Birgit Schlager, BSc

Low-Cost IoT Device Framework with Long-Term Availability

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn., Bernd Deutschmann

Institute of Electronics (IFE)

in cooperation with linked IP GmbH

Graz, März 2018

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

The aim of this thesis is designing an internet of things (IoT) device framework as a basis for the development of mechatronic IoT devices. An IoT device framework is a generic solution for recurring problem definitions in individual applications. The goal is to combine the advantages of commercial off-the-shelf (COTS) devices and the advantages of individual devices, that are developed from scratch. The high reusability of the framework results in a shorter time-to-market (TTM) and in a reduced total-cost-of-ownership (TCO).

Currently used hardware concepts for IoT devices are evaluated and compared. Based on these evaluations, the schematics of the IoT device framework and the layout of an exemplary IoT device are developed. Furthermore, possible software concepts are explained and the most suitable concept is described in more detail. During the design of the IoT device framework several challenges for both, schematics and printed circuit board (PCB) design, occurred and several guidelines had to be considered. This aspects are listed in the thesis. At the end the life-cycle relevant points are stated.

Kurzfassung

Das Ziel der Arbeit ist es, ein „Internet of Things“ (IoT) Device Framework zu entwickeln, das als Ausgangspunkt für die Entwicklung mechatronischer IoT-Geräte dient. Es werden die Vorteile von commercial off-the-shelf- (COTS-) Komponenten mit den Vorteilen von Geräten, die zur Gänze neu entwickelt werden, kombiniert. Ein IoT Device Framework stellt eine generische Lösung für wiederkehrende Problemstellungen in individuellen Anwendungsfällen dar. Im Vergleich zu derzeitigen Lösungen bringt das IoT Device Framework durch den hohen Wiederverwendbarkeitsgrad ein kürzeres time-to-market (TTM) und geringere total-cost-of-ownership (TCO) mit sich.

Im Rahmen der Arbeit werden verschiedene Hardware-Konzepte für IoT-Geräte verglichen und basierend darauf passende Konzepte und Komponenten für das Framework gewählt. Als Beispiel wurde ein Layout mit allen Komponenten des Frameworks entwickelt. Weiters werden verschiedene Softwarelösungen gegenübergestellt. Aspekte, die während dem Schaltplan- und Layoutdesign beachtet werden müssen, werden diskutiert. Am Ende der Arbeit werden Life-Cycle relevante Punkte erläutert.

Acknowledgement

I would like to thank everyone who helped me and who has contributed to this work. Particularly I want to thank…

… Prof. Bernd Deutschmann for being a great supervisor for both, my master’s but also for my bachelor’s thesis. Besides, he was an instructive mentor for my master’s program and helped me to compile my individual electronics study program.

… the Linked IP GmbH for providing a comfortable working environment and all the working tools but also for being a great employer during my entire study program.

… Dr. Rudolf Golser for being an incredible supervisor. He supported me with a lot of know-how and had an answer for every of the uncountable questions I had, but most importantly he provided the needed motivation to carry out this project.

… Prof. Marcel Baunach for being a great guidance when I was about to write this thesis. He supported me with lots of theoretical background and literature to the considered topics.

... my family and my friends. In particular I want to thank my mum, my dad, my little brother and Eva for being the essential support all over the time.

Contents

List of Abbreviations

6LoWPAN	IPv6 over Low power Wireless Personal Area Network
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
BOM	Bill Of Materials
CAN	Controller Area Network
CISC	Complex Instruction Set Computer
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DBC	Dual Board Computer
DDR	Double Data Rate
DMA	Direct Memory Access
DMIPS	Dhrystone Million Instructions Per Second
DRAM	Dynamic Random Access Memory
DTC	Device Tree Compiler
ECC	Error Correction Code
eCos	embedded Configurable operating system
EEPROM	Electrically Erasable Read Only Memory
EMC	ElectroMagnetic Compatibility
ESL	Equivalent Series L (inductance)
ESR	Equivalent Series Resistor
FPGA	Field Programmable Gate Array
GPOS	General Purpose Operating System
GPU	Graphics Processing Unit
HDMI	High-Definition Multimedia Interface
HSIC	High-Speed Inter Chip
HSYNC	Horizontal SYNChronization
I/O	Input/Output
I^2C	Inter-Integrated Circuit
I^2S	Inter-IC Sound
IC	Integrated Circuit
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Intellectual Property
LED	Light-Emitting Diodes
LTS	Long Term Support
LVDS	Low Voltage Differential Signal
MCU	MicroController Units
MDI	Media Dependent Interface
MII	Media Independent Interface
MLCC	Multi Layer Ceramic Chip
MMC	MultiMedia Card
MMU	Memory Management Unit
MPU	Memory Protection Unit, Micro Processing Unit
MTD	Memory Technology Device
MTU	Maximum Transmission Unit
ONFi	Open Nand Flash interface
OS	Operating SystemOperating System
OSI	Open Systems Interconnection
PC	Personal Computer
PCB	Printed Circuit Board

PHY	PHYsical layer
PLM	Product Lifecycle Management
PMIC	Power Management Integrated Circuit
PoE	Power over Ethernet
QR	Quick Response
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RMII	Reduced Media Independent Interface
RT	real time
RTAI	Real Time Application Interface
RTC	Real Time Clock
RTOS	Real Time Operating System
SBC	Single Board Computer
SDRAM	Synchronous Dynamic Random Access Memory
SiP	System-in-Package
SoB	System-on-Board
SoC	System-on-Chip
SoM	System-on-Module
SPI	Serial Peripheral Interface
SPU	Supervisory Processing Unit
SSD	Solid State Drive
TCO	Total-Cost-of-Ownership
TCP/IP	Transmission Control Protocol/Internet Protocol
TFTP	Trivial File Transfer Protocol
TTM	Time-To-Market
TVS	Transient Voltage Suppressor
UART	Universal Asynchronous Receiver Transmitter
UBI	Unsorted Block Image
UBIFS	Unsorted Block Image File System
ULPI	UTMI+ Low Pin Interface
USB	Universal Serial Bus
USB OTG	USB On-The-Go
UTMI	USB 2.0 Tranceiver Macrocell
VSYNC	Vertical SYNChronization

1 Introduction and Terminology

The term internet of things (IoT) was introduced since more and more mechatronic devices got connected to the internet. The trend is to connect even smallest devices to the internet with the main approach to distribute data collection devices as good as possible. Collected data from each device is then transferred using the internet and often merged in a central cloud storage.

A lot of recent surveys show that the IoT market was growing in the last years and forecasts show that it will even grow much more in the next years [1]. As an example, a forecast of Gartner, Inc. predicts that the number of used IoT devices worldwide will increase from 8.4 billion in 2017 to 20.4 billion connected devices in 2020 [2]. According to GrowthEnabler & MarketsandMarkets, the IoT market volume will grow from 194.68 billion USD in 2017 to 457.29 billion USD in 2020 [3].

This market growth makes the IoT market attractive for putting effort into studying and developing IoT technologies. IoT gateways and IoT devices are atomic computing units in the internet of things. Typically, the IoT gateway is the link between one or more IoT devices and the internet which can be seen in Figure 1. Its task is to translate communication protocols. Often the separation between IoT gateway and IoT device is not completely clear. Therefore, an IoT gateway and an IoT device can also be combined in one system.

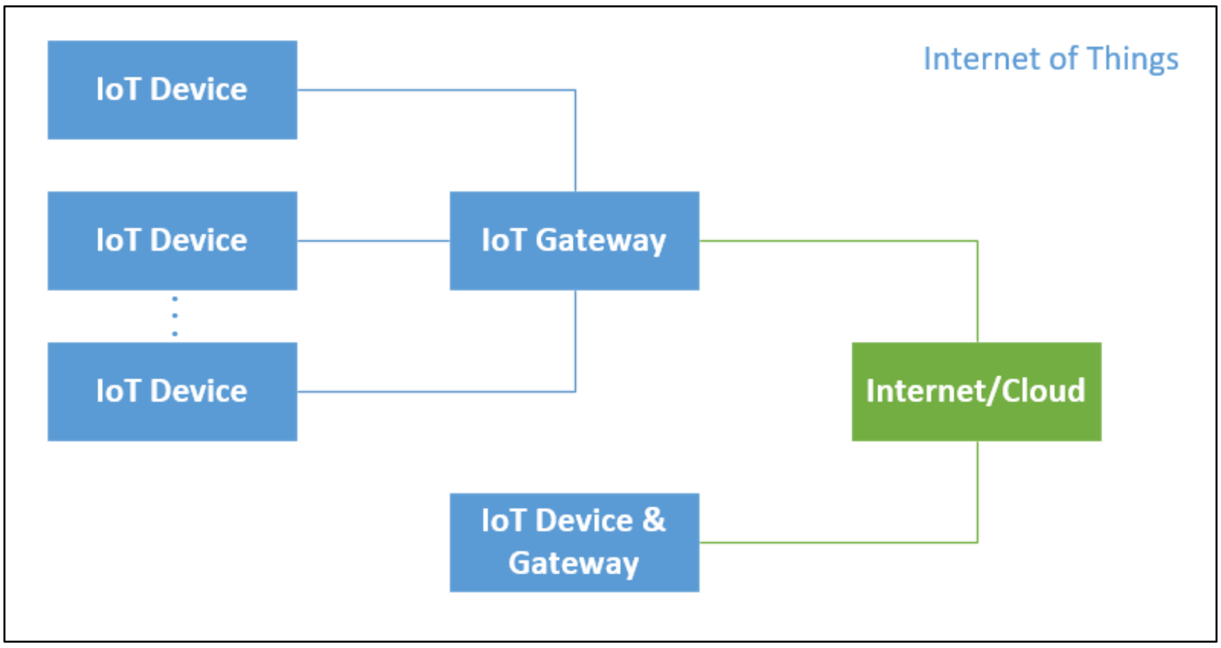

Figure 1 Internet of Things.

Within the scope of this thesis an IoT gateway and device framework is set up. The framework consists of several schematic modules which can be reused for developing a new IoT gateway, a new IoT device or a new combined IoT gateway and device (Figure 2). Due to the high degree of reusability, the developed IoT device framework provides an easy and fast development of customer and application specific IoT devices. Within the thesis, all schematic modules of the IoT device framework are used to develop an example printed circuit board (PCB), so an example IoT device. The functional blocks are separated clearly on schematic level but also on PCB level. Therefore, layout snippets of the PCB, which belong to a specific schematic module of the framework, can be reused.

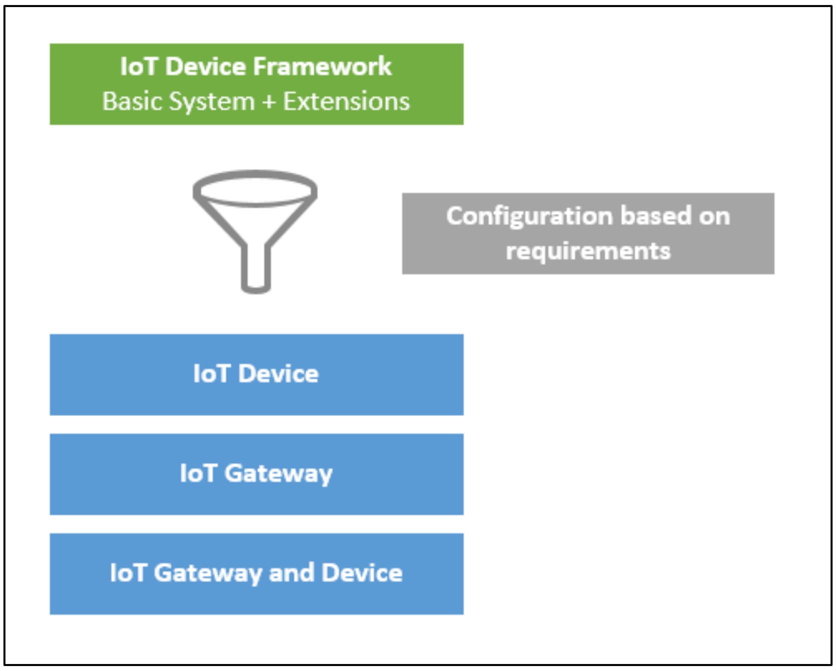

Figure 2 Basic Idea of IoT Device Framework.

The thesis first describes and compares theoretical solution concepts for developing an IoT device. Both, hardware and software aspects are considered while developing the IoT device framework. As the name of the thesis implies, further focuses lie on the long-term availability and the low-cost of the components that are used for the IoT device framework. The thesis consists of the following chapters:

In Chapter 2 several IoT hardware development concepts are discussed and compared:

The computing part of an IoT device can be classified in different levels of integration. In Section 2.1, **System-on-Chip (SoC)**, **System-on-Module (SoM)** and **Single-Board-Computer (SBC)** which are commonly used levels of integration are compared.

The main two approaches to develop an IoT device are the **Single-Board-Computer (SBC)** on the one hand, and the **Dual-Board-Computer (DBC)** on the other hand. An SBC integrates an SoC and all the other components of the IoT device on one PCB. In contrast to that, the DBC solution consists of two PCBs: an SoM and a carrier board which integrates and expands the SoM. The SoC as the processing unit of the device is integrated in the SoM when using the DBC approach. These two approaches are described in more detail in Section 2.2.

Focusing on Single-Board-Computers, it can be distinguished between two main types: **standard SBCs** and **individual SBCs**, respectively. Standard SBCs are off-the-shelf SBCs without any further need of hardware development. Compared to standard SBCs, individual ones are developed and designed for a specific application. The advantages and disadvantages of both solutions are compared and evaluated in Section 2.3.

In Chapter 3 the idea and the components of an IoT device framework are described in general:

The IoT device framework can be separated into two main parts: the **basic system** which contains the mandatory modules, and the **optional system extensions**. The basic system includes the modules, which guarantee the basic processing functionalities of the developed IoT device. Therefore, the components listed in 3.1 have to be part of the IoT device necessarily. According to the application's needs, optional extensions listed in 3.2 can be added to the IoT device. In this section, especially the structure and properties of possible interfaces are highlighted since one main property of the IoT device framework is the abstraction of peripherals.

In Chapter 4 possible software solutions are evaluated and compared:

There are two main approaches of running a software application on an IoT device which are described in more detail in Section 4.1. On the one hand, an application can be executed on **bare-metal**, therefore on the hardware directly. On the other hand, an **operating system** can be used as an abstraction layer between the hardware and the user application.
When it comes to select an operating system, the first step is to choose between a **real-time operating system (RTOS)** and a **general-purpose operating system (GPOS)**. The

advantages and disadvantages of both operating system approaches are compared in Section 4.2.

Furthermore, solutions how a GPOS can become real-time capable are discussed in Section 4.3. There are two main kernel architectures: **monolithic kernels** as a single-kernel approach and **microkernels** as dual-kernel approaches. The two main solutions for Linux are the **PREEMPT_RT** patch as a monolithic kernel approach and **Xenomai** as a microkernel approach.

In Chapter 5 latencies and data rates of interfaces are listed and discussed:

To ensure real-time capability also for abstracted periphery the **timing properties of interfaces** have to be considered. The most-common interfaces are compared regarding their **latencies** and **data rates**. Furthermore, these properties are compared to the latencies of the operating system. It is assessed if the interface or the operating system is the slower part and therefore constraining for the real-time capability of the periphery.

In Chapter 6 features of the developed and implemented IoT device framework are stated:

After considering several aspects and comparing different solutions for an IoT device, the IoT device framework which was developed so the **practical implementation of the IoT device framework** is described. Section 6.1 shows a block diagram of the framework. The used components are listed and discussed in Section 6.2. An SBC with all the modules of the IoT device framework has been designed. Properties like power consumption and dimensions of that developed SBC as a specific example for an IoT device. In Section 6.3 the software approach and in Section 6.4 the commissioning of the SBC are described.

In Chapter 7 the challenges of the schematic design are described:
Parts of the schematic are described in more detail. Since the SBC can be supplied by either Power over Ethernet (PoE) or by one of the two Mini-Universal Serial Bus (USB) connectors, especially the multiplexing of these different power supply alternatives is described in more detail. Furthermore, the power supply in production mode, so during commissioning, was a challenge and is described in this chapter.

In Chapter 8 the considered aspects while designing the PCB are listed:
Besides designing the schematics also designing the PCB came with a lot of challenges. Especially guidelines to achieve a high electromagnetic compatibility (EMC) were considered for the design of the PCB. Furthermore, the constraints for routing high speed signals (e.g. signals to the Dynamic Random Access Memory (DRAM)) are stated.

In Chapter 9 life-cycle relevant aspects are discussed:

As the IoT device framework should provide **long-term availability**, life-cycle relevant aspects such as component tracking, configuration management, software deployment and update functionality are stated.

2 Comparison of IoT Device Development Concepts

2.1 Levels of Integration: SoC, SoM and SBC/DBC

A system can be implemented in several levels of integration. There are three main levels. The first and smallest level of integration is the so-called System-on-Chip (SoC). As an extension of a SoC, a System-on-Module (SoM) can be developed which usually includes the parts of an SoC and some additional components. Extending the SoM further leads to either a Single-Board-Computer (SBC) or a Dual-Board-Computer (DBC). The three levels of integration are illustrated in Figure 3. A bigger ellipse stands for a higher number of components integrated. These definitions can also be found on many websites of SoM and SBC developers like in [4], [5].

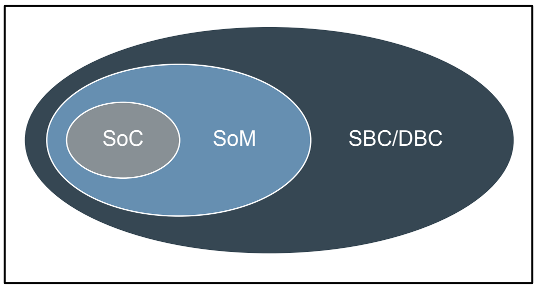

Figure 3 Levels of Integration.

A **System-on-Chip** integrates a system on a single chip. Similar to SoCs, a system can be integrated as a System-in-Package (SiP) which includes several chips in a package of an integrated circuit (IC). A SoC usually includes the processor core, caches, internal memory, interfaces for external memory, interfaces for connectivity, interfaces for multimedia, a real-time clock, direct memory access (DMA) controllers, timers, pulse width modulators, watchdogs and units for hardware security aspects [6]. Furthermore, physical layers (PHYs) for some interfaces are already part of the SoC but some PHYs still have to be added externally to the SoC as a separate IC. As an example, the block diagram of the i.MX6UL integrates the USB PHY whereas the Ethernet PHY has to be added externally [7]. A more detailed explanation of the integration of PHYs and a list of example interfaces is given in Section 3.2.3.

System-on-Modules are the next level of integration compared to SoCs. An SoM is a printed circuit board (PCB) which includes an SoC as one component. Additionally, it consists of external memory, so external random-access memory (RAM) as volatile memory and NAND flashes or NOR flashes as non-volatile memory. Such SoMs are typically attached to another PCB which is called carrier board or baseboard. On a carrier board, application specific components for example application specific interfaces and connectors can be added individually. As mentioned before, some PHYs are not integrated in the SoC so they are part of the SoM or the SBC. Since SoMs include the components which are necessary to do the main computing, in some literature the term Computer-on-Module (CoM) is used alternatively to SoM [4], [5].

Single-Board-Computers (SBCs) and also **Dual-Board-Computers** (DBCs) represent a further level of integration compared to SoMs. The difference between SBCs and DBCs is described in Section 2.2 but basically both have the same scope regarding the parts included. An SBC/DBC consists of the same components as an SoM, so it includes an SoC and external memory. Additionally, application specific interfaces and connectors are included on the SBC/DBC as well. Sometimes the term System-on-Board (SoB) is used for such SBCs but further in the thesis the term SBC will be used. The term SBC is used frequently by manufacturers like Toradex [4] and Phytec [5] whereas the term DBC is rarely used in the literature and by manufacturers.

2.2 Comparison of SBC Solutions with DBC Solutions

As described in Section 2.1, SBCs integrate an SoC, external memory and connectors on one PCB. In contrast to that, another main development concept is to design DBCs. As the name says, this solution implements the system on two PCBs. One PCB is a SoM which includes the SoC and external memory and the other PCB, called carrier board, integrates the SoM as the main component and extends it with the required connectors. DBCs can be realized in two ways. The SoM can either be soldered onto the carrier board or the SoM can be attached to the carrier board via connectors. The SoM as an additional intermediate development stage for DBCs compared to SBCs is shown in Figure 4.

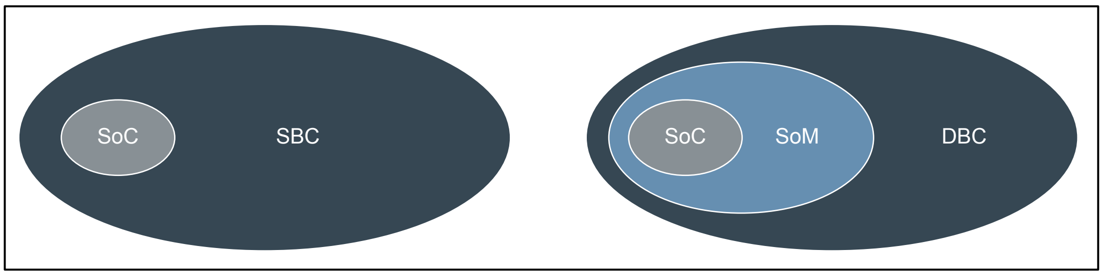

Figure 4 Comparison: SBC and DBC.

In the following the two solutions are described in more detail. Table 1 summarizes the properties of the two approaches.

The fact that SBCs only consist of one PCB has the big advantage that SBCs are more robust against mechanical stress than DBCs. Especially, if the SoM and the carrier board are combined via connectors a lot of mechanical stress can lead to a damage and therefore a high failure rate. In contrast to that, the development time, so the time-to-market (TTM), is lower when using DBC solutions instead of SBC solutions. An SBC has to be designed from scratch for each application as a whole. Contrary to SBCs, DBCs use an already developed SoM as the main part. Therefore, only the carrier board which is much simpler (lower number of layers, broader trace width, higher distance between tracks, bigger vias) has to be developed for the specific application. This leads further to the advantage that the layout of the SoC and the extending memory, which is the most critical part, is already done and verified. So, considering the development point of view, DBC solutions have a lower risk for failure than SBC solutions. When using the DBC approach, the SoM can either be developed in-house or it can be outsourced. The disadvantage of outsourcing the SoM is the dependency on the SoM manufacturer which leads to less flexibility in redesign. A further factor for deciding whether to implement the required system as an SBC or a DBC are the cost. An SBC is only profitable from a certain quantity since the higher development cost for an SBC have to be covered by the lower variable cost and therefore higher gross margin. This calculation is done in Section 2.2.1 [8].

As described above, both, SBCs and DBCs have advantages regarding different aspects. In the following table these advantages are summarized:

	SBC	**DBC (SoM + Carrier Board)**
time-to-market (TTM)	longer	shorter
development risk	higher	lower
profitability	higher quantities	lower quantities
mechanical robustness	higher	lower
prototype cost	higher	lower
bill of materials (BOM)	longer	shorter
redesign flexibility	higher	lower

Table 1 Comparison of SBCs and DBCs.

A lot of literature discusses on whether to choose an SBC or a DBC solution. The following two articles state advantages and disadvantages of the two solutions.

As a first example, the German electronics magazine “Elektronik: Fachmedien für industrielle Anwender und Entwickler” discusses the importance of DBCs in embedded applications in the article “Modular ohne Module“, which can be translated to “Modular without modules“. Having a modular system which contains a main board with several extension cards is common in the PC market but not suitable for a lot of applications, especially not for embedded and IoT applications. Such a modular system leads to the problem that it requires much space and the power supply unit is often oversized because also possible extensions have to be supplied with power too. On the opposite, implementing the system as an SBC leads to a static and non-scalable implementation. Therefore, the idea is to design a DBC consisting of an SoM and a carrier board as an intermediate solution [9].

As a second example, Toradex states that DBC solutions are more suitable for embedded product development than SBC solutions because of their design flexibility and design scalability [4], [10]. Besides Toradex having their Colibri product line, Phytec is another provider of DBC solutions with their phyBOARD product line. Phytec uses the term SBC but in this thesis it is referred as a DBC solution since it consists of a SoM and a carrier board. Both, Toradex and Colibri, extend their standard SoMs with an application specific carrier board [5].

As described in more detail in Chapter 3, the IoT device framework is designed for the development of SBC solutions. The IoT device framework which consists of several reusable schematic modules and reusable PCB snippets has the benefit that the typical disadvantages of a conventional SBC solution are minimized. If the IoT device framework is used for the development of SBCs the time-to-market is much lower than developing it from scratch. Furthermore, the critical parts like the schematic and layout of the SoC and RAM are already designed and verified so reusing it lowers the development risks.

2.2.1 Calculation: Crossover Point of SBC Solution

SBC solutions only pay off from a higher quantity compared to DBC solutions. The higher development cost of SBCs compared to DBCs have to be covered by their higher gross margin. A crossover point calculation, which should result in a specific quantity for the SBC solution, can be done. The crossover quantity can be calculated by using Equation (10).

The following formula symbols are used:

$\pi \ldots Profit$
$TR \ldots Total\ Revenue$
$TC \ldots Total\ Cost$
$TFC \ldots Total\ Fix\ Cost$
$TVC \ldots Total\ Variable\ Cost$
$NRE \ldots Non - Recurring\ Engineering$
$VC \ldots Variable\ Cost\ per\ unit$
$Q \ldots Crossover\ Quantity$
$P \ldots Price\ per\ unit$

The profit is the difference between the total revenue and the total cost. The total revenue can be calculated as the product of the quantity sold and the price per unit. The total cost consists of the total fix cost and the total variable cost where the total fix cost is the non-recurring engineering and the total variable cost are the variable cost per unit multiplied by the quantity.

Calculation of the profit for the SBC solution:

$$\pi_{SBC} = TR_{SBC} - TC_{SBC} \tag{1}$$

$$\pi_{SBC} = TR_{SBC} - (TFC_{SBC} + TVC_{SBC}) \tag{2}$$

$$\pi_{SBC} = Q \cdot P - NRE_{SBC} - Q \cdot VC_{SBC} \tag{3}$$

Calculation of the profit for the DBC solution:

$$\pi_{DBC} = TR_{DBC} - TC_{DBC} \tag{4}$$

$$\pi_{DBC} = TR_{DBC} - (TFC_{DBC} + TVC_{DBC}) \tag{5}$$

$$\pi_{DBC} = Q \cdot P - NRE_{DBC} - Q \cdot VC_{DBC} \tag{6}$$

The following calculations result in the minimum quantity for which the SBC solution is more profitable than the DBC solution:

$$\pi_{SBC} > \pi_{DBC} \tag{7}$$

$$Q \cdot P - NRE_{SBC} - Q \cdot VC_{SBC} > Q \cdot P - NRE_{DBC} - Q \cdot VC_{DBC} \tag{8}$$

$$-NRE_{SBC} - Q \cdot VC_{SBC} > -NRE_{DBC} - Q \cdot VC_{DBC} \tag{9}$$

$$Q > \frac{NRE_{SBC} - NRE_{DBC}}{VC_{DBC} - VC_{SBC}} \tag{10}$$

2.3 Comparison Standard SBC with Individual SBC

SBCs can be divided into two groups. On the one hand, standard SBCs and on the other hand individual SBCs. In this Section the properties of both solution are described in more detail and are summarized in Table 2.

Standard, so **commercially off-the-shelf (COTS) SBCs,** are solutions which are static and inflexible. They are sold as an already developed and finished product. Standard SBCs integrate a lot of functionalities and interfaces which are not used by the individual application. This overhead leads to unnecessary complex IoT devices which require a lot of space. Furthermore, the form factor of the PCB cannot be adapted to specific mechanical requirements. Since standard SBCs are usually designed generically, the needed application specific periphery has to be abstracted to separate PCBs which leads to a larger overall system. The big advantage of such off-the-shelf SBCs is that there are no non-recurring engineering cost [11].

Compared to standard IoT devices, an individual and therefore **custom SBC** is a solution developed for a specific application. It is possible to include only the needed interfaces and to adapt the PCB to its mechanical requirements such as PCB contour and component placement. An individual IoT device is paying off from a certain quantity on. The cost comparison between standard SBCs and individual SBCs is similar to the cost comparison between SBCs and DBCs. The calculation of the crossover quantity is done in Section 2.3.1 [11].

The following table summarizes the properties of standard SBCs compared to individual SBCs:

	Standard SBC	**Individual SBC**
time-to-market	shorter	longer
flexibility and scalability	only provided implementations	fully scalable
profitability	lower quantities	higher quantities
development risk	lower	higher

Table 2 Comparison of Standard SBC to Individual SBC.

2.3.1 Calculation: Crossover Point of Individual SBC

Similarly to SBC solutions in Section 2.2.1, individual SBCs pay off from a higher quantity compared to standard SBC solutions. The occurring development cost of the individual SBC have to be covered by its lower variable cost and therefore higher gross margin. Compared to an individual SBC, standard SBCs come without any non-recurring engineering cost. The crossover quantity can be calculated by using Equation (20).

The same formula symbols as in Section 2.2.1 are used.

Calculation of the profit for the individual SBC solution:

$$\pi_{ind} = TR_{ind} - TC_{ind} \tag{11}$$

$$\pi_{ind} = TR_{ind} - (TFC_{ind} + TVC_{ind}) \tag{12}$$

$$\pi_{ind} = Q \cdot P - NRE_{ind} - Q \cdot VC_{ind} \tag{13}$$

Calculation of the profit for the standard SBC solution:

$$\pi_{std} = TR_{std} - TC_{std} \tag{14}$$

$$\pi_{std} = TR_{std} - TVC_{std} \tag{15}$$

$$\pi_{std} = Q \cdot P - Q \cdot VC_{std} - \underbrace{NRE_{std}}_{=0} \tag{16}$$

The following calculations result in the minimum quantity for which the individual SBC solution is more profitable than the standard SBC solution:

$$\pi_{ind} > \pi_{std} \tag{17}$$

$$Q \cdot P - NRE_{ind} - Q \cdot VC_{ind} > Q \cdot P - Q \cdot VC_{std} \tag{18}$$

$$-NRE_{ind} - Q \cdot VC_{ind} > -Q \cdot VC_{std} \tag{19}$$

$$Q > \frac{NRE_{ind}}{VC_{std} - VC_{ind}} \tag{20}$$

3 IoT Device Framework

The comparisons regarding different IoT device development concepts, which are done in Chapter 2, lead to the idea of an **IoT device framework**.

The two main development concepts of IoT devices, which are described in Section 2.2, are SBCs and DBCs. SBCs have the disadvantage that they have to be designed from scratch for every new IoT device which can take a lot of development time. Therefore, a common way to decrease the time-to-market is to use DBC solutions instead of SBC solutions. The big disadvantages with DBC systems are that they require more space and the mechanical robustness decreases because of the required connection between the main board and the baseboard. The IoT device framework is a solution which provides low-cost single board computers with a low time-to-market. Due to the fact that the already verified and tested functional parts are clearly separated regarding schematics and layout, these parts can be reused easily for the development of a new application specific SBC. The reusability of the parts of the framework lead to a very fast development.

The IoT device framework is a modular hardware framework which consists of the basic system which is described in 3.1 and several extensions which are described in 3.2. The basic system includes the minimum functionalities for building up an IoT device. So, without any extensions it mainly includes the parts of a previously described SoM. By adding extensions to the basic system an IoT device can be configured to comply with the requirements for a specific and individual application. For example, the included power management extension with a power management IC (PMIC) or a supervisor controller can be added as an optional feature.

Besides the application specific extensions to the basic system, the basic system is scalable itself. The parts of the basic system have to be used mandatorily but they can be scaled-down or scaled-up regarding their performance and size of memory. For example, the RAM which is part of the basic system can be varied in size because it has a standardized package footprint, a standardized pin assignment and it is connected to the controller via standardized interfaces (JEDEC memory standard, e.g. DDR3 SDRAM standard which was published in 2012 [12]). Furthermore, the bus width of the connection to the RAM and the storage bandwidth can be scaled-down or scaled-up.

The idea of having a modular system with already defined and tested parts is commonly used in the area of SoC designs. Similar to the IoT device framework but on another level of abstraction, SoC designers reuse and integrate parts of the schematic and layout in integrated circuits (ICs). Such reusable parts are so-called intellectual property cores (IP cores) which should be designed as generic as possible. When developing a new SoC, the needed, reusable and already verified parts are selected, integrated and connected to the internal bus. A common bus architecture is the Advanced Microcontroller Bus Architecture (AMBA) which consists of several buses. After integrating the parts, the SoC has to be verified and tested as a whole [6], [13]. As a specific example, such a reusable design for SoCs is described and particularly evaluated for a Bluetooth baseband processor in [6]. In this thesis, such a modular hardware solution is described and developed on PCB level. Modules and therefore entire schematic sheets and layout snippets can be reused and integrated in a solution for an IoT device.

3.1 Basic System

The basic system consists of the mandatory parts which are shown in Figure 5. The main part of the basic system is the SoC as the computing unit. Further parts are the power management and the memory. The power management ensures that the main power supply voltage rails of the SBC are established properly. The main parts of the power management are the power supply voltage rails, the power sequencing and the reset logic. Furthermore, external memory on the SBC, including volatile and non-volatile memory, belongs to the basic system.

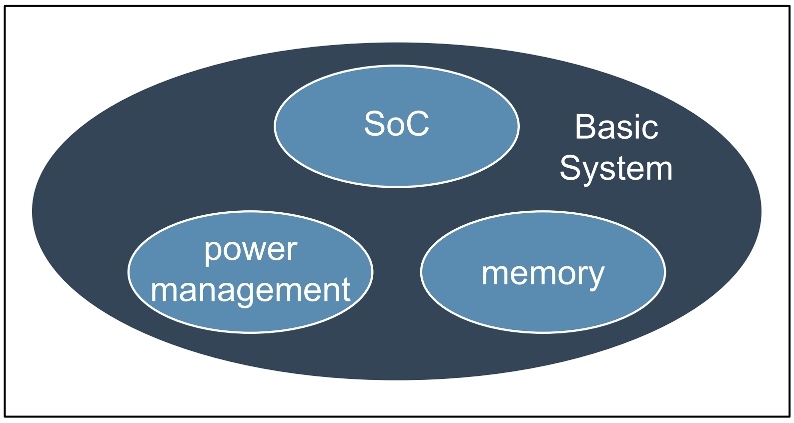

Figure 5 Components of a basic SBC.

3.1.1 SoC as the Computing Unit

The first thing that has to be selected appropriate to the system and its requirements is the computing unit of the basic system and therefore of the IoT device. Conventional computing units can be separated into **central processing units (CPUs)**, **micro processing units (MPUs)** and **microcontroller units (MCUs)**.

Among these types of computing units, the CPU is most powerful regarding processing power. In the past, a single CPU was used for centralizing the processing tasks of many smaller modules. Due to the fact that more and more devices are connected to a single CPU, the overall data became too high for only one processing unit. The resulting idea was to decentralize the computing power from one CPU to many MPUs or MCUs which have the advantage that the data is processed locally. CPUs are still used for personal computers (PC) but it is likely to use scaled down MPUs or MCUs for IoT devices.

Selecting between MPUs and MCUs depends on the factors of price, performance, power consumption, scalability of memory and connectivity. The basic difference between the architecture of MPUs and MCUs is that MCUs have a flash memory and power management units included. The included flash memory has the advantage that the start-up and the execution of applications is fast. The big disadvantage in embedding the flash memory is that it is not scalable so it has a constrained memory size. The limited memory size is an important aspect when it comes to the decision whether to run the application on bare-metal or if an abstraction layer like a real-time operating system (RTOS) or a general-purpose operating system (GPOS) needs to be implemented. An operating system is especially important if the security aspect plays an important role for the application. Due to the fact that the memory is inside the MCU, also the power supply for the memory can be embedded in the MCU. In contrast to that, choosing an MPU with separate memory ICs leads to the disadvantage that also the required power supply parts have to be added externally. Regarding memory protection and memory management, an MCU includes only a memory protection unit (MPU) whereas an MPU includes the more extensive memory management unit (MMU). Furthermore, MCUs have lower power consumption and power saving modes can be handled easier. Regarding performance MPUs are more powerful. For example, an MCU with an ARM Cortex-M4 has 150 DMIPS compared to an MPU with an ARM Cortex-A5 with 850 DMIPS. Since a general-purpose operating system requires about 300 to 400 DMIPS a MCU would be not suitable for running such an operating system. An RTOS on the contrary only requires 50 DMIPS so an MCU would be powerful enough. For high speed interfaces an operating system and thus an MPU is needed. A main advantage of the MCU is that it is cheaper than an MPU. As a result, MCUs are used for power and cost saving relevant applications where no operating system is used. MPUs are for more performance intensive and scalable solutions. [14] Depending on these comparisons, it was concluded that an MPU would be more suitable for the IoT device framework than an MCU. It is important to implement an operating system especially because of the required degree of security if a device is connected to the internet. The importance of security for devices in the internet is also shown in [15] where security is stated as the most important concern when developing an IoT device. Other advantages of using an operating system are stated in Chapter 4.

The trend in the last years was to develop so-called **SoCs** for embedded applications. According to the “Linux-Magazin”, such SoCs include, besides the general-purpose processing unit, components like a graphics processing unit (GPU), memory, a USB controller and USB PHY, power management units and wireless radios [16]. The integration of these components lead to a lot of benefits like the requirement of less space and lower cost. Therefore, considering the state-of-the-art shows that SoCs are the type of computing units that should be used for IoT applications.

The main architectures for such MPUs implemented in form of an SoC, are currently **Intel**, **ARM** and **MIPS**. In contrast to the Reduced Instruction Set Computer (RISC) architecture which is used in ARM and MIPS technologies, Intel uses the Complex Instruction Set Computer (CISC) architecture. The CISC architecture has longer instruction words than the RISC architecture. Therefore, a CISC needs fewer instructions to execute the same tasks. The CISC architecture is used for powerful PCs where a lot of computational power is needed. IoT devices usually do not need that much computational power, so the RISC architecture is sufficient to execute the tasks of an IoT device. Furthermore, the low power consumption of a RISC is essential for IoT devices. At first, it was a challenge to use the RISC architecture because it was not capable to run full general-purpose operating systems. In the last years operating systems which can be run on a RISC were developed (e.g. Windows 10 IoT Core) [17].

Intel has a very high share in the PC market but by using the CISC architecture, Intel was not able to enter in and contribute to the IoT market at the beginning. Due to the fact that the IoT sector was rapidly growing, Intel wanted to enter this market. To achieve that goal, they developed the Intel Atom E3900 series [18]. Furthermore, Intel licensed intellectual properties (IPs) of ARM about one year ago according to bloomberg.com [19]. The ARM architecture is the most suitable architecture for the IoT Device Framework. Also comparable solutions for IoT devices on the market mainly use the ARM architecture.

The ARM IP cores can be split up in several families: **ARM Cortex-M**, **ARM Cortex-A** and **ARM Cortex-R**. The ARM Cortex-M family is designed for implementing MCUs. So, the Cortex-M family is not suitable for the IoT Device Framework which requires an MPU to run an operating system. The ARM Cortex-M family can be used in very small end nodes of an IoT network where issues like security are not that important. So, the ARM Cortex-M family can be used for devices which abstract periphery and which are connect to the SBC or DBC via interfaces like CAN or USB. The Cortex-R family is hard real-time capable and is used for more critical applications (e.g. automotive or medical applications). For the IoT Device Framework the most suitable family is the ARM Cortex-A where the A stands for application. The A family supports operating systems like Linux, Android and Chrome. According to ARM, the family is qualified for applications such as smartphones, automotive systems, servers, wearables, tablets and robotics. Currently, the seventh version of the ARM architecture (ARMv7) is used most. The ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A9, ARM Cortex-A15 and ARM Cortex-A17 belong to that version of the ARM architecture. The ARM Cortex-A5, ARM Cortex-A7 and ARM Cortex-A9 are for low-power applications, whereas the ARM Cortex-A15 and ARM Cortex-A17 are used for applications which require higher performance [20].

3.1.2 Power Management

The power management of an SBC consists of three main parts:

- power supply voltage rails,
- power sequencing and
- reset logic.

The first task is to guarantee that the needed **power supply voltage rails** are generated with linear and switching regulators. Mostly linear regulators are implemented in the SoC already, whereas switching regulators have to be added externally. Common required power supply voltage rails are 1.2V, 1.35V, 1.8V, 2.5V and 3.3V.

The **power sequencing** is used to turn on or turn off power supply voltage rails in a specific order. As an example, some MPUs require the I/O voltage before the core voltage or the other way around, otherwise they do not behave properly and could be damaged. A further example is that the CPU has to be powered up before co-processors like a GPU [21].

The **reset logic** resets the entire system until all of the power supply voltage rails are turned up in the specific sequence. Otherwise a damage would be possible in the case that one component is already transmitting data but the receiving component is not powered on properly.

3.1.3 Memory

The memory blocks of the basic system are a Double Data Rate Synchronous Dynamic Random-Access Memory (DDR SDRAM) as a volatile memory and an Electrically Erasable Programmable Read-Only Memory (EEPROM), a NOR and a NAND flash as non-volatile memories. While starting up the system, the program code is loaded into the volatile RAM, which operates at faster speeds compared to the non-volatile memory.

Considering the **DDR SDRAM** there are several architectures: DDR2, LP-DDR2, DDR3 and DDR3L. The LP-DDR2 is the low power version of the DDR2 which is the predecessor of the DDR3 technology. Both, the DDR2 and the LP-DDR2 operate with a nominal voltage of 1.8 V. Compared to the DDR2, the LP-DDR2 has additional modes of operation if the SDRAM is not or rarely used. If the full bandwidth is used, the LP-DDR2 does not save much power compared to the DDR2. The DDR3L SDRAM is the low-voltage version of the DDR3 SDRAM. According to the JEDEC standard, the DDR3 and DDR3L are compatible to each other. For the power supply of a DDR3L SDRAM 1.35 V are used instead of 1.5 V which are used for driving a DDR3 SDRAM. The low-voltage supply comes along with less power consumption and therefore less heating of the DDR SDRAM but also less heating of the DDR SDRAM controller. A further advantage of using the DDR3L technology is that the core voltage of the processing unit is the same as the required voltage for the DDR3L SDRAM. The DDR3L technology is already commonly used by providers of SBCs. The most recent technology is the DDR4 SDRAM with a nominal voltage of 1.2 V [22], [23].

Besides the DDR3L SDRAM, non-volatile memory blocks (**NOR and NAND flash**) are used as non-volatile storage of the software. The properties of NOR and NAND flashes differ from each other. Basically, a NOR Flash has a shorter read latency. It does not need error correction codes (ECCs) because NOR Flashes have no problem with bit flipping. Since a NOR flash does not have any bad blocks, the bad block management which is necessary for NAND flashes is not needed. Compared to that, NAND flashes require ECCs, bad block management and wear-leveling to operate properly. On the one hand, this can be achieved by using a raw NAND flash with a filesystem, a device driver or a controller that guarantees the ECC, the bad block management and the wear-leveling. On the other hand, these functionalities can be guaranteed by using a managed NAND flash, which often has a high-speed MultiMediaCard (MMC) controller integrated for that task. The disadvantage of the NOR Flash is that it is more expensive than the NAND Flash for the same capacities [24]. Therefore, the NAND Flash is used to store the operating system, which has a too big memory footprint for storing it in a NOR Flash. The NOR Flash is used for the relatively small bootloader which often includes the unsorted block image (UBI) and UBI filesystem (UBIFS). Physical blocks of the memory technology device (MTD) are abstracted by logical blocks which belong to the UBI layer. The UBIFS is the filesystem which further abstracts the UBI layer [25]. The UBI layer does the wear leveling and the bad block management.

The difference between a raw NAND flash and a managed NAND flash can be seen in Figure 6. If a raw NAND is used, the common interface between the NAND controller and the NAND flash is the standardized Open NAND Flash Interface (ONFi). In contrast to that, if the NAND blocks are embedded into an e.MMC or an Solid State Drive (SSD) which are managed NAND flashes, the interface between the NAND controller and the memory is an MMC bus or a SDCard bus [26].

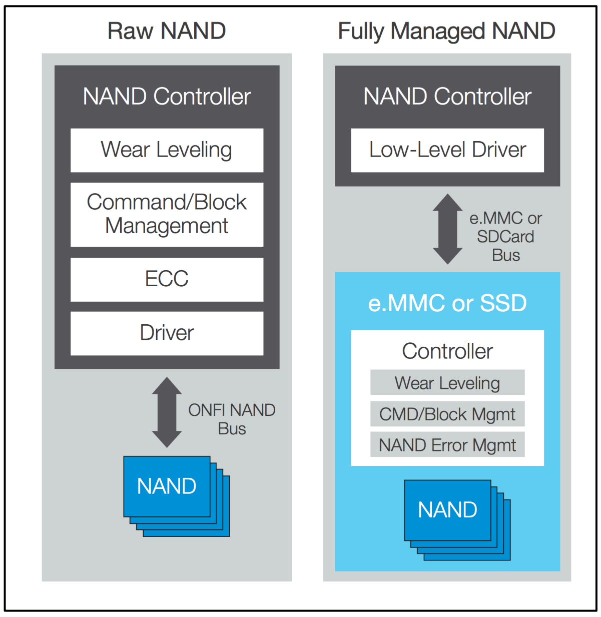

Figure 6 Comparison: Raw NAND Flash and Managed NAND Flash [24].

The **EEPROM** is used in applications where the needed amount of memory is low, so typically it is used for saving device specific information like the serial number of the device. Its advantage is that it has a very low power consumption. The most common interfaces for connecting an EEPROM are SPI, I^2C and Microwire [27].

3.2 Optional System Extensions

The basic SBC, which consists of the SoC as a computing unit, the power management and the memory, can be extended by further components considering the needs of the specific application. Extensions developed within the IoT device framework are a supervisory processing unit, a power management IC and some interfaces which can be seen in Figure 7.

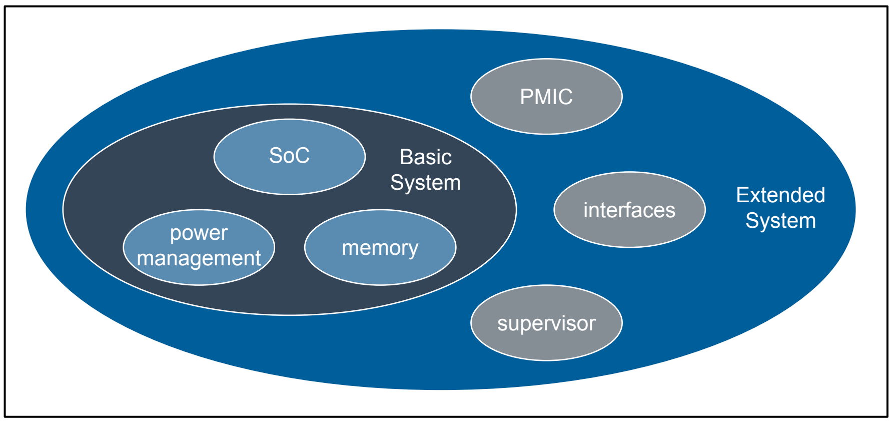

Figure 7 Components of an Extended SBC.

3.2.1 Supervisory Processing Unit

The supervisory processing unit (SPU) can be added as an option. The following tasks are done by the SPU:

- Configuration of the power management IC (PMIC),
- Power on PMIC and
- Continuous monitoring of power supply voltage rails

3.2.2 Power Management IC

Optionally, a Power Management IC (PMIC) can be added to the SBC for achieving advanced and easier power management which is described in Section 3.1.2.

The tasks of a power management IC are

- Providing power supply voltage rails,
- Power sequencing,
- Reset logic,
- Battery back-up logic,
- Manage stand-by modes and
- Power monitoring

Adding a PMIC is especially recommendable, if a lot of interfaces with different voltage rails are used. Furthermore, a PMIC makes required power sequencing easier, than without using a PMIC. A further task of the PMIC is to provide the needed reset logic and current limits can be set easily. It is advisable to use a PMIC if the SBC is not constrained in terms of space and if the needed quantity of the SBC is not high.

3.2.3 Interfaces

Interfaces can be added to the IoT device for connecting both, ICs within the system so on-board interfaces, but also for connecting other devices to the IoT device, so for example to abstract periphery. The High-Speed Inter-Chip (HSIC) interface is used for connecting ICs on-board. For the connectivity to other devices Ethernet, USB and CAN are typical interfaces that are used. For multimedia extensions, there is the possibility to add audio, display, touch panel and camera interfaces. A very important standard for wireless connections is Bluetooth.

In Figure 8 some possible interfaces are listed. The figure further shows on which level (SoC, SoM or SBC) the specific components of an interface especially the PHYs are located. A PHY or transceiver is the connection of the link layer (layer 2) and the physical layer (layer 1) of the Open Systems Interconnection (OSI) model.

As an example, the Universal Serial Bus (**USB**) PHY is mostly part of general-purpose SoCs whereas an FPGA can either integrate the PHY on-chip or it can be added externally. The integration of USB PHYs is possible, compared to Ethernet PHYs, because there is only one standardized physical layer for USB interfaces. The interface for USB 2.0 on the link layer is called USB 2.0 Tranceiver Macrocell Interface (UTMI). The UTMI standard was only defined for USB peripherals first. As an extension of the UTMI standard the UTMI+ standard was introduced later which provides the standardization for USB hosts and USB On-The-Go (USB OTG) peripherals [28]. According to an article of EE Times, another interface called UTMI+ Low Pin Interface (ULPI) was introduced in 2004. The new interface reduces the number of signals from up to 80 pins to only 12 or even only 8 pins [29]. The smaller number of signals and also the smaller package size of an USB PHY using ULPI instead of UTMI/UTMI+ makes it possible to connect a USB PHY externally (e.g. if an FPGA is used instead of a general-purpose SoC).
A further interface if a USB HUB is used, is the High-Speed Inter-Chip (HSIC) interface between the USB controller and the USB HUB. Since the PHY is typically included in the SoC,

only the connector has to be added for implementing a USB host, a USB device or a USB OTG. In the case of a USB host and a USB OTG it is recommendable to further add an overcurrent switch. Another PHY interface is the PHY Interface for PCI Express, SATA, USB, DisplayPort and Converged IO Architectures (PIPE) [30].

In contrast to the USB PHY, which is mostly integrated in the SoC, **Ethernet** PHYs have to be chosen and added as separate ICs. On one side the PHY is connected to the SoC with the media independent interface (MII). On the other side the PHY is connected to the physical medium with the medium dependent interface (MDI). For Ethernet, there are several forms of MIIs. The type of MII depends on the speed (10BASE for 10 Mbit/s, 100BASE for 100 Mbit/s, 1000BASE for 1 Gbit/s, …) whereas the type of the MDI depends on the physical medium (coaxial cable, optical fiber, twisted pair). Considering these two dimensions, the Ethernet PHY exists in several combinations of MIIs and MDIs [31].

A **CAN** interface requires a so-called CAN transceiver between the CAN controller and the physical CAN bus. The connection to the SoC is realized with a CAN_Rx and a CAN_Tx signal. The physical interface is realized with twisted-pair wires with the differential signals CAN_HIGH and CAN_LOW. The typical used connector for CAN interfaces is the D-SUB DE-9 connector.

An **audio interface** is realized using an audio codec (coder and decoder) between the controller and the connector. The interfaces Inter-Integrated Circuit (I^2C) for configuring and Inter-IC Sound (I^2S) for data transmission are used for the communication between the controller and the audio codec [32]. An alternative standard for that connection is Audio Codec ’97 (AC’97) [33].

High-definition multimedia interface (HDMI) can be used as a general multimedia interface since it transmits both, video and audio data [34]. Standard **display** interfaces are parallel single-ended signals, low-voltage differential signal (LVDS), MIPI DSI and the VESA DisplayPort [35]. A **touch panel** is typically connected via I^2C, SPI and MIPI Touch. A **camera** is connected to the controller via parallel single-ended signals, LVDS or MIPI CSI [36]. These interfaces are directly driven with the respective controller. **IOs** are typically connected with single-ended lines to a buffer. The buffer converts voltage levels and converts signal-ended signals to differential-signals if required. A **Bluetooth** chip or Bluetooth module for wireless transmission can be connected I^2C, SPI or Universal Asynchronous Receiver Transmitter (UART).

A further important aspect of the designed IoT device framework is that periphery can be abstracted easily to another PCB, due to the fact that standard interfaces like USB and CAN are implemented. This can be also seen in Figure 8 where an example USB device and an example USB composite device are connected. In bigger systems which require bigger bandwidths a PCIe is recommendable.

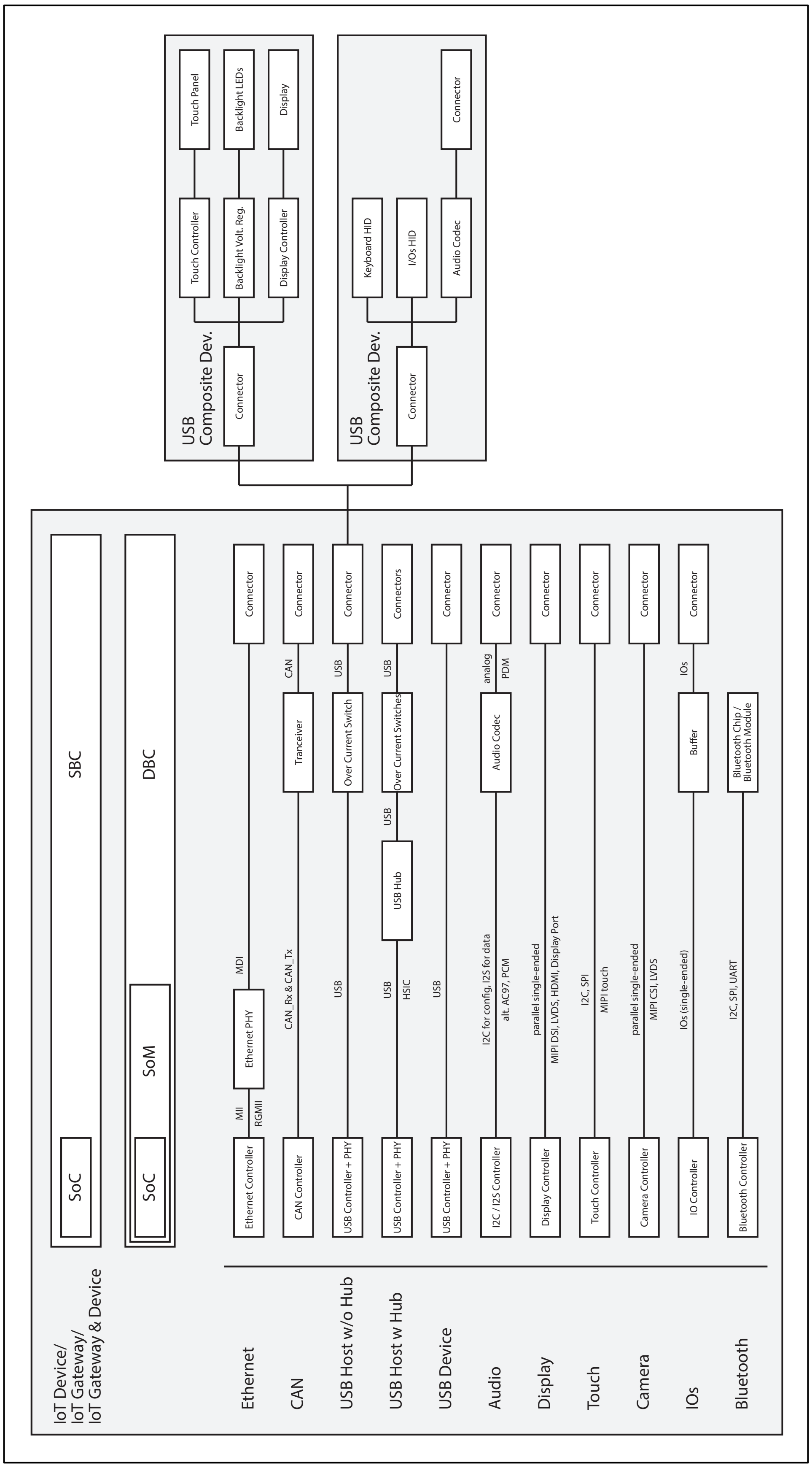

Figure 8 IoT Device Framework: Interfaces.

4 Survey on Appropriate Software Solutions

The previous Chapters 2 and 3 describe the IoT device framework regarding its hardware aspects. This chapter discusses and compares several software concepts which can be implemented for the IoT device.

The first question which has to be considered when designing the software concept is whether an **operating system** is needed or a **bare-metal** solution is enough. Further in this chapter it is discussed if the chosen operating system has to be **real-time capable** and which techniques can be used to achieve real-time capability in an operating system. Operating systems can be classified regarding the type of kernel they use. There are two main types of kernels. On the one hand, **monolithic kernels** as a single kernel approach and on the other hand **microkernels** as a dual kernel approach. Microkernels can be classified into pure micro kernels and hybrid micro kernels. It is further focused on Linux as a GPOS. The two main approaches (**PREEMPT_RT** patch and **Xenomai**) to make Linux OS real-time capable are discussed in more detail.

4.1 Comparison of Bare-Metal Programs with Using Operating Systems as an Abstraction Layer

Bare-metal programming means that the written program code is executed on the hardware, so the computing unit, directly. In contrast to a bare-metal program, a user application can be executed as a process in an operating system. Therefore, the operating system is the interface between the hardware and the user application. Its tasks are to manage the available hardware resources [37].

Having an operating system comes with a lot of additional overhead regarding memory, computing performance and power but accepting this compromise brings a lot of positive features compared to a bare-metal solution: management of tasks, time, memory, interrupts but also mechanisms for synchronization and communication. These features of an operating system lead to a software solution which is more fault-prone than a software which runs on bare-metal [38].

Using an operating system makes software development easier because a big problem can be broken down to several small problems which can be implemented using different tasks. A bare-metal solution on the other side, is just one big program. Therefore, software using an operating system is easier to develop in a team because smaller problems can be easily split up and distributed to the developers [38]. As a further advantage, software debugging is easier. These facts do not only lead to an easier and faster software development but also software maintenance afterwards.

An operating system brings the advantage that the synchronization between tasks is easier and safer. Messages are passed between tasks in a safe way whereas the synchronization in bare-metal is often realized using global variables which leads to a solution that is unsafe and prone to errors [38].

Due to the fact that the operating system abstracts the hardware, software applications are a hardware independent solution which can be ported to another hardware easily. Especially, when it comes to update the hardware of a system, using an operating system saves a lot of time [38].

Furthermore, standard drivers and standard protocols are often provided by an operating system and, compared to bare-metal programs, it is not required to use low-level languages which can be another source of errors. Depending on the operating system, security mechanisms are implemented.

Table 3 summarizes the aspects mentioned above.

	bare-metal	**operating system**
processing overhead	lower	higher
hardware portability	difficult	easy
Software development/maintenance	fault-prone and slower	easier and faster
synchronization	unsafe	safer
standard protocols and drivers	not provided	provided
memory footprint	smaller	larger
low-level programming languages	required	not required
security mechanisms	difficult to implement	already included

Table 3 Comparison of Bare-Metal with Operating System as an Abstraction Layer.

The following paragraph shows the increasing fault-tolerance when an operating system is used compared to the bare-metal program. [37] analyzes the behavior of an application after injecting soft errors (bit-flips in memory) in the embedded software which is running on an ARM Cortex-A9. On the one hand, the application is executed as bare-metal code, so directly on the hardware. On the other hand, the applications are executed within an operating system. The fault classification after injecting the soft errors is done by the simulation platform OVPSim into three classes: no error detected, the application hung and the application has finished but the data memory is corrupted (silent data corruption). The results show that the error rate which means hangs and silent data corruptions summed up, compared to injected error, is by far lower in the Linux environment than running the applications on bare-metal [37].

The high number of advantages lead to the conclusion that an operating system should be used to abstract the hardware of the IoT device. Especially, high security is required by devices which are connected to the internet because the internet is a potential point of attack of a system. Since the available memory is large enough, the larger memory footprint which comes with the operating system does not matter.

4.2 Comparison of Real-Time Operating Systems with General-Purpose Operating Systems

In the previous Section of the thesis it was concluded that an operating system should be used for internet of things applications. Operating systems which can be used as an abstraction layer for IoT devices can be divided into real-time operating systems (RTOS) and general-purpose operating systems (GPOS). This Section describes the advantages of both and states their applicability for IoT purposes. The properties of RTOS and GPOS are summarized in Table 4.

For determining whether an RTOS or a GPOS is the suitable operating system it is necessary to divide IoT devices in two classes: low-end IoT devices and high-end IoT devices. Low-End IoT devices usually run an RTOS whereas high-end IoT devices run a GPOS like Linux [39].

<u>Low-End IoT devices:</u>
Requirements for operating systems of low-end IoT devices are a small memory footprint, support for heterogeneous hardware, network connectivity, energy efficiency, real-time capabilities and security. Regarding the small memory footprint both, volatile and persistent memory have to be considered. The challenge is to lower the memory footprint but also having a good performance and a convenient API. It is important to support heterogeneous hardware which means to support different types of microcontroller/microprocessor architectures and families, different memory sizes and different communication technologies. Furthermore, it is very important to consider network connectivity, therefore supporting an IP protocol based

network stack to ensure a proper end-to-end communication over both, wireless links like IEEE 802.15.4 and Bluetooth but also wired links like Ethernet or bus systems. Besides, it is important that the timing is deterministic so the operating system should be real-time capable. This means having a maximum execution time and maximum latencies. Security can be increased if open source software is used and updates are installed on already running IoT devices [39].

The Internet Engineering Task Force (IETF) introduced the term “constrained nodes” for low-end IoT devices. According to the Internet Engineering Task Force (IETF), constrained nodes are limited regarding the maximum possible code complexity (flash memory), the size of state and buffers (RAM), the amount of computation feasible in a period of time, so the processing power, the available power, user interfaces and accessibility in deployment. Such constrained nodes are further separated into three classes [40]:

- Class 0 devices: << 10 kB data size (e.g. RAM) and << 100 kB code size (e.g. Flash),
- Class 1 devices: ~ 10 kB of data size (e.g. RAM) and ~ 100 kB code size (e.g. Flash),
- Class 2 devices: ~ 50 kB of data size (e.g. RAM) and ~ 250 kB code size (e.g. Flash).

Such low-end IoT devices do not have the capability to run a GPOS like a Linux. Therefore, it is common to choose an RTOS. Common used ones are FreeRTOS, RT-Thread, Contiki OS, RIOT and TinyOS which are open source [39].

High-End IoT devices:

Compared to RTOSs, GPOSs like Linux have the advantage that standard protocols are already integrated even on higher levels of the OSI model (layer 5, 6 and 7). Some RTOS support standard protocols only for lower layers of the OSI model (e.g. TCP/IP). Standard application programming interfaces (APIs) like the POSIX standard which are provided by GPOS make the development and adoption of applications much easier than applications that are written for RTOS. Due to the widespread usage of GPOS the technical support is good and they provide up-to-date security mechanisms which is an essential feature for devices connected to the internet. GPOS are typically processor and hardware compatible so they can be adapted to another processor or hardware platform easily. High-end IoT devices use mostly MPUs instead of MCUs. Therefore, it is required to handle the integrated memory management unit (MMU) in the MPU. An RTOS is not capable of running such an MMU, so high-end devices require a GPOS. On the contrary to that, RTOSs are more suitable for applications where an MCU with an integrated memory protection unit (MPU) is used.

The following table shows the differences between RTOS and GPOS summarized:

	RTOS	GPOS
timing determinism	provided	not provided
task scheduling	priority-based	fair
preemptive kernel	supported	not supported (only with patch as extension)
priority inversion	no	yes
memory footprint	smaller	larger
standard APIs	no	yes
network stack (standard protocols)	OSI Layer 2-4 (e.g. TCP/IP)	also higher levels (6LoWPAN)
use	low-end IoT devices	high-end IoT devices

Table 4 Comparison of RTOS with GPOS.

To conclude a GPOS should be chosen if the required resources are provided. Also, the IoT developer survey in 2016 shows that Linux is the most used operating systems for IoT devices with 73.1% [15]. This percentage increased for the year 2017 to 81.5% [41]. Due to the fact that the IoT device framework provides enough resources, a GPOS is the chosen solution. Compared to RTOS, a GPOS does not provide timing determinism of any kind typically. To overcome this problem, it is possible to add extensions to GPOSs which is described in Section 4.3.

4.3 Real-Time Extensions for General-Purpose Operating Systems

This section focuses on **Linux** as a general-purpose operating system in more detail. Due to the fact that the real-time capability is an important and required feature of the IoT device, two main approaches (**PREEMPT_RT** patch and **Xenomai**) for making a Linux **real-time** capable are compared [42].

A standard Linux distribution includes a monolithic kernel (single kernel) which is written in standard C. A big advantage of Linux compared to RTOS is its multitasking capability. A further advantage of using Linux is that standard protocols and drivers are already implemented and therefore easy to use. Since Linux is widely used, software bugs and security issues are fixed and patched fast.

Due to the fact that the standard Linux implements a fair scheduling algorithm, it is not real-time capable. An important fact about real-time systems is that they should have enough priority levels and priority inversion should not happen. The two main methods PREEMPT_RT patch and Xenomai to extend the standard non-real-time capable Linux, that it becomes real-time capable, so deterministic regarding time, are compared. These different methods to make the standard Linux real-time capable can be compared regarding their latencies, jitters, context switching time, preemption time, priority functionality, deadlock breaktime and semaphore shuffle breaktime [43], [44]. The limits for the speed of a real-time operating system are set by the hardware so the ARM architecture in this case.

Using a monolithic kernel means that the Linux has just a single kernel. This can be achieved by adding the PREEMPT_RT patch which changes the scheduling strategy from a non-preemptive one as used in the standard Linux to a preemptive one. This has the big advantage that real-time critical parts of the code do not have to be compiled as kernel modules. They can be run in user space in real-time [42]. Furthermore, the PREEMPT_RT patch implements a high resolution timer.

Besides, operating with a single kernel a second kernel, a so-called microkernel, can be added. In the case of Linux, one implementation of the microkernel approach is Xenomai (Adeos) [44]. It can be added as a patch to the standard Linux. Xenomai uses a real-time co-kernel and assigns real-time tasks to this second kernel. Another very similar solution which is called Real Time Application Interface (RTAI) exists. Comparing these two solutions, RTAI is more focused on very low latencies whereas Xenomai considers properties like maintainability and portability [45].

Besides PREEMPT_RT patch and Xenomai (co-kernel approach), real-time capability can also be achieved by using a real-time co-processor. Therefore, real-time tasks are assigned to this high priority co-processor.

Comparing the PREEMPT_RT patch with Xenomai leads to the conclusion that the worst latency is the same for both solutions but Xenomai leads to a lower jitter in the latency. Since the worst latency is the important parameter when real-time capability is evaluated, both solutions are almost equal regarding real-time capability. The big advantage of the PREEMPT_RT patch compared to Xenomai is that it is integrated into the Mainline Linux and it is much simpler to use [42].

In [46] the latencies and jitters of a Linux with the PREEMPT_RT patch and the Linux with Xenomai are compared. For a userspace task the worst latency for both solutions is about

95 μs but the jitter is much lower with the Xenomai solution. If kernel-tasks are compared Xenomai has worst latency of about 30 μs whereas the PREEMPT_RT patch has worst latency of about 90 μs.

A further benchmarking and analysis of different operating system is described in [44]. The metrics within this study are interrupt latency, task switching time, preemption time and deadlock break time. Compared to a real-time operating system in this case embedded configurable operating system (eCos), the RT patch and Xenomai have higher interrupt latency because eCos handles interrupts in a better way. Furthermore, also the deadlock breaktime is low, compared to the real time (RT) patch and Xenomai. In contrast to that, both, the RT patch and Xenomai, are better in task switching because Linux is designed for handling multiple tasks whereas eCos is designed for handle a single task. A further aspect where the RT patch and Xenomai can score is the preemption time [43], [44].

5 Latency and Data Rates of Interfaces

The latencies and data rates of used interfaces have to be considered to ensure that the maximum time (deadline) defined for the **real-time** system is not exceeded.

The latency for periphery is the sum of the following timing parameters [47]:

- Operating system processing delay (host),
- Interface transfer latency,
- Processing latency (periphery).

In a typical case, the three parameters listed above have to be taken into account twice since a request is done by the host and the periphery responds to the request. Regarding the operating system processing delay it has to be considered whether the requesting thread is a kernel space or user space thread. The processing latency of the periphery is either the operating system processing delay or the interrupt latency if the software is realized in bare-metal. The overall latency represents the time deadline for the real-time system.

Table 6 shows the data rate of SPI, USB, Ethernet, CAN and PCIe which are common interfaces for abstracting peripherals. In general, the latency can be calculated by using the specific payload size and dividing it by the data rate (Equation (21)) as this is the time needed to fill the buffer before the next frame is sent. In contrast to SPI, Ethernet, CAN and PCIe the latency of USB is specified in the standard.

$$latency = \frac{frame\ size}{data\ rate} \tag{21}$$

For example the **SPI** interface can be used with different data rates. Assuming a data rate of 20 Mbit/s with a payload size of 32 bit leads to a latency of 1.6 µs as it is calculated in Equation (22). SPI can also be used with other data rates.

$$latency_{SPI} = \frac{frame\ size_{SPI}}{data\ rate_{SPI}} = \frac{32\ bit}{20\ \frac{Mbit}{s}} = 1.6\ \mu s \tag{22}$$

The latencies for the **USB** interface are specified in the standard which also takes the maximum number of USB hubs into account. It is specified with 1000 µs for full-speed USB and with 125 µs for high-speed and SuperSpeed. The maximal cable length for full-speed and high-speed USB is 5 m whereas the maximum cable length for SuperSpeed USB is 3 m.

The **Ethernet** standard specifies the maximum transmission unit (MTU) with 1500 bytes which leads to a frame of 1536 bytes. This is 12288 in bits. Assuming a data rate of 1 Gbit/s the latency for a frame with maximum payload is 12.29 µs (Equation (23)). With 10 Gbit/s the latency is 1.23 µs (Equation (24)). This latency can be guaranteed if no collision occurs.

$$latency_{Ethernet,1Gb/s} = \frac{frame\ size_{Ethernet}}{data\ rate_{Ethernet,1Gb/s}} = \frac{12288\ bit}{1000\ \frac{Mbit}{s}} = 12.29\ \mu s \tag{23}$$

$$latency_{Ethernet,10Gb/s} = \frac{frame\ size_{Ethernet}}{data\ rate_{Ethernet,10Gb/s}} = \frac{12288\ bit}{10000\ \frac{Mbit}{s}} = 1.23\ \mu s \tag{24}$$

According to the **CAN** standard, the fastest specified data rate is 1 Mbit/s. A standard CAN frame with a frame ID of 11 bits has an overall length of 111 bits so the latency is calculated with 111 µs (Equation (25)). Using only half of the data rate, leads to a latency of 222 µs (Equation (26)). Since a frame with a lower frame ID has a higher priority this latency cannot be guaranteed in general. The latencies stated in Table 6 for CAN are ensured if a point to point connection is used instead of a bus with several nodes. The maximum cable length for a 1 Mbit/s connection is 40 m whereas the maximum cable length for a 500 kbit per second connection is 100 m.

$$latency_{CAN,1Mb/s} = \frac{frame\ size_{CAN}}{data\ rate_{CAN,1Mb/s}} = \frac{111\ bit}{1\ \frac{Mbit}{s}} = 111\ \mu s \tag{25}$$

$$latency_{CAN,500kb/s} = \frac{frame\ size_{CAN}}{data\ rate_{CAN,500kb/s}} = \frac{111\ bit}{500\ \frac{kbit}{s}} = 222\ \mu s \tag{26}$$

PCIe can be classified in five generations at the moment where fifth generation is still under development. The higher the generation, the higher is the data rate. Nowadays, generation 2 or 3 are commonly used for embedded applications. Furthermore, a PCIe connection can use either 1, 4, 8 or 16 lanes. Therefore the speed of each generation can be multiplied by the specific number of lanes. Table 5 states the data rates for different generation and number of lane combinations in MB/s (line codes 8b/10b for generation 1 and 2 and 128b/130b for generation 3 and 4 are already considered). Therefore, the number stated in that table has to be multiplied by 8 bits/byte to get the data rate in Mbit/s for the calculations of Table 6. The latency of three different PCIe modes (Gen1 x1, Gen3 x8, Gen4 x16) have been calculated as examples (Equations (27), (28) and (29)).

$$latency_{PCIe,Gen1\ x1} = \frac{frame\ size_{PCIe}}{data\ rate_{PCIe,Gen1\ x1}} = \frac{4096\ bit}{2000\ \frac{kbit}{s}} = 2.05\ \mu s \tag{27}$$

$$latency_{PCIe,Gen3\ x8} = \frac{frame\ size_{PCIe}}{data\ rate_{PCIe,Gen3\ x8}} = \frac{4096\ bit}{63040\ \frac{kbit}{s}} = 0.065\ \mu s \tag{28}$$

$$latency_{PCIe,Gen4\ x16} = \frac{frame\ size_{PCIe}}{data\ rate_{PCIe,Gen4\ x16}} = \frac{4096\ bit}{252000\ \frac{kbit}{s}} = 0.016\ \mu s \tag{29}$$

PCIe generation	x1 in MB/s	x4 in MB/s	x8 in MB/s	x16 in MB/s
Gen1	250	1000	2000	4000
Gen2	500	2000	4000	8000
Gen3	984.6	3938	7877	15754
Gen4	1969	7877	15754	31508
Gen5	3900	15800	31500	63000

Table 5 Data Rates of Different PCIe Generations.

Interface	Variant	Data Rate in Mbit/s	Maximum Payload Size in bit	Latency in µs
SPI	20Mbit/s, 32bit	20	32	1.6
USB	Full Speed (USB 1.1)	12 [48]	512	1000
	High Speed (USB 2.0)	480 [48]	8192	125
	SuperSpeed (USB 3.0)	5 000 [49]	8192	125
Ethernet	1Gb	1 000 [50]	12000	12
	10Gb	10 000 [50]	12000	1.2
CAN	1Mb	1	131	131
	500kb	0.5	131	262
PCIe	Gen1 x1	2000	4096	2.05
	Gen3 x8	63040	4096	0.065
	Gen4 x16	252000	4096	0.016

Table 6 Latency and Data Rate of Interfaces for Abstracted Peripherals.

6 Features of the IoT Device Framework

In this chapter the features of the developed IoT device framework are described and listed. Section 6.1 shows a block diagram with a rough overview of the hardware system. Section 6.2 describes the hardware in more detail. Relevant aspects such as the physical dimensions, the power consumption and the used components of the device are described. A possible software solution is evaluated in Section 6.3 and also the board bring up process which has to be done within the production process is described in Section 6.4. Figure 9 shows the IoT device framework with all modules of the schematics design included from a side view. As described in Chapter 3 not all of the schematic design modules have to be used to develop a new IoT device. Therefore, the PCB can be adapted to any specifications of a new IoT product. It is scalable regarding many parameters. The physical dimensions can be changed easily and also the power consumption can be reduced essentially.

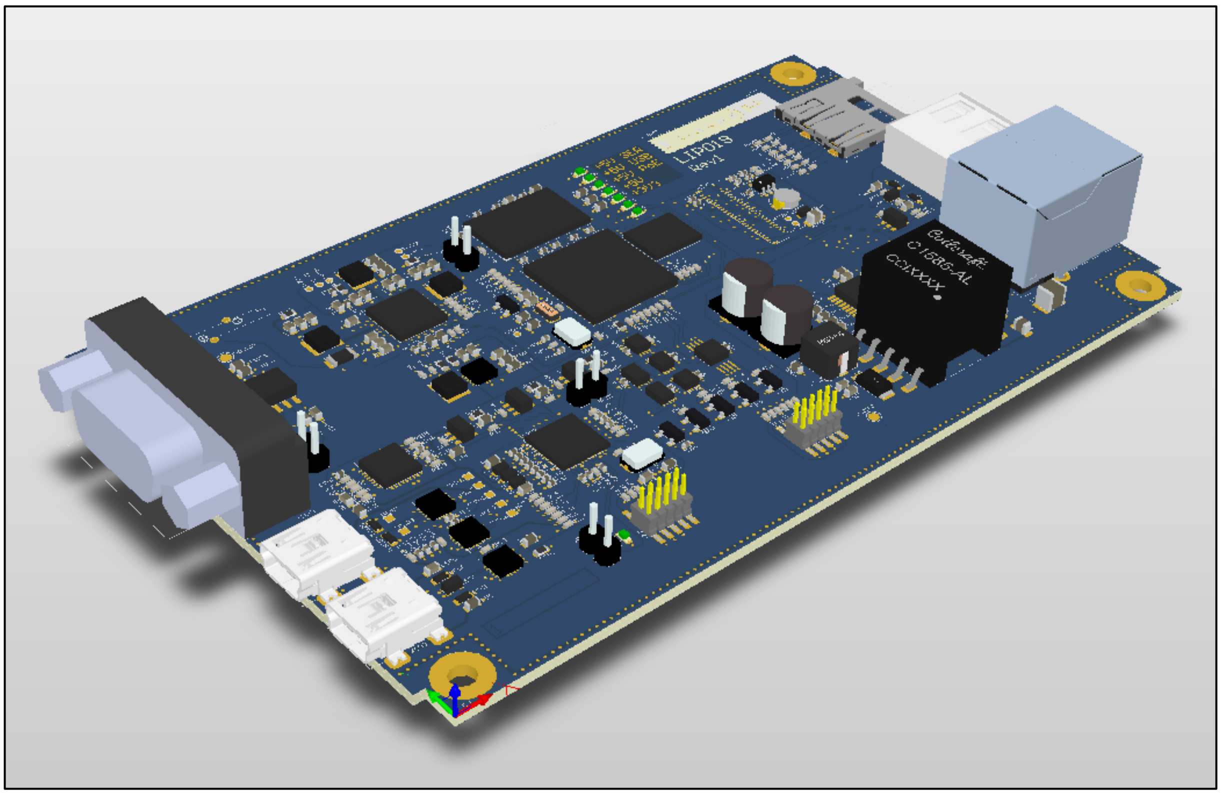

Figure 9 Developed IoT Device Framework.

6.1 Block Diagram

Figure 10 shows the block diagram of the developed IoT device framework.

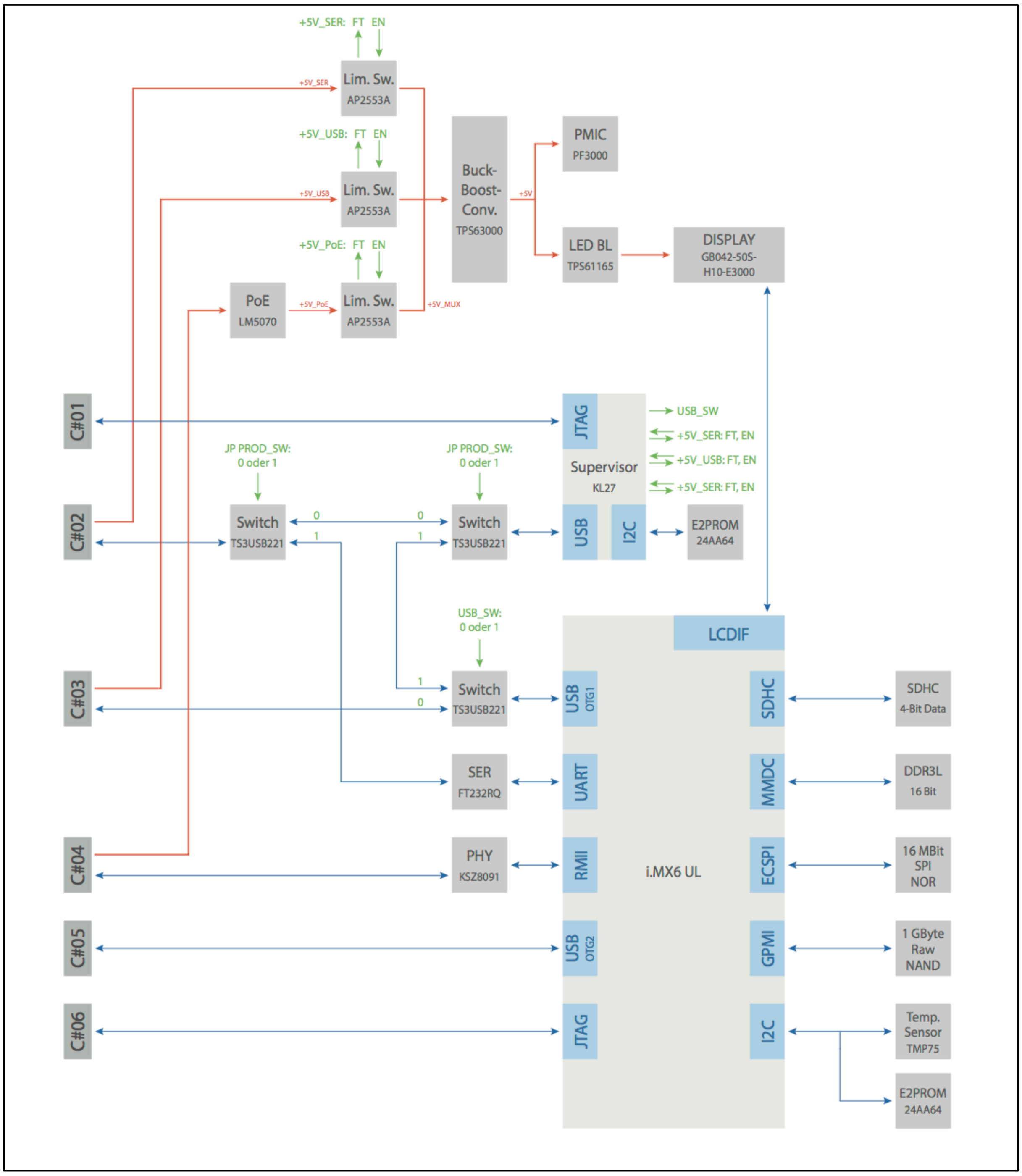

Figure 10 Block Diagram of Developed IoT Device Framework.

6.2 Hardware

The PCB, with all the components of the developed IoT device framework included can be seen in Figure 11 and Figure 12 where Figure 11 shows the top view and Figure 12 shows the bottom view of the PCB.

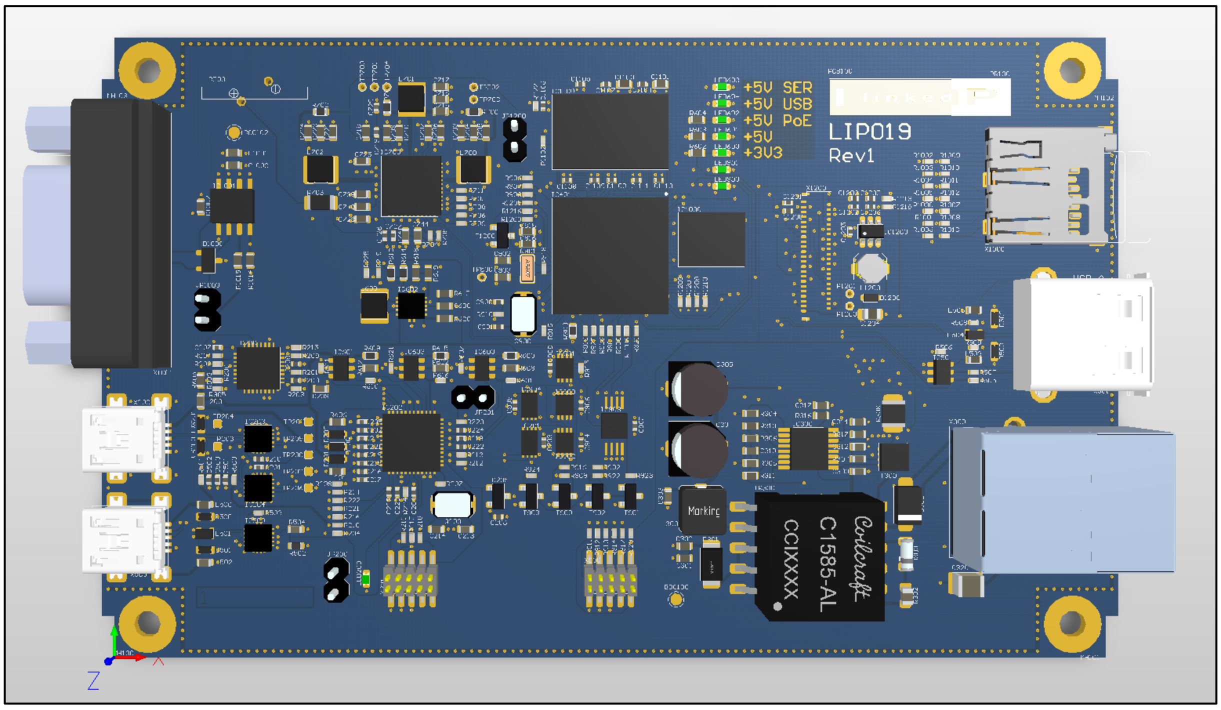

Figure 11 IoT Device Framework: PCB Top View.

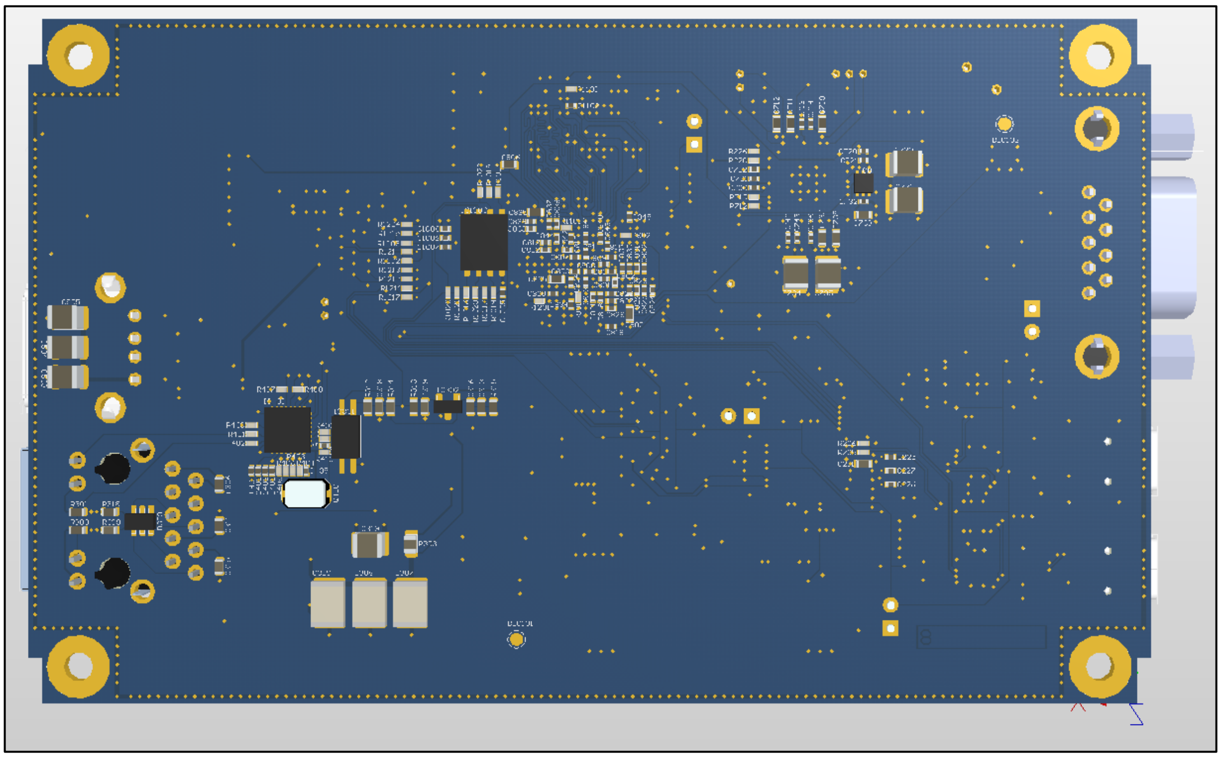

Figure 12 IoT Device Framework: PCB Bottom View.

6.2.1 Overview

When the hardware of an IoT device is developed, several aspects should be taken into account: physical dimensions, power consumption, data storage and processing, connectivity, and cost [51].

Including the basic system and all of the optional extensions, the **physical dimensions** of the PCB are 120 mm x 75 mm. The highest component on the top side is the RJ45 connector with a height of 13.95 mm and on the bottom side the multi-layer ceramic chip (MLCC) capacitors with a height of about 2 mm.

The used **components** for the IoT device framework are:

- SoC: NXP i.MX 6UL G2, ARM Cortex-A7, 528 MHz
- Power supply:
  - Power over Ethernet, 13 W
  - USB Mini (used as an USB Device), 2.5 W
  - USB Mini (used for the console), 2.5 W
- PMIC: NXP PF3000
- Memory:
  - DRAM: 256 M x 16 (4 GB)
  - NOR Flash: 2 M x 8 (16 MB)
  - NAND Flash: 128 M x 8 (1 GB)
- Supervisory processing unit: NXP KL27
- Real Time Clock (RTC)
- Interfaces:
  - Micro SD card
  - USB host
  - USB device
  - Ethernet incl. PoE
  - CAN interface
  - USB to serial
  - Display incl. touch panel

The **power consumption** is another important factor when a device is evaluated. It depends on many aspects like the mode of operation of the SoC, the used memory, the used interfaces and the mode (speed) of memory operations and data rates of the used interfaces.

The board can be supplied by one of the two Mini-USB connectors or by the PoE power adapter. The USB connectors provide 2.5 W and the PoE power adapter is designed to provide 13 W.

The calculation of the power consumption for full operation of the device can be seen in Table 7 where the power consumption of each component is the product of the voltage and the drawn current. Since the DC/DC converters between the input voltage and the supply voltage of the components have constrained efficiencies the calculated power consumption has to be divided by the efficiency of the DC/DC converter to get the actual power consumption. The efficiency of the converters in the PMIC and the TPS63000 have to be considered and in the case of a supply by PoE the efficiency of the PoE controller has to be taken into account as well. For calculating the power consumption of the display the efficiency of the led driver TPS61165 for the backlight has to be considered.

The calculation of the power consumption was separated into the power consumption of the basic system and the power consumption of the optional extensions as it can be seen in Table 7. The i.MX6UL, the DDR3L SDRAM, the NAND flash and the Ethernet PHY are the essential components for calculating the power consumption of the basic system (green boxes in Figure 13). The NOR flash is not considered in the calculation since it has a negligible power consumption and since it is only in operation at the boot process when all the other components are turned off or are not in full operation. The most important optional extensions for calculating the power consumption are the USB host, the micro SDCard, the touch panel, the CAN interface, the KL27 and the display.

The power consumption of the **i.MX6UL** core is about 675 mW since it is supplied by 1.35 V and the maximal supply current is 500 mA. For the high level of the i.MX6UL 3.3 V are used and the maximum current rating is 125 mA. Therefore the power consumption is 412.5 mW [52].

The **DDR3L SDRAM** is supplied by the 1.35 V power supply voltage rail. According to the document “i.MX 6UltraLite Power Consumption Measurement” the DDR3L SDRAM draws a current of about 100 mA for the case of a video playback. This leads to a power consumption of 135 mW.

In the datasheet of the **NAND flash** the maximal operating current for programming, reading and erasing is 30 mA. As the supply voltage is 3.3 V, the power consumption of the NAND flash is 99 mW.

The power consumption of the **Ethernet PHY** depends on the mode of operation. First it depends on the current which is drawn by the transceiver and the digital I/Os. These values are given for different nominal operating voltages (VDDA and VDDIO). In that case VDDA and VDDIO are both 3.3 V. Furthermore, the power consumption depends on the transfer mode of the interface. In the example, the highest possible power consumption has been taken from the datasheet. The maximum power consumption is achieved with a 100BASE-TX Full-duplex connection with 100 % utilization. The current consumption by the transceiver itself is 34 mA and the current consumption of the digital I/Os is 13 mA. Since both are supplied by 3.3 V, the two currents are adding up to 47 mA. The power consumption for that mode is 155.1 mW.

The **USB host** can provide up to 500 mA at a voltage level of 5 V. Therefore, the power consumption can achieve a level of 2500 mW.

The nominal voltage of the **CAN** interface is 5 V. According to the CAN transceiver datasheet, a typical current of 50 mA and a maximal current of 70 mA is drawn in normal mode when the bus is dominant. This leads to a power consumption of maximal 350 mA.

The **microSD** interface uses 3.3 V as a nominal voltage. The needed current depends on the speed mode. For the standard mode with 25 MHz the current during read or write operations is 100 mA. Considering high performance mode with 50 MHz this leads to a current of 200 mA which is a power consumption of 660 mW.

The **touch panel** is supplied by 3.3 V and draws a current of 12 mA. This leads to a power consumption of 39.6 mW. The **backlight** of the LCD display uses a nominal voltage of 26.5 V at 20 mA which means that the power consumption is 512 mW. The **LCD display** itself is supplied by 3.3 V. Assuming that it draws a current of about 30 mA, leads to a power consumption of about 99 mW.

In contrast to the components mentioned before, which are part of the i.MX6UL, the **SPU** has to be taken into account. The SPU is supplied by the 5 V power supply voltage and draws a current of about 50 mA. The power consumption is therefore about 250 mW.

The power consumption calculated for the different components has to be divided by the efficiency of the specific DC/DC converter to get the overall power consumption for the component. Figure 13 shows the hierarchy of the DC/DC converters of the PCB. The TPS63000 provides the stable 5 V and has a efficiency of about 90%. The TPS61165 for backlight of the display has an efficiency of about 85% for an output voltage of 24 V, an input voltage of 5 V and a current of 20 mA. The power consumption of the 5 V/4.2 V converter of the PMIC is the product of the voltage drop of 0.8 V and the output current of about 1.81A (0.81 A for the SW3 regulator with 1.35 V and 1 A as the maximum output current for the SW1A regulator). Therefore, the power consumption is 1.448 W (Equation (30)). The SW1A and SW3 regulators have an efficiency of about 85 %.

$$\begin{aligned} P_{5V/4.2V} &= U_{drop} \cdot I_{out} = (U_{in} - U_{out}) \cdot (I_{SW1A} + I_{SW3}) = (5\,V - 4.2\,V) \cdot (1\,A + 0.81\,A) \\ &= 1.448\,W \end{aligned} \tag{30}$$

Component	Voltage in V	Current in mA	Power of SoM in mW	Power of Periphery in mW
i.MX6UL Core (VDD_SOC_IN)	1.35	500	675	
DDR3L SDRAM (DRAM)	1.35	100	135	
Total 1.35 V			**810**	
VDD_HIGH_IN	3.3	125	412.5	
NAND	3.3	30	99	
Ethernet PHY (ENET1)	3.3	47	155.1	
microSD (SD1)	3.3	200		660
Touch Panel (UART4)	3.3	12		39.6
display	3.3	30		99
Total 3.3 V			**666.6**	**798.6**
USB Host (USB_OTG_2)	5	500		2500
CAN (UART2)	5	70		350
KL27 (SPU)	5	50		250
Total 5 V				**3100**
Backlight of display	26.5	20		512
Total 26.5 V				**512**

Table 7 Power Consumption without Efficiency of DC/DC converters.

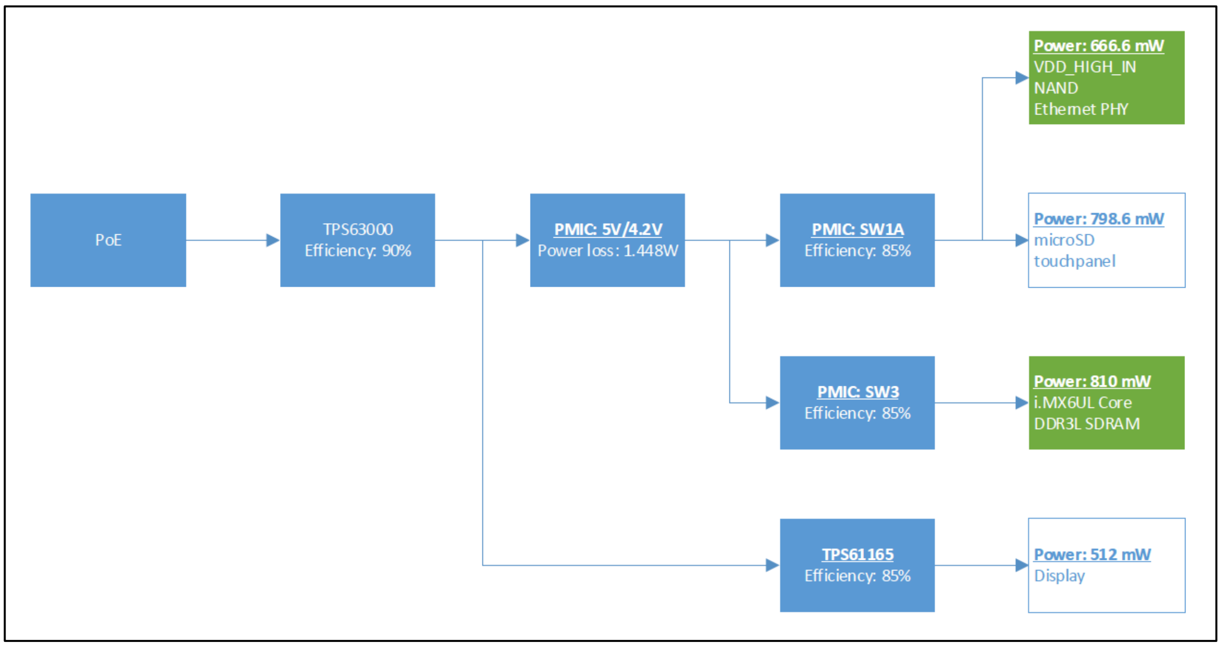

Figure 13 Efficiencies of Regulators.

Calculation:

$$P_{basic\ system} = \frac{\frac{666.6\ mW}{0.85} + \frac{810\ mW}{0.85} + 1448\ mW}{0.90} = 3539.08\ mW \tag{31}$$

$$P_{optional\ extensions} = \frac{\frac{798.6\ mW}{0.85} + 1448\ mW + \frac{512\ mW}{0.85}}{0.90} = 3322.09\ mW \tag{32}$$

$$P_{total} = \frac{P_{basic\ system} + P_{optional\ extensions}}{\eta_{PoE,controller}} = \frac{3539.08\ mW + 3192.68\ mW}{0.84} = 8168\ mW \tag{33}$$

P_{total} (Equation (33)) represents the total power consumption of the device. In case of a PoE supply this total power consumption has to be divided by the efficiency of the PoE controller. The evaluation board described in [53] shows an efficiency of 84%. This efficiency is taken for the calculation (Equation (33)). To be sure another 10 % can be added to this result so the power consumption of the device supplied by USB can be approximated with 9 W.

If the PCB is supplied by one of the two mini USB connectors not all of the periphery can be used since USB only provides 2.5 W.

6.2.2 SoC

As described in Section 3.1.1, the ARM architecture for the SoC is most suitable for IoT device applications. The ARM architecture is used by NXP for different embodiment variants. The i.MX6 series of NXP consists of eleven families. The last added families are the i.MX6UL and the i.MX6ULL. The trend is to scale-down the processors with the aim to make the development of very small IoT devices possible. As it can be seen in the table [54] even the smallest variants support security mechanisms, which is very important for devices in the internet [54].

Considering the stated aspects, the i.MX6UL was chosen for designing the IoT Device Framework. The i.MX6UL is based on an ARM Cortex-A7 core which is suitable for low-power devices. The ARM Cortex-A7 has a higher performance compared to the similar ARM Cortex-A5. According to NXP, is the i.MX6UL providing two important aspects of an IoT device: low-power operability and security. It can even run Linux. Furthermore, a big advantage which comes with using the i.MX6UL is that the IoT Device Framework is a solution which is scalable regarding the processing power because it is compatible with the entire i.MX MPU family [55]. Furthermore, the IoT Device Framework can be easily modified to use the i.MX6ULL due to its compatibility.

Four different device options of the i.MX6UL are provided by NXP. The IoT Device Framework uses the MCIMX6G2 which has compared to the MCIMX6G0 and the MCIMX6G1 a 24-bit parallel camera interface (CSI) and a 24-bit parallel display interface (LCD). Furthermore, it has two CAN interfaces, two Ethernet interfaces and two ADC converters with 10 channels. Therefore, the IoT Device Framework can be scaled down using the device options MCIMX6G0 or MCIMX6G1. There is the device option MCIMX6G3 which has more security features than the MCIMX6G2 option.

The i.MX6UL with ARM Cortex-A7 has 528 MHz with a factor of 1.9 DMIPS/MHz which equals to 1322 DMIPS. An operating system requires at least 300 to 400 DMIPS.

6.2.3 Power Management

There are three options to power the IoT device. The IoT device can be powered by using Power-over-Ethernet (PoE), by using the USB Device Connector or the USB connector which is used for the serial interface.

PoE uses the Ethernet cabling to get both, power supply and data. The provided power by PoE is 13 W according to the standard IEEE 802.3 af whereas the provided power when USB is used as a USB device is 2.5 W.

Power Supply Voltage Rails

The i.MX6UL should be supplied by a voltage between 1.275 V and 1.5 V and the DDR3L requires a voltage of 1.35 V. Due to the fact, that the power supply can be within a given tolerance both, the i.MX6UL and the DDR3L are supplied with the same voltage of 1.35 V. Furthermore, a voltage of 3.3 V have to be provided to the i.MX6UL for having basic functionalities. If other interfaces are used, other voltage rails have to be provided to the i.MX6UL as well.

Table 8 lists typical power supply voltage rails for different components of the SBC.

SBC component	**Power supply voltage rail**
SoC	1.35 V
DDR3L	1.35 V
GPIO, SDCard, Ethernet, touch panel, NAND flash, NOR flash	3.3 V

Table 8 Power Supply Voltage Rails.

Power Sequencing

As it is described in the datasheet of the i.MX6UL the power-up sequence, the power-down sequence and the steady state guidelines have to be considered while designing the system. Otherwise the current could be too high during power-up, the device is not booting or the processing unit can be even damaged in the worst case.

The power-up sequence is:

- The VDD_SNVS_IN supply must be available or connected to VDD_HIGH_IN before all the other power supplies,

- VDD_HIGH_IN must be available before VDD_SOC_IN and
- Further requirements: POR_B must be asserted at power-up until the last power level is turned on, guarantee that there is no back voltage from a supply to the 3.3 V voltage rail, no power-on restriction for USB_OTG1_VBUS and USB_OTG2_VBUS.

The power-down sequence is:

- VDD_SNVS_IN must be turned off after all the other voltage rails.

Further, an IO pin should not be driven when the power supply voltage rail for that pin is not available. In the datasheet it is stated that this could lead to an internal latch-up and malfunctions.

Reset Logic

Besides the power supply rails and the power sequencing, the reset logic is an important part of the power management.

The PMIC is first configured over I^2C by the SPU. When the configuration is finished the PWRON signal from the SPU to the PMIC is set. If the PWRON signal is set and the input voltage VIN is higher than the threshold UVDET the PF3000 goes into the ON mode. The power supply voltage rails are turned on and the power sequencing is done, as it is set in the configuration. If the power sequencing was done properly the PMIC sets the RESETBMCU to HIGH since it is an active LOW signal. If a fault occurs the signal RESETBMCU is set to LOW. The reset signal which is an open drain signal is furthermore connected to the JTAG connector.

6.2.4 Memory

<u>DDR3L SDRAM:</u>

The i.MX6UL supports the Low Power DDR2 (LP-DDR2), DDR3 and DDR3L versions of DDR SDRAMs with 16-bit. The DDR3L SDRAM is chosen for the IoT device, because it is the newest version provided by the i.MX6UL and it has a low power consumption.

<u>NOR Flash and NAND flash:</u>

The partitions of the NOR flash and NAND flash are shown in Figure 14. The NOR flash has a size of 1 MB where 768 kB are used for storing u-boot, 64 kB for storing the u-boot environment and 192 kB of unused memory which can be used for application specific data. The NAND flash has a size of 1 GB and consists of two equal sized UBI partitions with 512 MB. Each partition holds a Linux consisting of the Linux kernel, the root file system and the device tree.

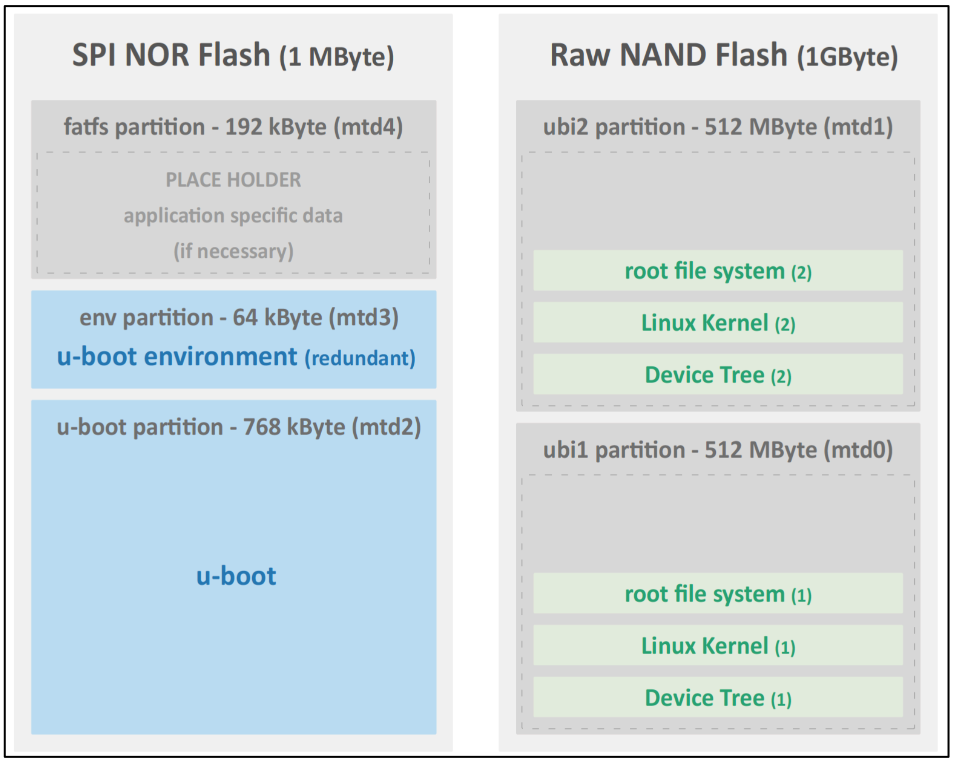
SPI NOR Flash (1 MByte)
fatfs partition - 192 kByte (mtd4)
PLACE HOLDER
application specific data
(if necessary)
env partition - 64 kByte (mtd3)
u-boot environment (redundant)
u-boot partition - 768 kByte (mtd2)
u-boot
Raw NAND Flash (1GByte)
ubi2 partition - 512 MByte (mtd1)
root file system (2)
Linux Kernel (2)
Device Tree (2)
ubi1 partition - 512 MByte (mtd0)
root file system (1)
Linux Kernel (1)
Device Tree (1)

Figure 14 Partitions of NOR Flash and NAND Flash.

<u>EEPROM:</u>

Device specific information like the serial number is stored in the EEPROM as described further in Chapter 9. The memory size is 64 kbit.

6.2.5 Supervisory Processing Unit

The IoT Device Framework uses an NXP KL27 controller as a supervisor. The Kinetis KL27 Microcontroller uses an ARM Cortex-M0 core. The supervisor is used for loading the bootloader into the NOR Flash and the Linux into the NAND Flash. Furthermore, several voltages can be monitored and in case of an error the system can be shut down. Besides, hardware can be controlled before starting the SoC.

6.2.6 Power Management IC

The PF3000 which is recommended to use as a PMIC for the i.MX6UL by NXP is used. The PF3000 contains four buck regulators, a boost regulator and six linear regulators. It is controlled via I2C and includes a one-time programmable memory for the configuration of the PMIC. The configuration includes the start-up sequence, start-up timing and the selection of the output voltage, frequency and soft start. The PF3000 includes a coin cell charger and provides the reference voltage for the DDR SDRAM.

6.2.7 Interfaces

The provided interfaces by the configured IoT device are:

- USB Host,
- USB Device,
- Ethernet,
- Parallel single-ended display interface,
- CAN and
- microSD.

6.3 Software

As a bootloader u-boot is recommendable. It is the most used bootloader. Its task is to initialize the hardware.

Linux includes the kernel, the device tree and the root file system. The kernel versions 4.1 and 4.4 as long-term support (LTS) mainline kernels are used. The device tree is compiled by the device tree compiler (DTC) and is forwarded as a parameter at the start of the kernel. The driver reads the data. The root file system is based on Debian.

6.4 Board Bring-Up

The commissioning consists of several steps. Firstly, the supervisor firmware is loaded over the serial wire debug (SWD) interface which uses two signals (SWDIO and SWCLK). It represents an alternative to JTAG [56]. Afterwards the bootloader u-boot is loaded into the serial NOR flash over USB and the supervisory processing unit. Then the commissioning Linux is transferred via the Ethernet interface to the DDR3L SDRAM. Therefore, the Trvial File Transfer Protocol (TFTP) protocol is used. After loading the soft- and firmware into the memories the commissioning Linux is booted with initrd which then loads the production Linux from the TFTP server into the RAM first and into the NAND flash afterwards. The system has to be rebooted to set the environment variables in the u-boot. It has to be set that the production Linux is booted from the NAND flash by default. Restarting the system again, the Linux is booted from the NAND flash.

7 Challenges in Schematics Design of the IoT device framework

The challenges in the schematics design of the IoT device framework were connecting memory and interfaces but also finding solutions for the power supply logic and for switching between different modes of operations (e.g. production mode and normal mode of operation).

7.1 Multiplexing Power Supply Alternatives

One challenge was the power supply. Since the board can be supplied by three different variants, a method for multiplexing these variants has to be designed. The board can either be supplied by the RJ45 connector so PoE, by the USB device connector, or by the USB connector which is used for the console.

For multiplexing, a current-limited power switch (AP2553A) shown in Figure 15 is used for each of the three power supply possibilities. Each current-limited power switch has an enable signal (+5V_PoE_EN, +5V_USB_EN, +5V_SER_EN) which is controlled by the SPU. So, only one of the three power supplies is forwarded to the +5V_MUX signal. The current-limited power switches signal the SPU in case of a fault with the specific signals (+5V_PoE_FT, +5V_USB_FT, +5V_SER_FT). The currents are limited with the externally added resistors to the ILIM pin of the switch. In the case of PoE this resistor has a value of 10 kΩ, which limits the current to about 2.1 A. If the +5V_USB or the +5V_SER are used, the current is limited to about 510 mA with a 39 kΩ resistor.

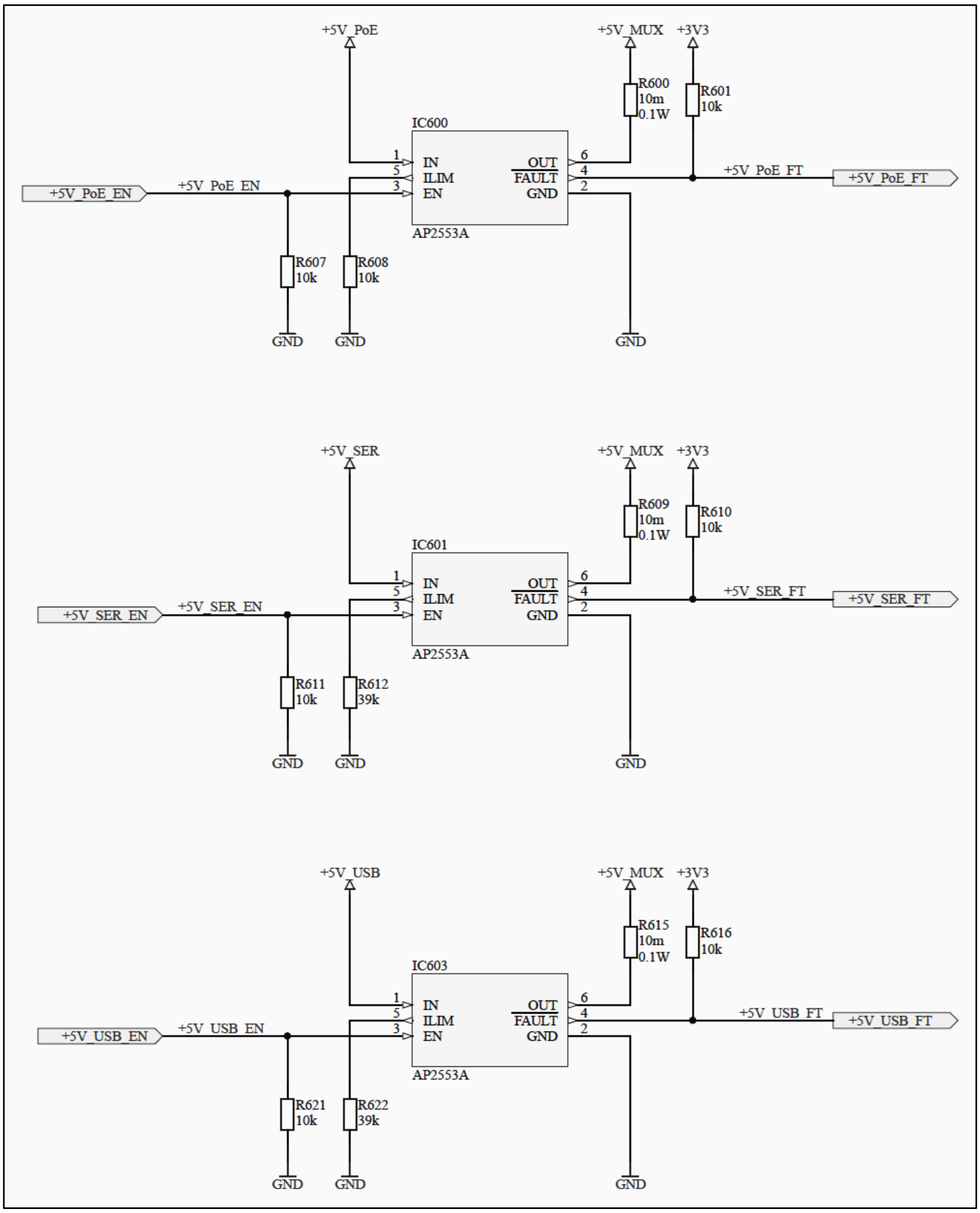

Figure 15 Current-Limited Power Switches for Power Supply Alternatives.

The +5V_MUX signal is connected to a buck-boost converter (TPS63000) which has the +5V power supply as an output. This is shown in Figure 16. The values for the external circuitry of the buck-boost-converter have been selected according to the data sheet. The values for R618 and R619 have been calculated with the following formula [57]:

$$\frac{R_{618}}{R_{619}} = \left(\frac{U_{out}}{U_{FB}} - 1\right) \tag{34}$$

Since the desired output voltage is 5 V and the feedback voltage is 500 mV the ratio between the two resistor values should be 9 as shown in Equation (35).

$$\frac{R_{618}}{R_{619}} = \left(\frac{U_{out}}{U_{FB}} - 1\right) = \frac{5\ V}{500\ mV} - 1 = 9 \tag{35}$$

The value for R618 is therefore chosen with 27 kΩ and the value for R619 is chosen with 3 kΩ.

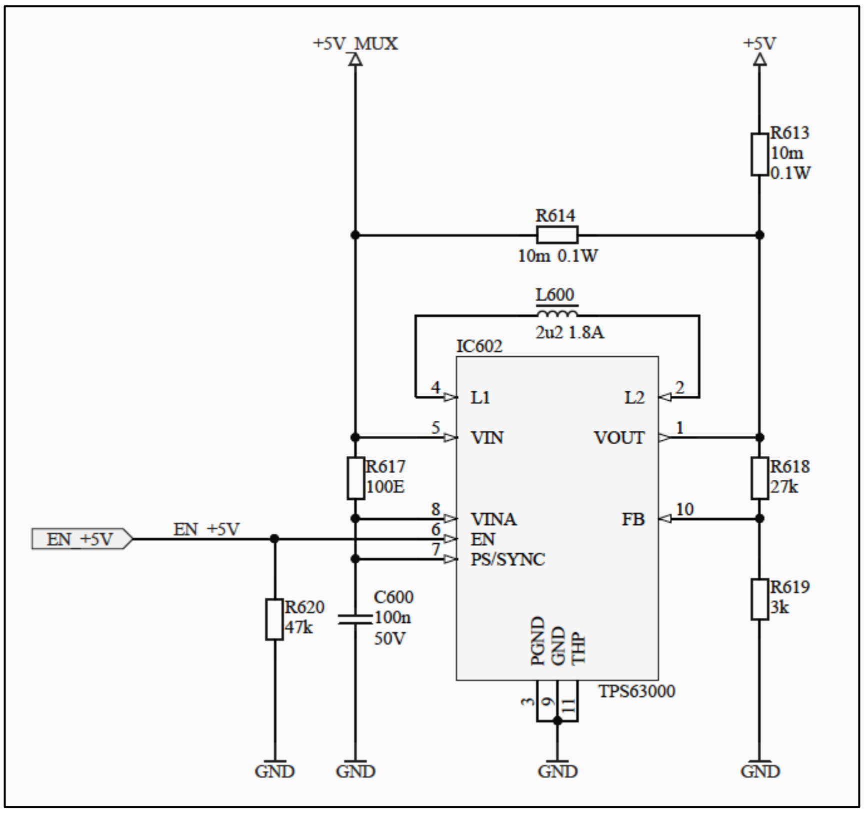

Figure 16 5 V Buck-Boost Converter.

For visualizing if the power supplies are available, light-emitting diodes (LEDs) with their specific pre-resistors are used (Figure 17). To ensure that the LEDs have the same brightness, the current has to be equal. Therefore, the resistors have to be chosen depending on the supply voltage. Since the typical forward voltage of the used LED is 2.2 V the voltage drop for the resistor of the 3.3 V supply is 1.1 V. The current is chosen with 5 mA. Therefore the resistor has to be 220 Ω. For the 5 V power supply the voltage drop for the resistor is 2.8 V. So, a resistor with 560 Ω is chosen to limit the current to 5 mA.

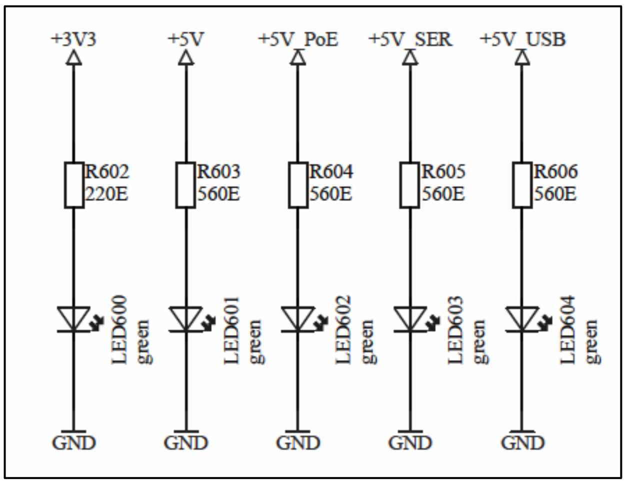

Figure 17 Power Supply LEDs.

7.2 Power over Ethernet (PoE)

The power adapter for PoE mainly consists of a power interface port and a pulse width modulation (PWM) controller which drives the PoE transformer. A 13 W PoE transformer is chosen with an output voltage of 5 V. Therefore, a current of 2.6 A is possible as it is calculated by using Equation (36).

$$I_{PoE} = \frac{P_{PoE}}{U_{PoE}} = \frac{13\,W}{5\,V} = 2.6\,A \tag{36}$$

The controller requires a feedback circuit which is realized with an optocoupler.

7.3 PMIC

The PMIC PF3000 is supplied by the 5 V power supply voltage rail. The PMIC generates the voltage which is needed as the input for its internal switching regulators itself. The switching regulators of the PMIC generate 3.3 V, 1.8 V and 1.35 V. The 1.8 V are currently not used by the PCB. Furthermore, the PMIC integrates several LDOs which are not used in this design. Also, the DRAM reference voltage which is required by the i.MX6UL is generated by the PMIC. A lithium coin battery is connected to the PMIC for supplying the RTC. For configuring the PMIC an I2C interface is used.

7.4 Power Supply for Production Mode

Since the PMIC is not set up in production mode from the very beginning, it has to be ensured that the SPU, the EEPROM and the reset generator are supplied in another way.

The 3.3 V supply in production mode is provided by the connector which is used for programming the SPU over the SWD interface at the very beginning.

In a further step, the SPU is supplied by +5 V, which is regulated by the SPU to 3.3 V. The +5 V supply for the SPU comes from one of the three alternatives as explained before: PoE, USB device connector or the USB connector which is used for the serial interface. Instead of using the current-limited power switches and the buck-boost converter, the three alternatives are connected via Schottky diodes in parallel as it can be seen in Figure 18.

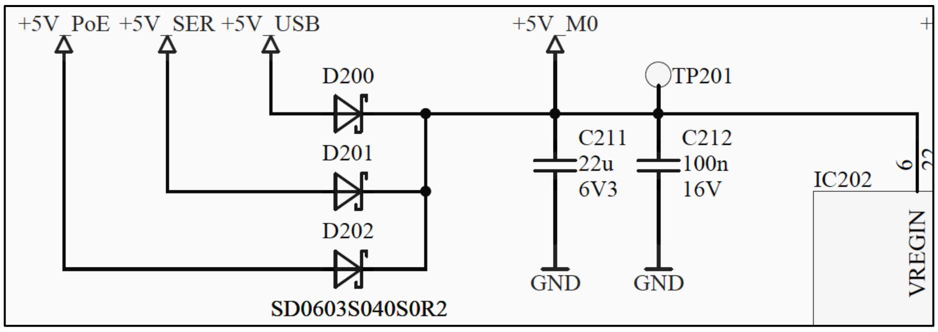

Figure 18 +5 V Supply of the SPU.

7.5 Switch Concept for Production Mode and Standard Mode of Operation

7.5.1 USB connector

The USB connector which is used for communicating with the SPU in production mode is used for the console in the standard mode of operation. Therefore, a jumper has been implemented which is set in production mode. This jumper controls a high-speed USB 2.0 switch which connects the USB connector to the SPU in production mode and to the UART to USB converter in the standard mode of operation. A second switch is used to connect the SPU in the normal mode of operation to the USB interface of the i.MX6UL, if the device is not connected over the USB device connector. This concept is shown in Figure 19.

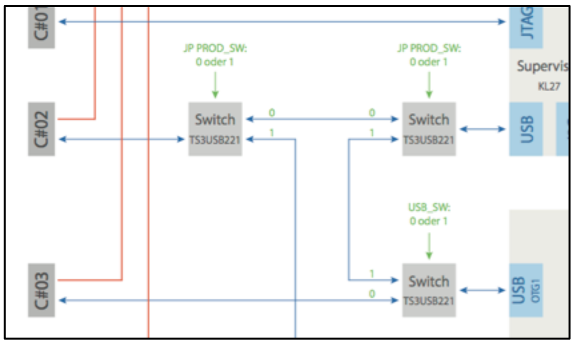

Figure 19 Switch Concept for Production Mode and Normal Mode of Operation.

7.5.1 EEPROM

The WP is pulled to VSS in production mode. Therefore, the read and write operations can be done in production mode. In the normal mode of operation it is pulled to VDD to inhibit write operations of the EEPROM.

7.5.2 I^{2}C interface

Dependent on the mode of operation different devices are connected to the I2C interface. The three different modes of operation are: **config PMIC mode**, **production mode** and the **normal mode of operation**. Figure 20 shows a flow chart with the different modes of operation.

If the CONF-PMIC signal is set to high the board is in the **config PMIC mode** regardless of the PROD-SW signal. In the config PMIC mode the i.MX6UL is just connected to the display and disconnected from all the other components (PMIC, EEPROM, temperature sensor, SPU). In this mode the SPU is connected to the temperature sensor, the EEPROM and the PMIC. Therefore, these components can be configured with the SPU.

If the CONF-PMIC signal is low, the PROD-SW controls whether the board is in production mode or in the normal mode of operation.

Since the PROD-SW signal is implemented as an active low signal, the board is in **production mode** if the PROD-SW signal is low. In this mode the i.MX6UL is connected to the display and the SPU is connected to the temperature sensor and the EEPROM. The PMIC is disconnected in that mode of operation.

If the PROD-SW signal is low the board is in its **normal mode of operation**. The i.MX6UL is connected to the temperature sensor, the EEPROM and the display. The SPU is disconnected in this mode.

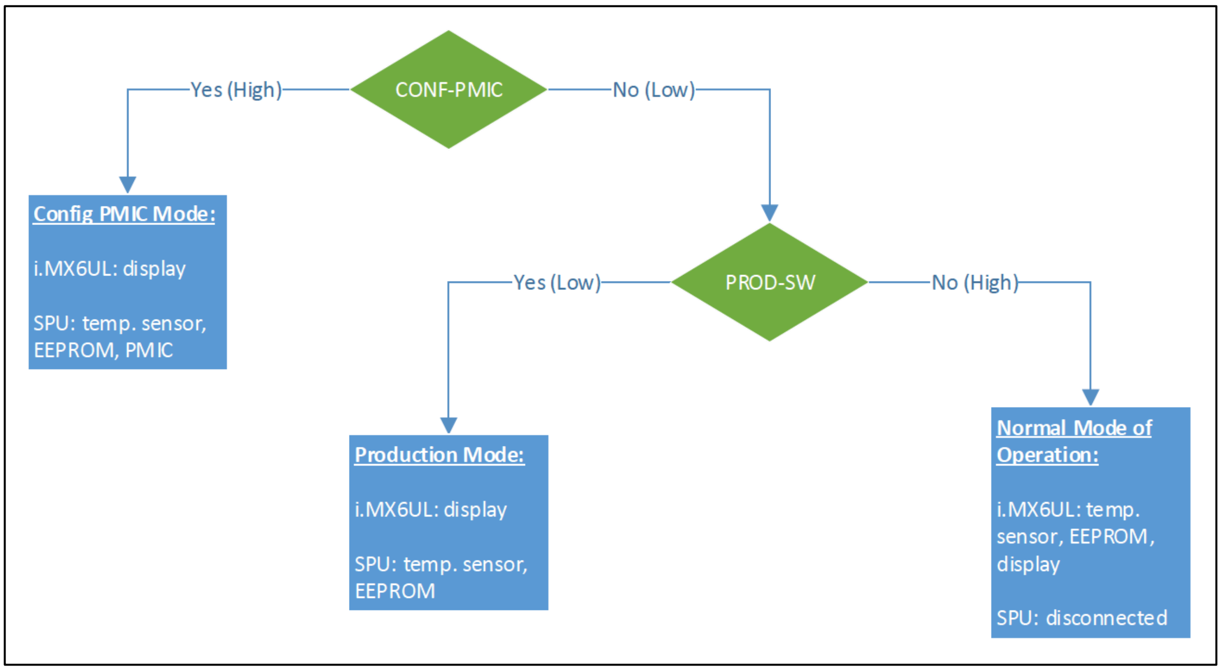

Figure 20 I2C in Different Modes of Operations.

7.6 Interfaces

The signals of the **JTAG** interface are connected to the controller by 220 Ω resistors.

The **Ethernet PHY** is connected to the RJ45 connector via the differential pair signals for receive and transmit (RX_P/RX_N and TX_P/TX_N). It is good practice to use transient voltage suppressor (TVS) diodes at the connector to comply with the standard IEC 61000-4-2 level 4 to 8 kV contact discharge and 15 kV air discharge. The PHY is connected via a reduced media independent interface (RMII) to the i.MX6UL. The Ethernet PHY requires its own 25 MHz quartz oscillator.

The differential pair of the **USB host** can be connected almost directly to the USB controller. It is good practice to add TVS diodes and a common-mode choke at the USB signals. Furthermore, ferrite beads should be connected to the 5V and the GND pin. The 5V supply signal for the connected USB device is controlled by a precision adjustable current-limited power switch. The switch limits the current dependent on the externally added resistor which is about 0.73 A (0.73 A @ 5 V = 3.65 W) in that case.

The **USB Device** connector is connected similar to the USB Host connector to the specific USB controller. It is recommendable to add TVS diodes, a common-mode choke and ferrite beads in the same way. Since the current is provided by the host device and the IoT device framework just draws current from the host device the current-limited switch is not needed. This switch has to be added on the host device.

The **CAN interface** uses a D-SUB DE-9 connector. Between the connector and the CAN controller there is a transceiver for converting the logic levels from 5 V to 3.3 V and for converting the differential CAN signal to a receive and transmit signal. Furthermore, the interface requires a 120 Ω termination resistor on both ends of the bus. Therefore, one is included in the framework, whereas the second one has to be implemented in the node on the other side of the bus. It is good practice to add TVS diodes to the CAN_HIGH and the CAN_LOW signals.

The **NAND flash** is connected via the Open NAND Flash interface which is described in the ONFi specification [58].

The **NOR flash** is connected via SPI, so it only requires four signals: Master Out Slave In (MOSI), Master In Slave Out (MISO), the clock signal and a chip select signal [52].

The **SD Card interface** consists of four data signals, a command signal, a clock signal and a card detect signal. The card detect signal shows if a card is inserted or not [52].

7.7 Parallel Single-Ended Display Interface

The connector for the display is connected via 18 parallel single-ended data signals, a clock signal, a data enable signal, a horizontal synchronization (HSYNC) and a vertical synchronization (VSYNC) signal. Besides, I2C is used for configuring the display and a reset signal is provided to the display connector. For controlling the backlight of the display an LED driver is used [52].

The components which were used for the LED driver shown in Figure 21 were calculated according to the datasheet. The only component which had to be calculated was the R1219 which is called R_{SET} in the datasheet. The feedback voltage between the FB pin and ground is regulated to 200 mV by the LED driver. The nominal current for the backlight of the display is given with 20 mA in the datasheet. The formula for calculating the value for R_{SET} is stated in the datasheet [59]:

$$R_{SET} = \frac{U_{FB}}{I_{LED}} = \frac{200\ mV}{20\ mA} = 10\ \Omega \tag{37}$$

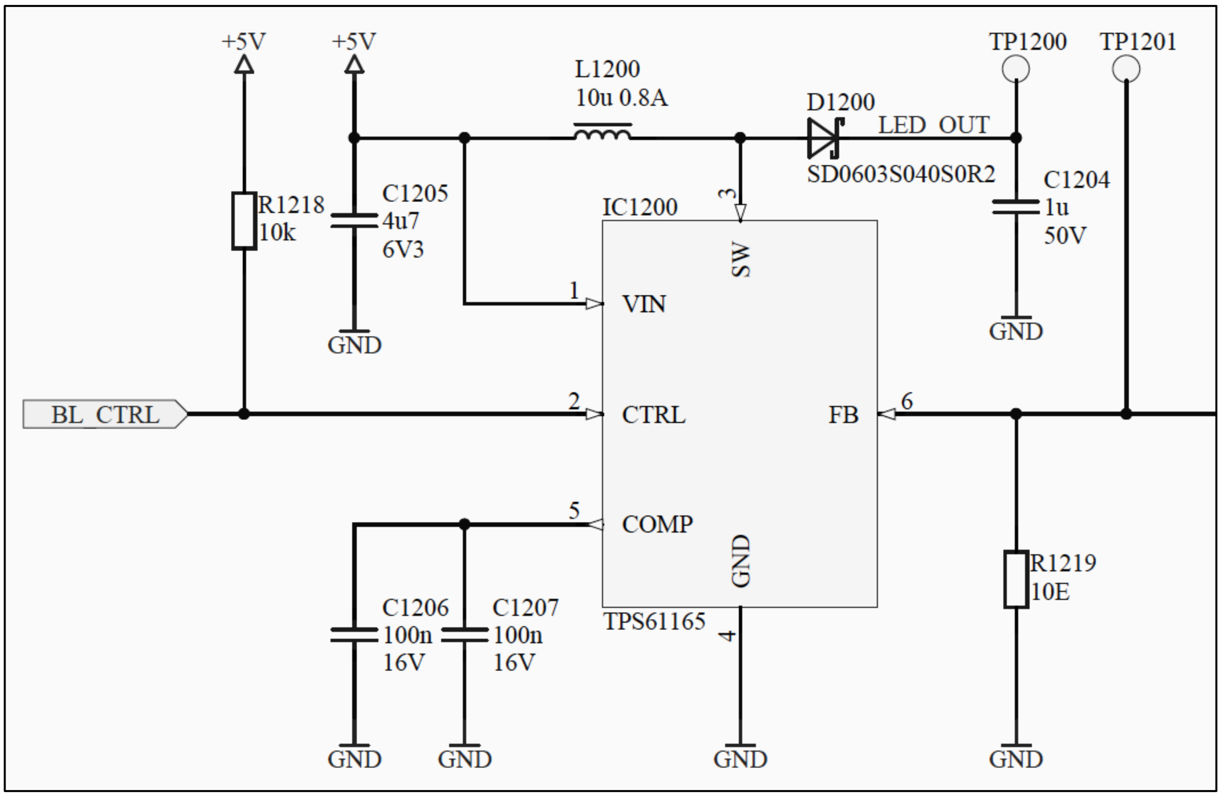

Figure 21 LED Driver Circuitry.

8 Considered Aspects for PCB Development of the IoT Device Framework

Developing the PCB requires the conformity with several design guidelines to achieve a good electromagnetic compatibility (EMC). The compliance of these guidelines avoids the need to redesign the PCB several times before it passes the tests for the EMC relevant certificates. According to the lecture “Design of Electronic Instruments and Systems” at the University of Technology in Graz, important aspects which have to be considered in the development stage of a PCB are [60]:

- placement of the connectors,
- floor planning of the PCB,
- number of PCB layers,
- separation of analog and digital ground,
- routing of the power supply tracks,
- routing of high frequency signals,
- usage and placement of decoupling capacitors,
- placement of components.

Furthermore, the document “Hardware Development Guide for the i.MX 6UltraLite Applications Processor” provided by NXP was considered for the development of the IoT device framework [61].

8.1 Placement of the Connectors

Connectors should be placed on the same side of the PCB if it is possible. Placing the connectors on opposite sides of the PCB comes with high electromagnetic emissions, due to the fact that a bigger distance between the ground signals of two connectors causes a bigger voltage drop. This is because the resistance increases proportional to the distance. Having the connectors on the same side of the PCB minimizes the distance and therefore the voltage drop between them.

In the case of the IoT device framework it was not possible to follow this guideline in all cases because of the lack of space on just a single side for all the connectors. Placing all the connectors on the same side is often not possible because mechanical requirements restrict it.

8.2 Floor Planning

The floor planning of the PCB already starts with the development of the schematic. Separating the functional blocks to different schematic sheets is essential for a good overview and a fast PCB design. In a simple design these functional separations could be just the separation to power, digital, analog and supply circuits. A good practice is to assign three or four digit designators where the first digit or digits represent the number of the corresponding schematic sheet. This method helps especially when it comes to design the PCB. It should be considered to place noisy parts close to the connector. Furthermore, susceptible parts should be placed further away from noisy parts on the PCB.

In the case of the developed IoT device framework the design was separated to the following schematic sheets:

- A **block diagram** which is the top schematic sheet of the hierarchy. It connects all the other subordinated schematic sheets,
- **Supervisory processing unit**: with the KL27 controller and the console,
- **Power-over-Ethernet** (PoE): with the RJ45 connector, the PoE interface IC, the 13 W transformer and the feedback loop with an optocoupler,
- **Ethernet**: Ethernet signals from the RJ45 connector of the PoE sheet and the Ethernet PHY,
- **USB**: USB host and USB device with an overcurrent switch,
- **Power supply**: the three power supplies for the PCB with an overcurrent switch each and a buck-boost converter for guarantee stable 5 V power supply,
- **PMIC** with a coin cell,
- **i.MX6 power**: with the circuitry of the power pins of the i.MX6,
- **i.MX6 control**: with a JTAG connector, an EEPROM and a temperature sensor,
- **NOR, NAND, SDCard and CAN** interface are placed on one schematic sheet but clearly separated within this sheet,
- **DDR3 SDRAM** and
- **Display**: with parallel display interface to a connector and a switching regulator for the backlight.

8.3 Number of PCB layers

Easy designs can be implemented with two layers either with or without a solid ground plane. Having a more complex design requires more layers.

A good way to implement a layer stack for a four layer design:

- Layer 1: signal,
- Layer 2: ground,
- Layer 3: power supply and
- Layer 4: signal.

The advantage of having the ground plane and the power supply plane on adjacent layers leads to a good decoupling between the signal layers. It should be avoided that the ground planes or power supply planes are separated by slots (e.g. because of vias in a row or tracks).

In the case of the developed IoT device framework eight layers are used. The layer stack is implemented as followed:

- Layer 1: signal,
- Layer 2: ground,
- Layer 3: signal,
- Layer 4: power with two zones: 3.3 V for the supervisor and 5 V,
- Layer 5: power with two zones: 3.3 V and 1.35 V,
- Layer 6: signal,
- Layer 7: ground and
- Layer 8: signal.

8.4 Separation of Analog and Digital Ground

The ground plane for digital and analog parts should be separated but it has to be considered that the current path of a signal and the current return path on the ground plane do not form a big loop. Therefore, a signal should be routed the same way on the signal plane as the current flows back on the ground plane.

8.5 Routing of the Power Supply

The routing of the power supply is a further important aspect. Big loops in tracks for the power supply should be avoided to achieve high EMC.

8.6 Routing of High Frequency Signals

It is very important to route high frequency (HF) signals as short as possible. A good way to achieve short tracks is to route them first so before all other signals. Furthermore, they should not be routed over a slot of any ground or power plane. It is further good practice to shield high frequency signal tracks with ground tracks on both sides of the HF track. Changing the layers should be avoided with HF tracks. In the IoT device framework the DDR3 SDRAM and USB use HF signals.

There are several constraints which have to be considered for routing the DDR3 SDRAM tracks. Especially, the maximum length of the tracks and the maximum mismatch of lengths to each other have to be considered when routing the DDR3 SDRAM. As recommended for HF signals, it should be avoided to change layers. Especially, the signals of the same byte lane should be routed on the same layer.

USB.org specifies the layout guidelines in [62]. For the USB interface a 90 Ohm differential impedance is required.

8.7 Usage and Placement of Decoupling Capacitors

Ripple and noise can lead to a lower noise margin and to a higher clock jitter of ICs. Decoupling capacitors stabilize voltages which are used to supply an IC. Furthermore, they decrease the size of the current loop from the source to the IC and back to the source because they are placed close to the supply pins of the IC. An ideal placement of the decoupling capacitor can be seen in Figure 22. The via to the ground plane should be placed right after the capacitor, so that the current return path is on the ground plane.

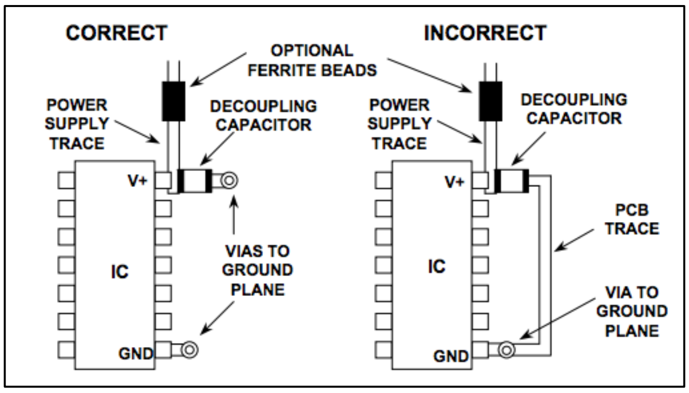

Figure 22 Placing of Decoupling Capacitors [63]**.**

Besides, placing decoupling capacitors close to the supply pin, the current return path, so the connection to the ground pin, should be also as short as possible. In a lot of cases a 100 nF decoupling capacitor is used on the PCB. Sometimes another 10 μF capacitor is used in parallel. The 100 nF has the advantage that its equivalent series resistor (ESR) is lower and it therefore has a faster response. Since a capacitor behaves like an inductor from a specific frequency the resonant frequency has to be considered for the selection. This can be explained by its equivalent series inductance (ESL). The resonant frequency can be calculated by using Equation (38).

$$f = \frac{1}{2\pi} \cdot \sqrt{LC} \tag{38}$$

For selecting the decoupling capacitor for the supply pins of the i.MX6UL, Table 9 in the “Hardware Development Guide for the i.MX 6UltraLite Applications Processor” was used as a reference [61].

8.8 Placement of Components

Especially placing inductors and capacitors for example in form of a filter is critical. It has to be avoided that there is a coupling between inductors and capacitors.

9 Product Lifecycle Management

The product lifecycle management (PLM) includes the component and product tracking, the configuration management, the software deployment and the update functionality of the device. A common way to solve the **component tracking** of a device is to introduce a barcode which includes all the relevant information of the device. Such information can be the name of the device, the article number and a unique serial number. In contrast to one-dimensional barcodes, the Quick Response (QR) code or the DataMatrix code are two-dimensional barcodes which are used more and more nowadays. Often the two-dimensional DataMatrix-Code according to the ISO/IEC 16022 with the error-correction code ECC200 is used. So even if there is a scratch, it is possible to correct the barcode. It is good practice to store the barcode and device specific information electronically in a non-volatile memory for example in an EEPROM or an unused partition of the NOR Flash. The software should be able to read out the used components and the versions of these components. The **configuration management** is separated into the device configuration and the applications configuration. The configuration management should evaluate if the specific hardware configuration (option or variant) is valid. The **software** consists of only one binary which contains all the parts (kernel, device tree, root file system, application and the update functionality). During production process an initial software package is loaded into the device. This initial package can be updated later via a browser where the web server runs on the device.

10 Conclusion

Using the internet of things (IoT) device framework to develop a new IoT device leads to an individual Single-Board-Computer (SBC) solution. The idea of an IoT device framework came up after the comparison of different hardware development concepts for an IoT device. Basically, the advantages of the usual SBC solution and the advantages of the DBC solution are combined and the disadvantages of both are reduced essentially. Using the IoT device framework for development of an IoT device leads to an application specific and robust solution in a short time because modules of the framework are reused and integrated in a new design. Since the development time is reduced significantly the time-to-market (TTM) and the total-cost-of-ownership (TCO) are reduced compared to standard SBCs which have to be designed from scratch.

Furthermore, using Linux for being the interface between the hardware and an application is the best solution since the device framework has enough resources for running such an operating system. A very important aspect is the real-time capability of the IoT device. Therefore the PREEMPT_RT patch can be used which is a simple solution compared to others.

The framework is designed in a way that periphery can either be integrated on the SBC so on the IoT device directly, or it can be easily abstracted over standard interfaces like CAN or USB to another device.

11 References

[1] L. Columbus, “2017 Roundup Of Internet Of Things Forecasts,” *Forbes*, 2017. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2017/12/10/2017-roundup-of-internet-of-things-forecasts/#4e2f39d31480. [Accessed: 30-Dec-2017].

[2] R. van der Meulen, “Gartner Says 8.4 Billion Connected ‘Things’ Will Be in Use in 2017, Up 31 Percent From 2016,” 2017. [Online]. Available: https://www.gartner.com/newsroom/id/3598917. [Accessed: 30-Dec-2017].

[3] GrowthEnabler, “Market Pulse Report, Internet of Things (IoT),” 2017.

[4] Toradex, “Kundenspezifische Single Board Computer,” 2018. [Online]. Available: https://www.toradex.com/de/customized-single-board-computer. [Accessed: 28-Jan-2018].

[5] Phytec, “Single Board Computer | PHYTEC,” 2018. [Online]. Available: http://www.phytec.de/produkte/single-board-computer/. [Accessed: 28-Jan-2018].

[6] R. Saleh *et al.*, “System-on-chip: Reuse and integration,” *Proc. IEEE*, vol. 94, no. 6, pp. 1050–1068, 2006.

[7] NXP Semiconductors, “IMX6UL-BD.png (800×480),” 2018. [Online]. Available: https://www.nxp.com/assets/images/en/block-diagrams/IMX6UL-BD.png. [Accessed: 28-Jan-2018].

[8] Viewpoint Systems, “Comparing Off-The-Shelf To Custom Designs For Industrial Embedded Systems - Viewpoint Systems.” [Online]. Available: https://www.viewpointusa.com/IE/wp/comparing-off-the-shelf-to-custom-designs-for-industrial-embedded-systems/. [Accessed: 13-Mar-2018].

[9] J. Kroll, “Single-Board-Computer: Modular ohne Module | Elektronik,” *Elektronik Fachmedium für industrielle Anwender und Entwickler*, 2012. [Online]. Available: http://www.elektroniknet.de/elektronik/embedded/modular-ohne-module-85390.html. [Accessed: 26-Dec-2017].

[10] P. Mohapatra, “Why choose System on Module over SBC for Embedded Development?,” 2015. [Online]. Available: https://www.toradex.com/de/blog/system-on-module-over-single-board-computer-for-embedded-development. [Accessed: 31-Dec-2017].

[11] J. Sarantakes, “COTS vs Custom: Top 5 Reasons to Go Custom for Enterprise Applications - Headspring,” 2016. [Online]. Available: https://headspring.com/2016/03/09/cots-vs-custom-top-5-reasons-go-custom-enterprise-applications/. [Accessed: 28-Jan-2018].

[12] Jedec, “DDR3 SDRAM STANDARD | JEDEC.” Jedec, 2012.

[13] S. Sinha, R., Roop, P., Basu, *Correct-by-Construction Approaches for SoC Design, ISBN: 978-1-4614-7863-8*. 2013.

[14] F. Gaillard and A. Eieland, “Microprocessor (MPU) or Microcontroller (MCU)?,” *Atmel*. pp. 1–4, 2013.

[15] Eclipse IoT Working Group, IEEE IoT, and AGILE IoT, “IoT Developer Survey 2016.” pp. 1–39, 2016.

[16] A. Busse, “ARM-Chips für Einplatinen-Computer,” *Linux-Magazin*, 2015. [Online]. Available: http://www.linux-magazin.de/ausgaben/2015/01/arm-socs/. [Accessed: 24-

Jan-2018].

[17] R. Mitchell, “Understanding the Differences Between ARM and x86 Processing Cores - News,” 2017. [Online]. Available: https://www.allaboutcircuits.com/news/understanding-the-differences-between-arm-and-x86-cores/. [Accessed: 26-Dec-2017].

[18] R. Merritt, “Intel, ARM Battle over IoT: Explosion of end nodes makes MCUs strategic,” *EE Times*, 2016. [Online]. Available: https://www.eetimes.com/document.asp?doc_id=1330662. [Accessed: 26-Dec-2017].

[19] I. King, “Intel Licenses ARM Technology to Boost Foundry Business - Bloomberg,” 2016. [Online]. Available: https://www.bloomberg.com/news/articles/2016-08-16/intel-licenses-arm-technology-in-move-to-boost-foundry-business. [Accessed: 26-Dec-2017].

[20] Arm Limited, “Processors – Arm,” 2018. [Online]. Available: https://www.arm.com/products/processors. [Accessed: 28-Jan-2018].

[21] I. Maxim Integrated Products, “Multiple Voltage Systems Need Supply-Voltage Sequencing,” 2002. [Online]. Available: https://www.maximintegrated.com/en/app-notes/index.mvp/id/1133. [Accessed: 01-Jan-2018].

[22] N. Rossetti and R. Lenk, “DDR Memories Require Efficient Power Management,” 2001. [Online]. Available: http://www.powerelectronics.com/content/ddr-memories-require-efficient-power-management. [Accessed: 28-Jan-2018].

[23] Micron Technology Inc., “DRAM | Memory and Storage,” 2018. [Online]. Available: https://www.micron.com/products/dram. [Accessed: 28-Jan-2018].

[24] Micron Technology Inc., “NOR / NAND Flash Guide,” no. Mlc. pp. 1–8, 2013.

[25] J. Jung, “Application of UBIFS for Embedded Linux Products,” *LinuxCon Japan*. pp. 1–21, 2010.

[26] Micron Technology Inc., “Choosing the Right NAND,” 2018. [Online]. Available: https://www.micron.com/products/nand-flash/choosing-the-right-nand. [Accessed: 24-Jan-2018].

[27] ON Semiconductor, “EEPROM Overview and Applications Tutorial - On Semiconductor | DigiKey,” 2016. [Online]. Available: https://www.digikey.com/en/ptm/o/on-semiconductor/eeprom-overview-and-applications/tutorial. [Accessed: 28-Jan-2018].

[28] B. Vertenten *et al.*, “UTMI+ Specification, Rev. 1.0.” pp. 1–26, 2004.

[29] EE Times, “Low-pin-count USB transceiver spec is released,” 2004. [Online]. Available: https://www.eetimes.com/document.asp?doc_id=1148806. [Accessed: 31-Dec-2017].

[30] B. Hosler *et al.*, “PHY Interface For the DisplayPort , and Converged IO Architectures, Version 5.0.” pp. 1–145, 2017.

[31] Microchip Technology Inc., “Ethernet theory of operation,” *Microchip Technology Inc., Application Note …*. pp. 1–26, 2008.

[32] Silvaco, “I2S Audio Interface,” 2018. [Online]. Available: https://www.silvaco.com/products/IP/i2s-audio-interface/index.html. [Accessed: 28-Jan-2018].

[33] Intel Corporation, “Audio Codec ’ 97, Rev. 2.3.” pp. 1–108, 1998.

[34] HDMI Licensing, “HDMI :: Manufacturer,” 2018. [Online]. Available: https://www.hdmi.org/manufacturer/index.aspx. [Accessed: 28-Jan-2018].

[35] Toshiba, “Display Interface Bridge | TOSHIBA Semiconductor &amp; Storage Products | Asia-Pacific,” 2018. [Online]. Available: https://toshiba.semicon-storage.com/ap-en/product/assp/interface-bridge/display-interface.html. [Accessed: 28-Jan-2018].

[36] Toshiba, “Camera Interface Bridge (HDMI(R) Interface Bridge included) | TOSHIBA Semiconductor &amp; Storage Products | Asia-Pacific,” 2018. [Online]. Available: https://toshiba.semicon-storage.com/ap-en/product/assp/interface-bridge/camera-interface.html. [Accessed: 28-Jan-2018].

[37] L. G. Casagrande and F. L. Kastensmidt, “Soft error analysis in embedded software developed with &amp; without operating system,” *2016 17th Latin-American Test Symp.*, pp. 147–152, 2016.

[38] T. N. B. Anh and S.-L. Tan, “Real-Time Operating Systems for Small Microcontrollers,” *IEEE Micro*, vol. 29, no. 5, pp. 30–45, 2009.

[39] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, “Operating Systems for Low-End Devices in the Internet of Things: A Survey,” *IEEE Internet Things J.*, vol. 3, no. 5, pp. 720–734, 2016.

[40] C. Bormann, M. Ersue, and A. Keranen, “RFC7228: Terminology for Constrained-Node Networks,” 2014. [Online]. Available: https://tools.ietf.org/html/rfc7228. [Accessed: 26-Dec-2017].

[41] Eclipse IoT Working Group, IEEE IoT, and AGILE IoT, “IoT Developer Survey 2017.” pp. 1–53, 2017.

[42] J. Altenberg, “Linux und Echtzeit.” pp. 1–38, 2006.

[43] H. Shah, R. Shah, U. Shah, and S. Deshmukh, “Performance parameters of RTOSs; Comparison of open source RTOSs and benchmarking techniques,” *2013 Int. Conf. Adv. Technol. Eng. ICATE 2013*, no. 101, 2013.

[44] M. D. Marieska, P. G. Hariyanto, M. F. Fauzan, A. I. Kistijantoro, and A. Manaf, “On Performance of Kernel Based and Embedded Real-Time Operating System : Benchmarking and Analysis,” *2011 Int. Conf. Adv. Comput. Sci. Inf. Syst.*, pp. 978–979, 2011.

[45] P. Feuerer, “Benchmark and Comparison of Real-Time Solutions Based On Embedded Linux,” *Hochschule Ulm, German*. pp. 1–95, 2007.

[46] J. H. Brown and B. Martin, “How fast is fast enough? choosing between Xenomai and Linux for real-time applications,” *proc. 12th Real-Time Linux Work.*, pp. 1–17, 2010.

[47] S. K. Maharana, “IMPROVING USB 3.0 WITH BETTER I/O MANAGEMENT,” *EE Times*, no. June, pp. 1–8, 2016.

[48] C. Peacock, “USB in a Nutshell, Making Sense of the USB Standard, Third Release,” 2002.

[49] A. Gupta, “USB 3.0 vs USB 2.0: A quick reference summary for the busy engineer | Embedded,” 2014. [Online]. Available: https://www.embedded.com/design/connectivity/4437961/USB-3-0-vs-USB-2-0--A-quick-reference-summary-for-the-busy-engineer. [Accessed: 02-Feb-2018].

[50] Qlogic, “Introduction to Ethernet Latency An Explanation of Latency and Latency Measurement, SN0330915-00 Rev. E,” *Qlogic Reports*, pp. 1–4, 2017.

[51] A. Gerber, “Choosing the best hardware for your next IoT project,” 2017. [Online]. Available: https://www.ibm.com/developerworks/library/iot-lp101-best-hardware-

devices-iot-project/index.html. [Accessed: 28-Jan-2018].

[52] NXP Semiconductors, “Applications Processors for Industrial Products, Rev. 2.2.” pp. 1–132, 2017.

[53] Texas Instruments, “AN-1358 LM5070 (AE) Evaluation Board.” pp. 1–11, 2013.

[54] NXP Semiconductors, “i.MX 6 Series Applications Processors: Multicore, Arm® Cortex®-A9 Core, Arm Cortex-M4 Core|NXP,” 2018. [Online]. Available: https://www.nxp.com/products/processors-and-microcontrollers/applications-processors/i.mx-applications-processors/i.mx-6-processors:IMX6X_SERIES. [Accessed: 26-Dec-2017].

[55] J. Zeng, “IoT evolution: Taking processing and analytics to the edge,” 2016. [Online]. Available: https://blog.nxp.com/internet-of-things-2/iot-evolution-taking-processing-and-analytics-to-the-edge. [Accessed: 26-Dec-2017].

[56] Silicon Laboratories Inc., “Programming Internal Flash Over the Serial Wire Debug Interface 1 Debug Interface Overview, Application Note AN0062, Rev. 1.02.” pp. 1–27, 2013.

[57] Texas Instruments, “TPS6300x High-Efficient Single Inductor Buck-Boost Converter With 1 . 8-A Switches.” pp. 1–17, 2015.

[58] Hynix Semiconductor *et al.*, “Open NAND Flash Interface Specification : Block Abstracted NAND, Rev. 2.0.” pp. 1–169, 2008.

[59] Texas Instruments, “TPS61165 High-Brightness , White LED Driver in WSON and SOT-23 Packages.” pp. 1–23, 2016.

[60] B. Deutschmann and G. Winkler, “EMC Aware PCB Design, Lecture Notes ‘Design of Electronic Instruments and Systems’, University of Technology, Institute of Electronics.” pp. 1–97, 2017.

[61] NXP Semiconductors, “Hardware Development Guide for the i . MX 6UltraLite Applications Processor, Rev. 2.” pp. 1–55, 2017.

[62] Intel Corporation, “High Speed USB Platform Design Guidelines, Rev. 1.0.” pp. 1–19, 2000.

[63] Analog Devices, “Decoupling Techniques,” *Application Note, MT-101 Tutorial*. pp. 1–14, 2009.

12 Appendix

12.1 List of Figures

12.2 List of Tables