frame was chosen as a unit of transfers because it is well known in industry.

The standard way of creating a USB network is to use the Ethernet Networking Control Model of the Communications Device Class (CDC). There exists, however, a rival protocol known as Microsoft RNDIS. The following two subsections will elaborate on these two schemes at some length.

7.1 Communications Device Class (CDC)

The standard[42] way of building network links over USB is to use the Communications Device Class, mostly known as CDC[22].

CDC supports protocols for connecting a large number of communication peripherals over USB, known as *device models*:

- A *direct line* (DL) model is used to connect devices to "POTS", an obscure acronym meaning *plain old telephone service*, listening in to the electrical signals transmitted on the wire
- A *datapump* is the same thing simplified, whereas
- An *abstract control model* supports using the device with AT-commands.[43] While the ACM model will accept commands, the DL model relies on the host to supply all signal processing and intelligence, something that is known as "host modem" or "winmodem"
- Further, two models for ISDN[44] lines are also specified.

Indeed, the CDC specification is possibly the most extensive and detailed device class, as USB was supposed to replace the RS232 links previously used by most computerized telephones, faxes and modems.

Network interfaces are however not regarded device models, but are more abstract in character and resemble low-level network protocols. The two networking models supported by CDC is Ethernet and ATM (Asynchronous Transfer Mode). The Ethernet model will enable two endpoints

[42]By *standard* we mean *de facto* as defined by the USB Implementers Forum, whereas there is no *de jure*-standardization for these protocols. The USB Implementers Forum also pool and control all patents related to USB technology.

[43]AT-commands are de facto modem control sequences once defined by the modem manufacturer Hayes.

[44]Integrated Services Digital Network

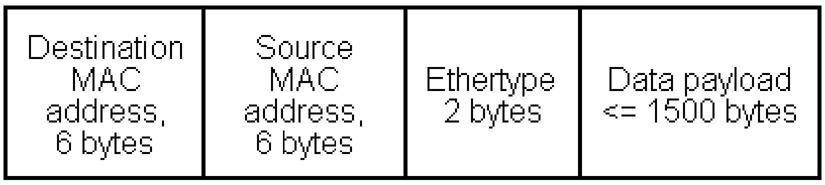

Destination MAC address, 6 bytes
Source MAC address, 6 bytes
Ethertype 2 bytes
Data payload <= 1500 bytes

Figure 5: The Ethernet II frame without its trailing 4 byte CRC checksum, as used in the USB Ethernet network model. A common mistake is to confuse this simple frame, used in practice, for 802.3 frames, only used in theory.

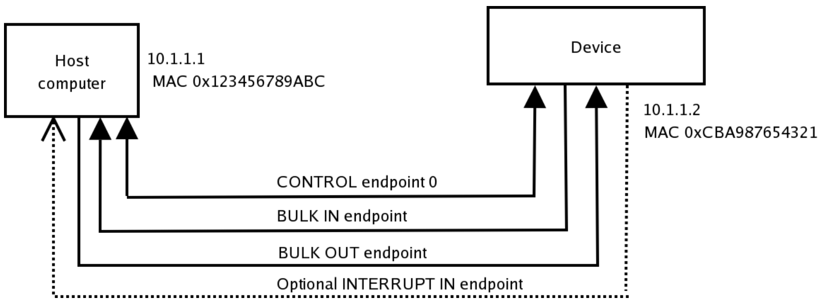


Figure 6: The two peers of our virtual Ethernet network, with their IP-addresses, Ethernet MAC addresses and four compulsory endpoints. (Endpoint 0 is in both directions and thus counts as two.)

to exchange Ethernet II frames, whereas the ATM model will similarly enable the exchange of ATM cells. We will only look into the Ethernet model here. The structure of Ethernet II frames is illustrated in Figure 5.

The network model interfaces use a split control/transfer mechanism: SETUP messages sent by the host controller to endpoint 0 are used to control the network devices, whereas the actual frames are transmitted on two BULK endpoints: one inbound and one outbound. Frames that are to be "sent" onto the network by the host controller are written to the BULK OUT endpoint (sans CRC checksum), and frames that appear on the BULK IN endpoint are regarded as frames decoded by some "network card" electronics and likewise appear without their CRCs.

The way the two network peers communicate across the USB bus is illustrated in Figure 6.

If the Ethernetworked device is an actual network card, it will during normal operations take the frames arriving at its BULK OUT endpoint, add a CRC and transmit it on the physical Ethernet cable, and vice versa. In our case, however, we want to connect a device as if it was a host on some "ethernet network" (which will only be a virtual concept), and thus we decode the Ethernet frames that arrive at the BULK OUT endpoint and conjure response frames by way of a TCP/IP stack. We will thus assume a MAC address for our virtual network card (a 6-byte number)[45] and report this as the MAC address for IP-address 10.1.1.2 (our device) when it is requested by an ARP broadcast. Notice that the host controller will also have to assume a random MAC address: the frames arriving at the BULK endpoints shall be complete, including the source MAC address field of the Ethernet frame.

Apart from these two BULK endpoints, the device will of course also support the compulsory endpoint 0 in both directions, totalling four endpoints. The control endpoint 0 is used for common USB handling and a few special CDC Ethernet commands, related to setting packet filters and retrieving statistics. If the device need to "talk back" to the host, an optional INTERRUPT IN endpoint can be utilized, but most CDC devices don't.

The typical device implementing CDC will present a device descriptor announcing that it belongs to the CDC class, containing one configuration with one interface. This interface, however, has to support two alternate interface settings: the default interface setting shall have none of the two BULK endpoints (or the optional INTERRUPT IN endpoint) and the secondary interface setting shall have all the endpoints (except for endpoint 0) declared. The alternate interface is used for activating the CDC Ethernet device: once the secondary alternate interface setting is selected, the network is on-line and frames may be transmitted in both directions on the BULK endpoints.

CDC has two distinct features that make it troublesome with some silicon: it requires that the device USB controller IP-block or discrete component support the *Set Alt Interface* command that is issued from the host controller to select the runtime alternate interface,[46] and it requires the ca-

[45]The CDC specification does not say anything about how MAC addresses are to be assigned. It is of course important that no two devices will collide on a network, and in our case the MAC address space for network consists of the host, the embedded system and any addressable subcomponents in our embedded system. The host will typically assume a MAC address at random, and the device and it's subcomponents will assume some different MAC addresses. It doesn't really matter if some MAC address in the embedded system collides with the address assumed by the host, because there are two duplex directed pipes involved, not any "true broadcast ethernet" simplex, so only one peer (behind a logically hidden direction of transfer) will respond to a message sent to a certain MAC address in practice. If the device was to be an *actual* ethernet device connected to an *actual* ethernet network, more caution is of course needed.

[46]For example the PXA2xx USB controllers from Intel and the USB controller built into

pability of the BULK pipes to transport a zero-length packet. This last feature is necessary, because the 1500+ bytes of an Ethernet frame may need to be transfered across the USB bus in several packages (a typical USB package is in the order of 64 or 512 bytes) and the driver will detect end-of-frame by detecting a packet that deviates from the full packet size. For example: a frame of 514 bytes on a system with a USB maximum packet size of 64 bytes[47] will be divided into 8×64 byte packages followed by one 2-byte package. The 2-byte package will be detected as less than the maximum frame size, and terminates the frame. If, however, the packet was to be exactly 512 bytes, 8×64 bytes is sent, then a zero-length package is sent to indicate that this is the end of the frame.

Some IP-core vendors and some discrete USB chips are, sadly, unable to handle one or the other of these two prerequisites. Both features are strictly speaking required for by the USB specification, but broken components have been produced anyway, probably because almost nobody was using these features at the time.

Implementing support for CDC is not totally straight-forward: the best way to gather the experience needed is to intercept USB traffic from some controller that has been known to work in practice.

7.2 RNDIS

The Remote Network Driver Interface Specification or RNDIS, is a scheme supplied by Microsoft Inc. for developing network links over USB[17]. It is an extension of NDIS, Microsofts Network Driver Interface Specification, used for creating Windows-centric network drivers.[48] The idea with RNDIS is quite impressive: it is to encapsulate *all* possible remote (i.e. not directly connected to the host computer bus) network devices in the foreseeable future. This includes not only network devices connected via USB, but also any network devices attached using FireWire (IEEE 1394), BlueTooth and InfiniBand. It is currently only defined for Ethernet II[49] and actually, it is only defined for USB. So RNDIS is currently (as of revision 1.1) a USB-Ethernet-only specification.

RNDIS has largely been the preferred solution among manufacturers of

certain SuperH chips from Hitachi and ST Microelectrics (often used with Windows CE) do not support this command.

[47]64 bytes is the maximum packet size for "full speed" USB devices, whereas a "high speed" devices will support up to 512 bytes in each packet.

[48]NDIS was created in cooperation with 3COM.

[49]Though Microsoft repeatedly confuse Ethernet II for "802.3" in their RNDIS documents.

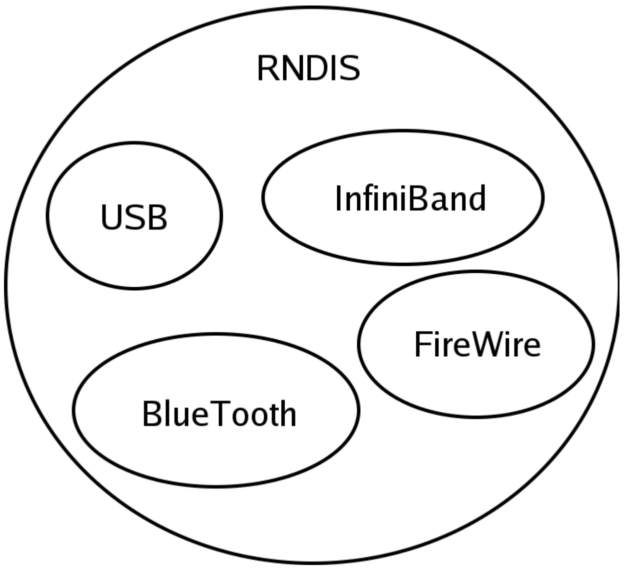


Figure 7: The RNDIS vision is simply illustrated in a Euler diagram like this: RNDIS is supposed to be the general framework for writing network drivers for network devices accessible on any future external computer bus.

USB network devices, since it is the only driver infrastructure supported by the widespread Windows family of operating systems. A number of cable modem (ADSL) manufacturers have however stayed with the CDC model and supply their own Windows drivers for these.

While the vision seen in Figure 7 may appeal to some, this idea has not generally caught on, at least not outside the world of Windows. It is currently unclear if Microsoft will some day support CDC devices as well, but as of now, such devices need to be supported using third party software whereas RNDIS comes bundled with e.g. Microsoft Windows XP. There is a driver development kit available from Microsoft that supports the host-side part of RNDIS, and a protocol specification that help out when developing the device-side drivers for RNDIS.

It should be noted that a Windows machine will not automatically detect and use an RNDIS device: an additional driver configuration file (*.inf*-file) is always needed. This is because the Windows kernel require these drivers to be associated with the precise *idVendor* and *idProduct* fields of the device descriptor for a networking device. The actual host-side driver is supported and developed by Microsoft however, though they have dropped support for legacy systems like Windows 98 and Windows ME which nevertheless have USB support and was available within the

RNDIS framework in the past.

An RNDIS device identifies itself as having a CDC interface, but one that uses a protocol that is vendor specific. It then lists itself as supporting an Abstract Call Management (ACM) function, which can receive and deliver encapsulated queries and responses, much like a modem accepting AT-commands.

The actual protocol used for RNDIS involves a rather complex Remote Procedure Call (RPC) scheme built from some 50 encapsulated messages that may be sent from the host controller to the device, and the result retrieved back to the host controller once the device signal that the message has been processed. The last signal, "message processed" is sent over an additional INTERRUPT IN endpoint, so that RNDIS will require one more endpoint than a simple CDC device. These messages are most likely a heritage from NDIS. The procedure for sending an RNDIS query and retrieveing the response is illustrated in Figure 8.

The encapsulated messages sent over endpoint 0 must subsequently be parsed and handled. One out of 6 basic message types can be sent and all of these must be supported, with the exception of *REMOTE NDIS INDICATE STATUS MSG*, which is generated by the device and may just as well never appear. All other messages go in the host $\rightarrow$ device direction and must be replied to in due time, or the network driver will conclude that device is dysfunctional and disconnect it.

Most of the message parser handles the numerous *REMOTE NDIS QUERY MSG* messages. Microsoft has listed a huge set of possible queries in their protocol specification, and in practice even more — undocumented — messages may appear. Most of these "mandatory" messages can however be answered with nonsense and zeroes: for example RNDIS mandates that a device shall be able to deliver transmission and receiveal statistics, and the device may just as well reply "0" to all of them. A particular feature of RNDIS is that the host controller can send a query named *OID GEN MEDIA CONNECT STATUS* that reports if the cable was plugged into the ethernet jack or not, in case this is an Ethernet controller. It may also request the network link speed and other things.

After sending an initial chunk of RNDIS commands, the driver will enter an idle state and thereafter only repeatedly query the device for link speed, media connection status and several packet transfer statistics. It will also repeatedly send "keepalive" messages to check that the device is actually there. In the meantime, packets may be transferred back and forth on the two BULK endpoints, just like on a CDC device. The packets sent on the BULK endpoints are not simple Ethernet frames however: they are prepended with RNDIS headers of 0x28 bytes which indicate the

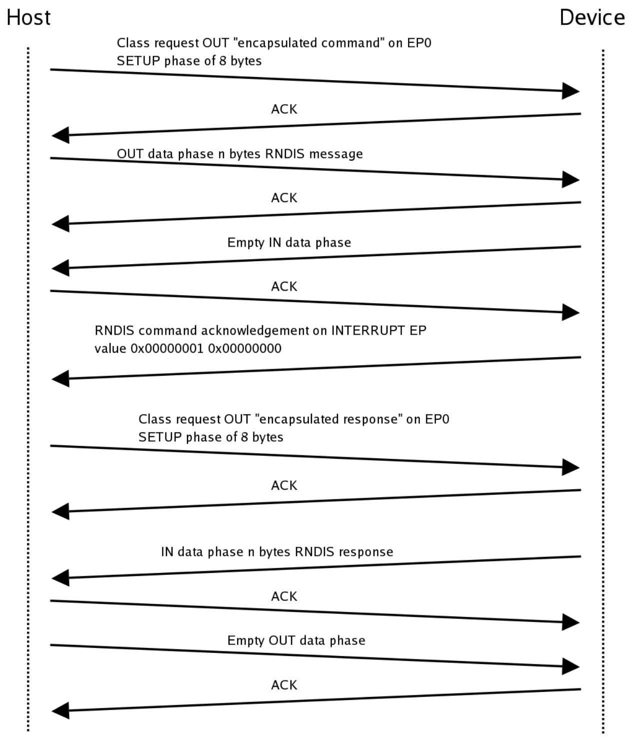

Host
Device
Class request OUT "encapsulated command" on EP0
SETUP phase of 8 bytes
ACK
OUT data phase n bytes RNDIS message
ACK
Empty IN data phase
ACK
RNDIS command acknowledgement on INTERRUPT EP
value 0x00000001 0x00000000
Class request OUT "encapsulated response" on EP0
SETUP phase of 8 bytes
ACK
IN data phase n bytes RNDIS response
ACK
Empty OUT data phase
ACK

Figure 8: The RNDIS RPC protocol as it appears on the USB bus. The RNDIS messages are sent back and forth in the data phase of the SETUP command on endpoint 0, with the INTERRUPT IN endpoint as a moderator: the device must first signal that the response is available on the INTERRUPT endpoint before next RNDIS command can be sent. It should be noted that in comparison with the USB CDC RNDIS is rather “talkative”.

number of bytes in this RNDIS frame followed by some obscure words of data containing pointers to extraneous out-of-band data[50] and per-packet information,[51] but which may just as well be set to 0.

In difference from CDC devices, RNDIS devices put part of the host controller responsibilities into the attached network controller. The most clear distinction is that there exists a command for setting the MAC address of the network controller, whereas in CDC this value is kept by the operating system driver. Here, the virtual network is obviously believed to begin within the remote network driver, and *not* within the operating system.

7.3 RNDIS and CDC combined

Belcarra Technologies Corporation have for some time supported a combination of RNDIS and CDC (and their custom CDC subset) for a module of the Linux operating system, known as the *Gadget API*. The Gadget API is a set of drivers for the Linux kernel which can be used when developing different embedded systems based on Linux[3][13].[52] Their source code has also been merged into the mainline Linux kernel. They have further developed a CDC driver for MacOS X and Mac OS 9, so that companies that use the Linux Gadget API can supply drivers for all major operating systems on the market, unless these already talk CDC or RNDIS natively.

The main trick involved in supporting both CDC and RNDIS at the same time is to present two device descriptors: one (the first) for Microsofts RNDIS, and the second for CDC. The reason for why the RNDIS descriptor must be presented first is that the Microsoft RNDIS driver will just examine the first device descriptor to see if it is an RNDIS device, whereas recent Linux kernels, for example, will examine any additional descriptors.[53]

The different nature of CDC and RNDIS make them put different requirements on the device USB controller.[54] When using both CDC and

[50]This is a legacy from NDIS, the Network Driver Infrastructure. It is used for media-specific information and priority tagging of packets.

[51]Microsoft says *per-packet-information* may be for example TCP checksums. (One might wonder what the network driver shall do with those, it evades me.)

[52]Apart from networking using RNDIS and CDC, the Gadget API support such things as USB file systems (USB Mass Storage).

[53]The Linux kernel will in fact make a qualified interface selection — Linux supports both CDC and RNDIS so it will first search through the descriptors to see if there is a CDC configuration to use, and in that case use the corresponding configuration. Else, it will fall back on the RNDIS configuration and use that instead.

[54]Belcarra even notes that some controllers do not even support the requirements of

RNDIS, the following is required from the hardware of the USB interface:

- Support for at least two different configurations. (CDC and RNDIS)
- Support for data following the SETUP command on both inbound and outbound CONTROL endpoint 0 transfers. (RNDIS)
- Two BULK endpoints, one for inbound and one for outbound Ethernet frames. (CDC and RNDIS)
- One inbound INTERRUPT IN endpoint. (RNDIS, otional for CDC)
- Support for the *set alt interface*-command on the CONTROL endpoint. (CDC)
- Support for zero-length transfers on the bulk endpoints. (CDC)

The following is required for the software part of the embedded system supporting both CDC and RNDIS:

- Low-level control of the USB interface, including the possibility to present several device configuration descriptors, interfaces and interface settings.
- A low-level Ethernet frame grabber that communicate directly with the endpoints.
- A low-level RNDIS parser that intercepts the messages sent to endpoint 0 and responds accordingly on the INTERRUPT IN + endpoint 0 RPC channel.

A print-out of the device descriptor used in practice, its configurations and interfaces as they were developed for the prototype can be found in appendix A. This dump has been included as a reference for implementers wanting to create the same kind of functionality: one of the major obstacles I found during implementation was the shortage of working device samples. A simple dump like this one was extremely hard to get at.

any of them and goes on to define a CDC subset that will work with only two bulk endpoints and no extras[3].

7.4 Single-threaded Networking Using TCP/IP on Embedded Systems

Network stack functionality such as TCP/IP, PPP, DHCP and even basic Ethernet is commonly known in the embedded systems field under the name *middleware*.[55] It is so called because it is mostly conceptually placed between the operating system and the higher-level applications for the system. Several companies and open source projects produce such functionality components for embedded systems, often in the form of software libraries.

TCP/IP for embedded systems with small memories was for a long time considered a "no can do". That is to say, before Peter Eliasson and Adam Dunkels created their TFE[56] cartridge for the Commodore C64 home computer. Adam Dunkels, working at SICS, the Swedish Institute of Computer Science, wrote a minimal TCP/IP-stack to go with this card, and such a stack would of course have to fit the memory footprint of a Commodore 64, i.e. significantly less than 64 kilobytes. The stack was named *uIP* ("micro-IP", μIP). It was fully compliant with the IETF RFC:s, would fit in a few kilobytes of memory and could survive with just a few hundred bytes of RAM[7]. He would also later write yet another small TCP/ IP-stack, named LW*IP* ("lightweight-IP") from scratch. This stack was a bit more general and extensible, and implemented a few optional features at the cost of a few extra kilobytes of memory[8].

One might wonder what kind of fallacies made several engineers believe that small-footprint and RFC compliant TCP/IP was not possible. The most obvious explanation may be that either they had seen the "small implementations" made in web cameras and the like, which were actually quite ugly hacks and as such not RFC compliant, and compared this to the code size of the BSD[57] TCP/IP stack, which happened to be the most widely available source code. Seeing that the fully functional, elegant BSD stack was several hundred kilobytes or even a few megabytes in size, whereas the small stacks were ugly and standards ignorant, may have caused a few uninformed people to believe that the difference in size

[55]The very word "middleware" is ambigous and should preferably be avoided. In the context of databases this word has a totally different meaning: it is taken to signify software components between a database and an application.

[56]TFE is an acronym for *The Final Ethernet*, a word-play with some popular Dutch cartridges produced by *RISKA H & P COMP.* with names like *The Final Cartridge* or *The Final Chesscard* produced primarily in the late 1980s.

[57]BSD is the Berkeley Software Distribution, a common POSIX-compliant UNIX-like operating system.

Stack	x86 platform	AVR platform
uIP	5188	5164
LWIP	14588	21756

Figure 9: Code size of the uIP and LWIP stacks in bytes. As can be seen, even the rather advanced LWIP stack does not even exceed 32 KB[7].

would equate the difference in quality, thus yielding the false assumption that you had to have a stack of a few hundred kilobytes to achieve decent TCP/IP functionality. I do not know if this explanation is true, but the assumption about TCP/IP stack size is very real, and kept several implementers of embedded systems from embedding TCP/IP into firmware.

Dunkels illustrates how a small-footprint TCP/IP stack is perfectly feasible in his published papers[7][8]. The hard facts showing the memory consumption for the stacks can be found in Figure 9.

When adding TCP/IP to embedded systems in general, and firmware loaders in particular, a number of important points arise:

- Embedded systems do not generally need the quite large and complex BSD socket API. Some embedded OS:es aiming for POSIX compliance may want to implement this API, but simple applications like firmware loaders do not need it.

- Execution in several embedded systems, and in almost all firmware loaders, is single-threaded. Such esoteric features as context-switching (and associated copying of entire TCP buffers) and semaphores for limited hardware resources is not needed.

The uIP and LWIP stacks make up for this. Execution is possible to keep within a single thread, accessing the hardware on its own.

The execution of the uIP and LWIP stacks is driven by *events*. Such an event is typically that an Ethernet frame arrives at the interface, or that a timer has triggered an interrupt. When an Ethernet frame arrives, it is checked to see if it is a valid IP packet and if this is the case, triggers a number of callback functions that will decode the packet and deliver it to applications that have registered as listeners. As TCP/IP need to drop packets or connections that time out, an additional timer interrupt is also needed. A second possibility is to have a hardware-driven mechanism for stacking up frames that arrive in a buffer, and have the buffer periodically polled by a timer interrupt. By using this simple interrupt-driven mechanism, the overhead incurred by using a full operating system and its task scheduler is removed, and the TCP/IP stack becomes both very small and very fast.

LWIP differs from uIP in that it has slightly more features: unlike uIP it also supports UDP, multiple network interfaces, sliding window (sending several TCP fragments at the same time, so as to account for transmission delays and loss in the network), congestion control, out-of-sequence data and buffering for retransmissions.[58] Since its initial implementation, a rudimentary BSD socket API has also been added to LWIP, but this part is optional and need not be compiled in. The LWIP project has also started a life of its own apart from its author, and lives as an open source project[14].

The licensing terms for both stacks is the so-called BSD license, which means that the stacks can be used in proprietary software, and that modifications to the stack need not be shared with the wider community.[59]

7.5 TCP/IP Configuration

When a device has working IP networking, the question arise of how this device shall behave in order to become a true network citizen configured with an IP-address, a network mask and a gateway.[60] Strictly speaking, in some contexts an IP-address is all that is needed, but whenever a device shall be internetworked — share information with other hosts on the Internet — it will need a netmask and a default gateway as well. There are often even more things to network configuration: name resolution in accordance with RFC 1034 and 1035 may also require the IP-address of a DNS (Domain Name System) server.

So how are the configuration parameters assigned to the embedded system network? A possibility is of course to "hardcode" the IP-address and other parameters as constants in the device firmware, or make them configurable through the embedded systems user interface if there is one. Mostly, however, such things are perceived as obscure, and it is easier for an end-user if configuration is accounted for with some automatic configuration protocol. This Section will deal with a few such protocols.

[58]Buffering for retransmissions means that packets are kept in a buffer in case they'd be lost on the network, so that they can be retransmitted if not acknowledged within a certain amount of time. The uIP stack solves retransmissions by requiring the application to be able to re-generate a piece of TCP data if need be.

[59]It is however a good idea to contribute and share modifications to the LWIP stack with the other developers: the project is advancing at high speed, and keeping a forked version of LWIP could prove very exhaustive, if changes made in the mainline stack are to be continually merged back to the forked codebase.

[60]The reader is assumed to be familiar with the concepts of network masks and gateways for IP networking, as mentioned earlier.

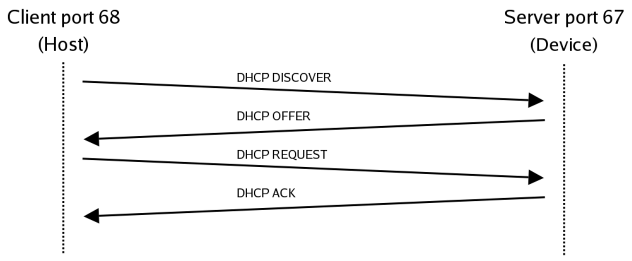


Figure 10: This Figure shows the basic outline of the DHCP protocol, as it is used when a client obtains an IP configuration from a server. More DHCP messages exist, but these are the most important.

7.5.1 DHCP

DHCP, the Dynamic Host Configuration Protocol, is defined in RFC 2131 as an extension of the earlier Bootstrap Protocol (BOOTP) from RFC 951.[61] DHCP is backwards compatible with BOOTP, so a client that only understand BOOTP may obtain an IP configuration from a DHCP server.

DHCP uses the UDP protocol and broadcast addresses. This is necessary, since we cannot address something that we don't know either hardware MAC address or IP-address for on the local network, so broadcasting is the only possibility. Further, two UDP ports are dedicated for this traffic: number 68 on the client side and number 67 on the server side.

The format of the UDP packets is basically that of BOOTP packets, appended with several *options*. The only part of the original BOOTP packet that is really used is the IP-address field and the hardware address field. The rest of the BOOTP structure is filled with zeroes (unless a BOOTP client was requesting the configuration of course). As BOOTP implementations are to ignore options it does not recognize, the entire DHCP superset mechanism has been implemented by way of such options, thus effectively achieveing backwards compatibility. A detailed list of available options can be found in RFC 1533. Except for the most rudimentary ones such as network mask and default gateway, there exist options for configuring DNS servers and even NTP servers for use with a certain client. A client must specify which options it is interested in, and must accept the fact that it is not mandatory that all configuration options that it wish for will be returned. For example the DHCP server may return a subnet mask, a gateway and a DNS, but no NTP server or NetBIOS over TCP/IP server.

[61] A newer version of DHCP which has been amended for use with IPv6 also exists as RFC 3315.

The basic DHCP operation can be seen in Figure 10: a newly attached client will locate a DHCP server using a *DHCP DISCOVER* message, and the server then offers a configuration using a *DHCP OFFER* message. This does not mean the configuration can be used: the DHCP server can offer the same configuration to several clients over time, if nothing is heard from the first machine. To confirm that the client really wants this address, it has to issue a *DHCP REQUEST* message,[62] and if the server still has this configuration available it will respond with a *DHCP ACK* message. After the ACK, the client may start to use it's new address. It is also possible that it issues a *DHCP NACK* message, denying the use of the requested address. The client will then have to iterate through the whole cycle again.

When the configured host is taken offline and subsequently reconnected to the network link, it will most often remember the last recently used DHCP server, and simply send a *DHCP REQUEST* to it immedately, requesting the re-use of the previously used address.

IP configurations are offered for a limited time span only, typically 7 days. After half this time, a client will usually send a new *DHCP REQUEST* in order to renew the configuration lease.

7.5.2 Zeroconf (Rendezvous)

The Zeroconf protocol, currently in an Internet Draft state (i.e. not published as an RFC) is already widely deployed in products developed by and for Apple Computers Mac OS X. Apple previously used the brand name *Apple Rendezvous* (french for "meeting") for this technology, which is inspired by their AppleTalk protocol. Zeroconf was developed for certain situations where AppleTalk would be able to auto-configure the network, whereas Internet protocols would not.

The typical use case for Zeroconf involves bridging two hosts with a cross-over ethernet cable. If the network interfaces of these two hosts are configured to use the TCP/IP protocol suite, typically nothing will happen, and neither host will be able to communicate with the other, because IP-address, network mask and gateway have not been configured. As none of the hosts can be assumed to be a DHCP or BOOTP server, there is no third party to ask for a configuration.

Zeroconf solves this problem. The algorithm used is basically the following:

- Provided that the network interface has not been assigned configuration parameters through DHCP or similar protocols;

[62]using unicast since it now knows the IP-address of the DHCP server

- Randomly select an IP-address between 169.254.1.0 and 169.254.254.255 in the 169.254.0.0/16 address space. The selection shall be done using an evenly-distributed random number generator. If a number has previously been used, it should be saved and tried first, before generating any new random IP-addresses.

- The address should then be claimed on the local network by the network interface before use, so that the address space is not corrupted. This is done by issuing a broadcast ARP[63] request for the desired IP-address. The device will fill in the *sender hardware address* of the ARP request, but set the *sender IP address* to all zeroes. This special ARP request packet is called an "ARP probe".

- The device issues a fixed number of such probe packets, randomly distributed over time.

- The device then waits a fixed time interval to see if there appears any ARP packets whatsoever, bearing the randomly generated IP-address, including both ARP request and response packets. It also checks to see if there are other ARP probe packets trying to probe the same address. If any such packet it found on the network, the address is regarded as occupied, and the process must be restarted by selecting a new random address and probing it.

- If no other peer on the network responds to the ARP probe, and if no other device is detected probing the network for the same address within a certain time interval, the device assumes a successful claim and takes the identity of the new IP-address.

- The device will then announce its new IP-address by transmitting a number of ARP probe packages with both the sender and target addresses set to the newly obtained address. This flushes any ARP caches present in other peers on the network.

- The IP-address is then continually defended: if ARP packets appear bearing the same IP-address as the device, but a different hardware address, the device shall either give in and try to assume a new IP-address or broadcast blocking ARP packets, responding to the ARP request and thus claiming the address.

In order to keep the defences up on the network, all ARP requests and responses must use broadcast: no peer may ignore any ARP packets.

[63]ARP is the Address Resolution Protocol, see RFC 826.

In order to use the network, you however need an IP-address to access your device. Unless these are statically assigned, or predictable (as with DHCP) you need some kind of naming service to be able to resolve a symbolic name on the local network into an IP-address. For this reason, the Zeroconf working group has drafted a multicast-based DNS mechanism which can be employed to use symbolic names on the local network. Another possibility is of course to use some custom broadcasting protocol.

7.5.3 What Address Should Be Used?

A practical consideration that will surface is of course what IP-addresses to choose for an embedded system connected to a host. The host and the device will share a small, private network, and as far as the host is not connected to the Internet any address can be chosen. If the host *is* connected to the Internet, there will exist at least one other network interface on the host computer and the system will have to have a routing table to decide for each packet which network interface shall dispatch it. If the network IP-addresses are the same for the two interfaces, confusion arises, and the packet will most likely be sent to the interface which was configured first and which therefore appears first in the routing table. This situation must be avoided at all costs, as it will in practice cause either network to "disappear" from the host.

RFC 3330 lists a number of special-use IP-addresses which can be consulted for deciding on which address to use in situations like these. Three of the address series listed therein are the networks 10.0.0.0/8,[64] 172.16.0.0/12 and 192.168.0.0/16 which are set aside for private Internets according to RFC 1918. These cannot be used, because we don't know if our embedded system is connected to a private Internet. It is a common practice for e.g. broadband routers and firewalls to define an internal network in one of these ranges and assign addresses on this network to hosts connected to the routes.

We *could* subclass the 127.0.0.0/8 network defined in RFC 1700 as *internal host loopback address*. It is common practice to use 127.0.0.0.1/31 for the local host, but the entire range 127.0.0.0/8 is actually reserved and mostly unused. However it is a matter of interpretation as to what *internal host* may mean. Does it mean that we could consider a device attached to the host using a USB cable as *internal*? The least dangerous interpretation is that we should *not* do this, because we do not know how operating system

[64]This notation reads: <network address>/<number of one-set bits from left of the subnet mask>, 10.0.0.0 with subnet mask 255.0.0.0 has the same meaning.

implementers may have interpreted it.

Instead, we decide to use the network 169.254.0.0/16. This network is defined in RFC 3330 as *link local*, and is said to be intended for *communication between hosts on a single link*. This range is specified to be used whenever we need to communicate between hosts by auto-configuration.

What is meant by "auto-configuration" is not clearly stated in this RFC, but taking into account that the IETF Working Group for Zeroconf (see previous Section) assigns addresses within this range gives at hand that this range has thus been cleverly reserved for Zeroconf before the actual RFC:s had been published. The drafts from the Zeroconf working group further define that a set of IP-enabled hosts are considered *link-local* if:[65]

- When any host A from that set sends a packet to any other host B in that set, using unicast, multicast or broadcast, the entire link-layer packet payload arrives unmodified, and
- A broadcast sent over that link by any host from that set of hosts can be received by every other host in that set.

Both requirements are fulfilled by the kind of network we have illustrated in Figure 3 and also match the solution implemented in the prototype.

If our device supports both Zeroconf and DHCP, we can run the following algorithm once the network interface is activated:

- The primary processor of the embedded system (there may be only one, typically the one dealing with the host connection) is dedicated as a DHCP server. It will first wait for a while and see if a DHCP discover/request message appears from the host. If that happens, it serves a static address in the 169.254.0.0/16 range and assign other addresses in that range to the subcomponents of the embedded system.
- The subcomponents of the embedded system may ask for their configuration using DHCP as well, as far as they wait for the host to be assigned first. They may detect that DHCP is in use by listening to see if messages are sent from the DHCP server. When they detect DHCP traffic, they can start requesting configurations for themselves.

[65]As Internet Drafts are not to be cited (see RFC 2026 Section 2.2), no source can be given for this definition until it's publicized.

- If no DHCP request from the host appears within a certain amount of time, the primary processor should go to Zeroconf mode and try to allocate an IP-address for the device in 169.254.0.0/16 using the Zeroconf algorithm.
- Addresses should be allocated for all subcomponents of the system independently, so when the subcomponents time out waiting for DHCP traffic or otherwise detect Zeroconf traffic on the network, they start employing the Zeroconf algorithm independently.

Since we have full control of the network end-to-end, we may use the same address range that has been reserved for Zeroconf when DHCP is used for address assignment: no unknown peers can possibly appear on this link and start behaving differently. If desired, there is also a document draft from the Zeroconf working group addressing *IPv4 address conflict detection* which may enable simultaneous use of DHCP and Zeroconf on the same network and same address range, making it possible for the two methods to interoperate without any special considerations.

On a side note, a third, less viable alternative for automatic configuration is said to be the NetBIOS protocol, originally developed by IBM and Sytec, but nowadays mostly known for its use in Microsoft Windows environments. It has been blessed as an Internet Standard in RFC 1001 and RFC 1002. However, this protocol (which was initially a link-level protocol as well) has merely been encapsulated in Internet Protocols for supporting the large amount of software that was tailored for NetBIOS in the 1980s, and has not been developed further since. I have not made any deeper investigations into the nature of NetBIOS configurations and name services.

8 Security Concerns

Andrew Huang who has written a practically-oriented and hands-on manual of embedded systems security named *Hacking the Xbox*[11] characterizes the core issue of embedded computing as *keeping control of the instruction pointer*. This is a simplification of course: embedded systems may exhibit a number of security hazards, from being dangerous by documenting too much, like a web camera placed in a bad spot, or by just building up a momentum of mental stress for the user.

Putting more high-level security concerns aside, and going for the main issue concerning firmware loading, we are back at the program counter. What is generally understood by the term *trusted computing* is that the program counter of the CPU will only point to trusted code, i.e. code that a manufacturer or user of a system has approved to be run on the system. This is to be distinguished from *malicious code*, which is code that the user does not want to execute on his or her system, such as a computer virus. The core security components of a system is often called its *trusted computing base* (TCB).

Whereas it is theoretically possible to have the CPU verify a signature for each machine-level instruction before running it, this is very hard to do in practice. The most common practice in embedded systems found in the wild is to verify an entire block of software before running it. A typical method is to checksum a block of code, such as an entire firmware, with something like an SHA-1 one-way hash,[66] and then encrypt this checksum with a symmetric or asymmetric cipher key and append it to the code block.[67] We call this a *signature*. By decrypting this block signature, recalculating the checksum and comparing the two, a certain software block can be verified against a certain key before executing any code. If the verification fails, the code is not executed.

Some systems, like the Microsoft Xbox, use a symmetric cipher for encrypting the signature checksum. This has the downside that the secret cipher key has to be stored somewhere inside the embedded system, and an experienced reverse-engineer is likely to be able to pry it out. Other systems like the Creative Nomad Jukebox from Creative Labs use an assymetric cipher, meaning that the embedded system will only contain a public key used to verify the signature, whereas the software has been signed with a private key. The latter solution should be preferred, as it is considerably more robust and can also be extended to use public key certificates if need be.

While the idea of signatures is in theory a perfect platform for secure execution of software, all interfaces to the system provides exploitable security holes. These holes always exist, but aren't necessarily easy to find and exploit. We will examine in some detail the holes that may exist in the firmware loader.

The whole issue really boils down to the old problem of keeping track of the program counter: what may distract the program counter away from authorized code?

[66]SHA-1 (an acronym for Secure Hash Algorithm) maps a binary sequence onto a 160-bit digest that should be unique enough to disable so called preimage attacks, i.e. constructing ad hoc attachment sequences which can make an arbitrary sequence match the digest.

[67]The reader is assumed to be familiar with concepts such as encryption, symmetric/asymmetric keys, public key certificates and similar things. If these things seem alien, please consult a textbook such as *Computer Security* by Gollmann[10].

If we look into the firmware loader using DFU as described in Section 6, or the networked alternative that was outlined in Section 7 we notice two things:

First: there is nothing that prevents the firmware file transfered over DFU or in the application layer of the network from containing an additional digital signature. Thus both solutions are compatible with trusted computing schemes.

Second: the lower levels of both concepts may be subject to quite simple buffer-overrun attacks. This goes for the USB low-level stack used by the system using either the DFU or a network, the DFU protocol implementation in the system (if that is used), or the rest of the network stack between the application and USB signals in Figure 4 on page 25.

Exploiting a buffer-overrun means that you locate a spot in the generated machine code of a program (device driver, protocol stack etc.) where an array variable storing incoming data can be indexed out of its memory domain. Doing so, you can potentially send in a custom data stream that will overwrite the data memory and reach the activation record[68] currently in use by the routine filling the buffer. There, it may overwrite the return address for the routine, effectively distracting the program counter to execute some arbitrary code that has been injected along with this malicious data. In this way all of the security measures introduced by signing the software is obliterated and the program counter is hijacked[6]. (Apart from corrupting the return address, other buffer-overrun attacks are possible.)

Of course you will need some additional knowledge of the hardware in order to truly exploit and take control of a system in this way, a task that typically will exceed the security breach process in time.

The only way to plug such security holes (as they are called) is to audit the code of the stack, over and over again. Not only will this remove a lot of exploitable code, but generally increase code quality and readability as well. Such attacks are easier to perform when programs use non-typesafe languages like C, so switching programming language is an option, however not always so easy to perform, since many embedded systems only have compilers for the C programming language. A few programmatic countermeasures are available,[69] however these often incur performance and/or memory utilization hits. Countermeasures that depend on features in the operating system are of course out of the question,

[68]Activation records are a concept of compiler technology. The reader is encouraged to consult a textbook on compilers if this concept is not understood.

[69]See Cowan et al[6].

as a firmware loader usually doesn't make use of the operating system.

The USB part of the stack includes the low-level drivers for the silicon used for handling transactions to/from endpoints on the device. By injecting special packets using specialized hardware or software a skilled intruder may for example send packets that exceed the maximum packet size for an endpoint, or add malicious code to the data phase following the SETUP command on the USB bus, as this will be interpreted by the USB driver. The BULK and INTERRUPT endpoints are not equally vulnerable: the data that arrive on these endpoints is usually passed on uninterpreted and unmodified to the next layer. In our case, data is only transported out of the device on the INTERRUPT endpoint when used for RNDIS, and it is thus not likely to be subject to an attack.

The LWIP stack presented in Section 7.4 presents a special dilemma: the code used in LWIP is open source, i.e. easily available off the Internet. One school of thinking (the minority) says this is a bad thing, stating that keeping source code secret hinders security exploits. The other school of thinking (the majority) calls this "security by obscurity" and notes that as buffer overruns are exploited by reading machine language and using disassemblers to find weaknesses in implementations, access to source code is not important to exploit buffer overruns: instead the characteristics of e.g. the compiler used and the host CPU are more important[18].[70]

Using LWIP in a project should typically involve feeding any patches for security vulnerabilities back to the project, so as to create a very good and secure TCP/IP stack. Not only do others get to use your fixes: you will also raise interest in these issues and get fixes and feedback from other developers with different interests.

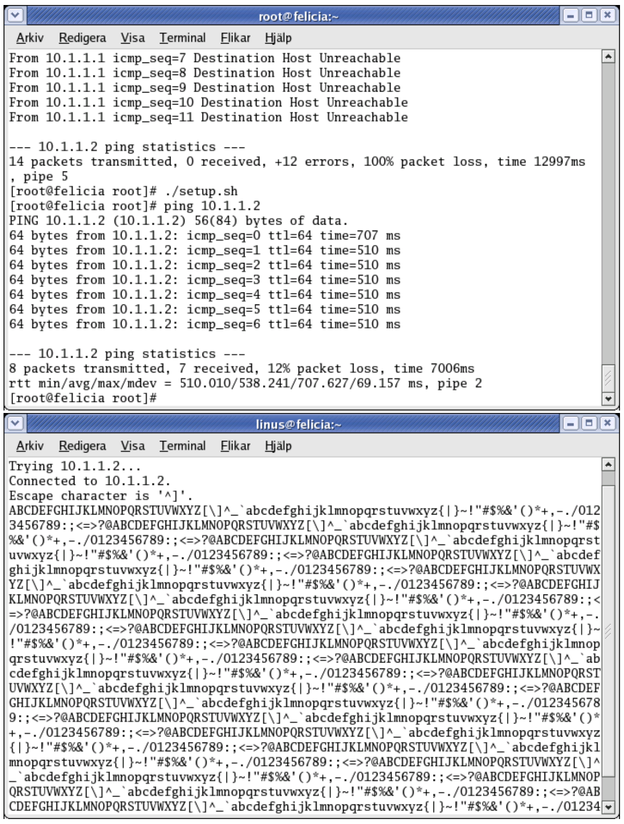
When some data has passed through the link and network layers, the applications using it must of course be equally security-aware and watch out for exploitable buffer overruns.

9 Implementation

This Section will present what I implemented in my prototype and how I went about doing it.

When I started my work on the embedded system, I first examined the hardware and software development environments used for this par-

[70]While I cite Stajano & Isozaki here, the notion about intruders using disassemblers is my own. In my opinion, Stajano & Isozaki show a lack of understanding as to how security vulnerabilities in code are actually exploited, by giving (limited) support to the notion that availability of source code simplifies attacks.



```
From 10.1.1.1 icmp_seq=7 Destination Host Unreachable
From 10.1.1.1 icmp_seq=8 Destination Host Unreachable
From 10.1.1.1 icmp_seq=9 Destination Host Unreachable
From 10.1.1.1 icmp_seq=10 Destination Host Unreachable
From 10.1.1.1 icmp_seq=11 Destination Host Unreachable

--- 10.1.1.2 ping statistics ---
14 packets transmitted, 0 received, +12 errors, 100% packet loss, time 12997ms
, pipe 5
[root@felicia root]# ./setup.sh
[root@felicia root]# ping 10.1.1.2
PING 10.1.1.2 (10.1.1.2) 56(84) bytes of data.
64 bytes from 10.1.1.2: icmp_seq=0 ttl=64 time=707 ms
64 bytes from 10.1.1.2: icmp_seq=1 ttl=64 time=510 ms
64 bytes from 10.1.1.2: icmp_seq=2 ttl=64 time=510 ms
64 bytes from 10.1.1.2: icmp_seq=3 ttl=64 time=510 ms
64 bytes from 10.1.1.2: icmp_seq=4 ttl=64 time=510 ms
64 bytes from 10.1.1.2: icmp_seq=5 ttl=64 time=510 ms
64 bytes from 10.1.1.2: icmp_seq=6 ttl=64 time=510 ms

--- 10.1.1.2 ping statistics ---
8 packets transmitted, 7 received, 12% packet loss, time 7006ms
rtt min/avg/max/mdev = 510.010/538.241/707.627/69.157 ms, pipe 2
[root@felicia root]#
```

```
Trying 10.1.1.2...
Connected to 10.1.1.2.
Escape character is '^]'.
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./012
3456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$
%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrst
uvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdef
ghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWX
YZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJ
KLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<
=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-.
/0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
qrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`ab
cdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRST
UVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEF
GHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./012345678
9:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*
+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijkl
mnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^
_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOP
QRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@AB
CDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./01234
```

Figure 11: These screenshots show *ping* and *chargen* running on Linux with the embedded system attached as a CDC device, with complete networking over USB. Ping was run successfully for the first time on 2004-07-09 at 12:57 and chargen was run on 2004-07-14.

ticular system. All larger companies have their own, slightly idiomatic software configuration management systems and change management / bug tracker / task dispatch systems and Ericsson Mobile Platforms is no exception. However, being an employee since many years, this was not much of a problem and I quickly became familiar with the development tools.

All code was implemented in the C programming language for the ARM 9E processor core. The programs were compiled using custom tools and transfered to the target using proprietary firmware loading mechanisms. Once started from within this framework, my own firmware loaders could start a life of their own.

9.1 USB What, Where and Why

The USB hardware was a well-documented IP-block that had been incorporated into one of the ASICs of the system, and was available through memory-mapped I/O-registers. The hardware could generate interrupts, and was supposed to also facilitate double-buffering and DMA[71] transfers of buffer contents; however this feature was seriously limited (slower than manual copying) and could not be used. The IP-block also had a double-buffering mechanism, so that incoming data (USB OUT transfers) could be retrieved from one buffer while the second was still being received. This feature was totally broken and could not be used at all.[72]

The USB IP-block could support fifteen inbound and outbound endpoints, apart from endpoint 0 (which had separate hardware registers and could be used without concern about what was going on on the other endpoints).

A software driver written in the C programming language was available for the USB IP-block. (This driver will be referred to as the *function driver* to distinguish it from the host side driver, which is implemented on the host PC side.) However this function driver would only support endpoint 0 control transfers and two BULK endpoints: one inbound and one outbound. This is typical for most function drivers that only want to use USB as a convenient data transfer channel, much the same way as RS232 interfaces were used in the past.

The hardware would only allow for *one* inbound and *one* outbound transfer to commence at the same time, as there was only one buffer for each type. (Apart from the transfer buffers for endpoint 0, which were independent of the other endpoints.) As I was only using one outbound endpoint apart from endpoint 0 in CDC+RNDIS, outbound transfers were no problem. However when using RNDIS, both one inbound BULK and one inbound INTERRUPT endpoint needed to be accessible at the same time. When using several inbound endpoints at the same time like this, a queueing mechanism should be devised, but in this case, as I was only using a single resource for two tasks, a simple buffer and semaphore-system was easier to implement. If either the BULK or INTERRUPT transaction detected that a transfer of the other type was commencing, it would simply buffer its packet and wait for the resource (i.e. the inbound endpoint) to become available.

[71]DMA is *Direct Memory Access*, and describes the feature of a computer system where a hardware unit may independently access the main memory without routing information through the CPU.

[72]A later revision of the IP-block fixes both these problems.

I also discovered that the function driver was not tested with regard to incoming data after the SETUP phase on endpoint 0, so code for handling this properly had to be written. Outgoing data on endpoint 0 is used for transfering all device descriptors and such things, so this was already implemented and usable.

9.2 USB CDC+RNDIS Networking

The network low-level function drivers were implemented as a two-stage rocket: first a CDC device was created, as this was the easiest part. Device, configuration and interface descriptors for a CDC Ethernet controller was presented to a host computer running Linux. Once the Linux CDC host driver started talking to this controller, interception of the USB traffic revealed missed details and helped out in speeding up the implementation procedure. USB traffic was captured and analyzed using a *USB Tracker*, a combined hardware/software USB sniffing tool developed by Ellisys.[73]

For generating some Ethernet frames on the USB bus without any working TCP/IP stack, the program in appendix B was used. This program use the low-level Linux network interface to generate an arbitrary packet on the ethernet interface associated with the CDC device. The function driver on the device side could respond by just copying the ethernet frame and sending it back the same way. This was how the rudimentary CDC implementation was bootstrapped (created from no working code base) and tested.[74]

Once the CDC Ethernet function driver was fully working, the LWIP TCP/IP stack was introduced into the source tree, adapted and debugged. LWIP was chosen over uIP for its support of UDP[75] and general extensibility, so as not to restrict future developments of more complex network functionality, for example LWIP has basic IPv6 support[76] under way, and this might be desirable in the future.

To debug the TCP/IP stack, some more intelligent packet analyzer than the Ellisys USB Tracker was needed, so for this I installed the sniffer program *Ethereal*,[77] which is perfect for analyzing high-level internet proto-

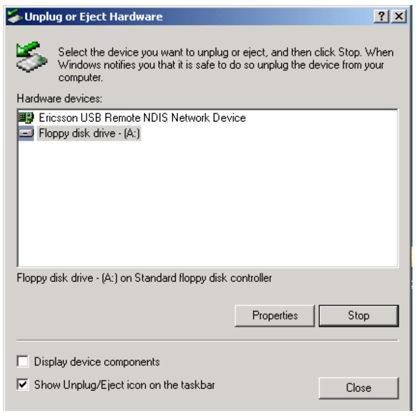
[73]See `http://www.ellisys.com/` tools like this can be quite expensive, the Ellisys tool cost around 10.000 SKR and is considered cheap.

[74]I have since been notified of such network tools as *hping* (`http://www.hping.org/`) whose only use is to create custom frames for different protocols, so the reader may want to examine a few other possible tools.

[75]UDP is the *User Datagram Protocol*, see RFC 768.

[76]IPv6 is the successor to the currently dominating, but limited IP version 4 protocol.

[77]See `http://www.ethereal.com/`


Unplug or Eject Hardware

Select the device you want to unplug or eject, and then click Stop. When Windows notifies you that it is safe to do so unplug the device from your computer.

Hardware devices:

Ericsson USB Remote NDIS Network Device
Floppy disk drive - (A:)

Floppy disk drive - (A:) on Standard floppy disk controller

Properties | Stop

☐ Display device components
☑ Show Unplug/Eject icon on the taskbar

Close

Figure 12: Once the RNDIS RPC protocol is implemented and responds as the Windows driver expect, the embedded system appears as a “network card” in Windows. The “Network and Dial-up Connections” control panel will also display an icon with the name “Local Area Connection 2” (if you previously had one) which gives you the opportunity to configure the network card with IP-address, netmask and gateway. (The first successful test was done 2004-09-13)

cols over Ethernet, and which is available for both Linux and Microsoft Windows.

A network interface module for LWIP utilizing the CDC USB Ethernet interface was written from scratch with structure based off the templates found in the LWIP ports tree (which is actually a bunch of examples of how to get LWIP up an running on different architectures, network interfaces and compilers). The USB CDC driver was very close to that of a common Ethernet driver. This network network interface was then integrated with the LWIP stack so that the entire stack as illustrated in Figure 4 on page 25 was up and running, see Figure 11.

The stack on the device was hardcoded to use IP-address 10.1.1.2, netmask 255.255.255.0 and gateway 10.1.1.1, whereas the host stack needed manual configuration by running this Linux command as root user:

```
# ifconfig eth1 10.1.1.1 netmask 255.255.255.0 up
```

These operations involved a lot of fixes here and there but was essentially a straight-forward task, much thanks to the very well written, well structured and easily portable code in LWIP. The support for UDP packages, IPv6 and BSD sockets was excluded (instead of sockets the so-called "raw API" using raw packets and callbacks as discussed in Section 7.4 on page 35 was used), but could be reintroduced if need be.

A few simple sample applications for TCP/IP in the framework of an Internet Daemon (*inetd*) were written for testing, for example *chargen*, a daemon that will respond by a repetitive stream of characters when a user TELNET:s[78] to port 19 of the system.

When the complete network stack was functional if attached to a Linux host, so far that *ping* (i.e. ICMP) and *chargen* (i.e. TCP) were fully functional, Microsoft RNDIS support was introduced. This involved writing a large RNDIS message parser and doing excessive bus monitoring. Here, a nasty problem with stack overflow in the IRQ stack appeared: if too many nested function calls were made (as in the RNDIS message parser) the stack would overflow and start corrupting main memory.[79] This error was finally resolved using a hardware debugging tool named *Trace 32* from Lauterbach Datentechnik GmbH. After the stack depth was increased by five times, no more sporadic errors would appear and the RNDIS function driver exhibited a stable behaviour.

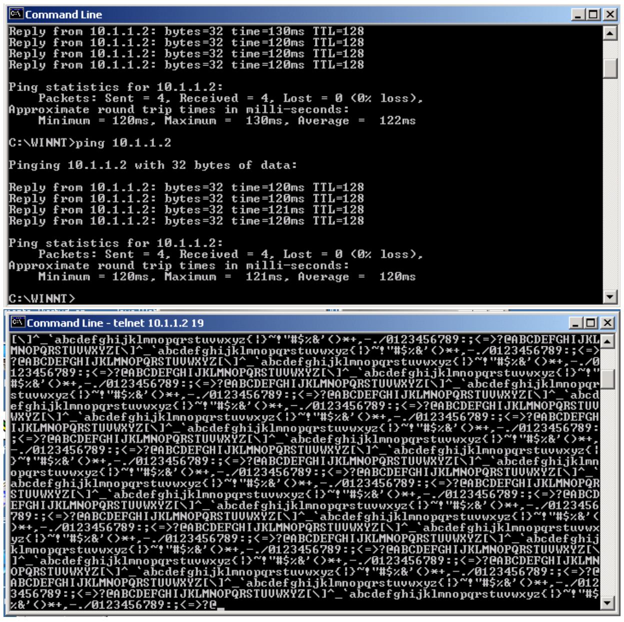
As the host operating systems were both using standard drivers to communicate with the system, no real programming had to be done on the host side. CDC works out-of-the-box with Linux, but RNDIS require you to supply a driver disk with all devices. In practice this is just an *.inf*-file which registers your *idVendor* and *idProduct* with Windows, plus two kernel driver files, both of which are supplied by Microsoft. On Windows XP (and presumably newer Windows systems as well) only the *.inf*-file was needed, as the drivers are included with the operating system.

I do not know why Windows does not autodetect and prime RNDIS drivers (as it does with, for example USB mass storage devices), but it presumably has to do with the fact that as RNDIS is just partly a standard device class, it is considered custom and as such should load custom drivers.

Microsoft have updated their RNDIS driver kits, and only support RNDIS under Windows 2000 and Windows XP nowadays, but using a combination of the present and an older kit, you can create RNDIS driver disks that also support Windows 95 SE, Windows 98 and Windows ME. (This was also done.) Apart from Windows 2000 it cannot be said to have been tested though, as my managed workplace would not allow me to

[78]TELNET is a standard protocol, but also a terminal-like program that opens a character stream console to a remote host on a TCP/IP network. Another program that has the same effect is *GNU netcat*.

[79]Notice that stack overflow is also a security hazard, albeit probably quite hard for an attacker to exploit.



```
Command Line
Reply from 10.1.1.2: bytes=32 time=130ms TTL=128
Reply from 10.1.1.2: bytes=32 time=120ms TTL=128
Reply from 10.1.1.2: bytes=32 time=120ms TTL=128
Reply from 10.1.1.2: bytes=32 time=120ms TTL=128

Ping statistics for 10.1.1.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 120ms, Maximum =  130ms, Average =  122ms

C:\WINNT>ping 10.1.1.2

Pinging 10.1.1.2 with 32 bytes of data:

Reply from 10.1.1.2: bytes=32 time=120ms TTL=128
Reply from 10.1.1.2: bytes=32 time=120ms TTL=128
Reply from 10.1.1.2: bytes=32 time=121ms TTL=128
Reply from 10.1.1.2: bytes=32 time=120ms TTL=128

Ping statistics for 10.1.1.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 120ms, Maximum =  121ms, Average =  120ms

C:\WINNT>
```

```
Command Line - telnet 10.1.1.2 19
[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKL
MNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>
?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0
123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqr
stuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
efghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUV
WXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGH
IJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:
;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,
-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|
}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmn
opqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQR
STUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCD
EFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456
789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'(
)*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwx
yz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghij
klmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN
OPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$%&'()*+,-./012
3456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~!"#$
%&'()*+,-./0123456789:;<=>?@_
```

Figure 13: These screenshots show *ping* and *chargen* (first successful test 2004-09-15) running on Microsoft Windows 2000 using RNDIS and the LWIP network stack.

install and test any other operating systems than Windows 2000.

When a proper RNDIS driver was in place and the device was responding correctly to all RNDIS messages, the device would appear as a network card, as can be seen in Figure 12. The *ping* command and the sample *chargen* daemon would then respond to requests in the same way as for CDC as can be seen in Figure 13, and the entire CDC+RNDIS network stack was thus implemented.

Close to the project end, a version 1.0.0 of LWIP was released. This was quite quickly integrated back into the codebase which had thereto been running the old 0.7.2 version.

9.3 Automatic Configuration

To support automatic configuration of the host computer I developed a lightweight DHCP server running on the embedded system. To understand the packet format and examine some traffic in real life I simply ran the Ehtereal packet sniffer in promiscuous mode on my home network, filtering out and saving DHCP traffic only while making a few DHCP requests against a server. I also obtained a log file from a large network, where DHCP had been running for some time, so I could observe how traffic would look in a large-scale deployment.

As DHCP uses UDP, the UDP support had to be turned on in the LWIP stack. This was no big operation, and the extra UDP code would only account for an additional 1.3 KB of the total LWIP code size.

When debugging the DHCP server, Ethereal was once again the perfect tool, when used together with the Linux CDC interface. Any broken DHCP packets were easily identified and the code could be finalized in just two days.

The very simplified DHCP server was not BOOTP compatible, and would assign the IP-address 169.254.1.1 and the network mask 255.255.255.0 to the first host connecting to the system. As there could be only one host system on a CDC+RNDIS connection, ever assigning the requesting host IP-address 169.254.1.1 was no problem. The device itself would assume the IP-address 169.254.1.2.

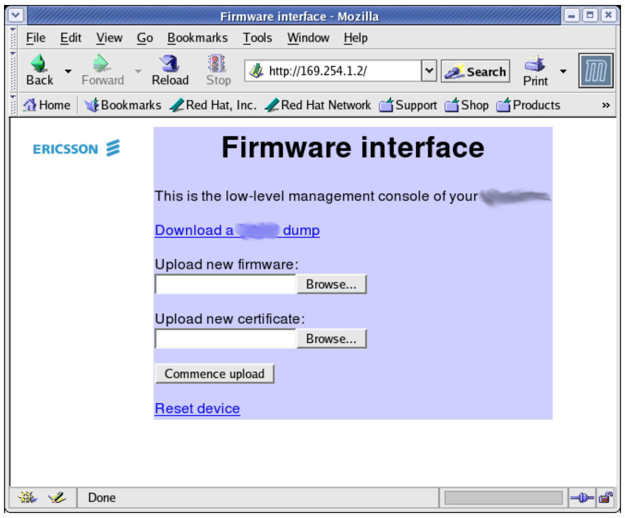
Network operations could then be performed from the host against IP-address 169.254.1.2 whenever the device was plugged in. No DNS entries were configured, as I did not implement a lightweight DNS server on the device, but this could optionally be done as a part of a larger project, so that e.g. *ping foo* would contact the device without any need for IP-addresses. (This may however raise conflicts with other DNS services that need to be resolved.)

9.4 Applications

Apart from the test daemons, a firmware loading daemon was implemented. This firmware loading daemon uses TCP and a custom protocol and a custom client for talking to the daemon from the host computer performing a firmware upgrade, configuration or diagnostic.

As the system used has a quite large and diversified proprietary firmware loader, all functionality could not be easily ported without forking the development source code trees, so a one-shot test (scrap) daemon was programmed to be thrown away as a proof of concept. This daemon could dump out regions of memory for diagnostics and reformat and program flash memory on the embedded system.

With this service daemon finished, the entire networked firmware loader concept was finally demonstrated and shown to work. Further studies that could have been performed are speed benchmarking for transfers, security auditing and the like, but because of time limitations, such measures will have to be put into future projects based on this first prototype.


Firmware interface - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://169.254.1.2/
Search
Print
Home Bookmarks Red Hat, Inc. Red Hat Network Support Shop Products
ERICSSON
Browse...
Browse...
Commence upload
Done

Firmware interface

This is the low-level management console of your [redacted]

Download a [redacted] dump

Upload new firmware:

Upload new certificate:

Reset device

Figure 14: A web interface for firmware modification

As a last step, a Firmware daemon in the form of a HTTP server was implemented. This had the immediate advantage of providing a simple Web-based user interface for firmware upgrade, maintenance, troubleshooting and modification, using nothing but standard protocols and description languages. A screen capture of the prototype web interface can be found in Figure 14.

9.5 Obtained Code Sizes

The code sizes obtained can be found in Figure 15. These code sizes are for an ARM processor in *thumb* mode. The difference between *thumb* and *normal* mode is basically that the instruction words are 16-bit instead of 32-bit in *thumb* mode, while some more complex instructions are unavailable. Using the *thumb* mode nominally results in less memory consumption for code, but there may be situations where several *thumb* instructions are needed to produce the same code that can be obtained by a single *normal* instruction. The compiler used is the IAR C/EC++ Compiler with the *-Z9* optimization switch (minimum size).

We see that the total footprint of the code is about $\frac{42210}{1024} \approx 41.2KB$, including a few demonstration programs. The RAM usage is dependant on how large TCP buffers you define, in this case the embedded system has several megabytes of RAM available so a large buffer of 128 KB was used. Apart from these 128 KB the LWIP stack would use circa 40 KB additional

Component	Code size	Constant size	Total size
USB stack	6574	1803	8557
LWIP TCP/IP stack	18616	107	18723
Basic Internet daemons	3414	1649	5063
System overhead	9000	1047	10047
Total	37604	4606	42210

Figure 15: The code sizes in bytes obtained for different parts of the prototype using the IAR C/EC++ Compiler. "System overhead" includes bootstrap code (processor initialization and similar) and a portion of the standard C library. Compare the TCP/IP stack size to the size of the uIP and LWIP stacks in Figure 9.

RAM for internal variables.

10 Conclusions

Concluding this thesis means stating if there was enough substance behind the concept of firmware loading using standardized protocols to go through the pains of doing a thesis on the subject.

I believe that there was. The text shows that there exists a number of protocols that can be used as a mixture for transfering firmware to a target system. We have seen that a viable alternative is to build a small network beteen a host system and an embedded system in order to transfer a firmware file and carry out any other things related to firmware loading on a level above electronics and tailormade protocols.

Through prototyping, I have showed that this can be done in practice, and even though the prototype only includes a single-processor system and a host system to manage it, the code sizes obtained show that this concept can both be deployed in hardware ROM code and in the future could also be deployed in a multi-core system, where each unit is autonomous and retrieves its firmware files and presents additional low-level interfaces over the network.

The ideas and implementation also illustrates the way to go with ever-increasing system complexity and unpredictable behaviours creeping up in designs. The robustness of networks will indeed be needed to keep large systems running during normal operations, and also adding a network of some sort to the very rudimentary single-threaded firmware upgrading process seems inevitable.

There are a number of security concerns with these solutions, most of which can be solved or taken as a given risk depending on the level of security required by the system in question.

The question remains if this thesis has made the world a better place to live in, generally speaking. I believe it actually does, because the ever-increasing plethora of different *ad hoc* solutions and protocols make up for a good part of the stress experienced by engineers in this field. Putting some standard in its place is always a good thing, as the acceptance of the TCP/IP suite has showed in the network world. Learning network administration today is a matter of understanding TCP/IP. It used to be a matter of understanding not only TCP/IP, but also SPX/IPX, NetBIOS/NetBEUI, the AppleTalk stack plus theorizing about numerous ITU-T X.nn-standards. The introduction of the USB bus has had a similar focusing effect in the area of peripheral connections.

References

[1] *A Brief History of the Internet*, The Internet Society
`http://www.isoc.org/internet/history/brief.shtml`
verifierad 2004-02-04

[2] Argonès, Xavier, González, José Luis and Rubio, Antonio: *Analysis and Solutions for Switching Noise Coupling in Mixed-Signal ICs*, Kluwer Academic Publishers, 1999, ISBN 0-7923-8504-7

[3] *Belcarra USB RNDIS/CDC White Paper*, version 1.0, January 15, 2003
`http://www.belcarra.com/rndis/Belcarra_RNDIS_Whitepaper_1.0.PDF`
verified 2004-09-29

[4] Benini, Luca and De Micheli, Giovanni: *Networks on Chips: A New SoC Paradigm*, IEEE Computer, January 2002.

[5] *Bits & bytes ur Datasaabs historia, tema flyg*, Datasaabs vänner, Linköping 1995, ISBN 91-972464-17.

[6] Cowan, Crispin, Wagle, Perry, Pu, Calto, Beattie, Steve, and Walpole, Jonathan: *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*, from *Proceedings of the DARPA Information Survivability Conference & Exposition Volume II of II*, IEEE, 1998.

[7] Dunkels, Adam: *Full TCP/IP for 8-Bit Architectures*, Swedish Institute of Computer Science.

[8] Dunkels, Adam: *Design and Implementation of the LWIP TCP/IP Stack*, Swedish Institute of Computer Science, February 20, 2001.

[9] Forouzan, Behrouz A.: *Data Communications and Networking*, 2nd edition, McGrav-Hill, Singapore, 2000, ISBN 0-27-232204-7

[10] Gollmann, Dieter: *Computer Security*, ISBN 0471978442

[11] Huang, Andrew "Bunnie": *Hacking the Xbox: An Introduction to Reverse Engineering*, No Starch Press, 2003, ISBN 1-59327-029-1

[12] *Libusb* - a software library. POSIX variants are available at:
`http://libusb.sourceforge.net/`
and a Microsoft Windows variant at:
`http://libusb-win32.sourceforge.net/`

[13] *Linux-USB Gadget API*, last modified August 2, 2004.
`http://www.linux-usb.org/gadget/`
verified 2004-09-29

[14] The LWIP project site at GNU:s Savannah project repository:
`http://savannah.nongnu.org/projects/lwip/`
verified 2004-10-01

[15] National Communications System Technology & Standards Division: *Federal Standard FS-1037C, Telecommunications: Glossary of Telecommunication Terms* August 7, 1996.

[16] Peacock, Craig: *USB in a NutShell - Making Sense of the USB Standard*, third relase, November 2002
`http://www.beyondlogic.org/usbnutshell/`
verified 2004-09-22

[17] *Remote NDIS Specification*, Rev 1.1, August 9, 2002, Microsoft Corporation.

[18] Stajano, Frank & Isoaki, Hiroshi: *Security Issues for Internet Appliances*, Proceedings of the 2002 Symposium on Applications and the Internet, IEEE 2002, ISSN 0-7695-1450-2/02.

[19] *Supported USB Classes*, Microsoft MSDN library, built thursday, sep 02, 2004. See also Jan Axelssons *USB Drivers Included in Windows* http://www.lvr.com/usbwin.htm
verified 2004-09-27

[20] *Universal Serial Bus Specification*, revison 2.0, April 27, 2000.

[21] *Universal Serial Bus Common Class Specification*, revision 1.0, December 16, 1997.

[22] *Universal Serial Bus Class Definitions for Communication Devices*, version 1.1, January 19, 1999.

[23] *Universal Serial Bus Device Class Specification for Device Firmware Upgrade*, version 1.1, August 5, 2004.

[24] *Wikipedia entry for the Apollo Guidance Computer* http://en.wikipedia.org/wiki/Apollo_Guidance_Computer
verified 2004-09-17

[25] *Wikipedia entry for Rube Goldberg* http://en.wikipedia.org/wiki/Rube_Goldberg
verified 2004-09-22

[26] Wolf, Wayne: *Computers as Components — Principles of Embedded Computer System Design*, Morgan Kaufmann Publishers, USA, 2001, ISBN 1-55860-693-9

A CDC+RNDIS Device descriptor

This is the device descriptor of the implemented CDC + RNDIS prototype as it appears when probed with the Linux **lsusb -v** command. Notice that clarifying string descriptors have been added to all interfaces.

```
Bus 002 Device 002: ID 0bdb:1010 Ericsson Business Mobile Networks BV
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               1.00
  bDeviceClass            2 Communications
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0        64
  idVendor           0x0bdb Ericsson Business Mobile Networks BV
  idProduct          0x1010
```

```
bcdDevice            1.00
iManufacturer           1 Ericsson Mobile Platforms AB
iProduct                0
iSerial                 0
bNumConfigurations      2
Configuration Descriptor:
  bLength                 9
  bDescriptorType         2
  wTotalLength           62
  bNumInterfaces          2
  bConfigurationValue     1
  iConfiguration          7 RNDIS
  bmAttributes         0xc0
    Self Powered
  MaxPower                0mA
  Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        0
    bAlternateSetting       0
    bNumEndpoints           1
    bInterfaceClass         2 Communications
    bInterfaceSubClass      2 Abstract (modem)
    bInterfaceProtocol    255 Vendor Specific (MSFT RNDIS?)
    iInterface              8 RNDIS Communications Control
      CDC Call Management:
        bmCapabilities       0x00
        bDataInterface        1
      CDC ACM:
        bmCapabilities       00
      CDC Union:
        bMasterInterface        0
        bSlaveInterface         1
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x85  EP 5 IN
      bmAttributes            3
        Transfer Type            Interrupt
        Synch Type               none
        Usage Type               Data
      wMaxPacketSize     0x0040  bytes 64 once
      bInterval              32
  Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        1
    bAlternateSetting       0
    bNumEndpoints           2
    bInterfaceClass        10 Data
    bInterfaceSubClass      0 Unused
    bInterfaceProtocol      0
    iInterface              6 Ethernet Data
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x84  EP 4 IN
      bmAttributes            2
        Transfer Type            Bulk
        Synch Type               none
        Usage Type               Data
      wMaxPacketSize     0x0040  bytes 64 once
```

```
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x03  EP 3 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               none
          Usage Type               Data
        wMaxPacketSize     0x0040  bytes 64 once
        bInterval               0
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength           80
    bNumInterfaces          2
    bConfigurationValue     2
    iConfiguration          3 CDC Ethernet
    bmAttributes         0xc0
      Self Powered
    MaxPower                0mA
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        0
      bAlternateSetting       0
      bNumEndpoints           1
      bInterfaceClass         2 Communications
      bInterfaceSubClass      6 Ethernet Networking
      bInterfaceProtocol      0
      iInterface              5 CDC Communications Control
        CDC Header:
          bcdCDC               1.10
        CDC Union:
          bMasterInterface        0
          bSlaveInterface         1
        CDC Ethernet:
          iMacAddress             6 Ethernet Data
          bmEthernetStatistics    0x00000000
          wMaxSegmentSize         1514
          wNumberMCFilters        0x0000
          bNumberPowerFilters     0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x85  EP 5 IN
        bmAttributes            3
          Transfer Type            Interrupt
          Synch Type               none
          Usage Type               Data
        wMaxPacketSize     0x0040  bytes 64 once
        bInterval              32
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        1
      bAlternateSetting       0
      bNumEndpoints           0
      bInterfaceClass        10 Data
      bInterfaceSubClass      0 Unused
      bInterfaceProtocol      0
      iInterface              0
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        1
      bAlternateSetting       1
      bNumEndpoints           2
      bInterfaceClass        10 Data
      bInterfaceSubClass      0 Unused
      bInterfaceProtocol      0
      iInterface              6 Ethernet Data
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x84  EP 4 IN
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               none
          Usage Type               Data
        wMaxPacketSize     0x0040  bytes 64 once
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x03  EP 3 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               none
          Usage Type               Data
        wMaxPacketSize     0x0040  bytes 64 once
        bInterval               0
  Language IDs: (length=4)
     0409 English(US)
```

B Raw Frame Generator

This is an Ethernet frame-generating Linux program in the C programming language, which just sends out a frame of 60 bytes, filled with zeroes. (This was trimmed to different packet sizes while testing.) As can be seen the MAC address of the interface is hardcoded to the address 0x123456789abc.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <features.h>    /* for the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR__ >= 1
#include <netpacket/packet.h>
#include <net/ethernet.h>     /* the L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>   /* The L2 protocols */
#endif
#include <sys/ioctl.h>
#include <net/if.h>

```

```
#define PACKETSIZE 60
#define MAXETHER 1514

static int get_ifindex(int sfd, char *intf)
{
  struct ifreq ifr;
  strcpy(ifr.ifr_name, intf);
  if ( ioctl(sfd , SIOCGIFINDEX, &ifr) < 0) {
    printf("Could not locate interface %s.\n", intf);
    exit(-1);
  }
  return ifr.ifr_ifindex;
}

int main(int argc, char *argv[])
{
  char *ifname="eth1";
  int sockfd;
  struct sockaddr_ll address;
  int proto = htons(ETH_P_ALL);
  int err;
  unsigned char buf[PACKETSIZE];

  sockfd = socket(PF_PACKET, SOCK_RAW, proto);
  if (sockfd < 0) {
    printf("Could not open socket\n");
    exit(-1);
  }
  memset ((void *)&address, 0, sizeof(struct sockaddr_ll));
  address.sll_family = AF_PACKET;
  address.sll_protocol = htons(ETH_P_ALL);
  address.sll_ifindex = get_ifindex(sockfd, ifname);
  address.sll_hatype = 1;
  // address.sll_pkttype;
  address.sll_halen = 8;
  address.sll_addr[0] = 0x12;
  address.sll_addr[1] = 0x34;
  address.sll_addr[2] = 0x56;
  address.sll_addr[3] = 0x78;
  address.sll_addr[4] = 0x9a;
  address.sll_addr[5] = 0xbc;

  // blank packet
  memset ((void *) &buf, 0, sizeof(buf));
  buf[0] = 0x12;
  buf[1] = 0x34;
  buf[2] = 0x56;
  buf[3] = 0x78;
  buf[4] = 0x9a;
  buf[5] = 0xbc;
  // Set packet type to IPv4
  buf[0x0c] = 0x08;
  buf[0x0d] = 0x00;
  // Set packet length
  buf[0x10] = (PACKETSIZE - 14) >> 8;
  buf[0x11] = (PACKETSIZE - 14) & 255;

  err = sendto(sockfd, &buf, PACKETSIZE, 0x00,
               (struct sockaddr *) &address, sizeof(struct sockaddr_ll));
  if (err < 0 ) {
    printf("Could not send packet... Code %d\n", err);
    exit(-1);
  }

  close(sockfd);
}
```