Sebastian Lassacher, BSc

Analysis and Implementation of a Cloud-Supported Update Delivery System for Embedded Devices

Masterarbeit

Zur Erlangung des akademischen Grades
Master of Science
Masterstudium Telematik

Eingereicht an der

Technische Universität Graz

Betreuer:
Eugen Brenner, Ao.Univ.-Prof. Dipl.-Ing. Dr.techn.

Institut für Technische Informatik

Graz, September 2018

Kurzfassung

Die Energiewende schreitet voran und das intelligente Energiemanagementsystem "SEMS" verteilt mit seinen ausgeklügelten Algorithmen Energie effizient in privaten Energiehaushalten. Neben der teureren Variante des Smarthome-Systems mit Alu-Gehäuse und Touch-Display gibt es auch eine kleinere und billigere Variante für eine Hutschienenmontage. Für den Nachfolger, der eine leistungsschwächere Elektronik beinhaltet, musste das gesamte Konzept zur Erstellung neuer Linux-Systeme überarbeitet werden. Der Schwerpunkt liegt dabei auf hoher Sicherheit und Updatebarkeit vor Ort beim Kunden.

Die Arbeit fokussiert nicht auf die Smart Home Anwendung selbst, sondern auf das dafür benötigte Linux-System. Ausfälle durch fehlerhafte Update-Prozesse sind nicht tolerierbar. Der Bootloader (U-Boot) und der Linux-Kernel in Verbindung mit seinem Dateisystem wurden soweit verändert, so dass die genannten Anforderungen erfüllt wurden. Die Hardwareauswahl erfolgte bereits zuvor. Das Build-System Yocto eignet sich sehr gut, um die gesamte Systemerstellung automatisiert und nachvollziehbar zu halten und die Anwendung von Sicherheitsupdates einfach zu machen. Die Verteilung der Updates wurde über eine Cloud-Infrastruktur ermöglicht, die verschiedene Szenarien bei der Update-Verteilung zulässt. Die erste Charge der Massenproduktion der neuen Energiemanagervariante wurde zum Teil in der Arbeit mit behandelt.

Abstract

The energy transition is a vital part of nowadays energy supply. The intelligent energy management system "SEMS" distributes energy with its sophisticated algorithms in domestic energy households. Besides the already established product variant, which is very expensive due to its aluminum housing and touch display, a new and cheaper generation of the smart home device had to be created for the market. The new version features low cost hardware and housing for mounting on top hat rails. A new concept for creating Linux system for a safe and secure way to update the system in-field at the customers' sites had to be accomplished.

The focus lies not on the running smart home application but on the Linux system needed for it. The used bootloader (U-Boot) and the Linux kernel together with its root file system (rootfs) had to be modified in a way to achieve the desired goals. Hardware selection was done prior to this work. Yocto, a building system for embedded devices, was used to be able to compile everything from the source in a reproduce-able way. Security is an important factor when dealing with embedded systems and Yocto makes it easy to apply security patches. The distribution of the updates was done with a cloud infrastructure which allows different scenarios for deploying updates. Partly, aspects of the first batch of mass production are covered in this thesis as well.

STATUTORY DECLARATION

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 marked all material which has been quoted either literally or by content from the used sources.

....
date (signature)

Danksagung

Bedanken möchte ich mich bei meinen Kollegen, mit denen ich gemeinsam 2013 dieses Projekt im Bereich der erneuerbaren Energien begonnen habe, aus dem danach auch die Firma LEVION entstand.

Weiters danke ich besonders meinem vorherigen Betreuer Dipl.-Ing. Dr.techn. Christian Kreiner, der mir bei schwierigen Problemstellungen stets einen Schubs in die richtige Richtung gab, sowie meinem derzeitigen Betreuer Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eugen Brenner.

Meinem Bruder sowie meinen Tanten danke ich für ihre tolle Unterstützung.

Besonderer Dank gilt meiner Mutter, die mir meine Ausbildung und somit auch das Studium erst ermöglicht hat und mir immer zur Seite stand.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

1.1 Motivation

In 2013, first products for so-called demand side management were available on the photovoltaics (PV) market. But none of the offered solutions were really easy to use (for the end customer as well as for the installer) and the system architecture offered no real flexibilities. Every household is very different from one another and the often fixed number of sensor inputs and controlling outputs of such devices limited the scenarios where they could be implemented. Demand side management is an enabler of the ongoing energy transformation.

Because of that a small group of people decided to build their own smart home solution focusing on energy management for domestic homes and small businesses. Due to the multi-vendor ecosystem in a typical household, one focus was to integrate devices from other vendors. The way to go was to keep as much intelligence at other devices as possible because of the vendors' expertise. For instance, a vendor for heating systems knows best how to design a controlling algorithm. Besides that, simplicity for all users was a major goal.

Soon it was clear that security was also a vital part of the new smart home application. Most vendors on the PV market hardly concerned themselves with security. The SEM (see figure 1.1) was developed as a high end, high price product equipped with Wireless-LAN, Ethernet and Z-Wave, a low power home automation standard mostly present in the United States.

This was only the first step on the hardware road map. The next step had been planned since the beginning: a cheaper version (of the SEM) having no fancy aluminum housing or touch display. For this second device, a new system needed to be built which hosts the same smart home appliance its bigger brother does.

Figure 1.1: The SEM - a marketing photo.

1.2 Goal

Prior to this work, a basic hardware selection for the display-less version of the smart home controller was already made. The goal was to create a suitable Linux system image for deployment on the chosen hardware. The resulting unbranded product was called "Compact Unbranded" which is shown in figure 1.2.

Figure 1.2: The "Compact Unbranded", a blank for various brands. After electric and housing assembly but before provisioning

Compared to its bigger brother, the housing of the Compact Unbranded is very simple and will not be described further in this thesis. However, the integration of the two buttons and three RGB light-emitting diode (LED)s is covered.

The created hardware is based on a SoM called "CHIP Pro" which is extended with a custom base board and one expansion board. Besides supporting the complete hardware, a way to update the complete system image in-field in a crash-safe way needs to be found. Of course, security is a major part when dealing with embedded systems.

The next step is the final system image bundled together with the boot loader and the smart home application itself. This image is needed for flashing the first batch of hardware and will be used in future batches too if there are no changes to the initial software. During production, a hardware test needs to be performed to ensure high quality standards. This testing process should be as fast and as automated as possible.

One additional output of the system creation process is the toolchain needed for compiling the smart home application for the target system. The creation of this application is not within the scope of this thesis.

Figure 1.3 shows one possible product with its applied branding. Various products

Figure 1.3: The SEMcompact - a marketing photo.

from various brands can be created based on the "Compact Unbranded".

Besides the update process on the target system, the distribution of its updates needed to be considered as well for the complete solution. This closes the gap between the update creation and their application in-field at customers' sites.

1.3 Arrangement

After the introduction containing motivation and goal for this thesis, the state of the art for this field of science is discussed together with its related work. Additionally, background information for a better understanding of the driving forces of the concept and the implementation related to it is presented. This covers update processes, embedded systems running Linux and smart home application characteristics.

The concept chapter discusses the exact requirements identified, the hardware target related considerations, tools already present and how they can be used. Further on, the customization of the Linux system and the bootloader are discussed together with the boot process itself. The partitioning of the NAND memory is described before moving on to the update process and update distribution. The chapter is closed with security considerations.

Based on the concept, the implementation chapter focuses on the actual changes made to the build system, starting with the general usage of Yocto and moving on to the new and changed recipes and layers of the building process. Those changes and additions are separated into the bootloader, the kernel and its rootfs. Extended by the auxiliary scripts needed for and used by the on-top running application.

Testing and results are presented in two different styles. First, the testing process for creating the systems for the first prototypes is described. In contrast to that, testing a complete production batch with automation is explained. The result section is organized in the same way.

This thesis closes with a conclusion and an outlook in its context.

1.4 State of the Art

Linux running on ARM systems is very common today [yBpLfW10]. Updating them in a secure way too. But back when the work on the first hardware started, there were no publications available on update crash-safe considerations for embedded systems. Similar update processes were present on the mobile platforms like Android and iOS. But those were commercial products.

Security threats are well known and can be categorized in many ways [RHM+12]. Depending on the product design, different possible attack scenarios may be applicable due to chosen interfaces to the outside world. Various smart home systems already exist but the interoperability is a big issue [PME+09]. Various standards and protocols are in place but most time they are not compatible to each other.

Technologies like "Boot to Qt" exist to reduce the system overhead [Sle13]. Often feature-rich software and tools are not needed to be installed on embedded systems. Normally, no one ever logs into a shell to retrieve information or performs actions. Therefore, reducing all shipped tools, programs and libraries to a minimum is a good idea. But it limits the flexibility a lot.

Architectural concepts to bring cloud services and smart home applications closer together exist [YWJ+10]. This specific work proposes a cloud architecture based on smart home. Smart home is a technology which is based on its predecessors the PC and the Internet. While first smart home application software was often installed on local PCs, another approach is to take the software and services not to other dedicated devices locally but to the cloud [YH11].

In the context of the market situation, for which the application running on the target system is designed, demand side load management concepts exist and are used for being able to integrate many decentralized energy producers (like PV systems) to the public grid [WHD+11]. The old approach of controlling the power production is not suitable anymore. Therefore, the demand side must be controllable up to some point. This is an enabler of the energy transition. Another system related to the resulting product based on the target system is one that uses Zigbee, a wireless sensor network, which is similar to Z-Wave [HL10].

Chapter 2

Related Work

In the context of the project described in the motivation section on page 1 there were a few related works by various people at Graz University of Technology.
First a comparison of various communication standards was done by Florian Hofer [Hof14]. The work contains an analysis of different aspects when it comes to selecting a proper wireless standard for domestic home applications.

For the first custom hardware built, Christian Mentin describes the challenges of creating a ready-to-sell hardware [Men16]. This was the first milestone in the previously mentioned project for running the own home control application on custom hardware. Another device developed for the very same project is an intelligent edge device used for domestic water heating. This work was done by Thomas Fuchs [Fuc17, Fuc18]. Both of those also take electromagnetic interference (EMI) and electromagnetic compatibility (EMC) into account.

When moving on from the hardware to the software running on it, especially on ARM platforms, a work about the Linux kernel which deals with similar challenges was done [yBpLfW10]. The main focus was also on providing a system for further applications. It also takes real-time performance into account. Various trade-offs between architecture and requirements were discussed while developing an embedded system with a graphical user interface (GUI) in a separate paper [LNNJ12]. Similar work when considering GUIs was done with a focus on the very same framework as used for the top level application in this work [JCZ12].

Several papers were published in the context of smart home and home control systems [WWSL09, NHC13, GDLZ11, WSES13]. Since the term smart home is difficult to define due to its huge scope for interpretation, those works differ widely. The first paper focuses on various connected devices at home with the possibility of remote control from anywhere thanks to the Internet. The application running on top of the target system is similar in many ways, but it focuses heavily on energy distribution. The second paper focuses on the Internet Protocol (IP) communications within smart homes. Various workloads have to be handled in smart homes. Some of them can be shifted to cloud computing. The next work discusses exactly these possibilities. The last paper takes a smart phone into account when running a smart home application.

Chapter 3

Background

This chapter describes various areas which impacts the hardware target, its resulting products and their update. The most important area is the renewable energy market and many smart home products are associated with it. The first section discusses different categories of such products and how they are characterized. Demand side management is one of the major topics when dealing with energy distribution. Therefore, the next section describes it in detail.

Besides market driven aspects, various purely technical ones like the security in Internet of Things (IoT) and domestic homes are very important for the system. Establishing trust in a distributed environment with the help of PKIs and different scenarios how to get access to devices are two central parts as well. Combined with the update processes, this is described later on in this chapter.

3.1 Smart Home Systems

The term "smart home" is difficult to define. It gathers a lot of different approaches for automating devices and processes at home. The marking strategies of various companies amplified the diversity of the meaning.

Some refer to a "smart home" as a quite simple and not very intelligent wall mounted control working together with motorized heating radiator valves. Others define a "smart home" as a system for smart energy distribution at home. Basically, there are devices talking to each other and some intelligence is embedded somewhere at home or even in the cloud.

Some differentiations are shown in the next sub sections. However, smart home systems are the enabler for demand side management at least for domestic homes, depending on how the term "smart home" is defined. On figure 3.1, an overview of the mentioned previous types is shown. The term demand side management it described in section 3.2.

3.1.1 Fully Cloud Based

As described in the previous section, smart home systems can be very different and this also applies to this sub section. There is a variety of systems which are only working with a connection to some cloud service. On the left side of figure 3.1, such systems are shown in a very simplified way. Often a local gateway is installed at the local (IP based) network. It connects to a remote cloud service. On-site they link together different devices. Rule based automation actions are triggered at the cloud system based on measurements not only limited to the devices installed on site. Users can control their devices often via a web portal or a mobile phone app. Other similar cloud-based systems also consist of only one single hardware device. Such a device is for instance a wireless local area network (WLAN) enabled smart power outlet. It connects via WLAN to the cloud service and can then be controlled like described previously. Both presented cases of cloud based smart home systems share one common problem: When the link between the local installed appliance and the cloud service is lost (for instance due to troubles with the Internet connection) then the automation stalls. Depending on the devices locally installed, they also suffer from those up-link connection losses.

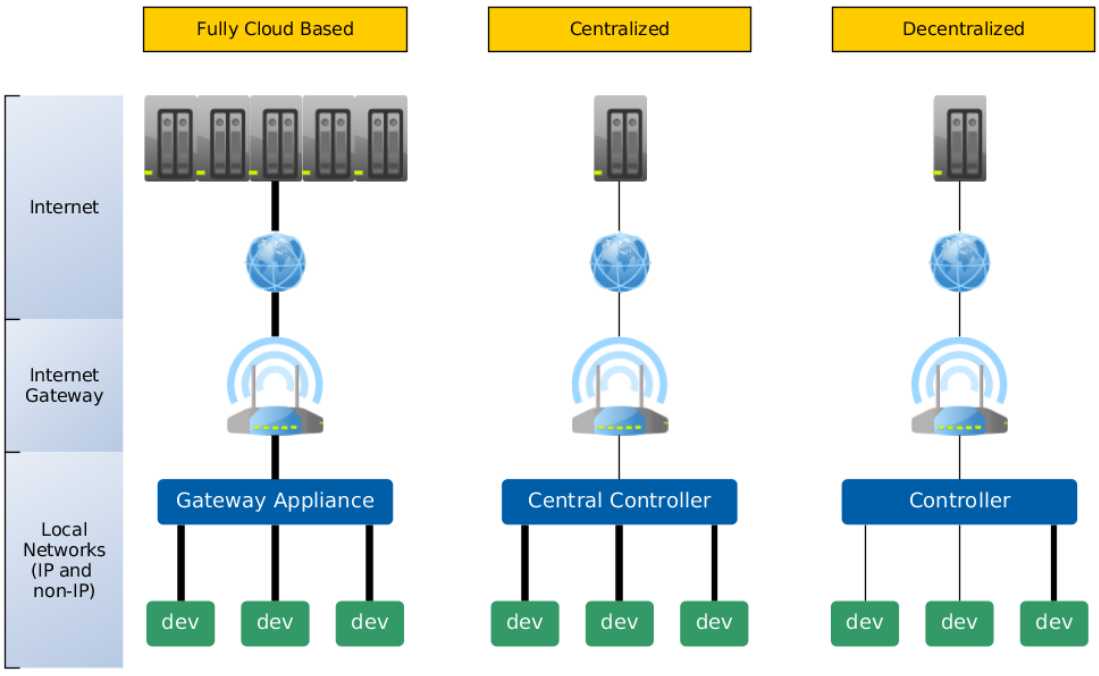

Figure 3.1: Three different types of basic smart home concepts. The line width indicates the dependency of the connection to always be available for operation.

3.1.2 Central Home Controller

Another category of smart home systems are those with a single central controller. Often devices (sensors and actors) are directly wired to a central location with the controller. All or at least most intelligence resides at that controller, meaning that a failure or outage

of it stalls the automation. In such scenarios a device connected to the single controller is useless when the connection to the central unit is lost, so this link is very important as well.

The connection to the Internet and various cloud services are auxiliary and not vital for most operations at the site. So the link to the Internet is only drawn with a fine line as shown in the middle column in figure 3.1. Central controllers can be a single hardware appliance as well as a modular (hardware) systems. Devices can be connected end-to-end or via a bus system, but they are not limited to wired connectivity at all.

3.1.3 (Fully) Decentralized

Decentralized smart home systems are not very different from centralized ones. The most significant distinction is that much more intelligence is moved from the central controller to the edge devices. Most devices can operate on their own without the need for a constant connection to the central controller. Also, multi controller scenarios are more likely than with the central controller concept explained before. The last column in figure 3.1 hardly contains any strong links. The controller still has an important role by keeping everything running together but its operation is more related to configuration and data aggregation related to itself and the connected devices. Those edge device's intelligence can still be executed on controllers in case of very simple devices. In that case a strong link still is needed between both parties. The Smart Energy Management System (SEMS) is most similar to that type of system.

3.2 Demand Side Management

A demand side management system (DSMS) can be described as a system with controlling algorithms designed for minimizing energy costs. They can do this by increasing and decreasing (mainly electrical) loads based on a given power budget on the house line or a price associated when buying or selling electrical energy. Other metrics for controlling are possible but not very common nowadays at domestic homes. All ideas of how to control consumers are based on the pricing in the end. This is true (for example) for the power grid of Austria or Germany where the availability for the energy supply is very high and power outages are seldom.

The basic idea of **DMS!** (**DMS!**) is to automatically achieve the lowest energy costs possible by using flexibilities in an energy household. Common flexibilities available today are for instance hot-water tanks or a pool with its pool pump. Hot-water tanks offer a great flexibility because of their large energy storage capacity. It does not matter when the water is heated up as long as there is always hot-water when it is needed. In the case of the pool pump a minimum runtime on a daily basis must be ensured to maintain water quality. But besides that, in general it does not really matter when the pump is turned on or off. But the conditions of various households are always different and so the amount of flexibility varies widely.

To illustrate such a DSMS operation on a daily basis figures 3.2 and 3.3 both show the PVs production, the directly used energy and the house line status over time. The

time span is one single day and is labeled on the x-axis. The values are averaged on a fifteen minute basis. Red colored areas represents energy bought from the energy supplier which is delivered on the connection to the public grid. The height represents the power. A reference value is plotted on the top left corner. Green areas are the opposite to the red ones. They indicate the energy delivered to the public grid - the sold energy. The blue areas show the energy which was produced and directly used at home. When this self usage is very high the overall system performs usually very well. Those blue areas combined with the green onces represent the energy production from the PV.

In the household on figure 3.2 all the energy produced is directly and completely used at first, no green areas are visible. But starting at about midday the usage drops. This means that the self consumption was excellent at first but then there was not enough potential to consume the rest of the available surplus energy. This could have many reasons, for instance when there is no one using hot-water during vacation or the storage capacity is not very high to name two possible causes.

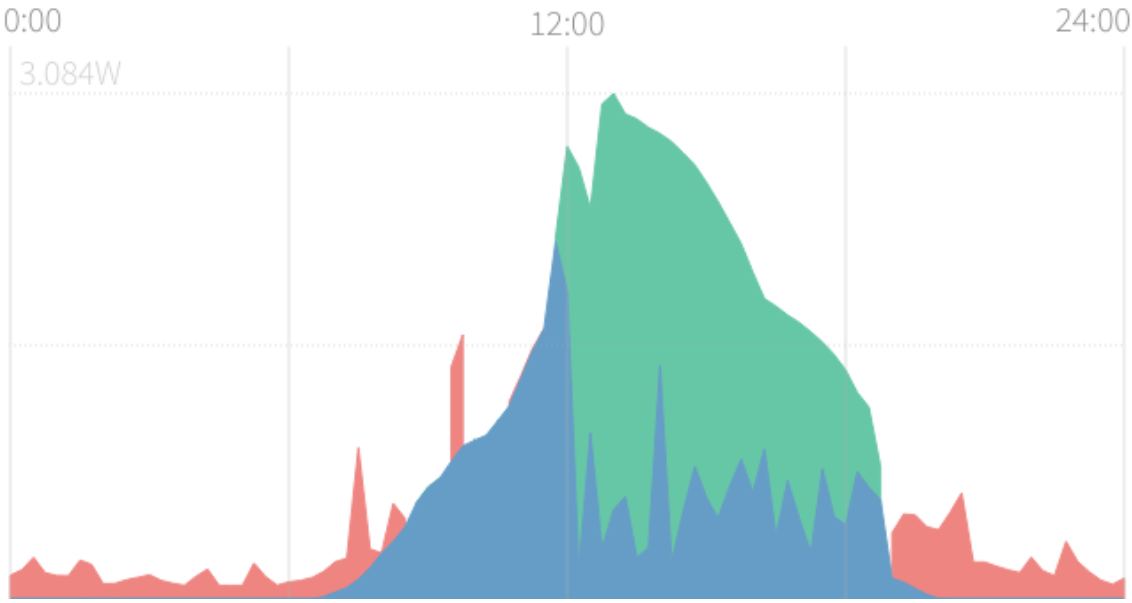

Figure 3.2: An energy household plotted over one day with too little capacity for that day (screen shot taken from a SEM).

Now when comparing the day above explained to the one as shown on figure 3.3 one important difference can be recognized immediately: Almost no green areas are visible. This household is different in multiple ways. First the PV system is differently mounted. Both charts show days with maximum production for the season. But the power distribution is not the same. In the first household the PV is mounted east and west and in the second one it is oriented south. Next, the second household has enough capacity to use all energy produced during the day. So, this system is running at an optimum: At night, hardly any energy is consumed from the public grid and during the days, hardly any is supplied to it. The self consumption is at a maximum. Optimizing this and reducing the

nightly load as much as possible is the task of a DSMS like SEMS within prosumers [1] and their PV systems.

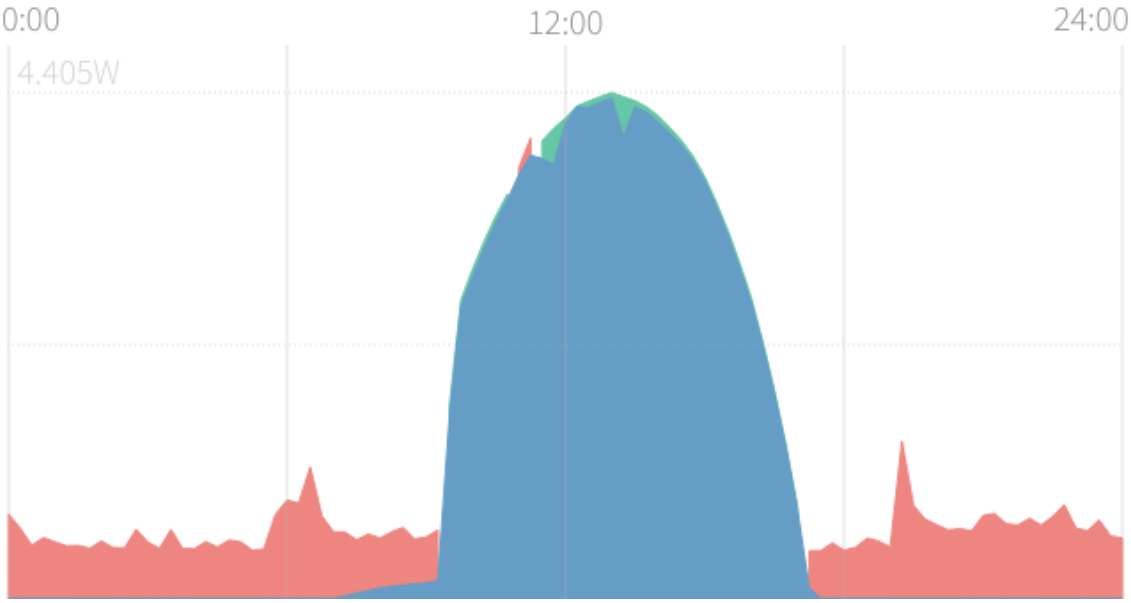

Figure 3.3: An energy household plotted over one day with optimal usage of the produced energy (screen shot taken from a SEM).

Two factors can be derived from such data. One is the self consumption rate (SCR) and the second one is the degree of autonomy (DOA). Both takes various energies over a certain period of time into account. The SCR can be defined as shown in equation 3.1. If no energy for a given time span is supplied to the public grid while something is produced, the resulting rate is at a maximum, which is 100%. In the opposite case if all produced energy is delivered to the public grid the rate is at a minimum of 0%. The directly used energy ($E_{directly\_used}$) is all the energy which is produced ($E_{produced}$) and not sold (E_{supply}).

$$SCR = \frac{E_{directly\_used}}{E_{produced}} = \frac{E_{produced} - E_{supply}}{E_{produced}} \tag{3.1}$$

The DOA represents the amount of energy which was used in total compared to the energy produced. This relationship is shown in equation 3.2. The basis for the factor which is expressed in percent is the total amount of used energy in a household. It peaks if all energy needed was produced at the very same energy household. The minimum is reached if all energy needed has to be bought from the public grid. The used energy (E_{used}) can be expressed by adding the produced ($E_{produced}$) to the bought energy (E_{bought}).

$$DOA = \frac{E_{produced}}{E_{used}} = \frac{E_{produced}}{E_{produced} + E_{bought}} \tag{3.2}$$

[1] A prosumer is a person who consumes and produces a product, in this case electrical energy.

Both equations do not contain any information about stored energy in battery systems. The longer the time spans for the calculations, the less important the influence of the stored energy. Battery systems are usually designed to store energy needed during one night [LVS15]. So when calculating the SCR or DOA for one day the impact is high in comparison to one month. When a battery systems State-of-Charge (SoC) was zero at the start of a time span and was 100% at the end, the stored energy would be taken into account as directly used. Since normally a battery system does not supply any energy to the public grid, in general this would be correct even if the energy has not been really used at the end of the time span. It would be fine in context of the SCR. When considering the DOA the stored energy could be taken into account as a reduction of the used energy which leads to a higher DOA, even over 100%. But since battery systems are only shifting energy over an amount of time this usually can be completely ignored for the calculations.

When dealing with power grids with a lesser general availability of the energy supply compared to Central Europe, then load management is an even more important term. Here the controlling algorithms are the enabler for a stable power grid to achieve higher availability for a single energy household or a whole region, especially when considering electric cars charging stations which are getting more and more every year.

3.3 Public Key Infrastructure

The PKIs are commonly used in today's technologies. The presence of a working PKI in the background is often not recognized by users if there are no verification errors. Two examples of PKI usages are secure web pages and Apps on mobile platforms. In both scenarios the user of an end device can trust a website or an App for being from a trusted source. The following subsection explains those use cases in more detail.

Basically, a PKI is a system for managing digital certificates. This includes procedures, roles and policies to be able to manage and use a PKI. Certificates are usually signed by other ("higher") ones which results in a hierarchy of certificates. On top of such a system is the root certificate which is self-signed. Commonly root certificates are known and trusted. All certificates which are signed by the associated private key are then trusted as well. Multiple layers of certificates signing one another result in so-called certificate chains. If something is signed by a leaf certificate's key, the complete chain of certificates must be provided to be able to determine if the chain and the final signature are valid.

If the provided signatures are correct it still depends on a client if it trusts the root certificate which backs the complete chain of trust. Depending on the use case where the PKI is used, the question of trusting various root certificates can be easy or rather complex as explained in the next subsection. When dealing with embedded systems there is often only one root certificate which can be trusted. Often software which is delivered via an update system or other important information needs to be verified and there is only one source providing those things. Therefore, often only one Certification Authority (CA) with its associated root certificate is used. Generally, at least one intermediate layer for security reasons often exists which issues all certificates for doing the actual signing work

of various things like software. If the private key of a root certificate leaks, the only real possibility to react is to distrust the complete CA. This is a real problem when only one CA exists and the entire security depends on it. So, protecting the root key at all costs is essential. Therefore, intermediate certificates are used, because revoking them is possible if for instance a system got hacked which had a copy of the private key.

3.3.1 Example Usages for PKIs

There are two use cases of PKIs which nearly everyone uses every day. One is secure web browsing and the second is installing and using various Apps on mobile platforms. Normally users of clients (a web browser or a mobile operating system (OS)) are only notified if a signature verification fails or the root certificate is untrusted. Depending on the client's policies the user can ignore the error and continue.

In context of secure web browsing, various trusted root certificates are used to determine if a certificate offered by a web server can be trusted or not. If the root certificate is trusted and all signatures are correct, the browser accepts the connection or otherwise reports an error. It works similar for the Apps on mobile platforms. Usually the mobile device does not allow installations of Apps which are not properly signed by an issued certificate from the central App-CA. Developer options sometimes allow to by-pass this security policy.

The main difference between both use cases is the number of trusted CAs. While using web browsers a big number of CAs is present which all are issuing certificates to various parties. All of those associated root certificates need to be trusted by the browser. A standard set of root certificates is usually maintained by the web browsers developer or maintainers. They have guidelines whether a CA is trusted or not. This is due to the fact that hardly any users of web browsers know the impact of trusting or distrusting CAs. To summarize that, there are a lot of CAs issuing certificates which are trusted or not due to some policies instated by the developers, the backing businesses or maintainers. Since there is more than one web browser vendor, this process of selecting trusted CAs needs to be done by all of them which might lead to different trusted CAs within common browsers.

When dealing with a big mobile OS where there is exactly one organization issuing certificates the number of trusted CA is mostly one. If there are multiple CAs within a single organization the management of trusted parties is also quite simple. The vendor of the mobile platform issues certificates to various developers which then can sign their Apps in order to be able to publish them. When everything is signed and published correctly other users can install the Apps and since they are signed by a trusted party the mobile device will allow it and the user can use them. Besides this software signing on mobile platforms, similar signing takes place on desktop and server OSes. The packet manager or installer can only install properly signed software by default.

3.3.2 Issuing Certificates

A CA can issue certificates. This happens by signing a Certificate Signing Request (CSR) with the signer's private key. The CSR is derived from the private key and some meta

data like the common name is added. After adding some meta data from the CA itself to the CSR the signing can take place and the certificate can be derived. This way the signing authority does not know the private key. The CA itself saves some data as well like the serial number for itself for tracking purposes. The top level CA is self-signed. Intermediate CAs are signed by the root certificate's key or other intermediate ones. While signing various flags can be set in the certificate. This includes usages for the key. This way leaf certificates cannot be used as a CA. A leaf certificate is normally used for web servers or code signing and represents the end of a certificate chain.

Besides the usage information, a start and end date for the certificates are set. So, the time in which certificates are valid is limited. Besides the expiration date, certificates also can be revoked. See subsection 3.3.3 for that.

3.3.3 Certificate Revocation

Certificates are issued as described before and they also can be revoked by the issuing CA if needed. This happens for instance when a private key leaks. In such cases the CA revokes the associated certificate and publishes the information in its certificate revocation list (CRL). The clients have to check the CRL too to determine if the certificate is still valid regardless of is expiration date. In the certificate the location of the CRL is included. When using a web browser this additional check can be easily performed since the system is usually connected to the Internet when surfing the web. But for embedded systems which are not always connected to the Internet this task is not trivial or sometimes even impossible. For instance when signed software resides on a USB media which could be plugged-in during start-up for side-loading purposes, the revocation information might be not accessible at all.

An alternative to CRLs is Online Certificate Status Protocol (OCSP). It's on a request/response basis like Hypertext Transfer Protocol (HTTP). Compared to CRLs less data is transported during a request and parsing the data is easier which makes the clients less complex. The answering party for OCSP is usually called OCSP responder and knowns based on the requests which host used a certificate at what time.

3.4 Security of Embedded Devices

Attacks can be divided in various ways [RHM+12]. This work puts them into three categories. Those include attacks via executing malicious code which is distributed without direct system access. Next are attacks over the network which exploit some weakness by buffer overflows or other flaws. Lastly, an attacker could have direct physical access which allows them to modify the hardware directly to get access. Figure 3.4 visually describes those three categories.

3.4.1 Foreign Code Execution

Without direct system access another way can be found to get the target system to do something in the attack's favor. This is usually injecting a malicious piece of code or

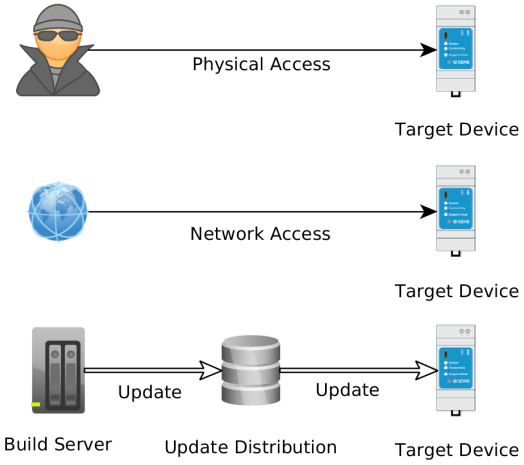

Figure 3.4: Different attack schemes on embedded systems.

software on some remote location in perspective of the target. A possible way might be to get access to the update infrastructure and mess with the updates.

Execution of the foreign software can be prevented even in the case of a hacked update system by signing the updates directly after building. This way the target system can check the integrity of the provided software. Basically, this means that as long the update is not corrupted before signing any manipulation can be detected. As a result, the build process before signing the update must be secured as well.

The management of the used PKI is critical to ensure security. A concept for issuing and using certificates and their associated keys needs to be created to avoid flaws in the signing chains. This is described in more detail at subsections 3.3 and 4.8.3. But one thing need to be added here: The target system also needs information about revoked certificates to be able to detect signatures from keys which were stolen and therefore revoked by the CA. So only determining the correctness of the signature for signed software is not enough, the chain and its revocation information need to be considered as well.

A very interesting topic for protecting a device should be named in association with not having any direct access to a system is social engineering. With the help of an user an action in favor of the attacker can be triggered. Besides that information could also be retrieved.

3.4.2 Exploit

When dealing with systems connected to a network or the Internet exploits can be used to get access to a device. In the worst case even root access can be gained, which is unrestricted access. Often those terms are used when talking about servers and securing them. It is very similar with embedded systems. A distinction should be made at this point

for the network access capabilities. First, there is access through the transport protocols UDP and TCP which might be allowed by some firewall rules if any firewall is present at all. So, the attacker in this case is at least more than one hop away from a network routing perspective. Second, there is being in the same network as the target system. Attacks over other protocols can be applied. Exploits are not limited to network access. They also might be used at other but local interfaces. More about that in the next section.

The first type of access with transport protocols can also occur when being in the same network. So, in the local network (Ethernet or WLAN) the threat is bigger because more is possible. This includes not having a dedicated firewall or intrusion detection system (IDS)/intrusion prevention system (IPS) between the target and the attacker. Often the term first hop security is used in this context.

Constantly applying security patches when vulnerabilities are discovered is an important measure to protect the system. When designing a product, the time span of support is very essential for security.

3.4.3 Direct System Access

When dealing with the threat of direct system access a distinction can be made between access which does not require the breaking of any seals on the device like USB ports and full access to all internals of the target. Normally a product is secured pretty well when focusing on those external interfaces because anyone like customers buying the product could have access to them. Also, it is often easy to protect the system in this case due to the fact that only wanted interfaces are accessible from outside.

But inside the housing the situation gets far more complicated. Since inside the device there are a lot of needed internal interfaces i.e. for connecting up various hardware components, protecting them can be a very hard task. Interfaces needed for initial flashing or simple an embedded memory card could be exposed on the PCB which often easily can be accessed when the housing is removed partly or in full.

When having access to the bootloader inside the device, it is possible to boot from another medium depending on the interface support of the hardware and the bootloader's capabilities (or configuration). At least if there are no measures taken against it. With another system booted or some configuration files changed it is possible to get root access to the device and do nearly anything an attacker wants. Also getting some sensitive information like credentials is a big threat. This includes the device's identity or saved passwords for cloud services and similar local services. Therefore, much asymmetric cryptographic should be used to reduce the possible damage when a system breach is not too hard for an attacker with unlimited physical access.

One possibility to avoid this is encrypting and/or sealing the device. With mechanisms like High Assurance Boot (HAB) it is possible to boot only a bootloader and a Linux kernel which are correctly signed. The signature is checked against a certificate burned on the processors internal one-time programmable non-volatile memory (OTP NVM). If sensitive information on the device must be protected at all cost a full system encryption must

be considered. This type of technique is generally known as secure boot in association with a Trusted Platform Module (TPM) as it is present on modern PCs.

When accessing the rootfs there is also the risk of finding possible exploits in association with network access which could not be detected before without direct access. So to summarize this section, there is very much work associated with protecting a device against direct access to its internals. Also, the hardware must provide various security features like HAB or something similar to be able to properly protect the system.

3.5 Updates on Mobile Platforms

Mobile phones with their various platforms are in some manners very similar to embedded system development. Thus, when designing an update process for custom systems, it is a good idea to look at mobile platforms. A distinction between system update and application update can also be made on mobile platforms. Apps are usually updated independently from each other and from the system. Developers do not have to care about update installation and distribution much here.

Apps are signed by the developers when building them with a key whose corresponding certificate is signed by the platform's central CA. Manifests provide information about compatibility and capabilities. App developers are not concerned with system updates at all. These have to be taken care of by the various device manufacturers. Depending on the platform, the update processes can be examined or not.

By contrast, in this work everything has to be done by one party due to the full stack nature of the product and its software. The update process, both system and application, have to be crash safe. Otherwise, after failed updates the product might not work at all and it would lead to a support case which can be very expensive.

Chapter 4

Concept

Previously to this work most of the hardware design for the target system was done and therefore hardware related requirements are present. This chapter starts with describing the requirements. The tools already provided and selected close the first part of this chapter. Further on, the concept of the work to be done to achieve the goal is presented in detail.

4.1 Requirements for the Target System

The Linux kernel was provided for the SoM. Driver selection, changes and modifications had to be done for the custom hardware itself where the SoM is only a part of it. The kernel and rootfs must support crash safe updating of the system (kernel and rootfs) as well as the application running on top of it.

The concepts created here for doing so should be applicable to other target systems as well. So, a general solution must be achieved. The update process itself must be as convenient as possible for the end user. Initial flashing of the finished products as well as provisioning them while manufacturing must be as simple as possible and should require as little additional hardware as possible.

The security of the system is vital so applying security patches to the used software in the rootfs or the kernel must be easy. Another aspect is to keep the system always running so mechanisms must be in place to detect hangs of any kind to avoid downtime as well as possible.

To insure low hardware costs, the new rootfs should be as small as possible. Unified access to hardware devices from the application running on top is a big advantage but it is no hard requirement. Application development for the target system should be pain free for developers creating it.

CPU	1GHz Allwinner R8 ARMv7 Cortex-A8
RAM	256MB DDR3
Storage	512MB SLC NAND
Connectivity	WiFi B/G/N & BT4.2
Certification	full

Table 4.1: Specification of the CHIP Pro (an excerpt).

4.2 Hardware Target

The target device is based on a SoM called "CHIP Pro". It is shown in figure 4.1. With its initial price of 16$ regardless of the quantity it is an outstanding SoM when comparing the price to the specifications. In addition to its built-in communications a custom UI for the housing and a Z-Wave module was needed. Therefore, electronics consisting of multiple PCBs was needed.

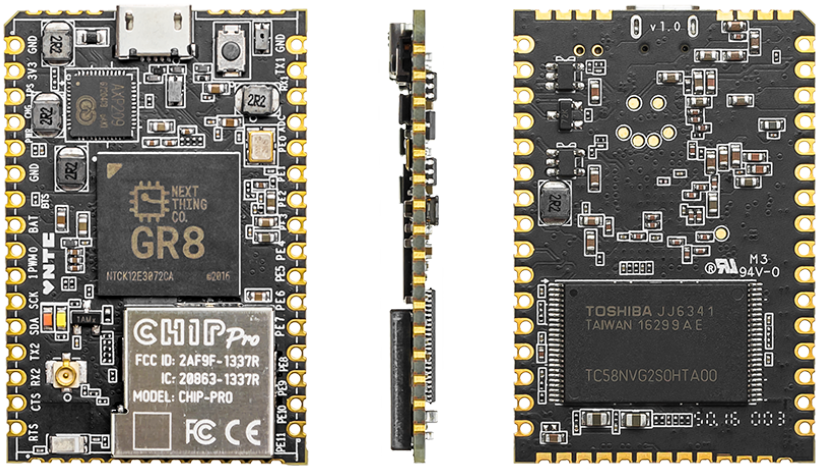

Figure 4.1: The heart of the target system: The CHIP Pro SoM.

Table 4.1 summarizes the most significant specifications for the CHIP pro. As stated in the last row, the module itself is fully certified. The single-level cell (SLC) NAND is an endurance non-volatile memory.

As mentioned before, the complete electronic assembly consists of multiple custom made PCBs and various components on it. A block diagram of the most basic components is explained later in section 4.4.1 on page 22.

Figure 4.2 shows the three custom boards side by side in a non-connected state. The left one is the expansion board with the exposed UI and the ZM5304, a Z-Wave module connected via universal asynchronous receiver-transmitter (UART) and one general-purpose input/output (GPIO) as reset line for resetting the module if needed. The UI consists of one USB On-The-Go (OTG) connector, two buttons with different heights and three

RGB-LEDs. The higher button is used for normal interactions with the user, the lower one is only for rebooting or hard-resetting the complete electronics.

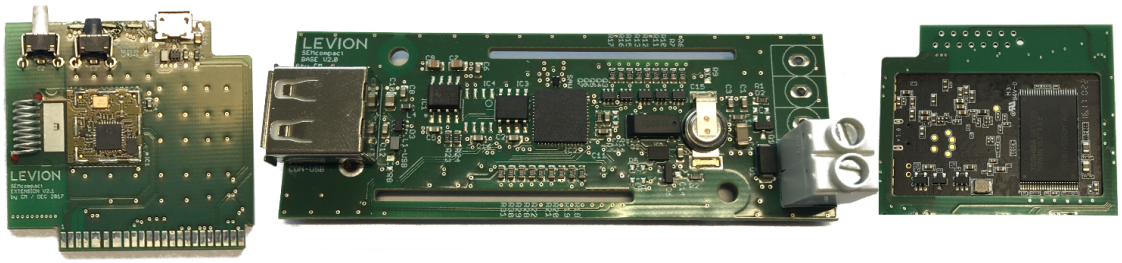

Figure 4.2: The three custom boards used: Communications and UI, base board and the CPU carrier board already assembled with the SoM.

The center PCB is the base board with one USB-A connector, the RTC with its big capacitor, the USB control and the power supply terminals. The last PCB is called the CPU board. It only holds the CHIP Pro, a debug header and a hardware identification.

In the last step of manufacturing the parts are soldered together. The complete assembly is shown in figure 4.3.

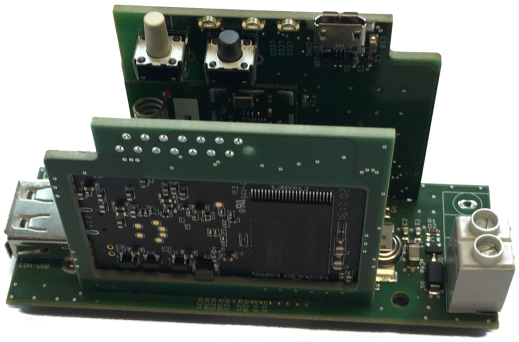

Figure 4.3: The complete electronic assembly for the Compact Unbranded.

4.3 Provided Tools And Software

As mentioned in section 4.1 there was already a Linux system for the CHIP Pro. More precise, all driver work was done for the Linux kernel and a rootfs based on Buildroot was available. In addition, all tools for bringing the system on the CHIP Pro were of course provided by the manufacturer. Buildroot was not very practical due to the need to manually integrate the needed software for the rootfs. Luckily, within the Yocto project, a so called meta layer was present for the used SoM. The layer is called meta-chip. Yocto as a build system was more suitable due to the recipe and layer structure for maintaining software and scripts for multiple hardware targets. Yocto is used very often nowadays by

various manufactures of various modules. See the next section for more information about that.

4.3.1 Building the System

The YP is an open source collaboration project that helps developers create custom Linux-based systems for embedded products, regardless of the hardware architecture [Pro18b]. The YP is a huge umbrella project housing the build engine OpenEmbedded with its meta data for embedded Linux and a reference distribution called Poky. Figure 4.4 summarizes that. With its abstractions Yocto is the perfect framework to develop a system with the desired properties for the use on different hardware targets.

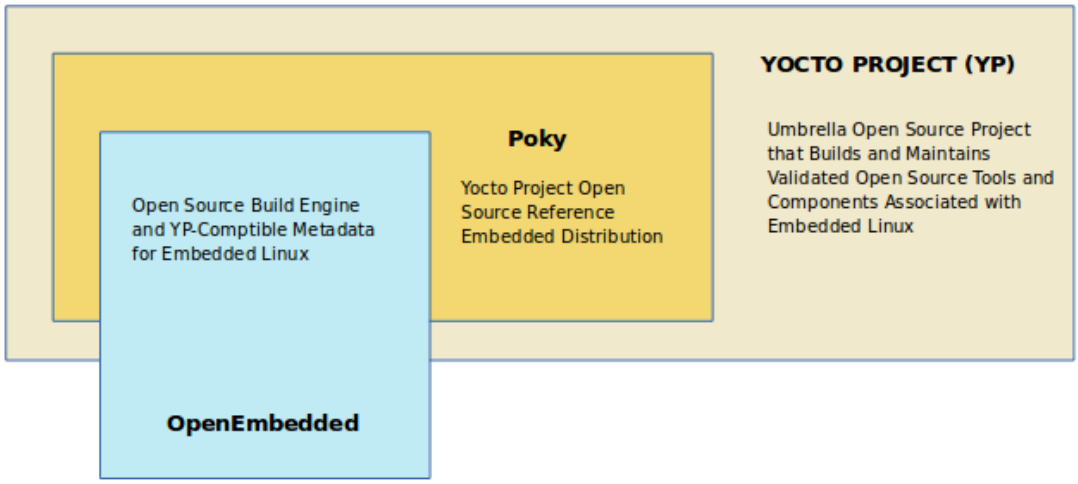

Figure 4.4: Base components in the YP [Pro18a].

The meta data consists of layers which have recipes and various configurations. Usually the layers' names start with "meta-" like "meta-sems" or "meta-qt5". New layers are required for handling on the one hand all common and hardware-independent, update-related recipes and on the other hand all recipes and configurations used for modifying and extending existing recipes for being able to run on the target hardware.

The layer for the common recipes will be call "meta-sems" and the second one specific for the hardware "meat-sems-chippro".

4.3.2 Toolchain for Application Development

The application running on top of the Linux system is based on Qt5 and its build process is managed by cmake. A software development kit (SDK) containing all tools and files needed for cross compiling the application must be provided by the build system used to build the bootloader, the kernel and the system itself.

The meta-qt5 layer contains all recipes and configuration not only for building the libraries for the target system with Yocto but also provides the target meta-toolchain-qt5

which builds the wanted SDK. Only one modification must be done for including also all needed cmake tools in the SDK.

4.3.3 Kernel Patches

One goal is to easily apply security patches to the kernel in the case of revealed security risks. The kernel itself does not need many modifications, so integrating security patches later on should not be a problem at all. It is even easier when the vendor of the SoM include them into a Yocto layer.

4.4 Customizing Linux for the Hardware Target

This section summarizes the changes needed to be made to achieve the desired goals for the target hardware. As stated in the previous sections, for the CHIP Pro, our SoM, most work was already done regarding kernel and device drivers support. Still some work had to be done to support our update process and the hardware devices on the custom boards.

4.4.1 Kernel Drivers

The first step is to select the needed kernel drivers for the hardware by configuring the kernel. Figure 4.5 demonstrates an overview of the hardware blocks which needed to be considered. Those blocks are located on the three boards: CPU, Base and Extension. On the CPU board there is the CHIP Pro, its own power management IC (PMIC) and the hardware ID, which is not shown here. This ID is just for assembled or not assembled pull down resistors for reading in the used custom hardware revision. The PMIC (a AXP209) is located on the CHIP pro but because of the so called PEK signal and its importance it is drawn separately. The AXP209 is connected besides other signals via an I2C bus. The status of the PEK signal is read from a register using I2C. The system is rebooted on short presses. On long presses the AXP209 shuts down immediately.

The base board hosts the control circuit for en- and disabling the USB interfaces and the RTC with its crystal and large capacitor. The RTC is connected via another I2C bus. Controlling the USB ports is done only by two GPIOs. This board also connects the CPU board with the extension board. Last but not least the extension board carriers the ZM5304, the used Z-Wave module and the UI with the USB OTG connector. The ZM5304 only needs a serial interface with one GPIO, no special driver is needed from a kernels perspective besides the standard serial driver. The UI has nine LED channels and two buttons in total. One is directly connected to the SoM and the other to the PMIC to be able to perform a hard reset if the system becomes unresponsive.

For all devices described above there were already drivers available. So, most work for the kernel here was to choose the correct position for the kernel modules: built-in or loadable from the rootfs. Mapping where the kernel finds which device is done by the device-tree. The only modification of a kernel driver is for the sun-xi watchdog on the CPU. This is needed so that the kernel does not stop it while initializing the device drivers during booting. The bootloader starts it to detect boot problems and if the driver shuts

down the watchdog it might get missed and the system does not get unstuck after possible hangs during boot.

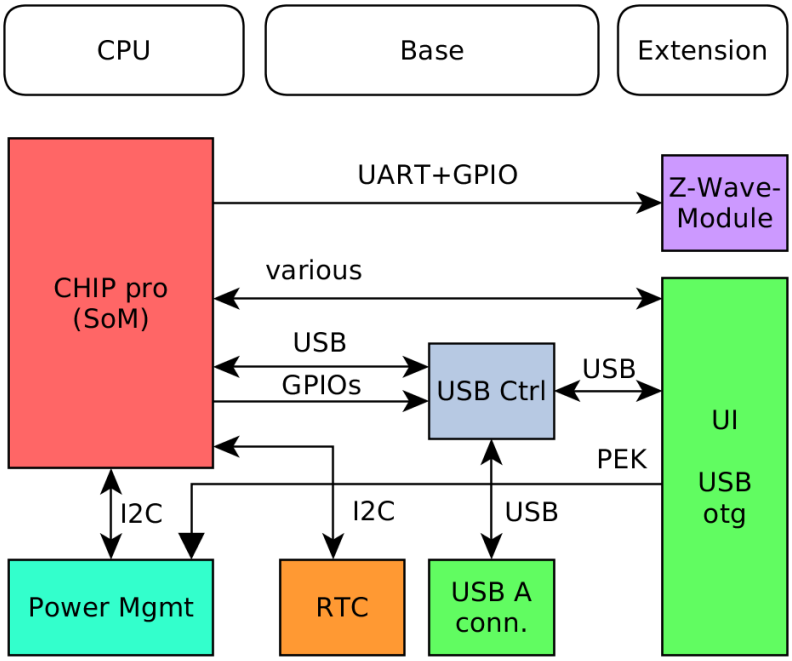

Figure 4.5: Block diagram for the given target hardware. The top labels indicating on which PCB the components are placed on.

4.4.2 Device-Tree

The device-tree is a description of the hardware where the kernel or the bootloader are running on. This is needed in embedded systems running the Linux kernel and associated bootloaders like U-Boot and barebox to get the information needed to run.

Those information contains sizes of the memories, CPU configurations, where to find devices, which drivers to load and so on. On classic PCs this information is provided by the Basic Input/Output System (BIOS). Before the device-tree was introduced this information was directly compiled into the Linux kernel in the well known board.c files. Also the initialization sequence was coded there. The big advantage of the device-tree systems is now that the same kernel can be used with different boards without the need for recompilation due to changed devices. This is achieved by providing the compiled device-tree to the kernel by the bootloader during boot. It is also possible to change the device-tree while the bootloader is active to dynamically decide about the use of various devices.

The device-tree file(s) must be written correctly to support all devices as shown in 4.5 and described in section 4.4. The device-tree system offers includes so that the description

of the hardware can be divided into different files. A simplified and general visualization of this is shown in figure 4.6: How many layers and includes a specific system or architecture has, various widely. For instance the CPU family could be NXPs i.MX6 CPU and various vendors offering SoMs include the base CPU dtsi-file in their own dtsi-files which then are used by their reference boards dts-files or by there customers for their boards. For large CPU families like the i.MX6 there are a lot of dtsi-files for the different variants of the CPU available on the market.

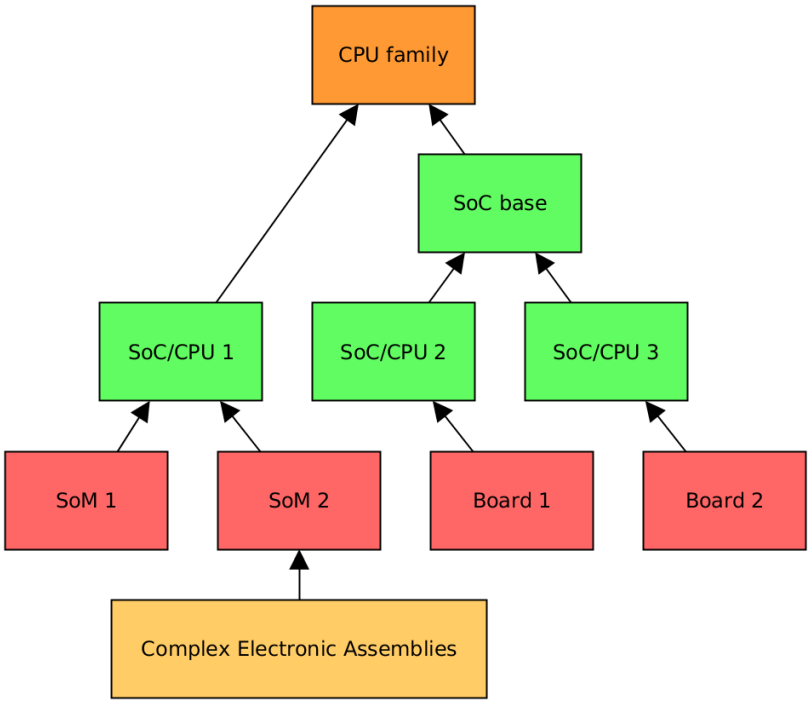

Figure 4.6: Simple representation of how device-tree includes work. Higher blocks (.dtsi-files) are used by bottom board blocks (.dts-files).

4.5 Boot Process

Booting a Linux system is not a challenge besides hardware related work. If all driver work is done there are normally no troubles firing up the hardware because of the previous work already done by the hardware manufactures. Would they not do this initial hardware integration for the Linux kernel for their products they sell, hardly any customers would buy them. Since the target is a custom hardware beside the SoM some driver related work needed to be done as described in the previous section.

To comply with the requirements, the boot process must be altered in a way that it supports a reliable and crash-safe method for updating both the system with its kernel and the application. Since the application update does only have to do with the rootfs to

provide services and scripts to perform the update, it does not affect the boot process itself.

Therefore, this section focuses only on the system update and all related tasks to perform to be able to achieve our goals. Independent from the components a general rule is that everything which is updated or written in the process needs to be redundant. Otherwise when the system goes down unexpectedly, the system might not be boot-able again and this possibly would mean that the system has to be replaced. Such support cases must be avoidable regardless of what the end user might have done.

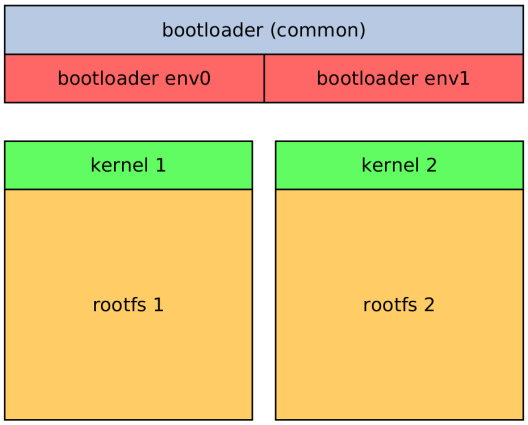

Figure 4.7: Dual boot system for safe system updates - a logical schema.

Figure 4.7 shows in a simplified way what needs to be redundant. The bootloader itself is hardly updateable generally. To be able to still update, it a secondary bootloader which is redundant as well would be required. For the target system the "u-boot" bootloader was already provided. It has to decide which kernel it has to load. Such information can be saved in its environment. Since it must be writeable to tell the bootloader after an update that is should now use the other system, the environment must be redundant.

To safely prepare the new system during an update it must not interfere with the currently running system at all. So, both the kernel and the rootfs must be redundant like the bootloader environment. This way the update process can always be interrupted, and the old system still would normally boot up. After the new system is completely prepared, the running system tells the bootloader via its environment that it should try the new system next time. After this step the last action to take is to reboot the system.

The following subsections explain the boot steps in more detail.

4.5.1 Bootloader

Our used bootloader (U-Boot) already has support for handling multiple environments. It has to be activated in the configuration needed for compilation. This also applies to

the tools for reading and writing the environment from a running Linux system. With this support enabled the environment is safe against sudden power losses. Recovering a possible damaged environment still must be handled.

4.5.1.1 Handling Redundant Systems and their Update

The bootloader has two tasks to perform every boot. One is the check if both environments are valid because one might have already crashed and in the case of performing an update, a second unexpected reset can also crash the only healthy environment, which leads to unknown behavior. The second task is to select the correct system to boot up.

In the left part of figure 4.8, there is a flow chart for loading the correct environment and restoring a possible damaged one. In the case that both environments are fine, no writing action is performed at all. This is the green path in the flow chart. If one of them is damaged, it is restored with the contents of the loaded, healthy one, which is shown as blue paths.

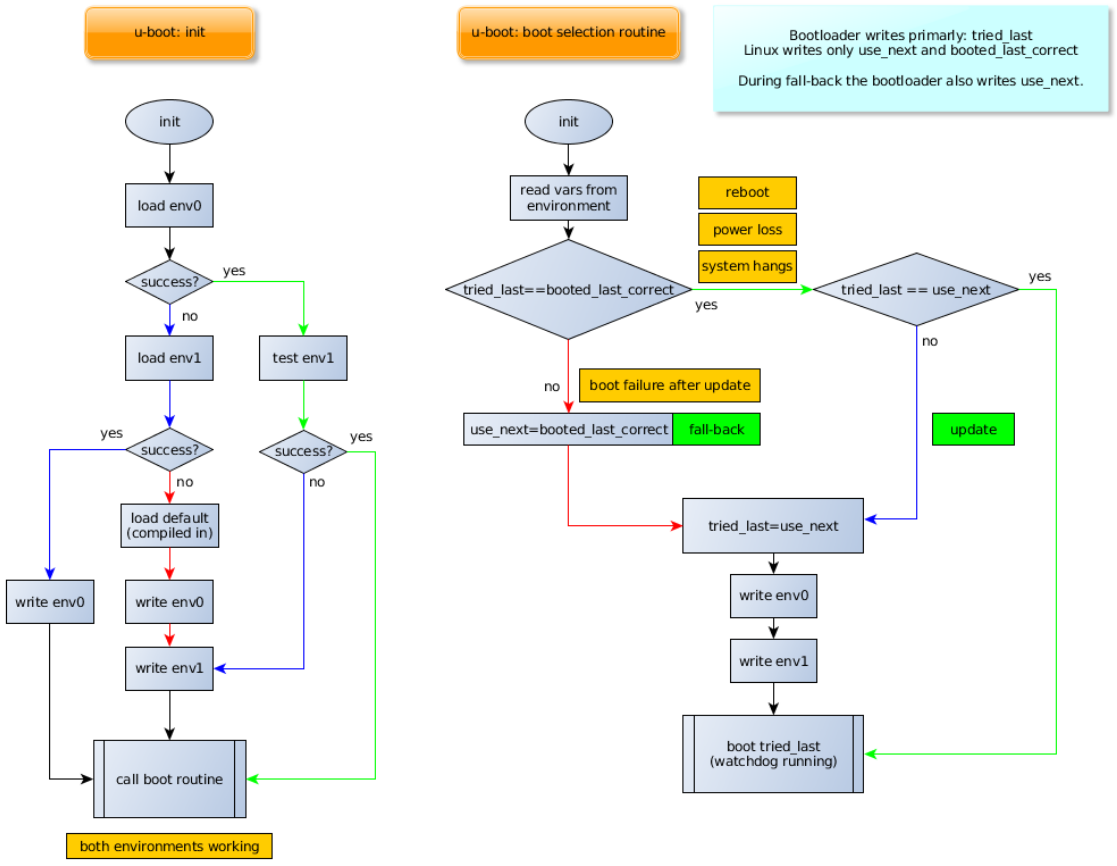

Figure 4.8: Environment loading/checking and system selection flow chart.

The red path indicates that both environments could not be read. This should never happen with the measures taken as already explained in this subsection. Still, it is possible that there are hardware faults associated with the used NAND memory. But this is very unlikely due to the fact that this environment is only written when performing an update. But still there is one case in which both environments are not readable and this is during the first ever boot up of the bootloader after initial flashing. In that case, the default environment is loaded, which is compiled into to bootloader binary.

Which system (first or second) the bootloader should try to start was determined by the previously running system. It only updates this information during updates. In addition to that the bootloader itself must keep track of what it has done on previous boots to be able to detect errors during boot and perform a fallback at the next start-up.

A summary of the selection process is drawn in figure 4.8 on the right side. After reading the variables they are analyzed in the following ways: If the last tried system is the same as the last system which was correctly booted, there was no problem with an update. If the last tried one is also the same as the next to use, no update is currently in process. This is shown in the chart as green path.

During an update (when correctly booted before) the use_next variable is updated by the running Linux system and this is detected by U-Boot - shown as blue path. The tried_last variable is now updated by U-Boot and the information must be preserved and therefore a write of the environment happens.

In the case that after an update the kernel or the complete system failed to boot anyhow it is detected by mismatching tried_last and booted_last_correct variables. The red path is executed and the use_next is restored as a fallback action. Like before the system to use changed and it has to be written together with the new tried_last value to the NAND memory for keeping the information until the next power-up and further on.

The selection process has one weakness: After an update when the system would be booted correctly but the start is interrupted somehow (i.e. power outage), a fallback action is always triggered. The complete update has to be started again. This is no problem at all because the system always will boot into a working system. This can only be done this way due to the lack of the reset cause of the system while executing the bootloader. In other architectures the information why the system was reset is provided and with that a power failure or watchdog reset can be distinguished. Based on that information the fallback action can be postponed and the new system can be tried again without the need for rerunning the complete system update.

At last it should be stated that the bootloader does not know anything about the rootfs. It only prepares the device-tree and starts the selected kernel based on the described flow. It is the kernel's responsibility to find the roofs and mount it.

4.5.1.2 Resulting Layout of the NAND Memory

Based on figures 4.7 and 4.8 and the explanations associated with them, the resulting layout of the NAND memory is shown in figure 4.9.

Figure 4.9: Principle used NAND layout. For details see table 5.1

The chosen layout is located directly used on the NAND. The bootloader and its redundant environments can only be located there. For flexibility, the remaining space is used for all kernels and rootfs images dynamically. It is formatted with Unsorted Block Images (UBI), which handles wear-leveling and volume management. On top of UBI one single volume is used, which is UBIFS formatted. The used bootloader can create UBI and its volume and can read from it during boot. The exact contents of the so-called image store are described in section 4.5.4 later on.

4.5.2 Kernel Startup

The kernel start-up is very straight-forward regarding the redundant systems present on the persistent memory. A normal device-tree enabled boot is performed. The device-tree is preloaded like the kernel prior to kernel execution. Besides the kernel, the bootloader must also be device-tree-enabled to perform this boot style.

Because of the presence of the UBIFS, which does not directly contain the rootfs, the kernel cannot directly mount it with its default implementations. Therefore, a common technique is used: the initramfs. It is a minimal Linux system very tiny in size which is loaded as the initial rootfs. It is simply configured in the kernel configuration. The initramfs can be preloaded like the device-tree prior to booting the kernel or it can be embedded in the kernel image itself.

Since the kernel size does not really matter when stored in the UBIFS it was embedded to make the updating process of the whole system easier. Otherwise, the initramfs must be updated parallel to the kernel and the rootfs as well, which would increase complexity of the boot and update processes.

Some boot arguments are given by the bootloader but those are not very special. The image store itself is given as the rootfs, so the initramfs does not need to mount it separately during start-up. Figure 4.10 shows both the kernel being stored inside the image store and the principle partitioning of one of these images into the kernel itself and the embedded initramfs.

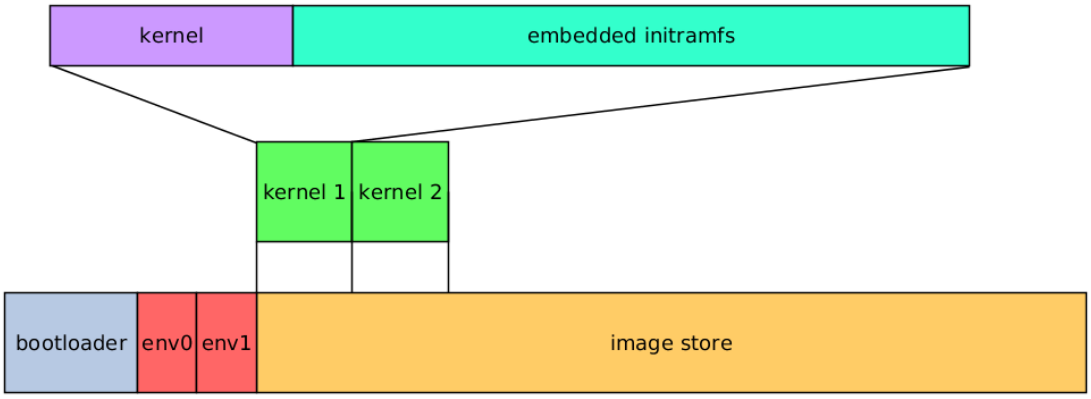

Figure 4.10: Kernel image contents: The kernel, saved at the UBIFS formatted image store has its own embedded initramfs.

4.5.3 Initramfs

In a normal Linux kernel boot process, the bootloader tells the kernel via the boot arguments where to find its rootfs. It can be on a NAND or NOR storage, a hard disk drive (HDD), a SD memory card or even on the network as a network file system (NFS) drive. In all these cases, the kernel can handle mounting the rootfs directly. Since the mount cannot be handled by these built-in routines with the chosen location, an initramfs has to be used, whose main goal is to find and mount the rootfs, then it carries on with the normal kernel boot procedure.

The initramfs is basically a tiny Linux rootfs with an init-script or a program and all files needed for executing it. Those typically are the various commands and the libraries used in such scripts. In addition, a minimal shell during development is helpful. The tasks of the used script are the following:

1. Prepare for mounting the rootfs.
2. Derive the filename for the rootfs based on the kernel name.
3. Mount the rootfs, which is a single squashfs formatted file located in the image store, as shown in figure 4.11.
4. Tell the kernel to swap the initramfs with the real rootfs and call the init of the finial file system.

It is a simple sequence of commands, which need to be executed to perform these tasks. The squashfs is a read-only file system which has high compression capabilities. See the next section for a more detailed explanation of it. The initramfs itself is a compressed cpio-archive, which is provided for the kernel build in a pre-built format.

4.5.4 Image Store

The major part of the NAND storage is used as one single UBI volume formatted with UBIFS. It houses various images, which are single files formatted in a way that fits their

purpose. Besides these single file images, one directory used by the bootloader is present containing the device-tree needed for the kernel boot. One file system containing all images has the advantage that all images can vary in size over time without any problems if the total space needed is not too large. This could not be achieved with partition on the NAND directly. A fixed layout would be present over the entire lifetime of the hardware. The reserved space for the images could be too little or too large. One other way possible is to use multiple volumes in UBI but this increases the complexity when updating the images.

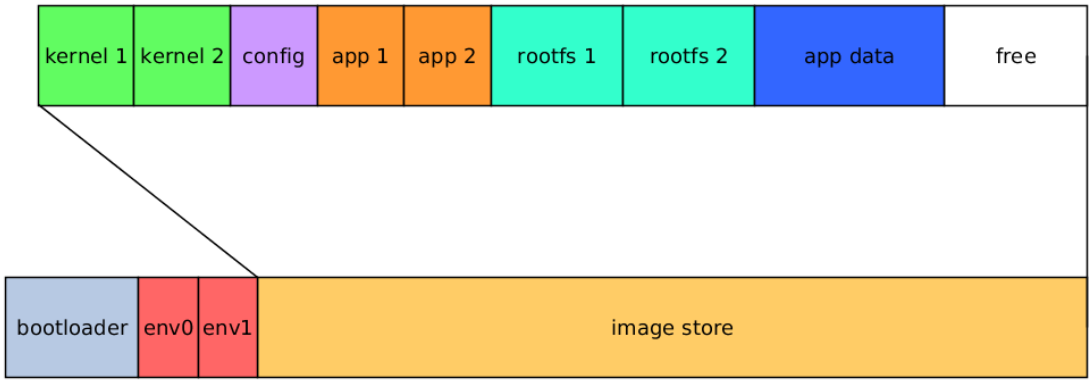

Figure 4.11: Image store contents.

In figure 4.11, all image files are displayed. Those images are

1. First kernel (active or used for updating)
2. Second kernel (active or used for updating)
3. The configuration container "conf" formatted with ext4
4. First application container (squashfs)
5. Second application container (squashfs)
6. A rootfs belonging to the first present kernel
7. Another rootfs belonging to the second kernel
8. Data storage for the application (ext4)

All the needed redundancy shown in figure 4.7 is present here. All the bootloader-related storage spaces are directly on the NAND (with exception of the device-tree), everything else in the image store. Loading the kernel is the responsibility of the bootloader, finding and mounting the rootfs is those of the kernel.

The free space is very important for updating. The new system file is downloaded and extracted before removing the old rootfs container. Because of this, the application data

is also fixed at one size to predict the available space during system updates.

Squashfs is mainly used for embedded devices with only a few megabytes of persistent memory like wireless routers. Since the rootfs should be crash-safe, making it read-only is vital to achieve this. Due to this possibility and squashfs featuring various compression methods it was chosen for the target system not only for the rootfs but also for the application containers. The resulting size when using squashfs is only a third when compared to ext4.

4.6 Image Management in Linux

All images menioned in the section above are managed by the running OS. The Linux system itself mounts the "conf" container automatically. This contains provisioning data including the hardware's serial number, among other things. Dispatching which application to launch is handled in a system service script. The application data is opened by the application. The complete image store is mounted in Linux at /mnt/images.

During application and system updates, the inactive containers inside the image store are replaced. Since everything updateable is redundant, it is safe to so.

4.6.1 Root Filesystem

The rootfs is read-only to avoid unwanted changes during runtime. Squashfs is used for this. Its characteristics are explained in section 4.5.4. Further on nothing can be saved there, a few modifications to the configuration files still must be done like updating the WLAN credentials, the serial number while provisioning or the access point name when the application is in setup mode. To satisfy the need of updating some files normally placed inside the rootfs, the "config" container is used. Every writeable system file is stored there and referenced by a symbolic link from the read-only rootfs.

Another possibility to still make the rootfs writeable is using overlay file systems like routers do. Since the hardware target only needs a few update specific pre-known files the previous solution fits better.

4.6.2 Configuration Container

The "config" container is mounted at /config and is used for write-able system files, provisioning data as well as important application configuration files. For maximum reliability ext4 is chosen as file system. During mounting, the journal and caching can be configured in a way that interrupted writes are detected immediately during mounting and the recovery is easy and fast. In addition, it is mounted with the "sync" option to avoid any write buffers delaying flushing the data to the disk. This way the application or system services can rely on the data being on disk when the write calls are returning. The journaling does some extra write load but the trade-off for fast recovery is acceptable.

4.6.3 Application

The application running on top of the system is stored very similarly at the image store, also being squashfs and redundant because the requirements and advantages are the same. The assigned mount points are /mnt/app1 /mnt/app2. The selection of which application to use is done by a script being part of the rootfs and started by a system service. The application start and updating process is out of scope of this work.

4.6.4 Application Data

The data spaces which can be used by the application are located in two places. One is inside the previously described configuration container at a specific path (/config/user-config) mainly used for configuration. The second place is at the mount point /persistent, where the application data container is mounted by the application itself. This separation is for distinguishing between important configuration and less important sensor data, persistent logs, and crash dumps.

4.6.4.1 Temporary Storage

As stated in section 4.5.4, there is free space on the image store for updating purposes. The system image is directly downloaded to the image store due to its large size, which is too large for keeping it in random access memory (RAM) especially when unpacking it. The chosen directory for this is beneath the image stores mount point at /mnt/images/tmp.

4.7 Updating the System

The task of updating the system is split in parts for the bootloader, the OS with its rootfs and the application running on top of it. Fetching, verifying and preparing is the responsibility of the application. Relocating the images to the correct paths while performing the update and telling the bootloader about the changes is the task of the system. Dispatching at boot time and starting the correct kernel is performed by the bootloader as described in section Handling Redundant Systems and their Update on page 26.

The interface between the application and the system therefore are some fixed locations for the update files and some service names for starting the update processes. The kernel exchanges information with the bootloader via the bootloaders redundant environment. After a successful update, the application must trigger reporting the state after start-up to complete the update.

4.7.1 Responsibilities for the Application

The application triggers the system update process by any means, directly by an operating user, or time controlled. A visual overview of the tasks to be performed by the application is found on the right side of figure 4.12. Downloading, extracting, and verifying the image is done after starting the update if there is any available. Before starting the system service associated with system updates, the kernel as well as the rootfs-container needed

to be placed in specific places so that the system's script called by the service can find it.

Besides triggering and performing the update after successfully booting the system and starting the application, another service must be started. This time it lets the system write back to the bootloader environment that the startup succeed.

4.7.2 Services and Scripts Provided by the System

The system update routine, a series of commands in a bash script, gathers further information needed before performing the update. This includes determining the currently used system image file name and also the location of the inactive kernel. The filename is read from the losetup command with tells the script about currently mapped image files. The NAND partition to use is queried from the bootloader environment. With this information, the old system image can be removed, and the new kernel flashed onto the correct place. After an integrity check, the new system to use is set in the bootloader environment and it is saved back to the corresponding NAND partitions. In the end, a reboot is performed. Figure 4.12 summarizes the described flow in the column "Linux Update".

The second system routine needed to be provided by the system is for determining if a system update was correctly performed. As by the previous routing a systemd-service is used for triggering the "Linux BootUp Check". The procedure is shown in the same figure as before. The application only triggers the check by starting the service. All tasks here are performed by the script backing the service. Basically, the environment is read and a simple comparison of two variables is performed. If there is a mismatch an update was performed and by setting the correct variable the bootloader knows at the next startup that the update was successful and therefore does not perform a fall-back to the old system.

With the systemd-services used to perform the update related tasks this unified interfaces are reusable on other systems as well.

4.8 Update Distribution

Up until this point, the concept chapter focused on the target system itself with its challenges and tasks. Besides that, the gap between the created system images and the start of the update routine on the running systems needed to be filled: This section focuses on update delivery from the build server to a few or all systems in-field. The concept of delivering updates can be applied to the application running on top of the considered system images as well. Since the bootloader is not updateable by design, the delivered over-the-air updates for the systems consist of the kernel and the rootfs.

In the first subsection, requirements for the software delivery are gathered. Derived from that, the update service itself is described. When dealing with updates, their authenticity must be guaranteed, therefore one subsection deals with the used PKI. This section closes with a closer look at identity management.

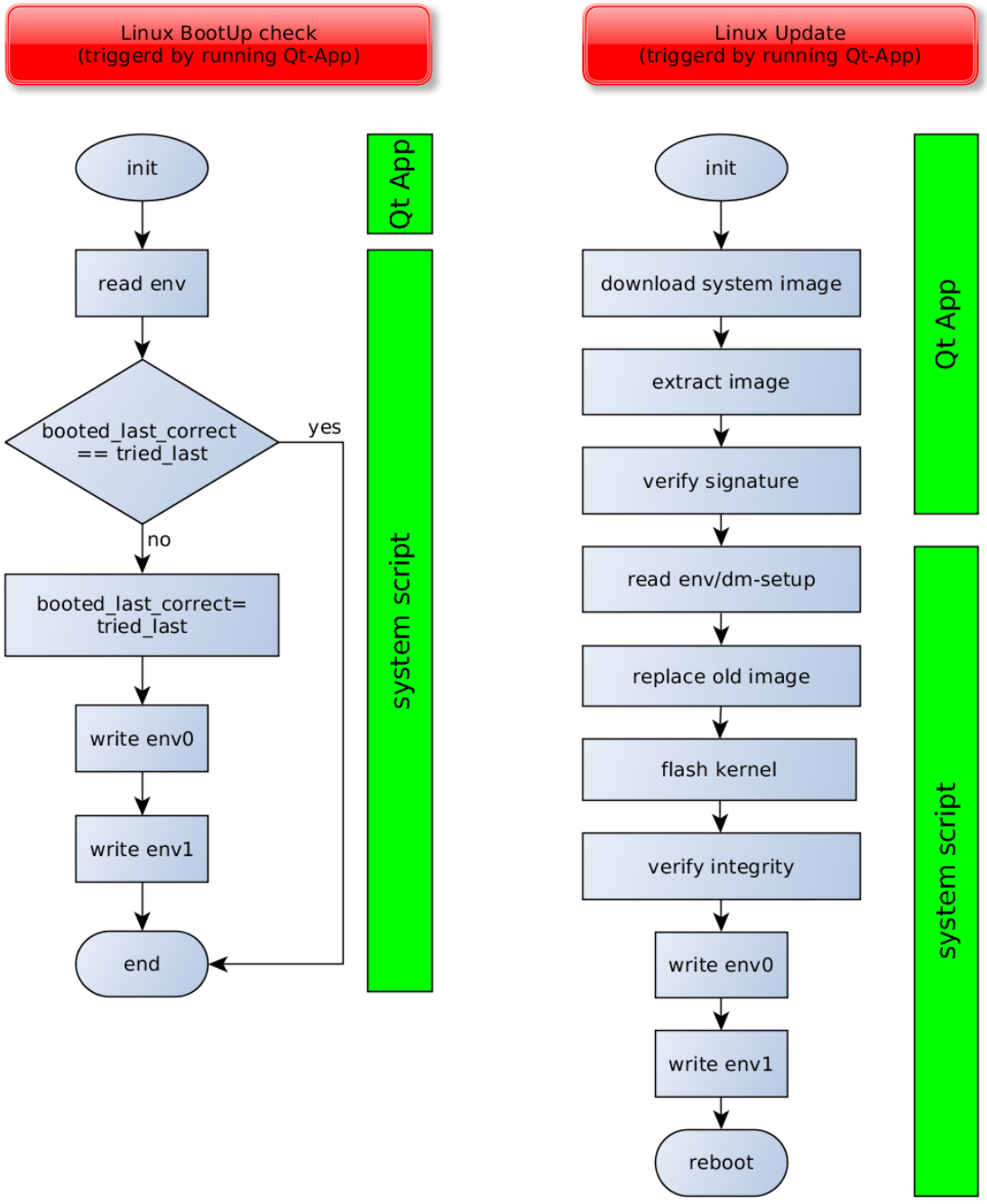

Figure 4.12: Tasks regarding a system update performed by Linux and its application

4.8.1 Requirements for the Software Distribution

The basic requirement for the update system is to take created updates and forwards them to the targeted systems. So, the idea is to define a format which updates must comply to, and the update service then can distributes them. This means the content is strictly separated from its delivery (path).

The service must provide a way to publish updates automatically by other services as well as manual publishing by authorized users. Also, the update distribution must be controllable. Lastly, the update must be fetchable by the target systems. Since nearly all system deployed in-field will not have a public IP address which would be reachable, the systems must fetch the updates themselves.

Update publishing and distribution control are internal tasks. No external stakeholders need to be considered for these tasks. To enable the possibility to define test groups as well as a group for early adopters some kind of deployment group management must be in place. As stated before, the updates must arrive on all systems in-field. The load of many systems fetching updates must be manageable.

Due to the long time spans, in which products are in stock at some warehouses, the update service must provide a long time support for its offered services. Otherwise, shipped products might not be updateable after several years. This applies to machine to machine interfaces like application programming interface (API)s in general.

The defined update formats with their meta data must be useable for the system updates as well as application updates. The version numbering should have a format to enable clients to detect the need to install all updates or only a few to avoid unnecessary (intermediate) updates.

Due to the requirement for targeted deployments, an identity management of some kind must be implemented. The update service must be able to distinguish between several clients. Those identities must be known by the service as well as the target system itself. This enforces the need for provisioning the products for (but not only for) the update service.

4.8.2 Update Service

The update service needs to be reachable by any party over the Internet. A web portal with an associated Representational State Transfer (REST) API for machine to machine communication achieves the desired goals. All manual operations are carried out on the web portal while services and clients access the API. Therefore, various ways of authentication are needed. The various tasks of publishing and distributing control can be carried out by different roles. Such roles might be development, testing department or product management. Together with the previously described deployment groups, this is visualized in figure 4.13. In the middle column, the basic life cycle of an update is shown. On the right side, different roles acting at different stages are associated with them. Example

targets on the left side represent clients who are fetching updates and their meta data.

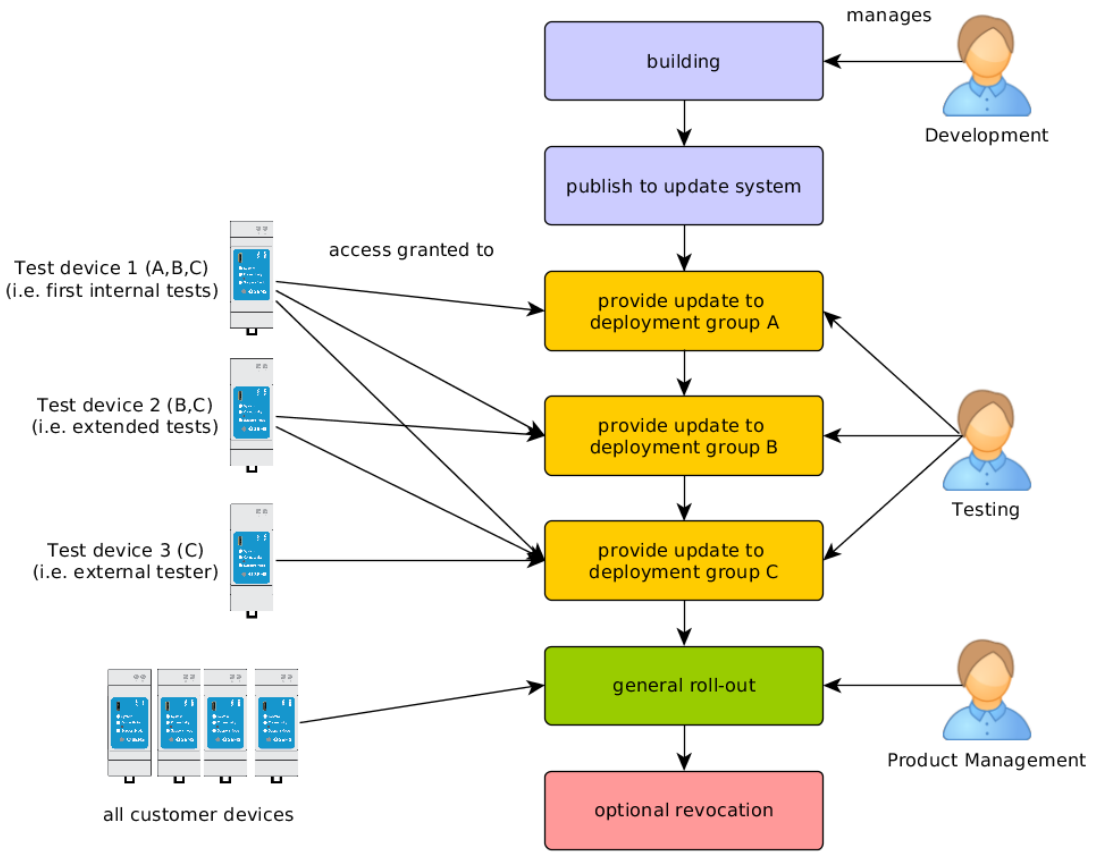

Figure 4.13: Principle life cycle for an update from creation until full roll-out.

Development usually builds and publishes the updates according to the development process in state. Most of the time, the building and publishing is fully automated and done by build servers. After an update is published in the update service, it needs further tests in isolated environments. The example in figure 4.13 defines three steps for testing and/or preview update delivery. Different systems can be targeted by the various groups. Three groups exist in the example: A, B and C. While test device 1 is enrolled in all groups, device number 2 only in B and C and the last test device in C. During the testing phase, the testing stuff deploys an update to the groups according to a test schedule or policies. If it is tested enough and considered ready for general (public) release to all registered clients (target systems at all customers), the last step in the chain is performed by the product management: the general roll-out. Usually it ends here. In some rare cases, if there is a major bug, an update can be permanently revoked. Then it is not available to any client any more. Such a revocation could happen at earlier stages as well. An alternative is to undo the general roll-out (or deployment on a group) stage. Systems in-field still have the update installed but other clients can be prevented from getting a faulty update.

Deployment groups are the enabler of various scenarios. Simple test groups can be organized as well as larger groups of customers. Any number of clients can be added to a group. Updates are deployed on a group. A client can get the meta data of all updates available to it. This includes all updates generally available as well as the updates deployed in all groups it participates.

Since various update types are available for the same target hardware, some differentiation has to take place. Besides the version numbering, the update's meta data contains a hardware identifier which represents the type of update and the systems which it is designed for (i.e. the used toolchain). All the clients, deployment groups, the association between them, and the available updates with the applied deployments are grouped together in a project. So, the top-level element of the update service is the project. Inside the project, clients, update and deployment groups are added, and relationships between them are defined.

Following subsections explain the roles and authentication in more detail. The update service's purpose is to distribute updates. It does not care about the validity or authenticity of them. This is done by signing the update by the build process and verifying the signatures after downloading updates on the target systems. Subsection 4.8.3 describes the used PKI in detail to be able to verify and validate updates.

For the requirement of a long time stable interface for the clients, the API updates must be available as different versions. This way old (less feature-rich) versions are still available when products with long-time outdated software try to access the update service. So, the clients can simple update their software and use newer versions of the API with it later on.

4.8.2.1 Roles

As shown in figure 4.13, multiple roles exist when taking a closer look at the update life cycle from the perspective of the update service. In the principle overview, only three roles are shown. In reality, all actions might be performed by only one person or bigger groups of people, depending on the size and organization of a business. On which basis decisions are made to perform various actions, is out of scope in this work.

In addition to the roles humans represent, services and clients also define different ones, such as the role of the build server publishing the updates, because update building and publishing often is fully automated. Another role is the one of the target system or client, which accesses the service for fetching new updates. The clients might also be differentiated furthermore according to their assignments to deployment groups. For this work, the focus is clearly on the client side and the update distributions. By design, there is no distinguishing on the web portal for different roles.

4.8.2.2 Authentication

The types of authentication can be divided in two categories. Those on the web portal and those on the APIs. Within the web portal a single type was chosen. The OAuth 2 standard allows external authentication providers, and a claim system for authorization can be used. Since there is no further distinguishing on the web portal regarding user roles, the claims are not used. Each user able to access the web portal (which is in fact determined by one special claim) can perform all actions the portal supports. This also includes the management of API keys, which are used for authentication when using the management functions not via the portal but via a dedicated API. This offers the possibility for automating management. For instance, the build server can publish an update or even deploy it on the first test group.

Besides the management portal and API, the interface for the products in-field is needed. It is also an API, but this time with another authentication. The clients identify themselves by their provisioned serial number and access key. When requesting information or updates, also the project name is needed since the clients are registered within a project. The type of authentication was chosen due to its simplicity and therefore easier integration when compared to OAuth 2 or similar mechanisms.

4.8.3 Update-PKI

The update service itself is responsible for storing updates including their meta data, organizing groups and deployments and also delivering updates, as described in section 4.8.2. Since it defines an interface how meta data and the update itself must be structured, it does not care about the content or authenticity. But the update services clients (the product's software) must verify the update. It is only allowed to accept updates from trusted sources for security reasons. A signing infrastructure, a PKI, is used to be able to establish trust. A trusted root certificate is installed on all devices and updates are only accepted if they are signed with a private key associated to a certificate directly or indirectly signed by the root CA. Besides updates, other special side-loaded software can be signed. This enables for instance an USB stick with a provisioning or test software to be executed instead of the main application.

To illustrate the structure of the used PKI figure 4.14 gives an overview. The trusted root certificate is in the top left corner. Beneath, there is the intermediate CA which is signed by the root CA. The intermediate issues the next layer of certificates with their keys for normal signing operations. All certificates carry a common name (CN) which is listed in the corresponding field. The purpose of the intermediate CA is to sign the certificates actual in use for doing the software signing work. This extra layer would not be needed at all, but it is there for security reasons. Over time, several working keys are needed for different entities who sign software. This is generally good for reducing the impact of a compromised key to a minimum. If a build server is hacked and its private key gets stolen, the corresponding certificate must be revoked and with it all signatures ever made using it. Every update and other software need to be rebuilt and signed with a new key. With multiple keys only a little part of the PKI are compromised, meaning not

all signatures have to be remade.

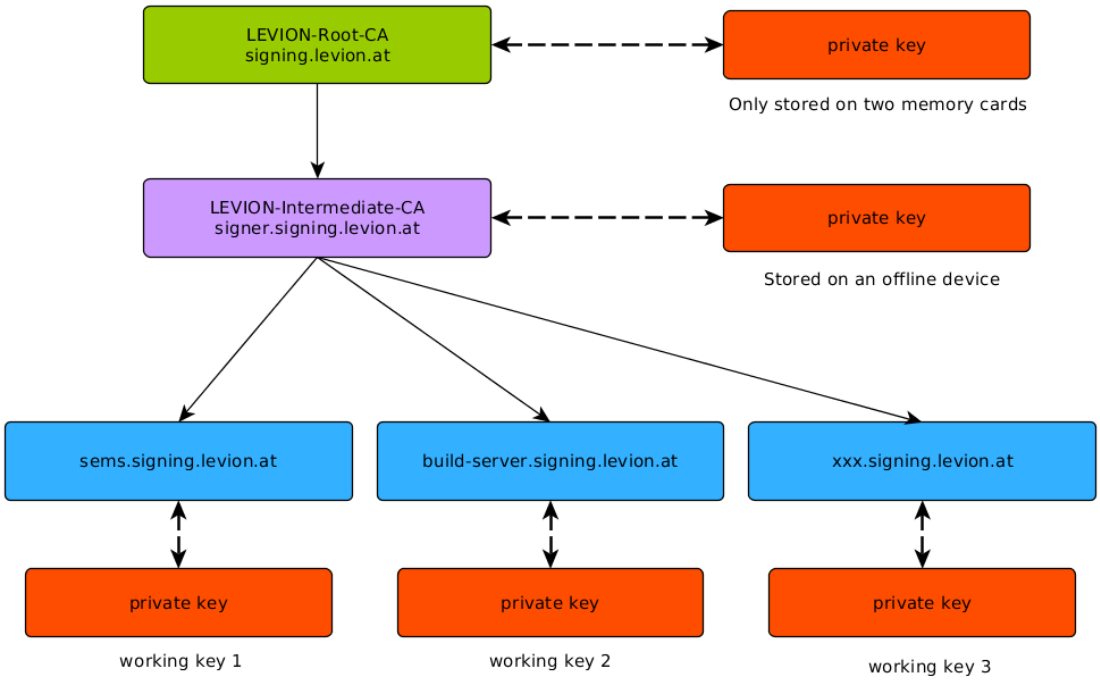

Figure 4.14: Basic structure of the used PKI for software (update) signing.

Even more important is the protection of the root and intermediate CA since all sub trees are revoked as well if a CA is revoked by a top level one. Since the leakage of the root CAs private key is fatal and the complete CA is then useless and cannot even be revoked, so protecting it is a top priority. Therefore, the root CAs private key and associated issuing data is stored only off-line on two memory cards. Normally, the storage media are only needed if the intermediate CA needs to be revoked and replaced by a new one. Both cards are encrypted and located on different sites for maximum security.

The intermediate CA can be revoked and therefore does not need such high security measures. But to minimize the risk of leaking the private key and maintain convenience for using the CA, it is stored on a dedicated laptop which has not been connected to any network since the creation of the root CA and never will be. This way this special system just needs to be booted and a new certificate can be issued. Its internal storage is redundant to avoid failure within the storage devices. With those measures the complete PKI should be well protected. In the case of leaking a leaf (a working) key the intermediate CA needs to revoke the corresponding certificate and the new CRL needs to be published on the correct locations as announced by the all issued certificate's meta data.

4.8.4 Embedded Identity Management

As stated before in section Requirements for the Software Distribution on page 35, all systems must be provisioned in order to have distinct identities for the update service. Otherwise it would not be able to distinguish between clients and therefore it would not be possible to deploy software on only a little targeted group of devices. During manufacturing, the products must be provisioned with the correct update system setup (Uniform Resource Locator (URL) and project ID) as well as its credentials which are the serial number and a generated random secret.

During the mentioned provisioning phase a provisioner must generate the secret and then store it with the configuration on the product itself as well as on the update service. This way the service has a list of all provisioned systems for managing the deployment groups and being able to authenticate the clients.

If the target system is hacked in any way the identity might get stolen but then the attacker only can request information and updates as the victim could have, but not more thanks to the per serial number (identity) generated secret.

4.9 Securing the System

To protect the hardware target with its running system and application, a few measures must be taken in order to make the system secure. The scope of the system configuration is to protect against mostly network related attacks. Verifying signatures during updates is not a concern for the rootfs but for the running main application on top. A few basic measures are explained at the next subsections. Besides them, constant security updates are required because of the huge number of various software components used at the system. Triggering updates in general is the responsibility of the top level application as described in section 4.7.

Therefore, this section does not cover secure software delivery. Due to the enormous effort to protect a system against direct internal hardware attacks, hardly any measures are taken to protect the system by that means. Risks exposed by such access depend on how the application stores sensitive information. Other systems might get exploited more easily due to the knowledge an attacker could gain by directly accessing the systems. Besides that, only one system is compromised in such a case. This poses no very high threat for all (other) systems.

4.9.1 SSH-Access

For developing software and third level support issues a Secure Shell (SSH) access to the system is a big asset. While developing applications, direct system access is important and very common. Because drawing a line between development and pure productive systems is very hard, it is nearly impossible in a market where every installation varies a lot. Some tests only can be performed in-field or even on customer sites. Besides that, some rare support cases need very deep inspection of the system. As a result, the SSH

root access needed for such actions is unified for all systems regardless of their purpose regarding development. It is vital to protect this access by all means. Systems could be easily hijacked with open or weak SSH access protection.

As a main counter measure against break-in attacks at SSH only public key authentication is enabled, meaning that the attacker would need the private key and its password to be granted access to the systems. The private key must be well protected and must not ever be unprotectedly saved, transmitted or accessed. Also, the systems where development and third level support work is done need good protection too. Therefore, policies must be enforced.

Besides support and development SSH offers another possibility. This is a secure connection for a text-based console for the application. Since the application offers such a console, it needs protection too. There is nothing better than to use SSH here. Besides using SSH public keys for authentication, a password could be used for convenience, but only if it is not set to a default password. This would be like no password is set at all. The SSH service can be configured in a way that the password authentication is white listed for one specific user only. And the access to the application console is granted after a password is set.

4.9.2 Read-Only File Systems

One way to protect systems against changes is making them read-only. It gets a lot harder for an attacker who got access to the system by exploiting some weakness if he or she cannot modify anything or hardly something. At the next reboot every change would be lost. While some write access is needed for correct system operation, the number of writeable files is very limited.

So besides making the system very robust against power losses using it read-only improves the security too. To make permanent changes at the system, the complete system image needs to be made writeable in order to do serious damage. The boot routines need to be manipulated to do this. This requires very sophisticated skills and experience with the target system due to mechanisms like the watchdog and its kernel configuration options. Shortly, it is not impossible, but not easy either.

A measure to detect system manipulation would be to sign off any changes to the file system and also save signatures for the deployed system and kernel images. The few writeable files cannot do much damage since they lead only to some failures of selected services.

4.9.3 Dedicated User for Applications

Applications should never run with root privileges because they could have weaknesses which can be exploited. Besides the read-only system, there are some files which could be manipulated like the system's identity (the serial number). Access to some configuration files of the on top running main application could be read and security relevant informa-

tion extracted.

Using a dedicated user with little privileges is the common way to deal with this situation. All services especially networked enabled ones must run with their own user. Since this is the default case in most common Linux services when using Yocto the work of creating own users is mostly automated. Only for our main application this needs to be done separately.

4.9.4 Disabling Unused Devices and Services

One security risk when dealing with provided distributions from hardware manufactures is that a lot of unwanted services might be enabled by default which often are network enabled and therefore could be exploited. To minimize the risk here every software is disselected by default an only needed software packages are white listed. Yocto supports this process by the use of custom targets and package groups. Only a validation of the running services when the final system or rootfs is built has to be done.

4.9.5 Firewall

When taking all measures as explained in the previous sections a firewall still could be used to increase the security of the system even more. However, this does not have a huge impact and a lot of adjustments have to be performed by the application. The background for this is that the system's rootfs does not really know anything about the on top running networking requirements. Several tasks like discovering other network enabled devices or providing various interfaces to other networking devices are hardly known before when building or updating the system. The standard default blacklisting and opening selected ports is not working here very well, even when all unneeded ports are already closed.

The biggest impact of a firewall could be the defense from unknown or incorrect network behavior in general. This entails for instance retrying various passwords for application authentication. But still the application must alert the system about suspicious events. A service called fail2ban could be used for this purpose. It sources its needed information from log files.

The target hardware mainly is designed for installations in domestic homes. So direct access from devices across the Internet is not possible per default. A port forward must be configured at the gateway or a public IP address must be assigned to the hardware to make access besides from the local network possible. Both require networking skills. Therefore, the network access is very limited by default.

4.10 Ensuring Always-On

One goal for the new system is to ensure that it is always on-line. Different things can happen which lead to a stall at the boot process or during normal operation. This includes

1. hangs due to an error after one system update try,

2. unexpected hardware failures and

3. stalls in the application.

A common technique to discover stalls and recover from them is the use of a watchdog device. Basically, something has to be written to the watchdog on a regular basis or it resets the whole hardware. Since different programs and procedures are running on the same hardware until the application is executed, a full chain or responsibility for keeping the watchdog happy must be created. The scheme for the target hardware is shown in figure 4.15.

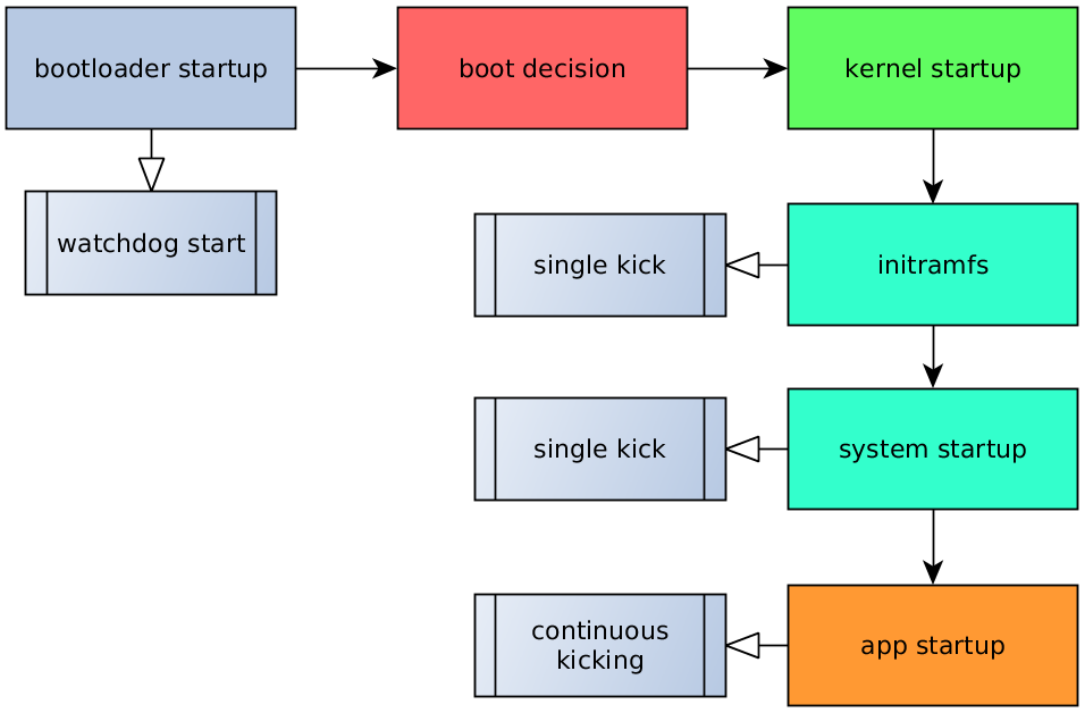

Figure 4.15: Watchdog chain of responsibility.

During hardware initialization, the bootloader starts the watchdog and kicks it one time. The boot procedure continues by selecting the correct system, loading and starting to execute it. When loading the watchdog driver in the Linux kernel, the watchdog is not disabled. When the initramfs' script is executed, one single kick is performed. The same applies for one single startup script in the rootfs. Both kicks are needed because of the very low possible timeout of only 16s on the target hardware.

When the application is started, it kicks the watchdog on a regular basis. At this point, the system is considered as booted correctly. Simultaneously, the write-back of the successful start is written to the bootloader environment, see section Updating the System on page 32 for more details. On other CPU platforms with a much higher watchdog

timeout, there are no intermediate kicks during the boot needed. Just the start of the bootloader and the next kick would be performed directly by the application.

If there are internal unrecoverable errors detected in parts of the application itself, it also stops kicking the watchdog. This way the system never should stall for longer than a few seconds.

Chapter 5

Implementation

This chapter at first describes how the build system with its input is used. All tasks needed for building the new target system with its atomic update capabilities are defined as recipes in specific so-called layers. After that, the focus is on the work needed for the system update within the bootloader, kernel and rootfs in distinct sections. Last but not least, the tools provided for application development are described.

5.1 Build System - Yocto

For the C.H.I.P, another product from the same vendor (Next Thing Co.) as the used SoM CHIP Pro, a complete build solution was done. Both are very similar. Due to a community effort, the CHIP Pro was already integrated into the meta-chip Yocto layer. Next Thing Co. offered a build solution using Buildroot.

The basis for the used setup was the Yocto enabled build solution with the recipes and definitions for the CHIP Pro. In addition to the already existing layers, meta-qt5 was added to the build system. To get all recipes and configurations for all hardware independently grouped together, the layer meta-sems was created. A second layer for all hardware related modifications and additions due to the developed custom boards was also added.

The following subsections describe these layers in a more detailed way.

5.1.1 Existing Layers

Figure 5.1 gives an overview of all layers used. On top are the most basic layers used in Yocto. Poky is the name of the reference distribution shipped. BitBake, a generic task executer, is used for resolving all dependencies between layers and their recipes, performing source fetching, analyzing recipes and performing the build itself.

Based on these layers, the hardware related layer meta-chip is the next according to its priority. Since the repository CHIP-SDK, which is not a direct part of the layer structure, its presence is vital for creating and flashing the SoM. It is also installed by the script responsible for the initial setup of the whole build system. Therefore it also found a place

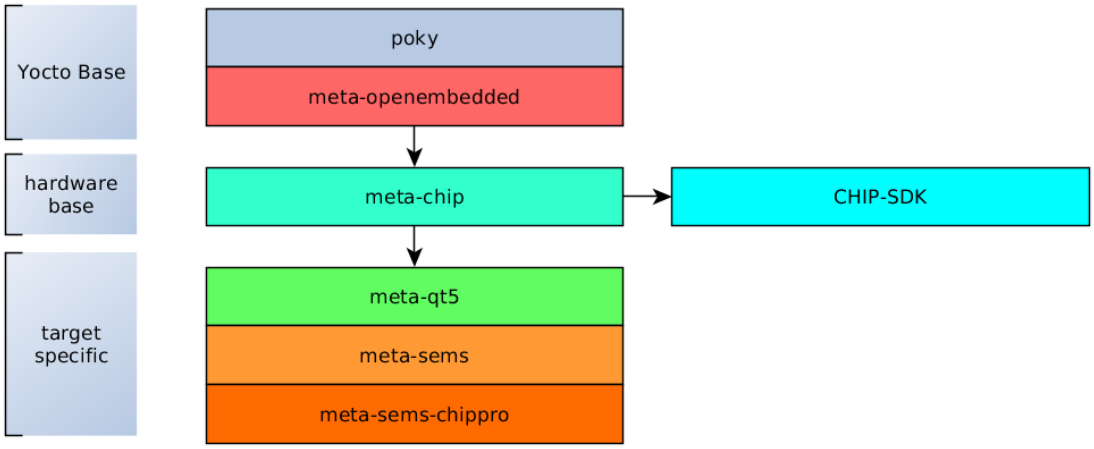

Figure 5.1: Uses layers within the build system.

in figure 5.1 next to the meta-chip layer due to its strong association.

The last preexisting layer is the one for the QT framework, called meta-qt5. Hardly any changes had to be made in more important layers to overwrite the recipe's definitions of it. Such changes were related to the outline of the build to include or exclude several QT components.

5.1.2 New Layers Introduced

As stated in the previous section, new layers had to be introduced. Since one goal is the reusability of the hardware-independent update routines, an own layer for this was created. So, everything which is not related to the specific hardware target is placed here. Since a lot of those recipes consist of bash scripts or service definitions, this data is directly stored within them most of the time. Common recipe alterations are placed here too. See sections 5.1.3 and 5.1.4 for more details.

To perform kernel driver selection, embed the initramfs or configure the image recipe responsible for which software is installed on the target, the layer meta-sems-chippro was used. It mainly consists of modifications of existing layers. New recipes are limited to hardware related or product specific services or identifications. These include for instance the service for polling the reset button state from the PMIC or the SSH host keys.

```
1 # We have a conf and classes directory, add to BBPATH
  BBPATH .= ":${LAYERDIR}"
3
  # We have recipes-* directories, add to BBFILES
5 BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
    ${LAYERDIR}/recipes-*/*/*.bbappend"
7
```

```
BBFILE_COLLECTIONS += "meta-sems-chippro"
BBFILE_PATTERN_meta-sems-chippro = "^${LAYERDIR}/"
BBFILE_PRIORITY_meta-sems-chippro = "51"
```

Listing 5.1: The layer configuration file for the meta-sems-chippro layer.

Listing 5.1 shows an example layer configuration which is the minimum a layer needs. Besides setting the path which is common to most layers, the recipes shipped with the layers have to be explicitly referenced here. In this case, line 5 includes all recipes (*.bb files) and all appends to existing recipes (*.bbappend) stored at directories starting with "recipes-" and being in a directory below them. The last noteable statement is the last one. It defines the priority of the layer. This layer is the most important one which overrules all others. The meta-sems layer has priority 50 which makes it the second most important.

5.1.3 Recipe Creation

The majority of new recipes contains various bash scripts and systemd service files. One example is displayed in listing 5.2. All recipes have a few lines in common. For instance, the used license or a description. In the example, the compatible architecturs is set to “all”, meaning there is no restriction for which target architecture it can be used.

A very common variable is called SRC_URI. It specifies the location for the source needed for building or deploying whatever the recipes describe. Most times, this is a download URL or relative path to files which are directly shipped with the recipes. This is the case for the shown recipes. The bash scripts and service files do not need to be compiled.

Since nothing special had to be done besides directly installing the files, the core element in the shown recipe is the "do_install" function starting at line 19. It is executed when BitBake reaches the "install" phase. Were there something to prepare, the function's name would be "do_prepare". A simple sequence of install commands is executed which copies files with the correct permission to the rootfs or more specific to the package which is later on installed at the rootfs.

The last two functions starting at lines 32 and 36 are more special cases. Since the "bootupcheck.sh" is slightly different on two different hardware targets but mostly very common, it makes sense to keep the files together at one place. Depending on the selected machine in the central "local.conf" configuration file, one of these functions might get selected. In both cases they append a command to the common do_install routine. This is achieved by the special syntax as seen in line 32 or 36. "chippro" is the machine name for our target hardware and "levion-phytec-v2" would be another machine.

A machine in the Yocto context is a target system for that the kernel, bootloader and device-tree name are designed.

```
DESCRIPTION = "SEMS update scripts"
LICENSE = "CLOSED"
PACKAGE_ARCH = "all"

```

```
SRC_URI = " \
  file://start-system-update.sh \
  file://start-update.sh \
  file://bootupcheck-chippro.sh \
  file://bootupcheck-phytec.sh \
  file://sems-system-update.service \
  file://sems-update.service \
  "

FILES_${PN} += "/system-scripts/ \
                /lib/systemd/system "

S = "${WORKDIR}"

do_install() {
  install -d ${D}/system-scripts
  install -m 755 ${S}/start-system-update.sh ${D}/system-scripts/
  install -m 755 ${S}/start-update.sh ${D}/system-scripts/


  install -d ${D}/lib
  install -d ${D}/lib/systemd
  install -d ${D}/lib/systemd/system
  install -m 644 ${S}/sems-system-update.service ${D}/lib/systemd/system/
  install -m 644 ${S}/sems-update.service ${D}/lib/systemd/system/
}

do_install_append_levion-phytec-v2 () {
  install -m 755 ${S}/bootupcheck-phytec.sh ${D}/system-scripts/bootupcheck.
    sh
}

do_install_append_chippro () {
  install -m 755 ${S}/bootupcheck-chippro.sh ${D}/system-scripts/bootupcheck
    .sh
}
```

Listing 5.2: The new recipe file update-scripts_0.9.bb included in meta-sems.

5.1.4 Recipe Alteration

Altering a recipe or a configuration file is a very common task when working with Yocto. Often a single "define" for compiling has to be passed to the build process of a specific recipe or simply a name of a variable for a recipe itself is overwritten. Another often used case is shown in listing 5.3 besides other interesting overwrites and additions.

Lines 3 to 8 define an extra file source for the initial recipe for the Linux kernel used with the target machine. Three files need to be sourced from this recipe instead of the original one. This includes the kernel configuration "defconfig" as well as the prebuilt initramfs and the patch for altering the device-tree file. With the FILESEXTRAPATHS_prepend variable set the defconfig for example is used before any of the same name in recipes from layers with lower priority. With this technique, most files easily can be shadowed and replaced with the wanted ones.

Line 10 sets the Linux kernel version used for building to 4.4.66. To get the correct system name for the kernels identification and also the resulting system image name, line 15 is used. This is vital to the update process because the initramfs derives the name from the kernel to find its rootfs stored in the squashfs image file. See section Initramfs on page 29.

The script building the updates for the update system also needs the kernel's name, as configured previously, so the build configuration has to be copied to the deployment directory after build. Since this does not interfere with the existing deploy task, the defined function which appends one single copy command can be found at the lines 17 to 21. To enable USB OTG support, the corresponding variable to loading kernel modules by default is used as displayed in line 24.

The rest of the file is just a reminder how to build the initramfs as a comment. Instead of previously building the initramfs manually and adding it to the recipe, a separate recipe which builds the minimal image needed could be created. But for simplicity reasons, this was not done during this work.

```
2
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
4
SRC_URI_chippro_append = " file://0001-dtb.patch \
6                          file://defconfig \
                           file://initramfs.cpio \
8                        "

10 LINUX_VERSION = "4.4.66"

12 # configure the current system version here
   # note: the squashfs image name is also derived from this
14 # example: SEM-MK90 will result in "4.4.66-SEM-MK90" with uname -r and
       system-v90.img
   CONFIG_LOCALVERSION_OVERWRITE = "SEM-MK11"
16
   do_deploy_append() {
18
     # copy kernel config to deployment dir
20   cp ${B}/.config ${DEPLOYDIR}/kernel-defconfig
   }
22
   # needed for usb otg
24 KERNEL_MODULE_AUTOLOAD += "sunxi"

26 # pack initramfs (when standing at its root dir):
   # $ find . | cpio -o -H newc  > ../initramfs.cpio
28 # dont forget that you need two devices under /dev
   # mknod -m 622 console c 5 1
30 # mknod -m 622 tty0 c 4 0
   # those need root priveliges to create and they are not created when
       unpacking
32 # the cpio archive as non root
```

Listing 5.3: The recipe altering the preexisting kernel recipe: linux-chip_git.bbappend.

Besides altering an existing recipe or one bbappend-file, those files can be blacklisted or masked. The variable is called BBMASK and has to be specified in a layer configuration file like the one shown in listing 5.1. This often makes it easy to skip a single bbappend-file in another layer, which would mess up things.

5.1.5 Further Configurations and Files

Beside the layers with their configuration and recipes, there are other important files: the machine configurations. As an example listing, 5.4 defines the target machine. This includes the used bootloader, kernel selection, tune options for the CPU architecture or the setup for the serial console.

Most noteable in this file are the EXTRA_IMAGEDEPENDS and KERNEL_MODULE_AUTOLOAD variables. Those where extended for correct build dependencies and enabling USB OTG as a client console support.

```
#@TYPE: Machine
#@NAME: C.H.I.P. PRO board
#@DESCRIPTION: Machine configuration for C.H.I.P. PRO board

MACHINE_EXTRA_RRECOMMENDS = " kernel-modules kernel-devicetree"

EXTRA_IMAGEDEPENDS += "u-boot \
                       sems-config-default \
                       sem-compact-app \
                       "

DEFAULTTUNE ?= "cortexa8t-neon"
include conf/machine/include/tune-cortexa8.inc

IMAGE_FSTYPES += "tar"

# ;ttyG0 would be USB OTG
SERIAL_CONSOLE = "115200;ttyS0"

PREFERRED_PROVIDER_virtual/kernel ?= "linux-chip"
PREFERRED_PROVIDER_u-boot ?= "u-boot-chip"

KERNEL_IMAGETYPE = "zImage"
KERNEL_DEVICETREE = "ntc-gr8-crumb.dtb"

SPL_BINARY = "spl/sunxi-spl.bin"
UBOOT_BINARY = "u-boot-dtb.bin"
UBOOT_MACHINE = "chippro_defconfig"

MACHINE_FEATURES = "usbgadget usbhost wifi"

KERNEL_MODULE_AUTOLOAD += " \
        libcomposite \
        usb_f_ecm \
        usb_f_eem \
```

```
36          usb_f_rndis \
            sun4i-lradc \
38          g_serial \
            "
40  # ethernet AND serial over usb gadget would be g_cdc instead of g_serial

42  # Include wifi modules and firmware

44  MACHINE_EXTRA_RRECOMMENDS += "kernel-module-8723ds rtl8723ds rtl8723bs-bt
        axp209"
```

Listing 5.4: The hardware targets machine configuration file.

5.1.6 Priorities for all Recipes, Layers and Configurations

One of the most important aspect to keep in mind is the layer priority. Various configuration files are selected from layers with higher priorities when found multiple times. Also, layer priority defines in which order the recipes with their appends are handled. The used priorities are shown in listing 5.5. Basically, this explains figure 5.1 in more detail. As shown at the paths, there are more layers embedded in poky and meta-openembedded.

The priority is set in the layer configuration file. Apart from the two new layers listed last, all priorities are the default ones. Since the meta-sems layer defines more common system recipes, it is not as important as the target specific layer (in the case of our target meta-sems-chip).

```
layer                 path                                          priority
2  ==========================================================================
   meta                  sources/poky/meta                                  5
4  meta-poky             sources/poky/meta-poky                             5
   meta-yocto-bsp        sources/poky/meta-yocto-bsp                        5
6  meta-chip             sources/meta-chip                                  6
   meta-oe               sources/meta-openembedded/meta-oe                  6
8  meta-networking       sources/meta-openembedded/meta-networking          5
   meta-python           sources/meta-openembedded/meta-python              7
10 meta-webserver        sources/meta-openembedded/meta-webserver           6
   meta-qt5              sources/meta-qt5                                   7
12 meta-sems             sources/meta-sems                                 50
   meta-sems-chippro     sources/meta-sems-chippro                         51
```

Listing 5.5: Output of the bitbake-layers show-layers command: All layers with their priorities.

5.2 Bootloader

The used bootloader is called U-Boot or "Das U-Boot" and is very common in embedded systems. Most hardware is brought up by this one or barebox, another popular bootloader. A few modifications had to be made to support the boot and update process as described in the concept chapter.

Since the device-tree is used both by the bootloader and the kernel and the impact of the modifications are higher for the kernel, it is described in the following section (5.3 on 53).

5.2.1 Customization

Most requirements are already met by the default U-Boot configuration used for the SoM. The first thing to do was to enable the redundant environment in the NAND partition. This was done by several patches which also include early driving the LEDs to indicate a working bootloader after power-up. The watchdog driver had to be added to the source because it was not included.

The default watchdog operation in U-Boot was not the desired one: Depending on whether it was a warm or a cold start, it behaved differently. This could be resolved with patches. Yocto made it easy to integrate patches, as shown in the very trivial bbappend-file at listing 5.6.

```
1
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
3
SRC_URI_append := " file://0001-redundant_env.patch \
5                   file://0002-chippro_config.patch \
                    file://0003-watchdog.patch \
7                   file://0004-led-drive.patch \
                    file://0005-watchdog-disable-kick-when-polling-serial.
    patch \
9                   file://0006-watchdog-warm-start-issue.patch \
                    file://0007-wdt.patch \
11                  "
```

Listing 5.6: The very simple u-boot-chip_git.bbappend file used for modifying the bootloader to comply with the requirements.

Besides configuring the bootloader build for redundant environments, also the build for the tool used at Linux for data exchange had to be made aware of both environments. The tool is called u-boot-fw-utils. This was only a minor task achieved by changing its build configuration.

5.2.2 Configuration

Based on section Resulting Layout of the NAND Memory on page 28, the resulting NAND layout is shown in detail in listing 5.1. The Secondary Program Loader (SPL) is needed to be able to load U-Boot. The program code in the CPUs read-only memory (ROM) is executed at power-up, which loads and starts the SPL, which then loads and starts U-Boot.

#	name	size	offset	purpose
0	spl	0x00400000 (4MB)	0x00000000 (0MB)	1st stage bootloader
1	spl-backup	0x00400000 (4MB)	0x00400000 (4MB)	1st stage bootloader (backup)
2	uboot	0x00400000 (4MB)	0x00800000 (8MB)	2nd stage bootloader
3	env	0x00400000 (4MB)	0x00c00000 (12MB)	u-boot environment
4	env-backup	0x00400000 (4MB)	0x01000000 (16MB)	u-boot environment (backup)
5	UBI	0x1ec00000 (492MB)	0x01400000 (20MB)	image store (kernels, rootfs, ...)

Table 5.1: Resulting NAND layout used. The bootloader as shown in figure 4.9 consists of multiple parts.

5.3 Kernel

The kernel modifications can be divided into driver selection and build configuration, driver and hardware usage based on device-tree and the initramfs for mounting the rootfs. The kernel itself is not aware of the update process. The userspace applications and tools handle the update process and the bootloader performs the selection of the kernel to use and thus the rootfs associated with it.

5.3.1 Customization

All drivers needed for the hardware target where already present. To keep the kernel footprint small, only the necessary drivers, or more precisely, kernel modules where selected. To reduce the size even more, some modules are not directly built into the kernel but are loadable from the rootfs. Only drivers needed for booting and mounting the rootfs must be built-in.

Like at the bootloader, the watchdog driver had to be modified as well. When the kernel initializes the driver, it stops the watchdog which would lead to possible undetected hangs. The watchdog has to detect such hangs. Besides that, the kernel configuration option CONFIG_WATCHDOG_NOWAYOUT was activated to avoid turning off the watchdog in application crashes when those have the watchdog device opened.

5.3.2 Initramfs

As mentioned before the initramfs is vital to the boot process since the rootfs cannot be mounted directly with the kernels built-it routines. It is built into the kernel directly instead of being loaded like the device-tree by the bootloader. The device-tree is only needed once since it should not be updateable. Embedding is done in the kernel configuration.

The initramfs consists of a minimal rootfs containing only a few commands and tools to be able to mount the related real rootfs. The core part is the "init"-script as shown in listing 5.7:

It is a simple bash script with a linear control flow starting with setting the PATH environment variable to find the commands shipped with the initramfs and a message. A

few essential special kernel file systems must then be mounted as shown from line 9 to 16. For debugging purposes each command has an echo command before it.

Line 19 opens the UBIFS formatted image store. In the next step the rather complex line extracts the system version from the kernel's name as reported by uname and completes the path to the rootfs image located inside the previously mounted image store with it. With this, the rootfs gets directly mounted at /sysroot. When this is done, the watchdog is triggered once due to its short timeout. The switch_root command at last is executed which performs the switch to the real rootfs.

```
1  #!/bin/sh
2
3  export PATH=/bin:/usr/bin:/usr/sbin:/sbin
4
5  echo
   echo "### Startup ###"
7  echo
8
9  echo "Creating block devices"
   mount -t devtmpfs none /dev
11
   echo "Mounting /proc filesystem"
13 mount -t proc /proc /proc
14
15 echo "Mounting sysfs"
   mount -t sysfs none /sys
17
   echo "Mounting image storage"
19 mount -t ubifs -o noatime,nodiratime,sync ubi0:root /mnt/images
20
21 echo "Mounting root filesystem image"
   mount -o loop,ro /mnt/images/system-v$(uname -r | sed -n 's/^.*-SEM-MK//p').
       img /sysroot
23
   echo "Trigger watchdog"
25 echo "123" > /dev/watchdog
   exec switch_root /sysroot /sbin/init
```

Listing 5.7: The main part of the initramfs: the "init"-script responsible for finding and mounting the real rootfs and starting the so called init-process there.

5.3.3 Device-Tree

Most of the configuration for the used hardware and its drivers is done in the device-tree. The listings 5.8 and 5.9 are two example sections of the dts-file.

The first example shows how the RTC is defined. At first, line 3 references a previously defined I2C bus referenced by the label i2c1. The following three lines reference a block to the pin definitions used with the bus. "okay" means that the bus is enabled.

The embedded block defines the RTC itself on the bus i2c1, starting with a label, a descriptor and the bus address. The reference to the use-able driver is stated at line 8. Line 9 is a parameter for the used driver followed by the bus address again.

```
1
&i2c1 {
3   pinctrl-names = "default";
    pinctrl-0 = <&i2c1_pins_a>;
5   status = "okay";

7   pcf85063: rtc@51 {
      compatible = "nxp,pcf85063";
9     quartz_load = "12.5pF";
      reg = <0x51>;
11  };
};
```

Listing 5.8: The device-tree file used: definition for the RTC connected via I2C.

The second example is more complex since a NAND device is not as simple as a RTC to configure. For the sake of simplicity, not all lines are explained, only the most important ones. The new definition for the NAND starts at line 7, which resides inside the block for the used interface. Some hardware related parameters can be found in lines 8 to 13.

Embedded in this block for each NAND partition, a block is used to describe the various sections on the memory. They are all structured as followed:

1. The name for the partition as label, followed by the address it starts with, separated by '@'.
2. The used name which is the same as the previously used label for the block.
3. A descriptor for the start and the size of the partition.

```
1
&nfc {
3   pinctrl-names = "default";
    pinctrl-0 = <&nand_pins_a &nand_cs0_pins_a &nand_rb0_pins_a>;
5   status = "okay";

7   nand@0 {
      #address-cells = <2>;
9     #size-cells = <2>;
      reg = <0>;
11    allwinner,rb = <0>;
      nand-ecc-mode = "hw";
13    nand-on-flash-bbt;

15    spl@0 {
        label = "SPL";
17      reg = /bits/ 64 <0x0 0x400000>;
      };
19
      spl-backup@400000 {
21      label = "SPL.backup";
```

```
        reg = /bits/ 64 <0x400000 0x400000>;
23      };

25      u-boot@800000 {
          label = "U-Boot";
27        reg = /bits/ 64 <0x800000 0x400000>;
        };
29
        env@c00000 {
31        label = "env";
          reg = /bits/ 64 <0xc00000 0x400000>;
33      };

35      env-backup@1000000 {
          label = "env.backup";
37        reg = /bits/ 64 <0x1000000 0x400000>;
        };
39
        rootfs@1400000 {
41        label = "rootfs";
          reg = /bits/ 64 <0x1400000 0x1ec00000>;
43      };
    };
45  };
```

Listing 5.9: The device-tree file used: definition for the NAND device.

5.3.4 Building and Bundling

The initramfs is pre-built and not created by the build system directly. It is provided to the kernel during the preparation stage before compiling. The kernel, its configuration and the device-tree are copied to the deployment directory at the deploy stage. The tools provided in the CHIP-SDK are used to post-process those files together with the rootfs to get the needed files for directly flashing onto the hardware.

5.4 Root Filesystem

The rootfs' contents are defined by the image recipe, which basically is a list of all recipes needed to be built and installed. Listing 5.10 gives an overview of all used recipes besides the minimal rootfs selected by chip-hwup-image.bb. Since not all the work could be done by recipe definitions, some post-processing tasks needed to be done manually. The do_rootfs_postprocess is a command sequence which defines the work needed to be done. The function's contents in the listing are only a subset of all tasks needed to be done and should only be a representative. The variable ROOTFS_POSTPROCESS_COMMAND at the end of the recipe declares the previously explained functions for execution when finishing the image building process.

```
1
IMAGE_INSTALL += " \
3       u-boot-fw-utils \
```

```
    uboot-scripts \
    update-scripts \
    e2fsprogs \
    openssl \
    openvpn \
    nano \
    ntpdate \
    sqlite3 \
    htop \
    qtbase \
    qtbase-tools \
    qtbase-plugins \
    qtscript \
    qtquick1 \
    qtquick1-plugins \
    qtxmlpatterns \
    qtdeclarative \
    qtdeclarative-qmlplugins \
    qtserialport \
    qtwebsockets \
    sems-application-essentials \
    sems-maintainer-scripts \
    sems-openvpn \
    sems-system-miscellaneous \
    sems-wpa-supplicant \
    sems-openssh-identity-chipprov1 \
    sems-identity \
    sems-console \
    sems-system-migration \
    sems-accesspoint \
    ca-certificates \
    bash \
    bash-completion \
    tzdata \
    tzdata-europe \
    openssh \
    i2c-tools \
    power-button \
    rsync \
    netcat-openbsd \
    hostapd \
    tcpdump \
    avahi-utils \
    pam-plugin-exec \
    sem-compact-datastore \
    mtd-utils \
    mtd-utils-ubifs \
    mtd-utils-misc \
    "

do_rootfs_postprocess () {

  ## we have to remove the normally provided wpa_supplicant.conf file
  ## but we need the rest of the wpa-supplicant package
  rm ${IMAGE_ROOTFS}/etc/wpa_supplicant.conf
  ln -s /config/wpa_supplicant.conf ${IMAGE_ROOTFS}/etc/wpa_supplicant.conf
```

```
61    ## same with resolv.conf
      rm ${IMAGE_ROOTFS}/etc/resolv.conf
63    ln -s /tmp/resolv.conf ${IMAGE_ROOTFS}/etc/resolv.conf

65    # remove link and make mount point for ramdisk
      rm ${IMAGE_ROOTFS}/var/tmp
67    mkdir ${IMAGE_ROOTFS}/var/tmp

69    # add sems user (EXTRA_USERS_PARAMS is executed later, because sed could
        never overwrite the PW)
      echo "sems:x:65533:" >> ${IMAGE_ROOTFS}/etc/group
71    echo "sems:x:65533:65533:SEM Console User:/home/sems/:/var/run/application
        -console" >> ${IMAGE_ROOTFS}/etc/passwd
      echo "sems:!:17422:0:99999:7:::" >> ${IMAGE_ROOTFS}/etc/shadow
73  }

75  ROOTFS_POSTPROCESS_COMMAND += "do_rootfs_postprocess; "
```

Listing 5.10: The image recipe append file includes all software deployed on the target rootfs (chip-hwup-image.bbappend).

5.4.1 Software and Libraries Shipped

The rootfs contains various software packages and libraries mainly already provided by an existing recipe in the used layers. An example would be the selected ones in listing 5.10 between line 3 and 23 or 34 and 51. Those are various tools for U-Boot, time synchronization, file system management, secure connections or even maintenance and development. The Qt5 framework is divided in several packages which are therefore selected separately in the image recipe.

All the selected recipes starting with "sems-" are newly introduced ones from the meta-sems layer. Those were created and contain mainly bash scripts, identities and services. Important services and helpers are discussed in subsection 5.4.3.

5.4.2 Root File System Preparation

The resulting rootfs needed is a single file used for deployment which basically is just a squashfs container. This means that it is a complete file system in a read-only mode with a high compression factor. Since there is no need for a writeable system, a lot of overhead is avoided that way and compression can be done very easily with the non-changeable contents.

The YOCTO build process, which is responsible for assembling the rootfs with the contents of the selected packages, can be configured with different options how to output the result. Multiple structural forms are simultaneously possible. Very common formats are a tar archive or a SD memory card image. The used way is not that common for packing the rootfs for deployment in the target system's case. It is a tar archive which later on is post-processed.

After finishing the build with BitBake, post-processing tools are needed to prepare the resulting software for the update system or initial flashing. The vendor from the SoM provided such tools. The only tasks needed to be performed were removing the boot folder from the rootfs, pack the rest to the squashfs container and create suite-able packages.

There are two packages needed. One is for the update system itself which are the container and the corresponding kernel together in an archive. The second is for flashing the hardware itself. But here the structural format was already defined by the provided post-processing tools. Basically, the same container as before is placed together with the previously extracted boot folder in a new file system which normally serves as the rootfs. This file system is exactly the same image store as described previously in subsection 4.5.4 on page 29. So, other containers are also deployed there as shown on figure 4.11.

5.4.3 Services and Helper Routines

All processes regarding the system and application updates are initiated by the running application. Since systemd is (among other things) used for service management, all vital update processes are started through it. This includes the service for the application itself which is auto-started at boot time and two further services for the updates itself. Those are triggered by the application and systemd provides the interface for that.

The update services are mostly hardware independent. From an application perspective there is no difference at all. However, under the hood, the correct commands needed to be executed to comply with the bootloader environment interface used at the hardware target.

The following subsection describes the provided services and tools in more detail.

5.4.3.1 Application

As stated before, the application is started automatically at boot. The correct order of all auto-started services at boot is important so that the system is in the correct state when starting the application to comply with its start-up dependencies. For instance, migration scripts after system updates must be finished before the application starts. The systemd service for the application simply starts the so-called start. dispatcher which is responsible for selecting the correct application. Due to crash safe updating, the application is redundant just like the system itself. After dispatching, the init file for the selected application is executed.

```
1 [Unit]
  Description=Smarthome Software Stacks
3 Requires=syslog.service
  After=syslog.service
5
  [Service]
7 EnvironmentFile=/etc/profile.sems
  WorkingDirectory=/tmp
```

```
9  ExecStart=/system-scripts/start-dispatcher.sh
   PIDFile=/var/run/sems-systemcontrol.pid
11 Type=simple
   Restart=always
13 #TimeoutStopSec=5s

15 [Install]
   WantedBy=multi-user.target
```

Listing 5.11: The service definition sems.service for the application.

Listing 5.11 shows the applications service definition. The hard requirement before starting the service are listed in the unit section just after name and description at line 3 and 4. It is the system's logging service called syslog which is de facto the standard on Linux systems. One specialty of the application is that it is written with Qt5. Therefore, a few variables in the execution environment must be set prior to running as shown on line 7. Further on the path to the file called at service start with some basic service related settings is defined. The last line is only needed when the service can be activated and deactivated, but this is not possible in our read-only system and therefore will not be needed at all. Still, it is nice to have it there when porting the application to other non-embedded systems.

5.4.3.2 Application Update

The interface for using the system update consists of a systemd service to start and a fixed file location for placing the update file. In the case of the application, the new one must be present on a known path. Listing 5.12 shows the quite simple service definitions. It is a manually started service, which simply executes the update script for the application which is described, later on. Most definitions are the same as for the normal application service.

```
   [Unit]
2  Description=Smarthome Software Update
   Requires=syslog.service
4  After=syslog.service

6  [Service]
   EnvironmentFile=/etc/profile.sems
8  WorkingDirectory=/
   ExecStart=/system-scripts/start-update.sh
10 PIDFile=/var/run/sems-update.pid
   Type=simple
12 Restart=no
```

Listing 5.12: The service definition sems-update.service for application updates.

The core script for the application update is shown at listing 5.13. The first lines up to 11 are some basic definitions and default values for saving the detection states of the old and new update format. Before the application was packed as squasfs the files were directly swapped out inside a writeable ext4 container. For better readability, the legacy paths between line 65 and 66 are left out here.

The first task is to detect the unused application number. This is done after a safety check starting at line 15. Next is the detection of the update format. In the old variant, it was a folder which could not be mounted like the squashfs container. This check is performed from line 32 to 43. A similar detection is done for the update to be replaced. Normally, a squashfs to squashfs update is performed by simply deleting the old update and moving the new one to the old one's location. The commands performing this task are listed at line 70 to 75.

```
#!/bin/sh

USE=$(cat /config/apptouse)
FALLBACK=$( cat /config/apptousefallback)

NOTIFY=/tmp/updatesadlyfailedfile
UPDATE_CONTAINER=/tmp/newapp.img

USE_SQUASHFS="no"
OLD_IS_SQUASHFS="no"

rm $NOTIFY &> /dev/null

if [ $USE -ne "1" ] && [ $USE -ne "2" ]
then
  CURRENT=$FALLBACK
else
  CURRENT=$USE
fi

if [ $CURRENT -eq "1" ]
then
  START="2"
else
  START="1"
fi

DEST_EXT4="/mnt/app$START"
DEST_SQUASHFS="/mnt/images/app${START}.img"

# first check for app container
if [ -f $UPDATE_CONTAINER ]
then

  # test mount new container
  TEMP=$(mktemp -d)
  mount $UPDATE_CONTAINER $TEMP
  [[ $? -ne 0 ]] && echo "error could not mount new app container" > $NOTIFY
    && echo "update-error" && exit

  umount $TEMP
  rmdir $TEMP
  USE_SQUASHFS="yes"

else
```

```
  # else go on with old style
  if [ ! -d /tmp/updatebins ]
  then
    echo "update dir not existing" > $NOTIFY
    exit 1
  fi
fi


file $DEST_SQUASHFS | grep Squashfs > /dev/null
if [ $? -eq 0 ]
then
  OLD_IS_SQUASHFS="yes"
fi

#umount container which is not used (to be sure)
umount $DEST_SQUASHFS 2> /dev/null

if [ $OLD_IS_SQUASHFS = "no" ]
then
else
  if [ $USE_SQUASHFS = "yes" ]
  then # squashfs to squashfs
    # remove old
    rm ${DEST_SQUASHFS}

    # relocate new
    mv $UPDATE_CONTAINER ${DEST_SQUASHFS}
    [[ $? -ne 0 ]] && echo "error moving app container" > $NOTIFY && echo "
    update-error" && exit

  else # this case is not supposed to happen (squashfs -> ext4)
    echo "error: downgrade from squashfs not supported" > $NOTIFY
    exit

  fi
fi

sync
echo $START > /config/apptouse
sync
echo $START > /config/apptousefallback
sync

systemctl stop sems.service
systemctl start sems.service
```

Listing 5.13: The core script for the application update (short version without legacy paths).

Regardless of the update formats, the application to be used for the next start-up is written back to the configuration folder, so the start dispatcher selects the correct one in future application starts.

5.4.3.3 System Update

The system update process interface is very similar to the one described in the previous section. For the more complex system update the rootfs container and the kernel image files must both be moved to specific locations prior to starting the update via systemd. The definition for the service is shown in listing 5.14. Only the executed script at service start is different and as before shown in addition.

```
[Unit]
Description=Smarthome Software Update
Requires=syslog.service
After=syslog.service

[Service]
EnvironmentFile=/etc/profile.sems
WorkingDirectory=/
ExecStart=/system-scripts/start-system-update.sh
PIDFile=/var/run/sems-system-update.pid
Type=simple
Restart=no
```

Listing 5.14: The service definition sems-system-update.service for system updates.

The core part of the complete system update process is displayed at listing 5.15. It is a bash script performing all essentials tasks. The basic operations are determining which kernel name will be the next, removing unused rootfs images from the image store and relocate the new system. Lastly, the information to boot the other system is written to the bootloader environment. All operations are written to a log file as declared at line 5.

The locations for the new kernel and the new system are specified in lines 3 and (implicitly) in line 48. The path for the new kernel is determined by reading the current system number from the bootloader environment starting at line 10 and ending at 28. Next, the current system number is read from the currently mounted rootfs container at line 31 and all systems but the running one are removed from the persistent memory (the image store). Finally, both images are moved at line 48 and 52.

```
#!/bin/sh

KERNEL_LOCATION=/tmp/newkernel

LOGFILE=/var/log/sysupdate.log

[ ! -e $KERNEL_LOCATION ] && echo "error: $KERNEL_LOCATION (kernel) not
    existing " >> $LOGFILE && exit 1

echo "Loading environment variables"
USE_NEXT=$(fw_printenv -n use_next)
RET=$?
USE_NEXT_NEW=$USE_NEXT

if [ $RET -ne 0 ]
then
```

```
16    echo "Cannot read environment" >> $LOGFILE
      exit 2
18  fi

20  #getting kernel name
    if [  $USE_NEXT -ne 1 ]
22  then
      USE_NEXT_NEW=1
24    KERNEL_NAME=/mnt/images/boot/zImage1
    else
26    USE_NEXT_NEW=2
      KERNEL_NAME=/mnt/images/boot/zImage2
28  fi

30  #preparing system
    CURRENT_SYS=$(losetup -a | grep "system-v" | grep "/mnt" | cut -d" " -f3 |
        grep -o '[^/]*$')
32  SYSTEMS=$(ls -1 /mnt/images/ | grep system-v)

34  echo "searching for obsolete system images" >> $LOGFILE

36  for SYS in $SYSTEMS
    do
38    if [ $SYS = $CURRENT_SYS ]
      then
40      echo "$SYS is the running system" >> $LOGFILE
      else
42      echo "$SYS is not the running system, deleting" >> $LOGFILE
        rm "/mnt/images/${SYS}"
44    fi
    done
46
    echo "relocating new system image" >> $LOGFILE
48  mv /mnt/images/tmp/system-v* /mnt/images/
    [ $? -ne 0 ] && echo "error moving new system image" >> $LOGFILE && exit 3
50
    echo "copy kernel" >> $LOGFILE
52  cp $KERNEL_LOCATION $KERNEL_NAME
    [ $? -ne 0 ] && echo "error copying new kernel " >> $LOGFILE && exit 4
54
    echo "preparing and saving" >> $LOGFILE
56
    fw_setenv use_next $USE_NEXT_NEW
58  RET=$?
    if [ $RET -ne 0 ]
60  then
      echo "error writing environment. Update ready but u-boot maybe don't get
        it" >> $LOGFILE
62  fi

64  echo "every thing looks fine" >> $LOGFILE
    reboot
66  exit 0
```

Listing 5.15: The core script for the system update.

When everything worked fine up to this point, the bootloader is informed by writing

its environment and on success, the system is finally rebooted to finish the system update. Those operations are between line 57 and 65.

5.5 Tools Provided for Application Development

Yocto provides a way to create complete toolchains for a given hardware target. The output format is a single bash script which contains the relocateable complete SDK with all tools needed for application development and cross compiling.

The out-of-the-box settings were quite good and only one cmake related change was introduced with a recipe append file. When the resulting bash script is executed, it uncompresses and installs the contents to a chosen directory. For the use of cmake the included environment file can be sourced prior to building any software using cmake. Nothing further has to be done to be able to cross compile. This makes installing and using multiple toolchains very easy.

Chapter 6

Testing and Results

The testing and validation process can be divided in two major parts. The first covers tests and evaluations during development like checking for correct operation of all drivers and processes. Ensuring the correct operation of all products during mass production represents the second part.

Like the tests the results are grouped by the very same major topics. The first part primarily consists of evaluations of the related results which are simpler ones like pure measurements of sizes and boot times. The following sections discuss the tests and their results in detail.

6.1 Performed Tests

The first step regarding testing was the evaluation of the bootloader and Linux. After that all auxiliary tools like various helper scripts and the update processes for the system and the application were tested. Next manual tests were performed to check the hardware itself while building the first prototypes. Those testing processes then were automated in the best possible way to test the first batch in a mass production manner.

6.1.1 Evaluation During Development

Evaluation of the various involved components was a continuous process since most tasks performed during development depend on one another. This is as easy as the testing of various hardware parts, for example the RTC depends on a running Linux which itself depends on a working bootloader.

The used bootloader and Linux kernel were already running on the hardware target as stated in various sections of the concept and implementation chapters. This firstly allowed the integration of the various device drivers before modifying the kernel and the bootloader for the desired updating capabilities. While performing both tasks an interactive test process took place.

With all drives running, the watchdog functionality and update processes were tested in-depth. After fully integrating the complete software, some test cases needed to be

defined to be able to test multiple electronics for correct operation. Since the correct operation of the RTC, the Z-Wave module or the UI did not matter for booting Linux such faulty hardware parts and devices easily could be missed. Also, errors while soldering could be the root cause of some hardware malfunction.

6.1.2 Testing the First Batch

After the first prototypes were tested and the product design was validated, preparing the mass production was on the schedule. All previously manually performed steps had to be automated as much as possible. This subsection focuses on the tasks for the hardware target itself and its associated test software running on it. Additionally, a second software was needed for displaying testing instructions and information gathering, which runs on another device. Such devices could either be an embedded system with a display attached, a simple PC or a laptop.

The test software running on the target was built as a dedicated image which was stored on a USB stick. This medium then was plugged in during testing before powering up the targets. This lead to starting the software from the stick instead of the main application. Dispatching which software to launch was done during the early start-up stage of the main software. This early stage routine is a separate binary for achieving this. Software on external media must be signed with the correct signing authority in order to get it accepted and started.

After the dedicated test software is started, it validates that it is running on the correct hardware target at then starts to perform the automatic tests like testing WLAN or the Z-Wave module. This stage is indicated by a special LED combination. During this stage the test user which previously connected the USB stick and powered the DuT did not have to do anything. This offers the possibility to parallelize the process. Figure 6.1 shows three DuTs with their tiny USB stick on the lower side and their power supply connected. Optionally, the test operator can already continue with the following step, which will be explained next, before the LED state shows that the initial check has finished.

With the LEDs, the test software indicates the readiness for the connection to the second test software running on a separate device with a display to track the results. The connection is done via the USB OTG jack on the top side, as shown on figure 6.2. If the automatic tests fails at this point, the overall test result is set to failed. The result is stored and the test user is informed about it and instructed to discard the DuT. If there are no problems, the manual tests are performed.

After the automatic stage of testing and connecting the DuT via USB, the test user has to perform some manual tests like pushing buttons or checking the LED channels for correct functionality. The test software with the display instructs step by step what to do. When all tests are performed, the test result for the DuT is saved. On success, a provisioning can be performed optionally. This means that the Compact Unbranded is branded, gets its labels including the serial number and the related information to the

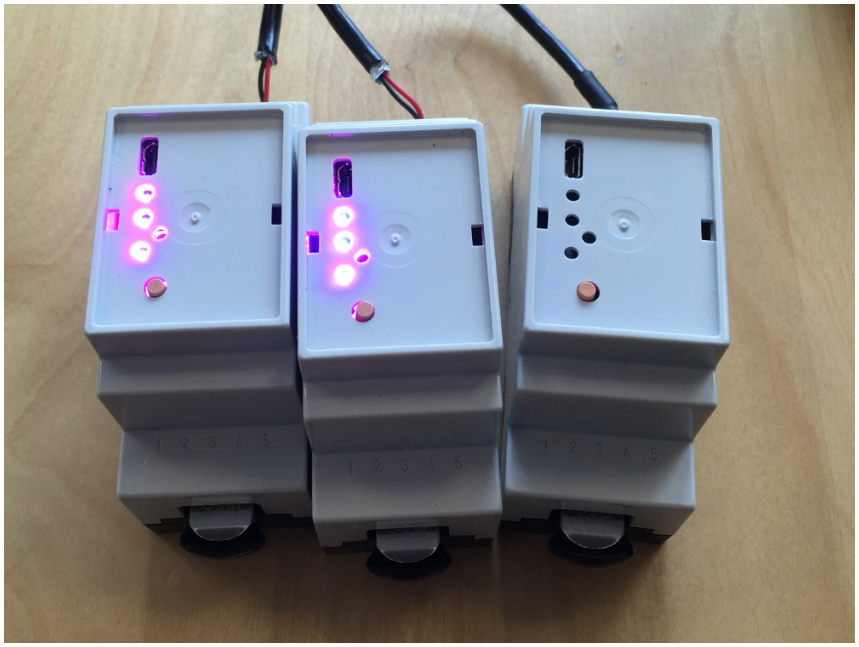

Figure 6.1: Multiple Compact Unbranded connected to power and plugged in USB stick with test software.

Figure 6.2: Single Compact Unbranded (top side) while testing with connected test display.

Name	Description	Size in MiB
u-boot.bin	bootloader image	3.4
kernel	kernel image with initramfs	6.2
system-v10.img	rootfs (squashfs container)	49

Table 6.1: Resulting sizes of various major software components.

branding and identity is stored on the device. After that the resulting product like the SEMcompact is finished.

Basically, all actions performed for the first batch are the same as for upcoming mass production which clearly have a quality assurance (QA) strong background. Therefore, keeping track of all serial numbers involved while testing is one main part of the second test software, which keeps track of all test processes.

6.2 Results

The same structure applies for this section as for the Performed Tests section. At first, evaluation of correct operation for prototypes and some numbers associated with it are discussed and then results and the challenges while testing the first batch in a mass production manner is explained.

6.2.1 Prototypes

Most evaluation and testing work was done with the first two hardware prototypes. Also, a few bugs in the schematics were discovered during this time. While being half-way with the implementations for the bootloader and the Linux, the second-generation prototype arrived with the correct schematics and routing which made a lot of the work easier. There was no need for further workarounds. Since most results are very simple due to the binary outcome when evaluating the compliance with the specification, a few further results like boot times and images sizes are discussed in the following sub sections.

6.2.1.1 Sizes of Finished Images

Table 6.1 shows an overview of the most important components handled when flashing the hardware target or creating updates. The resulting U-Boot image is usually less than 4mebibyte (MiB) in size as the reserved size is the same with most hardware targets in general. The kernel's size is very big compared to embedded systems due to the embedded initramfs. The displayed size of the rootfs is for compressed squashfs container as shipped at updates and saved on the image store.

Step	Indication	Duration in s	Total s
Power-on	all LEDs white	0	0
H/W and bootloader init	first LED violet	0.25	0.25
Kernel drivers loaded	second LED violet	8.1	8.35
Application start	third LED violet	20.9	29.25
Application loaded	normal LED operation	16	45.25

Table 6.2: The resulting time spans needed for various steps in the boot process.

6.2.1.2 Boot Times

One performance indication is the time the product needs to perform a complete start including fully starting the application running on it. Since the application itself was out of the scope of this work, its time consumption is listed in table 6.2, but not explained in more detail.

A few steps had to be defined to be able to determine the various time spans. Since the LEDs represent the states of the device including the boot process and power-up, they define a few useable points already. The hardware itself always turns on all LED channels to show for a short amount of time that all channels are working and the device got power. This is done in hardware and no software is running at this point. Therefore, it happens immediately after applying power and marks the start point for the time measurements.

When U-Boot finishes with the start process and all hardware related drivers are loaded, the LEDs are switched on and off differently. The same applies to the kernel when all drivers are loaded. When the application is starting up it does the same. So in all stages, when the first initialization is done, at least one LED changes its color. Regardless of the stage, at least one LED channel glows solidly - at least during startup. When the application has finished its startup, the LEDs are blinking and switched on or off like the application it defines.

6.2.1.3 Hardware Limitations

Due to the low price for the Compact Unbranded there are some hardware limitations. The hardware watchdog capabilities are very limited on the SoM: One limitation is the very short maximum possible timeout and the other the missing reset cause on startup which is associated in the case of a timeout. The maximum timeout of 16 seconds is far shorter than the complete boot time including boot strapping the application at its start. Thus, during different stages while booting, the watchdog had to be kicked. This is explained at section Ensuring Always-On and its associated figure 4.15. In short, it was not an unsolvable problem, but it caused extra work.

Feature-rich hardware platforms normally provide a register for the bootloader to determine the reset cause. The reset cause most common is a normal start which is often referenced as PWR or power. In the case of the hardware watchdog resetting the CPU

Type	First run	Second run
RTC	41	4
Boot	5	5
Z-Wave	9	9
USB	18	18
LED	5	5
Total errors	78	41
Affected DuTs	69	30

Table 6.3: Detected errors and affected DuTs during first batch of mass production.

or the whole SoM the watchdog as a reset cause can be read from the very same register. With this, it is possible to differentiate between a failed update and a power failure during the first startup after updating. Because of that a new try when booting up the freshly installed system cannot be performed after a power failure. The complete system update process including downloading and preparing the update itself must be re-performed. Since such an update is hardly started and power outages are not common in central Europe, this is not really a problem.

6.2.2 First Batch

Compared to the manual test and evaluation performed with the prototype systems the results for the first batch differ in their types. Testing them is mainly due to QA reasons and therefore the results sum up the detected errors and possible causes for them. After the first test run, the test software running on the DuT itself needed to be reworked slightly as it produced a lot of false test results.

The device count for the batch was 445. In the first run a lot of RTC errors were detected which lead to a second run. Table 6.3 shows the detected errors by type. Some DuTs had multiple errors so the total faulty device count at first was 69. In addition, while testing the first few devices the WLAN test nearly always failed. So before the mentioned first run, the test software on the DuT was already reworked slightly.

When a DuT does not start at all, restarts frequently or unattended, it is accounted as a boot error. When a hardware component does not work properly like the RTC or the Z-Wave module, it is accounted at their corresponding entries in the table. If either the test software from the USB stick does not start or a connection to the second test software cannot be established, the errors are treaded as the same for statistical purposes. At last, the detected malfunctions by the test operator at the LEDs are accounted separately, as shown in the table.

Some devices suffer from multiple errors, so the total error count is higher than the faulty DuT count. As mentioned before, there were a lot of RTC errors after the first

run, so a manual re-evaluation was performed for these devices. As it turned out, most of them were working properly and the test software needed to be reworked. After testing the devices again, only four of them still had the same issue.

Additional USB errors occurred during testing which were not DuT related. The frequent use of the very same micro-USB cables lead to fast wear-offs and the USB connections often failed after some time. This issue can only be detected by the test operators.

While developing the test software a run prior to the listed first run in table 6.3 was performed to test and evaluate the first tests automatically performed. The results were not tracked at that time. But a lot more errors were detected there due to the fact that at the manufacturer's site, the manual soldering work was not performed very well. Therefore, a lot of LED channels did not work at first. Those devices could mostly be repaired. There were additional problems but they numbers were not very high.

As stated in the very same table, a lot of devices still are not working properly. To be more exact 6.7% are still faulty and might not be repairable. But at the time of writing, there was no need for that. To be even more exact, some devices were marked as faulty by the manufacturer due to their first hardware tests. The total count of devices ordered was 479 what leads to an error rate of 13.4%, which is quite high. Error rates while manufacturing electronics are usually still at about 3% even with very high volumes. When manufacturing the next batch, the common errors are known, and the manufacturer will be able to decrease the error rate to get closer to the mentioned 3%.

Chapter 7

Conclusion and Outlook

For the given hardware target, all goals could be reached. Due to some hardware limitations, some trade-offs had to be made. The used bootloader U-Boot and the Linux kernel now support crash-safe updating. This includes all auxiliary tools, which were created to achieve this. All drivers necessary were already there, so it was not difficult to integrate them.

Almost the complete bootloader, kernel and rootfs builds are fully automated to be able to create everything from source. Only a few manual steps must be performed for the image generation both for initial flashing and for creating updates needed for over-the-air delivery.

All new software components which are mainly Yocto layers and recipes can be used with other hardware targets very easily. All patches and custom scripts are included there. Only the hardware specific post-processing tools are specific to the target. So if another SoM or custom hardware already has Yocto recipes, it should not be much work to achieve the same goals as in this thesis. All recipes needed for the application and support are mostly hardware independent and should be useable without any modification. This includes scripts, services and software like the Qt5 framework, sending debug information or tools for formatting persistent storages during first startup.

The update service which distributes the created updates (system and application) works very well. Multiple projects and hardware IDs are already registered there, managing 700 products of two different brands. Due to its well-defined abstraction within the update format, it can be used without modifications in future projects.

A complete testing concept for display-less DuTs was designed and implemented in a side track. While most tests are only depending on the hardware target itself (used components, the UI design and so on) common parts like central serial number or test result tracking can be reused without much modifications in other projects.

In future, there is some space for improvement like improving the Yocto meta-layer structure. Since this was the first time working with this build system, not every modification could be done in own layers. Some configurations needed to be changed in the

existing layers. This could be done more cleanly in future. The generation of the initramfs could be automated as well. Currently, it is manually packed and provided to the build process as an archive. The post-processing for creating a system update, namely the squashfs container, could also be done with Yocto, since it is only a new output format for the rootfs. Also, a developer image for deployment while development might be possible more easily then, with the rootfs creation more automated. This would be a gain because otherwise, it is very time consuming to debug some issues with the read-only file system, and a dedicated developer image just could be writeable.

For the built products, the Compact Unbranded and its brands like the SEMcompact, the used SoM is not available any more due to bankruptcy of the vendor during the testing phase of this work. So, a new SoM needs to be selected and integrated. The custom hardware around it was already built for exchanging the SoM. Since, as previously described, all layers and recipes are designed for application to other hardware targets, this should not be a big problem. It only might be some extra work if the bootloader is not U-Boot.

Not covered by the test and evaluation process are long term effects. Due to an issue with the NAND driver for the used processor family, a few systems did fail at customer's sites. These needed to be started with another kernel sideloaded to be able to boot at least one time correctly. After that, it was possible to update the systems to the newer versions while running the faulty kernel driver. Avoiding this would mean to set up some tests, which aim to test long time effects such as the persistent memory getting fully written on more than one time, so the NAND has to actually erase sectors. This normally would not happen very fast since there are 512megabyte (MB) of persistent storage available and the application hardly writes anything on it. Therefore, several weeks or months might pass before the first blocks need to be erased and cause such problems.

Appendix A

List Of Abbreviations

API application programming interface

BIOS Basic Input/Output System

CA Certification Authority

CN common name

CPU central processing unit

CRL certificate revocation list

CSR Certificate Signing Request

DOA degree of autonomy

DSM demand side management

DSMS demand side management system

DuT device under test

EMC electromagnetic compatibility

EMI electromagnetic interference

EMS energy management system

GPIO general-purpose input/output

GUI graphical user interface

HAB High Assurance Boot

HDD hard disk drive

HTTP Hypertext Transfer Protocol

I2C Inter-Integrated Circuit

IDS intrusion detection system

IoT Internet of Things

IP Internet Protocol

IPS intrusion prevention system

LED light-emitting diode

MB megabyte

MiB mebibyte

NFS network file system

OCSP Online Certificate Status Protocol

OS operating system

OTG On-The-Go

OTP NVM one-time programmable non-volatile memory

PCB printed circuit board

PKI public key infrastructure

PMIC power management IC

PV photovoltaics

QA quality assurance

RAM random access memory

REST Representational State Transfer

ROM read-only memory

rootfs root file system

RTC real-time clock

SCR self consumption rate

SDK software development kit

SEMcompact Smart Energy Manager Compact

SEM Smart Energy Manager

SEMS Smart Energy Management System

SLC single-level cell

SoC State-of-Charge

SoM System-on-Module

SPL Secondary Program Loader

SSH Secure Shell

TPM Trusted Platform Module

UART universal asynchronous receiver-transmitter

UBIFS Unsorted Block Image File System

UBI Unsorted Block Images

UI user interface

URL Uniform Resource Locator

USB Universal Serial Bus

WLAN wireless local area network

YP Yocto Project

Bibliography

[Fuc17] Thomas Fuchs. Hardwareentwicklung und emv messungen eines elektronischen gerätes. Technical report, Graz University of Technology, February 2017.

[Fuc18] Thomas Fuchs. Design und entwicklung eines smart home devices zur steuerung von variablen verbrauchern zur effizienten nutzung von erneuerbaren energien. Master's thesis, Graz University of Technology, July 2018.

[GDLZ11] Haijun Gu, Yufeng Diao, Wei Liu, and Xueqian Zhang. The design of smart home platform based on cloud computing. In *Electronic and Mechanical Engineering and Information Technology (EMEIT), 2011 International Conference on*, volume 8, pages 3919–3922, Aug 2011.

[HL10] Dae-Man Han and Jae-Hyun Lim. Smart home energy management system using ieee 802.15.4 and zigbee. *IEEE Trans. on Consum. Electron.*, 56(3):1403–1410, August 2010.

[Hof14] Florian Hofer. Home automation communication standards. Technical report, Graz University of Technology, April 2014.

[JCZ12] Tao Jiang, Zhe Chen, and Qiu Zhang. A brief analysis of the embedded-qt4 technique based on linux. In *Computing, Measurement, Control and Sensor Network (CMCSN), 2012 International Conference on*, pages 332–335, July 2012.

[LNNJ12] M.S. Loft, S.S. Nielsen, K. Norskov, and J.B. Jorgensen. Interplay between requirements, software architecture, and hardware constraints in the development of a home control user interface. In *Twin Peaks of Requirements and Architecture (Twin Peaks), 2012 IEEE First International Workshop on the*, pages 1–6, Sept 2012.

[LVS15] Matthew T. Lawder, Vilayanur Viswanathan, and Venkat R. Subramanian. Balancing autonomy and utilization of solar power and battery storage for demand based microgrids. *Journal of Power Sources*, 279:645 – 655, 2015. 9th International Conference on Lead-Acid Batteries – LABAT 2014.

[Men16] Christian Mentin. Entwurf, aufbau und analyse einer skalier- und erweiterbaren systemplattform zur entwicklung eines smart energy managers. Master's thesis, Graz University of Technology, April 2016.

[NHC13] Liu Ningqing, Yan Haiyang, and Guan Chunmeng. Design and implementation of a smart home control system. In *Instrumentation, Measurement, Computer, Communication and Control (IMCCC), 2013 Third International Conference on*, pages 1535–1538, Sept 2013.

[PME+09] N. Papadopoulos, A. Meliones, D. Economou, I. Karras, and I. Liverezas. A connected home platform and development framework for smart home control applications. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, pages 402–409, June 2009.

[Pro18a] Yocto Project. Yocto project reference manual. `https://www.yoctoproject.org/docs/2.5/overview-manual/overview-manual.html`, 2018. Accessed: 2018-07-07.

[Pro18b] Yocto Project. Yocto project – it's not an embedded linux distribution – it creates a custom one for you. `https://www.yoctoproject.org/`, 2018. Accessed: 2018-07-07.

[RHM+12] J.F. Ruiz, R. Harjani, A. Mana, V. Desnitsky, I. Kotenko, and A. Chechulin. A methodology for the analysis and modeling of security threats and attacks for systems of embedded components. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 261–268, Feb 2012.

[Sle13] Gunnar Sletta. Introducing boot to qt – a technology preview. `http://http://blog.qt.io/blog/2013/05/21/introducing-boot-to-qt-a-technology-preview/`, 2013. Accessed: 2018-07-07.

[WHD+11] P. Wang, J. Y. Huang, Y. Ding, P. Loh, and L. Goel. Demand side load management of smart grids using intelligent trading/metering/ billing system. In *2011 IEEE Trondheim PowerTech*, pages 1–6, June 2011.

[WSES13] Haidong Wang, J. Saboune, and A. El Saddik. Control your smart home with an autonomously mobile smartphone. In *Multimedia and Expo Workshops (ICMEW), 2013 IEEE International Conference on*, pages 1–6, July 2013.

[WWSL09] Zhenxing Wang, Shutao Wei, Linxiang Shi, and Zhongyuan Liu. The analysis and implementation of smart home control system. In *Information Management and Engineering, 2009. ICIME '09. International Conference on*, pages 546–549, April 2009.

[yBpLfW10] Chun yue Bi, Yun peng Liu, and Ren fang Wang. Research of key technologies for embedded linux based on arm. In *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, volume 8, pages V8–373–V8–378, Oct 2010.

[YH11] Xiaojing Ye and Junwei Huang. A framework for cloud-based smart home. In *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, volume 2, pages 894–897, Dec 2011.

[YWJ+10] Yongquan Yang, Zhiqiang Wei, Dongning Jia, Yanping Cong, and Ruobing Shan. A cloud architecture based on smart home. In *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on*, volume 2, pages 440–443, March 2010.