Delft University of Technology
Master's Thesis in Embedded Systems

Design and Implementation of Fault-Tolerant Firmware Updating Methods for Time-Critical Battery Products

Chiel de Roest

embedded
software

TU Delft
Delft University of Technology

Design and Implementation of Fault-Tolerant Firmware Updating Methods for Time-Critical Battery Products

Master's Thesis in Embedded Systems

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Chiel de Roest
M.A.deRoest@student.tudelft.nl
mderoest@tesla.com
chielderoest@gmail.com

March 2017

Author
Chiel de Roest (M.A.deRoest@student.tudelft.nl, mderoest@tesla.com)

Title
Design and Implementation of Fault-Tolerant Firmware Updating Methods for Time-Critical Battery Products

MSc presentation
April 25, 2017

Abstract

The world is transitioning to a low carbon power generation system, and local energy storage plays a key role in this transition. Many of such energy storage systems comprises batteries and power electronics that is controlled by firmware. Uptime of these systems is vital, and consequently any downtime should be avoided.

This thesis discusses the impact of firmware updating on the time-critical embedded systems, in particular energy storage systems. Concretely, firmware update procedures for the Tesla Powerwall product is designed, implemented and evaluated. The engineered solution offers a flexible update method: it supports multiple physical layers for data communication and the used communication speed does not affect the downtime of the product. The developed update procedure is shown to be reliable and has been used tens of thousands of times to date. The downtime has shown to be mainly dependent on the speed of the CAN bus, where the file transfer dominates the update process duration.

Preface

This thesis presents the work I performed over the last one and a half year at Tesla, Inc. in Palo Alto, California. This document and the accompanying thesis defense form the last piece of my Embedded Systems master's program.

I would like to thank Tarmigan Casebolt for being my mentor, and for convincing me to perform my thesis work at Tesla and brainstorming about thesis subjects. His mentoring has made the work I did very pleasurable and he has made me a better engineer in every way. I would like to thank Michael Carmel, my manager, for giving me the opportunity to perform my thesis work in his group and for reviewing my work. In addition, I would like to thank all my colleagues at Tesla for their daily motivation.

I would like to thank Przemysław Pawełczak and Koen Langendoen for their ideas and for reviewing my work. Lastly, I would like to thank my family for their support during my studies.

Chiel de Roest

Delft, March 2017

Contents

Chapter 1

Introduction

We're running the most dangerous experiment in history right now, which is to see how much carbon dioxide the atmosphere can handle before there is an environmental catastrophe.

Elon Musk

1.1 Renewable Energy

Over the next decades, the world is transitioning to a low carbon power generation system. More and more energy will be sourced from alternative energy sources such as solar power, wind power, and hydropower. Moreover, an increasing number of home owners have the desire to being self-sufficient in their energy generation and consumption. Stationary energy storage solutions are playing a key role in the reliable and economic operation of grid energy solutions with significant renewable power sources [1, p. 519].

On April 30th 2015, Tesla, Inc. introduced the *Tesla Powerwall*: a stationary energy storage system designed for residential usage.[1] A Powerwall enables home owners to charge the Powerwall during times when solar panels are generating electricity, or during off-peak hours when the price of energy is low, and discharge during high on-peak hours. Another use case for the Powerwall is to provide backup power during a power grid outage.

The system layout of a home power system using the Powerwall is depicted in Figure 1.1. An inverter is connected to both solar panels and to the Powerwall. The inverter runs optimization strategies to make the best use of the available power and energy.

[1]https://www.teslamotors.com/powerwall

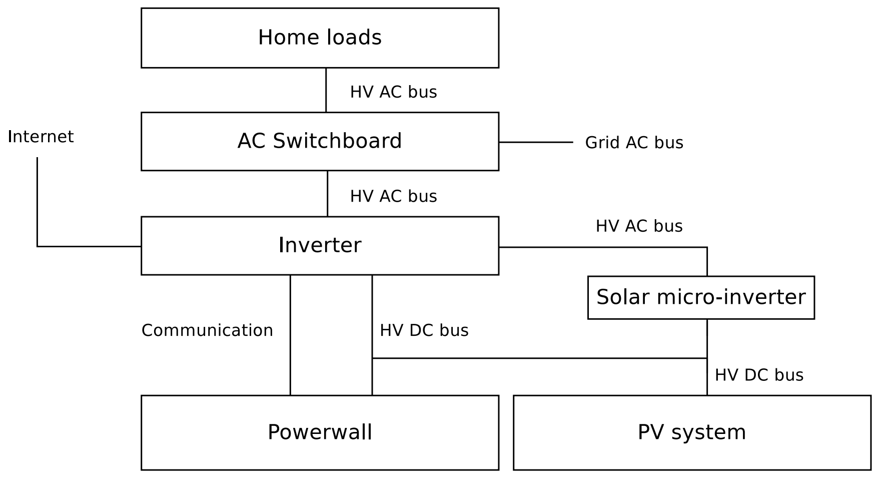


Figure 1.1: Overview of a residential stationary energy storage system that includes a solar generation installation. An inverter is connected by a DC bus to the Powerwall and to DC-coupled solar installations. Alternatively, solar installations are AC-coupled by solar micro-inverters. The inverter is connected to the home's electrical panel, which is the point of connection of the house with the electrical grid. Home loads, consuming AC voltage, are connected to the electrical panel. Most modern inverters are connected to the Internet, allowing monitoring of the system's performance and remote operation.

The inverter is connected to the Internet to allow data logging, remote diagnostics and firmware updating capabilities. A communications bus connects the inverter with the Powerwall, and enables the inverter to send operational parameters to and obtain operational information from the Powerwall.

1.2 Problem Statement

This thesis encompasses the challenge of updating the firmware of time-critical products. Such systems are required to have as little as possible downtime. Actions such as updating of the firmware of these products could introduce temporary or permanent (e.g. if a firmware update fails) downtime in the product's functioning.

The main statement is the following: *to what extent can the negative impacts of firmware updates of time-critical battery systems be minimized*?

1.3 Contributions

The work as presented in this thesis is the design, implementation and evaluation of a timely and robust firmware update method for a time-critical embedded systems product, the Tesla Powerwall.

Numerous design requirements, decisions and limitations are discussed with regard to topics such as the firmware update method's robustness, duration and extendibility. The result of this thesis is a reliable and extendible solution for firmware updating of embedded systems. Moreover, downtime of the Powerwall as introduced by the firmware update is minimized. The design can be applied to similar types of time-critical embedded systems.

This thesis is structured as follows:

- Chapter 2 discusses the system layout and restrictions of the Powerwall product. In addition, several detailed requirements for the Powerwall's firmware update process are specified.
- Chapter 4 discusses the requirements and design of the firmware update process. In addition, this chapter discusses a number of theoretical parameters of the design, such as firmware update duration and memory space usage.
- Chapter 5 evaluates the actual parameters the implemented firmware update solution. In particular, the duration of each step of the firmware update process is measured and the results are discussed.
- Chapter 6 discusses the conclusions to the implemented firmware update solution and discusses in what manner future work can improve the implemented solution.

Chapter 2

System Layout

2.1 Powerwall Overview

The inverter is responsible for determining the optimal power distribution from solar panels, to and from Powerwalls, and to and from the utility grid. To control the power going into and out of a Powerwall, the inverter has to communicate with the Powerwall in order to control power conversion by the Powerwall. In addition, the inverter can obtain operational parameters from the Powerwall, e.g. state of charge of the battery.

As shown in Figure 1.1, a communication bus is connected between an inverter and a Powerwall. The Powerwall integrates two physical layers, which both can be used as means for communication with inverters. Specifically, the physical layers are:

1. A RS485 bus, operating at a communication speed of 9,600 bps and using serial port configuration *8-N-1* (i.e. 8 data bits, no parity bits, one stop bit). Moreover, this RS485 bus strictly transports the *Modbus RTU* application layer protocol, as specified in [2].

2. A Controller Area Network (CAN) bus, operating at a communication speed of 250 kbps.

The Modbus protocol is the de facto standard communication protocol for communication between electronic control units (ECUs) in energy storage systems. The CAN bus is implemented in accordance to ISO 11898 specification [3]. Powerwalls implement addressing schemes for both physical layers, allowing an inverter to be connected to multiple Powerwalls on the same communications bus.

The communication busses internal to the Powerwall are depicted in Figure 2.1. Internally to the Powerwall, the RS485 bus that is connected to the inverter is connected to the Powerwall's *Interface ECU*. This ECU acts as a Modbus slave device and responds to Modbus requests from a Modbus

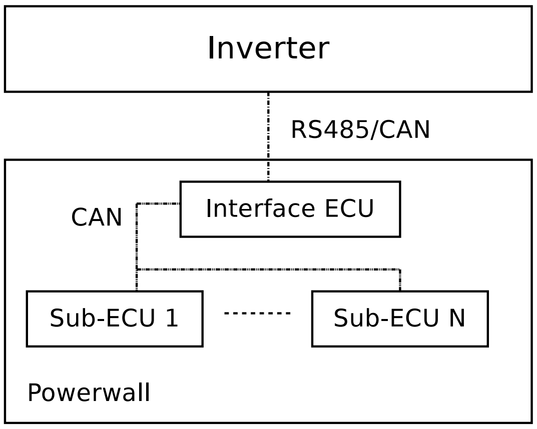


Figure 2.1: System-level overview of communication busses between a Tesla Powerwall and a third-party inverter. An inverter is connected to a Powerwall using either a RS485 bus or a CAN bus. Internally to the Powerwall, both RS485 and CAN busses are connected to the Powerwall's Interface ECU. The Interface ECU is connected by a CAN bus to various other of the Powerwall's internal ECUs.

master, a role taken by the inverter. The CAN bus that is connected to the inverter is connected to all ECUs in the Powerwall.

A Powerwall consists of one *Interface ECU* and a number of *sub-ECUs*. The precise functionality of these sub-ECUs are out of the scope of this thesis and are therefore referred to as *sub-ECUs*.

Next to its microcontroller's flash memory, the Interface ECU is equipped with additional microcontroller-external non-volatile memory for use cases such as data logging and firmware updating. This external memory contains 1MB of non-volatile memory.

2.2 Update Process Requirements

As aforementioned, the purpose of this thesis is to engineer means to update the firmware of a Powerwall. This constitutes updating all of the ECUs in the Powerwall: one Interface ECU and an arbitrary number of sub-ECUs.

The following requirements and restrictions are identified as means for updating firmware of ECUs in a Powerwall:

1. It shall be possible to update the firmware of all of the Powerwall's ECUs.

2. The Powerwall shall remain operational for as long as possible; down-time of the Powerwall should be kept to a minimum.

3. The first Powerwall to be deployed shall be able to be updated, hence the design and implementation of the updating procedure shall be such that new updating features and capabilities can be added in subsequent firmware releases and existing Powerwalls can be updated to firmware with newer updating capabilities.

4. It shall be possible to update Powerwalls over both RS485 and CAN physical layers.

5. The firmware update process should be as transparent and as simple to the inverter as possible.

6. There should be a means for an external party to inquire about any errors that may have occurred in the update process.

7. The Powerwall should always be able to accept a new firmware update, even after a previous failed firmware update.

8. There shall be a notion of firmware versioning across all Powerwall ECUs, and both upgrades and downgrades should be possible.

9. There shall be a notion of ECU hardware identification, and an ECU should never execute firmware that is not built for that ECU.

10. The only communication over the RS485 bus is the Modbus protocol as specified in [2].

Chapter 3

Related Work

As introduced in Chapters 1 and 2, the Powerwall is a device that requires the updating of the firmware of its microcontrollers. Much research has been done over the last decades into methods for firmware updating of electronic devices. Such devices range from hard disk drives to entire automotive vehicles [4, 5].

Dynamic Software Updating (DSU) The process of Dynamic Software Updating (DSU) allows for software to be upgraded while being executed [6, 7]. With DSU, code is updated by analyzing the present and new code, assembling a set of new data objects and functions, and load this new set onto the target device. The actual update of data objects and functions can only be performed under certain conditions, e.g. when these data objects and functions are not in use [8, p. 393-398].

DSU can upgrade code in a running system and thus provides almost continuous availability of the system [9, p. 12] [8, p. 398]. However, replacing small pieces of code can only be performed when the system allows for small pieces of memory to be erased. Microcontrollers equipped with non-volatile memory technologies such as NAND-type flash are not capable to perform per-byte flash erasure. Placing all data and executing all code out of RAM is not seen as an option since the size of a microcontroller's RAM is generally much smaller than its flash memory. Therefore, the program size would be limited by the size of the RAM.

Moreover, for DSU, a supervisory device needs to be able to perform inspection of object files in order to assemble new data objects and functions. Felser et al. call this supervisory device the *manager*, and would require megabytes of memory [8, p. 390]. However, a Powerwall does not have a device with such resources, thus DSU does not seem to be applicable to the Powerwall.

Duplicate Firmware Images Many approaches for firmware updating of devices require such devices to be equipped with enough memory to hold two firmware images. Tarra et al. describe a firmware update method in which two firmware images are available at all times [10]. Upon startup of the device, the bootloader can either start the primary or secondary firmware image, dependent on the verification result of both images.

The Chromium OS also uses a secondary memory location to use as fall-back in the case of a failed update [11]. A secondary partition is used to write the new updated firmware to. When an update has finished, the OS will switch to the secondary partition, yielding a seamless update.

Of the Powerwall's ECUs, only the Interface ECU is equipped with external non-volatile memory. Storing a second firmware image at a sub-ECU requires half of the sub-ECU's microcontroller flash to be reserved. This restriction is not acceptable, thus storing two complete firmware images at a sub-ECU is not feasible.

Since the Powerwall's Interface ECU is equipped with external non-volatile memory, storing a second firmware image there is possible. However, the Interface ECU's microcontroller is not able to execute code from this external memory. The image stored in external memory has to be transferred to the microcontroller's internal flash memory in order to execute this image. Whenever the image stored in microcontroller flash is not valid, the image from external memory can be transferred to the microcontroller's flash and be executed.

Fault Tolerance Liu et al. describe a mechanism for verifying the contents of a firmware image before it is consumed [12]. This verification can be done by either the host device or the device that is being updated. Verification of firmware images before the firmware update takes place ensures that no unsuitable code is loaded on the device.

The mechanism describes verification on the following pieces of data:

- Vendor identification code.
- Model name supported by the firmware.
- Specific instructions at specific memory addresses.

Liu et al. describe that the above checks ensure that no unsuitable code is loaded [12, p. 3]. A vendor identifier and a model name can be statically included in or together with the firmware file and this can be easily checked for by a host device or by the device to be updated. Verifying these pieces of data satisfies Requirement 9 from Section 2.2.

However, verifying instruction and data within a firmware file does not seem feasible. The main challenge is to identify a set of conditions that guarantees that a firmware image is suitable. Instead, additional focus can be placed into the release and validation process of firmware images. Extensive validation of firmware images is a more feasible approach than verifying specific instructions at specific memory addresses by the update process.

Ferlitsch describes a similar concept of querying the device to be updated for its device type identifier [13]. Using this device type identifier, the appropriate firmware file is used to update the device. This process makes sure that only the suitable firmware files are loaded onto devices.

Chapter 4

Design and Implementation

4.1 Inverter-Powerwall Interaction

The first major design decision is to determine what component is responsible for what part of the firmware update process. From the requirements as discussed in Section 2.2, it becomes apparent that the inverter's role shall be minimal. For minimalistic inverter functionality, the two main tasks for an inverter to update a Powerwall are:

1. Transfer data to the Powerwall.

2. Notify the Powerwall that all necessary data has been transferred.

In regards to the inverter's roll, the abovementioned steps are the only required steps for a Powerwall to perform a firmware update.

In addition, there should also be a mechanism to obtain the Powerwall's firmware update status. Without such a mechanism, the inverter is not able to determine whether the Powerwall is in progress of or has finished with a firmware update. The inverter needs such information to be able to properly control the Powerwall. For multi-Powerwall setups, an inverter will update all Powerwalls in a sequential fashion. An inverter starts updating the first Powerwall and keeps the rest of the Powerwalls in a normal operating mode. Once the first Powerwall signals that it has finished updating, the inverter will put the first Powerwall in a normal operating mode and advances to updating the second Powerwall.

The inverter can fulfill the aforementioned two tasks at two levels:

1. The inverter transfers controls a Powerwall's update per ECU: for each of the Powerwall's ECU, the inverter transfers data to the ECU and notifies the ECU that the data has been transferred.

2. The inverter transfers data to a Powerwall and notifies the Powerwall.

Since only the Powerwall's Interface ECU is connected to the inverter, the first method requires that the inverter has access to each ECU within the Powerwall, which, as can be seen in Figure 2.1, is not the case. For Modbus-connected systems, the Interface ECU would need to tunnel data to each ECU and, to accomplish this, there needs to be an additional addressing scheme to address each of the Powerwall's ECUs from the inverter.

Also, while an ECU is being updated, its Powerwall would be non-operational for the duration of this data transfer. Subsequently, by the first method, the Powerwall will remain in a non-operational state for the cumulative time it takes for each of the Powerwall's ECUs update to finish.

The second method does not require a connection between the inverter and all of the Powerwall's ECUs. Instead, this method requires a direct connection to solely update the Powerwall's Interface ECU. The inverter now only communicates with an imaginary Powerwall ECU, instead of a number of individual ECUs. More precisely, this method abstracts what constitutes a Powerwall in terms of ECUs and gives a generic "Powerwall interface" to the inverter.

The method of exposing a single Powerwall interface to the inverter can be broken down further:

1. The inverter transfers data to the Powerwall, which is directly used for updating each ECU.

2. The inverter transfers data to a Powerwall, which is cached in the Powerwall. Following a notification from the inverter, the Powerwall starts updating each ECU from the data it has previously received.

The principal difference between these two methods is that in the second method, the actual updating process of the ECUs can be expedited. The data receiving logic can be part of the normal operational code of the Powerwall's Interface ECU, and the other Powerwall ECUs do not play a role in this initial data caching process. Consequently, the Powerwall can remain operational during this initial data transfer process.

The disadvantage of this caching strategy is that the cache size inherently limits the number of ECUs that can be updated. If more ECUs need to be updated than can fit in the cache, the cache would have to be reused. In such a scenario, an inverter is equipped with a set of data packages, and for each such package the inverter transfers the data and initiates an update to let the Powerwall update a select set of its ECUs.

Due to an increased update time compared to the second method, the first method would increase the non-operational time of the Powerwall. The advantage of a streaming-based update strategy is that it does not require a cache in the Powerwall, making the Powerwall PCBA cheaper than the second method. Following this logic, the first method does not inherently put bounds on the number of ECUs it can update.

Also, a streaming-based update strategy is less capable of keeping the system in a non-operational state for as short as possible and is not always able to perform the necessary precondition checks on the data that is transferred. For example, a consistency error in a firmware image will only be caught at the end of processing a certain firmware image chunk, which might be midway of a firmware image download. Special measures would need to be taken to ensure that the consistency check failure does not leave the system non-operational for a long time. A caching-based update strategy can perform such checks before deciding to use a firmware file. By rejecting an update early on in the update process, a caching-based update strategy keeps the non-operational time of the Powerwall to the minimum.

Based on all of the above constraints and considerations, the caching strategy has been chosen. The reasons for this decision was that the non-operational time is kept to a minimal in this strategy.

4.2 Data Transfer

An inverter has the responsibility to obtain a firmware package and transfer it to the Powerwall. A firmware package can be obtained in a number of methods, e.g. over a local interface such as an SD card that can be plugged into the inverter, or a remote inverter interface such as an inverter's monitoring and diagnostics webserver. These methods are out of the scope of this thesis and are henceforth not further discussed.

4.2.1 Data Transfer over RS485 Bus

As discussed in Section 2.2, all data transfers on the RS485 bus shall be over the Modbus protocol. This includes any data that is needed for the firmware update process.

The Modbus specification specifies a number of function codes that enable a Modbus master device to perform certain operations on a Modbus slave device. Each such function code is a specific operation on the Modbus slave, such as reading of operational parameters and writing commands. For an overview of all Modbus function codes, refer to the Modbus specification. [14]

In order to make firmware updating over Modbus possible, appropriate Modbus function codes shall be used to implement the update procedure as specified in Section 4.1. The Modus specification specifies a dedicated function code for the transfer of non-operational data: function code *Write File Record*. This function code allows transferring of data file in chunks, so-called *file records*.

As per the Modbus specification, a Modbus frame consists of a maximum of 256 bytes: one Modbus address byte, followed by one function code byte, followed by the Modbus function code's data bytes, followed by two CRC bytes. The Modbus write file record function code itself consists of eight metadata bytes. Effectively, using the Modbus write file record function code, a maximum of 244 data bytes can be transferred per Modbus transaction.

A major downside to this function code is that, as per the Modbus specification, the Modbus response to a Modbus write file record function code frame is the echo of the Modbus request. Consequently, this Modbus function's response frame puts a 100% overhead on the data transmission rate. In addition, as marking of start of frame, Modbus frames shall be separated by at least 3.5 characters worth of *dead time* [2, p. 13].

If n_{data} is the number of data bytes per Modbus transaction, then n_{tr} is the number of bytes required to be transferred on the RS485 bus to transfer n_{data} over the Modbus protocol, which can be computed as

$$n_{\text{tr}}(n_{\text{data}}) = \underbrace{2 \cdot (2 + 8 + n_{\text{data}} + 2)}_{\text{Modbus frames}} + \underbrace{2 \cdot 3.5}_{\text{Dead time}} . \tag{4.1}$$

For example, when $n_{\text{data}} = 244$, for every byte transferred over the Modbus protocol, $\frac{519}{244} \approx 2.13$ bytes need to be transferred over the RS485 bus. The efficiency of the Modbus Write File Record protocol n_{eff} for a certain n_{data} is

$$n_{\text{eff}}(n_{\text{data})} = \frac{n_{\text{data}}}{n_{\text{tr}}}. \tag{4.2}$$

The highest efficiency can be obtained by using the highest value for n_{data}. Lower values for n_{data} can be desirable in order maintain timely communication of non-firmware update related information.

Firmware Package Transfer Time On a RS485 bus operating at a baudrate Bd baud 8-N-1 configuration, the number of bytes per second that can be transmitted over the bus can be computed as

$$B_{\text{8-N-1}}(Bd) = \frac{Bd}{n_{\text{startbits}} + n_{\text{databits}} + n_{\text{stopbits}}} = \frac{Bd}{10} \text{ bytes per second.} \tag{4.3}$$

Using Equations (4.2) and (4.3), the maximum throughput of data transfers using the Modbus Write File Record function code with $n_{\text{data}} = 244$ over a RS485 bus operating at 9,600 baud and configuration 8-N-1 can be computed as

$$\begin{aligned} B_{\text{Modbus,8-N-1}}(Bd, n_{\text{data}}) &= B_{\text{8-N-1}}(Bd) \cdot n_{\text{eff}}(n_{\text{data}}) \\ &\approx 960 \cdot 0.470 \\ &\approx 451 \text{ bytes per second.} \end{aligned} \tag{4.4}$$

The firmware package transfer time t_{pt} can be calculated by adding the transfer duration over Modbus t_{Modbus} and the duration of storing it in the

Interface ECU's external memory, t_{memory}. Since the external memory can hold 1 MB, the assumption is made that the largest firmware package is 1 MB as well.

The total data transfer time of the largest firmware package over the Modbus protocol over the RS485 bus, t_{Modbus}, can be calculated as

$$t_{\text{Modbus}} = \frac{1\,\text{MB}}{B_{\text{Modbus,8-N-1}}(Bd, n_{\text{data}})} \approx 2{,}323 \text{ seconds}, \tag{4.5}$$

where n_{data} is 244 and Bd is 9,600 bps. Observe that this calculation does not take into account any errors e.g. Modbus frame errors. Note that in addition to the transfer time over Modbus, the duration to store the received data in external non-volatile memory also needs to be taken into account.

4.2.2 Data Transfer over CAN Bus

From Section 2.2, there is no requirement for any particular application-level protocol that needs to be used on the CAN bus. For the entire update process, the UDS protocol is chosen as application level protocol for all firmware update related communications over CAN [15]. The UDS protocol is widely used in, mostly automotive, applications and is a very well established protocol within the Tesla organization. Using UDS will bring a broad set of libraries, tools and support. No other application level protocols are considered.

Firmware Package Transfer Time Transferring a set of bytes using the UDS protocol requires the use of three UDS services: *RequestDownload*, *TransferData* and *RequestTransferExit*. [15, p. 231] A firmware package consists of a continuous set of data, hence the transfer of a firmware package over UDS requires one *RequestDownload* transaction, followed by n *TransferData* transactions, followed by one *RequestTransferExit*.

To limit the size of the receive buffer in the UDS server, the maximum size of a UDS transaction is limited to 256 bytes per transaction. Since the UDS *TransferData* transaction adds two bytes of overhead to the data it transfers, the maximum number of data bytes that can be transferred by one UDS *TransferData* transaction is 254 bytes. [15, p. 237-238]

The *ISO-TP* protocol is a transport layer protocol for CAN, which is used to transfer UDS transactions. [16] Appendix A describes in detail how a set of n bytes is transferred over the UDS, ISO-TP and CAN protocols.

Transferring a *TransferData* transaction over ISO-TP introduces additional overhead. All CAN frames consist of eight data bytes, as required by the ISO-TP specification [16, p. 31]. To transfer 254 data bytes over the UDS, ISO-TP and CAN protocols, a total of 39 eight-byte CAN frames are needed.

The transfer of 1 MB over UDS requires $\frac{1\text{MB}}{254} = 4{,}128$ UDS *TransferData* transactions are needed. One such transaction requires the transmission of 39 eight-byte CAN frames, thus the number of 8-byte CAN frames needed to send 1 MB of data is $4{,}128 \cdot 39 = 160{,}992$ CAN messages.

The CAN bus connecting inverters to the Powerwall's Interface ECU operates at a baudrate of 250 kpbs. Since only 11-bit CAN identifiers are used, the maximum number of eight-byte CAN messages is estimated to be 1,968 frames per second. Note that this rate is for a 100% utilized CAN bus and consists of solely UDS messages.

Using the CAN frame rate and the number of CAN frames needed to transfer 254 data bytes, the throughput of a file transfer over a UDS on the CAN bus is calculated as:

$$\begin{aligned} B_{\text{UDS,B250}} &\approx \frac{1{,}968}{39} \approx 50 \text{ UDS transactions per second} \\ &\approx 50 \cdot 254 = 12{,}817 \text{ bytes per second} \end{aligned} \tag{4.6}$$

Using (4.6), a 1 MB file can be transferred over the CAN bus in $\frac{1\text{MB}}{12{,}817} =$ 81.8 seconds.

4.3 Update Procedure

Section 4.2 discusses the different ways in which a firmware package is transferred onto the Powerwall's Interface ECU. This section discusses how the Powerwall's Interface ECU uses the firmware package to update its own firmware and that of all of the Powerwall's subcomponents. Also, this section discusses design for reliability and how this affects various design decisions.

4.3.1 Sequence of Updating

As per the requirements as specified in Section 2.2, the Powerwall shall support changes of the Powerwall-internal update procedure. The Powerwall's Interface ECU is responsible for the Powerwall-internal update procedure, such as distributing the contents of firmware packages across the Powerwall's ECUs.

Changes in the Powerwall-internal update procedure requires a change in the Interface ECU's firmware. For the Powerwall to be able to have a forward-compatible update procedure, the Interface ECU needs to be updated as early in the update procedure as possible. After updating of the Interface ECU's firmware, any subsequent steps in the update procedure will be executed using the update procedure that is implemented and active in the just updated Interface ECU firmware. This approach allows for changes such as the format of firmware packages and communication between the

Interface ECU and the Powerwall's subcomponent ECUs. Also, as long as the Interface ECU can be updated, any bugs in the update procedure can be resolved and even deployed Powerwalls can be updated to the new firmware.

4.3.2 Updating of the Interface ECU

The Powerwall's Interface ECU consists of a microcontroller with both flash memory within the microcontroller and non-volatile memory external to the microcontroller. Per Requirement 1 in Section 2.2, it shall be possible to update the internal flash memory of this ECU, such that new firmware can be executed.

From the requirements as described in Section 2.2, the process of overwriting this memory shall be fault-tolerant: there shall be no path in the firmware update process that leads to the ECU not being able to execute valid application code. Invalidating parts, or all, of the contents stored in this memory may lead to non-operational behavior of this ECU. Since this ECU acts as the Powerwall's communications interface, non-operational behavior may lead to the Powerwall not being able to be communicated with, making any subsequent firmware update attempt of the Powerwall impossible. Alternative paths would be needed to repair a system in such a state, for example by using JTAG to directly program the ECU's internal memory with known good firmware.

The above observation implicates the need for a secondary application that is decoupled from the primary application. This secondary application, which is called the ECU's *bootloader* for the remainder of this document, will safeguard the update process of the primary firmware application, which is called the ECU's *application firmware.*

To avoid the need to also update the bootloader, the bootloader's design should be as simple as possible. The bootloader shall be programmed once onto the microcontroller's flash memory (e.g. using JTAG) and not be changed afterwards. Extended firmware validation shall be conducted on the bootloader to ensure reliable execution of all of its functionalities.

Careful design should be made in assigning firmware update responsibilities between the ECU's application firmware and the bootloader. A design could be to put most firmware update functionality in the bootloader, and none in the application firmware. Assuming a problem-free bootloader, such a design would guarantee that the ECU is always updatable. However, such a design would not allow the functionality of the update process to be extended after the first Powerwall is deployed, which violates Requirement 3 from Section 2.2.

To satisfy the aforementioned requirement, more functionality needs to be placed into updatable firmware, which is the application firmware. The bootloader's role it solely responsible for transferring the memory contents from the secondary memory location to the primary memory location. A power interruption during this transfer would leave the primary memory location with a partial firmware image. To account for this, additional firmware image validation checks need to be performed in the bootloader, such that the bootloader is able to validate the memory contents of both the primary and secondary memory locations. Moreover, such a validation check is also needed for the bootloader to know whether the secondary memory location holds a valid firmware image.

Per Requirement 9 from Section 2.2, the update process shall enforce that solely firmware images that are built for a certain ECU are written to their respective ECUs. Part of the validation check is to verify that the firmware image as stored in the secondary memory space is built for the correct ECU. As such, the bootloader will be equipped with an ECU identifier, which will be used to compare against the firmware image's ECU identifier. By putting these validation steps in the bootloader, there is always the bootloader that catches a firmware image validation bug in the application firmware.

At a high-level, the functions of the bootloader are:

1. Act as the code start point.
2. Checking for valid firmware image in secondary location.
   (a) Consistency check of the firmware image to make sure that firmware image is complete and contains no errors.
   (b) Validate that firmware image is valid for this ECU.
3. Copying of application firmware from secondary application space to primary application space.
4. Invalidate contents of secondary application space after copying to primary application space.
5. Start primary application code.

The bootloader requires the following functionalities from the application firmware:

1. Application firmware puts a binary-encoded firmware image in a specified location in the secondary memory.
2. Application firmware resets ECU, whereafter bootloader starts.

The bootloader compiles into its firmware the ECU identifier and the memory location in the secondary memory where the application puts new firmware images. Similarly, metadata such as the ECU identifier and consistency check information (e.g. CRC) is attached to firmware images.

```
if secondary firmware consistency check passes and is valid for this ECU then
    repeat
        bootloader copies secondary firmware into primary firmware memory space;
    until secondary firmware is equal to primary firmware and primary firmware consistency check passes;
    jump to primary application;
else
    if primary firmware passes consistency check then
        jump to primary application;
    else
        remain in bootloader indefinitely;
    end
end
```

Algorithm 1: Interface ECU bootloading logic. Upon power-on of the Interface ECU, the bootloader starts and verifies the consistency of the secondary firmware. If the secondary firmware is not valid, the bootloader determines whether if should start the primary application. If the secondary firmware is valid, and is different from the current firmware application, the bootloader copies the secondary firmware application into its primary memory. After verifying that transfer was successful, the bootloader starts primary application, which takes over execution of the processor.

Bootloader Control Flow The logic implemented of the Interface ECU's bootloader is depicted in Algorithm 1. Upon powering of the Interface ECU, the bootloader is executed. The first task for the bootloader is to obtain information about the firmware image stored in the secondary memory location: a) does the firmware image pass the consistency check, and b) is the image built for this particular ECU. If either checks are not successful, the existing primary application firmware is started, granted that it passes these checks. If the firmware image satisfies both conditions, the bootloader checks if the firmware images in the secondary and primary memory locations are different. If they are the same, then no update is needed, and the bootloader starts the primary application firmware. If they are not the same, the bootloader erases the primary memory location and copies the contents of the secondary memory location into the primary memory location. As for verifying of a successful copying action, a final consistency check is done on the primary memory space, whereafter the primary application firmware is started.

Table 4.1 describes various scenarios and how the previously discussed bootloader control flow handles these scenarios. There should always be a valid image available: either in the primary or the secondary memory. Also,

Scenario	**Recovery path**
Power cut while firmware application is writing new image to secondary memory location.	Bootloader does not pass secondary memory validation check. Primary memory still contains a valid firmware image, which the bootloader starts.
Power cut after bootloader has validated firmware image in secondary memory location and has erased primary memory in preparation for copying data from second to primary memory.	Secondary memory still contains a valid firmware image, thus bootloader passes secondary memory validation check. Since primary memory contains a different image, bootloader starts copying secondary to primary memory, and subsequently starts the primary firmware application.
Power cut while bootloader has partly copied the firmware image from the secondary memory location to the primary firmware location.	Same as above.
Power cut after bootloader is done copying the firmware image from the secondary memory location to the primary firmware location.	Secondary memory still contains a valid firmware image, thus bootloader passes secondary memory validation check.
Bootloader has validated that firmware image in secondary memory is valid. During copying, data as transferred over the communications bus is corrupted.	Post-copy check verifies the firmware image in the primary memory, which will not pass. The bootloader will reattempt copying of data from secondary memory to primary memory.

Figure 4.1: List of non-nominal firmware update scenarios and handling by the bootloader during a firmware update of the Powerwall's Interface ECU.

only when the bootloader has verified that both consistency check and ECU identifier check have passed for the firmware image in the primary memory, the bootloader will actually start this firmware. Solely hardware-related errors, e.g. erroneous communications bus or flash chip, can put this ECU in a state that makes it unable to be updated.

Bootloader in RAM One of the functions of the bootloader is to read, write and erase sectors of the microcontroller's internal flash memory. The code that executes these functions generally cannot run out of the same flash memory sectors as what is being written to or erased from. Due to this limitation, parts of the bootloader code is copied to and executed out of RAM in order to perform these sector operations. In particular, the C-functions that are responsible for writing to and erasing from microcontroller's internal flash are placed in a location in RAM. Upon start of the bootloader, these functions are copied into the predefined location in RAM.

4.3.3 Updating of Sub-ECUs

As depicted in Figure 2.1, the only connection between Powerwall's sub-ECUs and the Interface ECU is through a CAN bus. Unlike the Powerwall's Interface ECU, these ECUs are not equipped with external memory.

Without external memory, there are two designs of updating these microcontrollers:

1. Evenly divide the microcontroller's flash memory into primary and secondary memory locations. As a firmware image is received over CAN, the firmware application writes the firmware image to the secondary memory. After rebooting, the bootloader copies the contents from secondary into the primary memory.

2. Use a bootloader that receives a firmware image over CAN and writes it directly into the primary memory.

Design 1 puts a limit on the size of the firmware application that is possible for updating. Evenly dividing the microcontroller's flash memory space, while also reserving some room for bootloader code, leaves less than half of the available flash memory for application firmware. This limit is too strict, so this design is not chosen.

Design 2 requires a bootloader that is able to receive a firmware image over CAN and directly writes it to primary memory. The maximum size of the application firmware is almost the total size of the microcontroller's flash memory; there only needs to be some space reserved for the bootloader. The application firmware's only responsibility is to be able to be commanded to reset the microcontroller, which starts the bootloader.

The major downside of this design is that it is not guaranteed that a sub-ECU is always equipped with a valid firmware image. Upon receiving of a new firmware image, the existing application firmware image is erased from flash memory. If the new firmware image is not valid, the microcontroller is left in a non-operational state and runs bootloader code. However, this scenario is deemed acceptable, since a new firmware update can be attempted afterwards. Also, this design provides the fastest update time as, unlike Design 1, there is no intermediate storage of the firmware image. For these reasons, this design is chosen over the other design.

Bootloader Design The Powerwall's Interface ECU obtains a firmware package, which may contain a firmware image appropriate for a particular sub-ECU. The Interface ECU then executes a sequence of steps in order to get this firmware image into the sub-ECU's flash memory and have the sub-ECU execute that firmware. To support these steps, the sub-ECU's bootloader needs to implement functionality that allows the Interface ECU to perform the following steps:

1. Reset sub-ECU's microcontroller such that the bootloader is started.
2. Obtain sub-ECU's ECU identifier information.
3. Obtain sub-ECU's application firmware version.
4. Initiate sub-ECU to erase flash memory.
5. Transfer firmware image to sub-ECU.
6. Initiate sub-ECU's bootloader to perform consistency check on received firmware image.
7. Initiate sub-ECU's bootloader to start executing the application firmware.

Since this bootloader is in no way tied to the microcontroller's built-in bootloader, the protocol that is used to perform these steps can be chosen freely. Either a custom protocol is designed, or an existing protocol is used. As discussed in Section 4.2.2, this project already uses UDS for downloading firmware images over CAN from the inverter to the Powerwall's Interface ECU. Since all the aforementioned steps can be performed over UDS, and since UDS is already implemented, UDS is chosen for the firmware update process of sub-ECUs. Choosing a different protocol means the implementation and validation of two communication protocols, which is deemed unreasonable in the available time. In the chosen design, the Interface ECU acts as a *UDS client* and the sub-ECUs act as *UDS servers*.

The bootloader's decision tree is depicted in Algorithm 2. The bootloader waits for a maximum duration of 10 milliseconds for a UDS request, after which the bootloader will start the application firmware, if any. This allows for fast bootup time of the ECU in the case where no firmware update is being performed. The 10 ms time window is chosen to allow sufficient time for the Interface ECU to initiate an update while keeping the bootup time very short. Upon receiving of a UDS request, the bootloader will stay in the bootloader until the Interface ECU initiates it to start the application firmware.

The bootloader always verifies the application firmware prior to running it. This enforces requirement 9 from Section 2.2.

The transmission of a firmware image requires the bootloader's UDS server to support three UDS services: UDS services *RequestDownload*, *TransferData* and *RequestTransferExit* [15, p. 231-249]. Usage of these three UDS services to transfer a firmware file is depicted in Algorithm 3.

For every data section of the firmware image to transfer, the Interface ECU has to use one *RequestDownload* transaction, containing the following information:

- Start address of the data section on the UDS server.

```
Loop
    if received UDS request then
        /* bootloader resets UDS timeout timer */
        bootloader handles UDS request;
    end
    if bootloader times out on UDS requests or reset command has
      been received then
        if primary firmware passes consistency check then
            jump to primary application;
        end
    end
EndLoop
```

Algorithm 2: Powerwall sub-ECU bootloader logic. Upon power-on of the sub-ECU, the bootloader starts and waits a fixed amount of time (10 ms) for a UDS request. If no UDS request is received, the bootloader verifies the contents of the microcontroller's flash memory, and if valid, jumps to the application firmware. If an update is initiated while the sub-ECU is executing the bootloader, the bootloader will handle any UDS requests needed for a firmware update.

```
foreach segment in firmware file do
    issue UDS RequestDownload transaction;
    /* RequestDownload response contains maximum data size
       per TransferData transaction */
    while bytes left in segment do
        $blockSize = min$(bytes left in segment, maximum data size)
         issue UDS TransferData transaction, transferring blockSize
         bytes;
    end
    issue UDS RequestTransferExit transaction;
end
```

Algorithm 3: Procedure for transferring a firmware file over UDS, using UDS *RequestDownload*, *TransferData* and *RequestTransferExit* services. A firmware file describes the memory layout discontinuous segments. For each segment, a *RequestDownload* transaction is needed, specifying the start address and segment size. A UDS server response with *blockSize*: the maximum number of bytes per *TransferData* transaction. The UDS client transfers up to *blockSize* bytes per *TransferData* transactions to transfer the binary contents of the segment. Finally, the segment transfer is exited by issuing a *RequestTransferExit* transaction.

- Size of the data section to transfer.

After the transfer of the data section has been negotiated, a number of *TransferData* transactions are being performed. These transactions have the data section of the firmware image as payload. The transfer of the data section is terminated by a *RequestTransferExit* transaction.

After the firmware file has been transferred, the bootloader verifies that it is equipped with a valid firmware image that is built for its ECU. The bootloader has a hardcoded ECU identifier in its own flash memory, and validates that against the firmware image's fixed-addressed metadata field's ECU identifier field. If this check does not pass, the Interface ECU has probably attempted to update using an incorrect firmware image. When all checks pass, the sub-ECU's bootloader starts its application.

4.4 Powerwall Updater

This section discusses how the Powerwall's Interface ECU updates all ECUs in a Powerwall. It also discusses the format of a Powerwall firmware package.

4.4.1 Interface ECU External Memory

As discussed in Section 4.1, the Powerwall's Interface ECU is responsible for the updating of all the ECUs within a Powerwall. The Interface ECU receives a firmware package from an inverter and saves this package into its external flash memory.

The external flash memory stores two pieces of data: the Interface ECU's secondary application and the Powerwall firmware package. This size of the external flash memory directly puts a bound on the number of ECUs that can be updated using one firmware package. Multiple firmware packages might be needed in order to update all ECUs of a Powerwall.

4.4.2 Updating Logic

As discussed in Section 4.2, the inverter transfers a Powerwall firmware package to the Powerwall, and subsequently triggers the Powerwall to start the update process. Upon receiving the trigger to start the update, the Interface ECU verifies the consistency of the contents of the received data as stored on its external flash memory.

Once the package is validated, the Interface ECU starts updating the Powerwall's ECUs. As discussed in Section 4.3.1, the Interface ECU is always updated first, followed by the sub-ECUs.

The Interface ECU determines what file in the firmware package can be used for updating itself. A firmware image can be used when its ECU identifier corresponds to the Interface ECU's ECU identifier. The Interface ECU verifies that this firmware image is different than its currently running firmware, and copies the secondary firmware image into secondary memory.

After validating that the secondary memory now contains a valid firmware image, the Interface ECU resets itself such that the bootloader starts. As discussed in 4.3.2, the Interface ECU's bootloader copies the contents of the secondary memory space and writes it into the primary memory space, and finally starts the new application firmware.

After the Interface ECU restarts, the first thing it does is verifying that that the firmware version of the code that it is executing is the same as the firmware version of the firmware image in the firmware package. If the versions do not match, the update has failed, and the other ECUs are not updated. Retries are not performed at this stage since the same problem could make the Interface ECU unavailable for an extended time. An external trigger is required in order for the Interface ECU to restart the update process.

If the update of the Interface ECU did succeed, the Interface ECU determines what other ECUs can be updated. For each ECU, the Interface ECU verifies that there is a valid new firmware file available within the firmware package, and updates the ECU according to the update protocol as discussed in Section 4.3.3.

4.4.3 Powerwall Firmware Package

As discussed in Section 4.4.1, the Interface ECU obtains a firmware package from the inverter, stores it in its external memory, and uses it to update the Powerwall's ECUs. This package contains all data needed for the Interface ECU to determine if it can and should update these ECUs.

Requirements The firmware package should satisfy a number of requirement described in Section 2.2:

1. From requirement 3, the file format should be forward compatible: it should be possible to add more data or features at a later time. Any version of the Interface ECU's application firmware should be able to handle any firmware package, although not all features or data of the firmware package may be used necessarily. For example, older versions of the Interface ECU firmware may only support certain features and should disregard any new features that have been added to a newly downloaded firmware package.

2. In order to update all ECUs in a Powerwall, firmware packages should be able to contain multiple firmware files. Allowing updating of all ECUs with one firmware packages yields the minimum system downtime, which is required per requirement 2.

3. There should be means for checking for the consistency of firmware packages. Using an inconsistent firmware package may lead to an

ECU not being able to execute its application firmware, which leads to additional downtime.

In addition to these requirements, the means to interpret firmware packages should be on the Interface ECU.

Firmware File Metadata Powerwall firmware packages should contain one or more files, and there should be means to determine what file can be used to update what ECU. Per firmware file, the following properties need to be known:

1. ECU identifier, consisting of:
    (a) Component ID, e.g. Interface ECU, sub-ECU.
    (b) Hardware ID revision.
2. Checksum for consistency checking of the firmware file.
3. Version identifier of the firmware.
4. Type of encoding of the firmware file.
5. Identifier of the firmware file in the file archive corresponding to this data.

All this metadata of a firmware file can either be put together with each firmware file, or can be stored in one central *manifest file*. Having a manifest file has the benefit of having one central place were all information is stored, and only one piece of data needs to be read to get information about the contents of all files in the file archive.

Since the above list may need to be expanded, the manifest itself should also be forward compatible. Also, storing any information in a binary format is highly preferred over an ASCII-encoded format. ASCII files are by definition at least twice as large in size as the data that they encode, and string parsing would need to be done in the Interface ECU's application firmware.

Hence, a binary encoded format is preferred. Binary formats such as *BSON*[1] and *Efficient XML Interchange*[2] are considered, but no lightweight decoders have been found for these file formats. Existing libraries require a significant amount of available RAM or flash or both.

Google's *protocol buffers* format is a binary-encoded format that is forward compatible, and various lightweight decoders for most programming languages are widely available[3]. An example of a very lightweight decoding

[1]http://bsonspec.org/spec.html
[2]https://www.w3.org/XML/EXI/
[3]https://developers.google.com/protocol-buffers/

```
message Manifest {
        repeated FirmwareListing firmware_listings = 1;
}
message FirmwareListing {
        enum ECU_identifier {
                unkown = 0;
                interface_ecu = 1;
                sub_ecu = 2;
        }
        optional ECU_identifier ecu_id = 1;
        optional uint32 hardware_revision = 2;
        optional string filename = 3;
        optional uint32 firmware_checksum = 4;
        // ID 5 is deprecated, should not be used
        enum ImageFormat {
                 binary_segmented = 0;
        }
        optional ImageFormat image_format = 6;
}
```

Figure 4.2: Protocol buffers description for the Powerwall firmware package's manifest. A `Manifest` consists of a number of `FirmwareListing`s. Each `FirmwareListing` contains all metadata of a firmware file. This description is used to encode manifest files on the build servers, and in the Interface ECU's application firmware to decode the manifest file data in firmware packages.

library is the *nanopb* library[4]. The nanopb library allows for each protocol-buffer signal a maximum size to be defined. This allows for fully statically-allocated data structures without the need for any dynamic memory allocation. Because of the availability of a lightweight file format decoder, Google's protocol-buffer format is chosen to encode the manifest file.

Figure 4.2 shows a `.proto` file describing how a manifest is structured. A *Manifest* consists of a number of *FirmwareListing*s, each representing the metadata of a firmware file. The `.proto` file is used to encode manifest files on the build servers, and in the Interface ECU's application firmware to decode the manifest file data in firmware packages.

File Archive Powerwall firmware packages should contain one or more files, which should be stored in a file archive. The only requirements for this file archive format are the support for file archiving, i.e. file concatenation, and it should be an open file format. Support for additional features such as consistency checking are not strictly needed since these checks can also

[4]https://koti.kapsi.fi/jpa/nanopb/

Format	**Decoding complexity**	**Decoding memory footprint**
ar	++	++
tar	++	+
zip	−	−

Table 4.1: Comparison between decoding complexity and memory footprint for various popular file archive formats.

be implemented in higher levels. Also, there is no need for error correcting codes.

Because of their simplicity and popularity, the *ar*, *tar* and *zip* file archive formats are considered. The *RAR* format has not been considered since it is a proprietary format.

Table 4.1 depicts a comparative analysis between the considered file formats, comparing the complexity and memory footprint of a file format decoder. The *ar* and *tar* formats are by far the simplest – mostly because of their limited set of features. Also, both file formats allow decoding in a sequential order: the decoder starts with the first byte and strictly advances in the file.

The *zip* format requires to first obtain a global header, which is located at the end of the file. To determine the end of the file, the length of the file needs to be known, which is not needed for *ar* and *tar* formats. Also, the *zip* format incorporates to legacy support for all of its prior file format versions, which also increases decoding complexity. Unlike the *ar* and *tar* formats, the *zip* natively supports file compression. Due to its complexity, the *zip* format is not chosen. File compression can always be added at a later stage at a different level, e.g. compress firmware files within an *ar* or *tar* archive.

Between the *ar* and *tar* formats, the only difference is that the *ar* archive puts more restrictions on the length of filenames, which is 16 and 100 characters for *ar* and *tar*, respectively. Moreover, the *tar* archive allows for more file metadata to be stored, resulting in larger headers and a less-efficient file format compared to the *ar* format. However, supporting longer filenames has been considered more important than a slightly less-efficient file format. Hence, the *tar* format is the chosen archive format.

Conveniently, the manifest file, having a hardcoded filename, is stored in the tar archive. Alternatively, a predefined memory region would have to be reserved to store this file, which puts additional limitations on expansion of the manifest. By storing it in the tar archive, the only limitation is the predefined filename.

Next to the manifest, a set of firmware files is stored in the tar archive. For each firmware file, a `FirmwareListing` exists in the manifest, of which the filename is set to the filename of the file in the tar archive. Filenames are the means to link `FirmwareListing`s with firmware files in the tar archive.

Integrity In order to verify that a Powerwall firmware package has been successfully received on the Interface ECU, there should be a means of verifying the consistency of the firmware package.

As means for checking the integrity, a checksum is used. Since the Interface ECU already has the lookup table for CRC-32 and its functions implemented in the bootloader, this CRC-32 is used[5]. Reusing this CRC lookup table directly saves 1 kB of flash space on the Interface ECU.

Using a CRC-32 is deemed sufficient since

- CAN and Modbus have their own CRC, any errors in a firmware package are burst errors, making CRC-32 a good contestant [17, p. 66].
- Additional integrity checks are done at the firmware file level.

Equipping a Powerwall firmware package with a checksum allows for the Interface ECU to verify the integrity of all data in the firmware package: the file archive and any metadata.

Firmware Image Powerwall firmware packages contain file archives that mainly contain firmware images. These firmware images contain data that should end up in each ECU's microcontroller internal flash memory.

Each firmware should contain a representation of the binary data together with the location in flash memory to program this binary data. Since this binary data is not necessary continuous, it could be advantageous to store multiple of such representations instead of one single sparse region of binary data.

There are a large number of file formats that describe firmware images, most notably the *Intel HEX* and *SREC* formats [18] [19, p. 5] [20, p. 2]. Since the size of the external memory on the Interface ECU is very limited, a file format needs to be as efficient as possible. However, file formats such as *Intel HEX* and *SREC* use ASCII text encoding in such a way that the encoded data is at least twice as large as the binary content they encode.

A new file format has been designed that encodes a set of segments of a binary image in a binary format. A file encoded with this binary segmented encoding contains the following data:

1. A *file header*, containing the file's total data size.
2. A number of segments, each containing:
   (a) A header containing the segment's target start address, the segment data size and a checksum of the segment image.

[5]http://reveng.sourceforge.net/crc-catalogue/17plus.htm

(b) Binary encoded data that encompasses the segment.

This format is designed such that it fully describes the binary data and their memory addresses, and allows for decoding in small or large chunks. Also, due to the binary encoding of all segments, the encoding is very space efficient in the sense that the size of an encoded file is almost equal to the size of the binary information it encodes. Adding compression to encoded files would even further decrease the file size, but supporting compression is not yet considered.

As discussed in Sections 4.3.2 and 4.3.3, bootloaders of all ECUs need to be able to verify that a firmware image corresponds to the bootloader's ECU identifier. Next to that, bootloaders should also be able to verify the integrity of a firmware image.

To achieve both requirements, firmware files themselves also need to be equipped with metadata. More specifically, the firmware image should contain an ECU identifier and a means for checking the firmware image's integrity.

As discussed previously, CRC-32 is used by both the bootloader and by application firmware to verify their firmware images. Similar to the integrity checks in firmware packages, firmware images are equipped with a CRC-32 checksum.

For the bootloader to start the application firmware, the application firmware's *code entry point* should also be known by the bootloader. Since this the code entry point is not necessarily the same across firmware images, firmware images should have the code entry pointed stored in a predefined location.

All aforementioned three pieces of metadata of the firmware file are located in a predefined location within the firmware file such that bootloaders can obtain this information. The build system generates firmware images in this format, and bootloaders of all ECUs are able to handle firmware images according to this format. Firmware applications themselves also implement functionality to read their own firmware image metadata, such that they can report this information.

Chapter 5

Results and Discussion

This chapter discusses the performance of the Powerwall's firmware updating procedure. Based on empirical data, analysis is done to compare theoretical estimates with real-world systems.

5.1 Firmware Package Transfer

This section discusses the performance of firmware package transfer from inverters to Powerwalls, of which its design is discussed in Section 4.2.

5.1.1 Package Transfer over Modbus

As discussed in Section 4.2.1, the RS485 bus between inverters and Powerwalls transfers Modbus packets, which is named the *Modbus channel* from here on. As determined by (4.4), for $n_{\text{data}} = 244$, a theoretical throughput of 451 bytes per second can be obtained over this channel.

Test Setup A Linux-based x86 PC is connected by a USB-to-RS485 converter[1] to the Powerwall's RS485 bus. The length of the twisted pair cable is 10 feet long.

To perform Modbus transactions, a PC-based program is developed to act as the Modbus master and will be the entity that transfers a Powerwall firmware package to the Powerwall. This tool implements the Powerwall Modbus firmware update protocol. The performance of this PC-based program, the Linux serial port driver, and the USB-to-RS485 converter is separately verified to not introduce any inter-byte and inter-frame delays. This setup has been verified by continuously sending Modbus requests and receiving Modbus responses between the PC and a Powerwall. Using a

[1] http://www.gearmo.com/shop/usb-to-rs485-rs422-converter-ftdi-chip-with-terminals/

logic analyzer[2], the Modbus transactions on the RS485 bus were verified to strictly follow the 8-N-1 serial communications configuration as described in Section 2.

The firmware package being transferred consists of 896 kB of randomly generated data. The PC sends Modbus Write File Record transactions with $n_{\text{data}} = 128$, resulting in a total of 7,168 Modbus transactions. Using equation (4.1), a theoretical throughput of 428 bytes per second can be obtained. For this test, choosing 128 instead of 244 for n_{data} is justified by a slightly less sophisticated PC tool while still being able to exercise and validate the performance of the Modbus transfer protocol.

The time between the first and last Modbus transaction is considered the overall package transfer time. To characterize the transfer time of Modbus transactions, a general purpose pin is used. This pin is set high by the microcontroller when it receives the first byte of a Modbus request, and is set low when it has sent the last byte of the Modbus response. This microcontroller pin is sampled by a logic analyzer. The Modbus transaction time is equal to the time that the pin is high. The time taken for this extra handling by the microcontroller to actuate the pin is assumed to be negligible.

Results Figure 5.1 depicts the Modbus file transfer times of the 896 kB firmware packages. The figure on the bottom of Figure 5.1 depicts a detailed set of the first 122 Modbus Write File Record transactions.

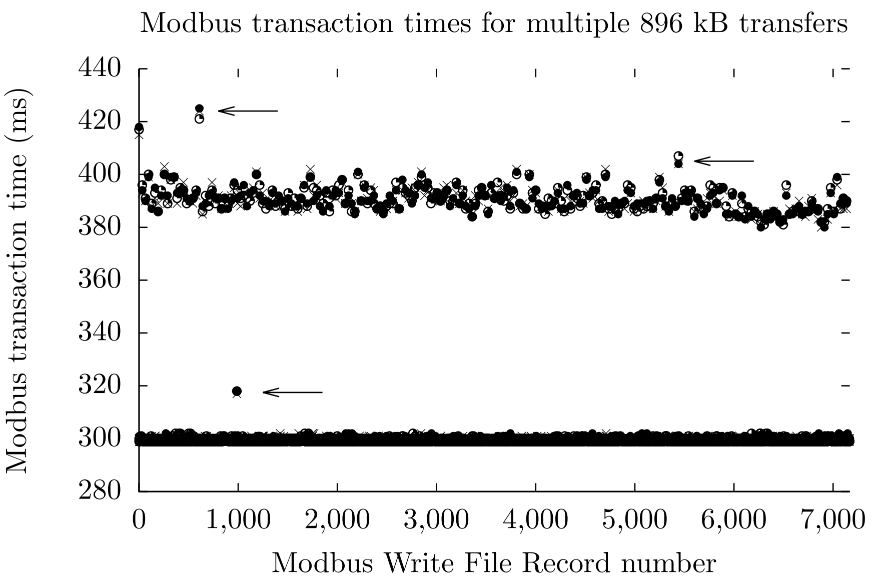
As can be seen, for each 32nd transaction, the transaction time is significantly higher. This is caused by the Interface ECU's Modbus File Record data flushing logic, which happens on every 32nd Modbus File Record. The Modbus response for each 32nd transaction is returned after the data flushing has finished. To characterize the duration of the data flushing, a second microcontroller pin is used: the pin is set high upon starting the data flushing and set low upon finishing the data flushing.

Figure 5.2 depicts the duration of each of the data flush operations from Figure 5.1. The mean data flush operation takes 90.60 milliseconds with a standard deviation of 5.12 milliseconds. Over the three test cases, the average cumulative duration of the data flush operations is 20.3 seconds.

Figure 5.3 depicts the total firmware package transfer time of the test as described above. On average, the package transfer takes 2,173 seconds with a standard deviation of 118 milliseconds. Naturally, using a less reliable Modbus channel (e.g. improper RS485 bus) will create more errors in the Modbus transactions, and thereby introduce a longer and less consistent transfer times.

A transfer time of 2,173 seconds to transfer 896 kB of data yields a throughput of 422 bytes per second. The aforementioned throughput of

[2]https://www.saleae.com/



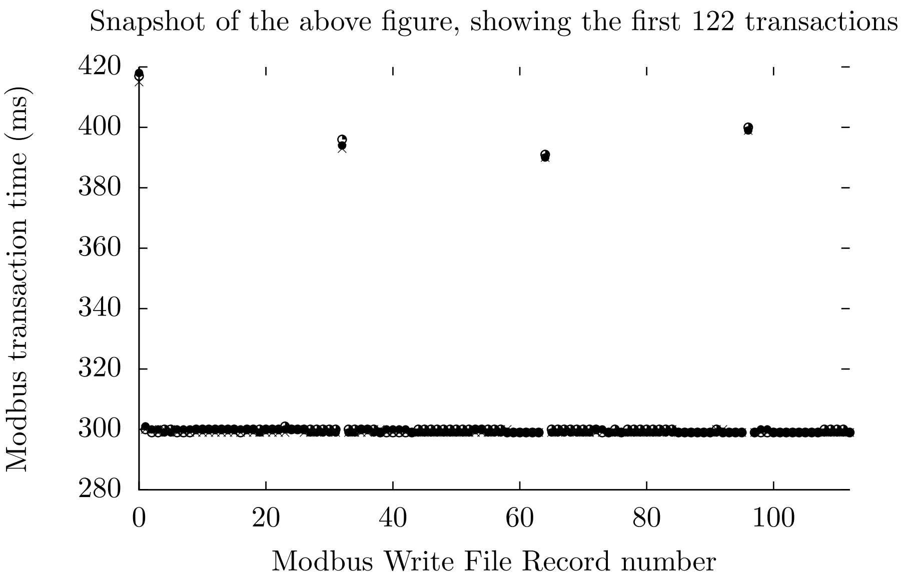


Figure 5.1: Duration of Modbus Write File Record transactions for three file transfers of size 896 kB, each using 7,168 transactions. The nominal transaction duration is 300 milliseconds. As can be seen in more detail in the bottom-most figure, the duration of each 32nd transaction takes is about 80-100 milliseconds longer than other transactions. This is due to the additional data flushing logic that happens on every 32nd transaction. In the top-most figure, arrows indicate how certain transactions consistently deviate from the nominal transaction duration.

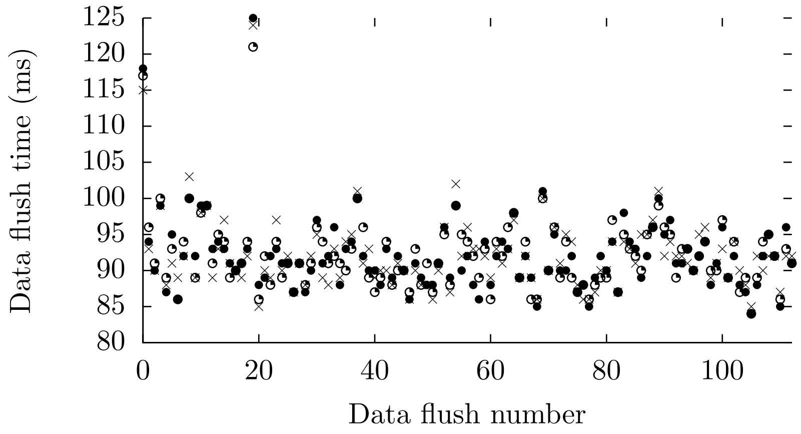


Figure 5.2: Duration of Modbus File Record data flush operations for the three file transfers of size 896 kB of Figure 5.1. The mean data flush time is 90.60 milliseconds and the standard deviation is 5.12 milliseconds.

428 bytes per second is not obtained due to the added duration of the data flushing logic. Removing the duration of the data flushing logic yields a total duration of $2,173 - 20.3 = 2,153$ seconds, which would yield a throughput of 426 bytes per second. Thus, removing the duration of data flush operations yields a throughput of 99.5% of the theoretical maximum throughput. With the data flush operations included, the throughput is 98.6% of the theoretical maximum throughput.

5.1.2 Package Transfer over CAN

To evaluate file transfers over a CAN bus, a similar test approach as for Modbus file transfer is taken. An x86 PC connected to a USB-to-CAN device[3] is connected a Powerwall's Interface ECU's CAN bus. The PC executes a tool that implements the Powerwall UDS firmware update protocol. During these tests, all CAN messages that are not part of the update procedure are disabled. Timestamps of the file transfers are obtained by connecting a logic analyzer to the CAN bus and observe the timestamp of the first UDS request message and the last UDS response message.

Figure 5.4 depicts a histogram of 104 file transfers over UDS, each file being 896 kB in size. The duration between the first UDS *RequestDownload* transaction and the final UDS *RequestTransferExit* transaction is shown. Of the 104 file transfers, the average transfer time is 87.449 seconds and the standard deviation 89 milliseconds.

Equation (4.6) estimated a transfer rate of 12.817 bytes per seconds over UDS, which equates to about 72 seconds for an 896 kB file. The discrepancy

[3]http://www.peak-system.com/PCAN-USB.199.0.html

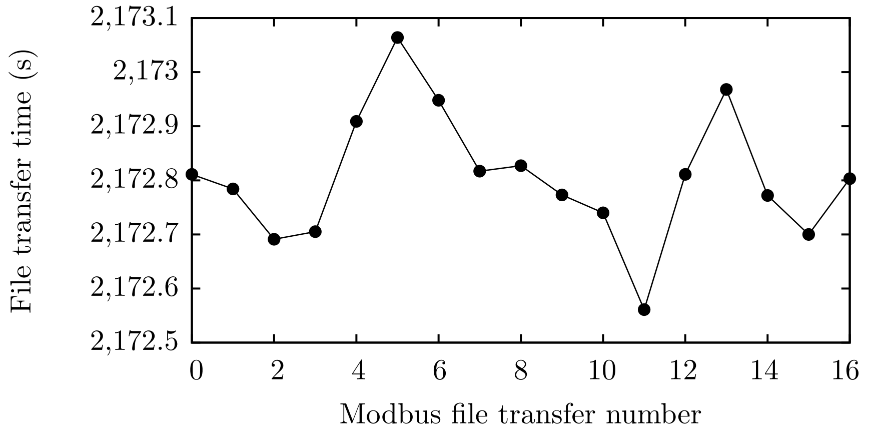


Figure 5.3: Total file transfer duration of several transfers of 896 kB worth of data using Modbus Write File Record transactions. Across these seventeen samples, the average file transfer duration is 2,172.805 seconds, and the standard deviation is 119 milliseconds.

between the observed data and the estimate is mainly the bus load of the CAN bus. During all tests, the bus load averaged to about 70%, and not 100% as the theoretical estimate assumes. Also, other higher-priority code is being executed on the Interface ECU's microcontroller during a file transfer.

Close inspection of the time between CAN frames show that the time between the last UDS request frame and final UDS response CAN frames of a UDS *TransferData* transaction has a considerable delay of 2.8 milliseconds. During this period, the Interface ECU is flushing the received bytes into external memory.

5.2 Firmware Package Validation

As discussed in 4.4.2, the Interface ECU executes a consistency check on a received firmware package. For this check, the entire firmware package is streamed in 500 byte blocks from the Interface ECU's external flash memory into RAM, and incrementally the checksum is computed. The firmware package is considered consistent when the streamed data is consistent.

The rate at which the consistency check is performed is such that 100% of CPU is used for a short period of time. Due to the watchdog mechanism of the Interface ECU, the full utilization of CPU time during the consistency check resulted in the Interface ECU's watchdog being triggered. A delay is intentionally introduced to overcome this problem; after loading a 500 byte block from external memory, a 1 millisecond delay is introduced before loading the next block. This 1 millisecond delay allows the idle task to be

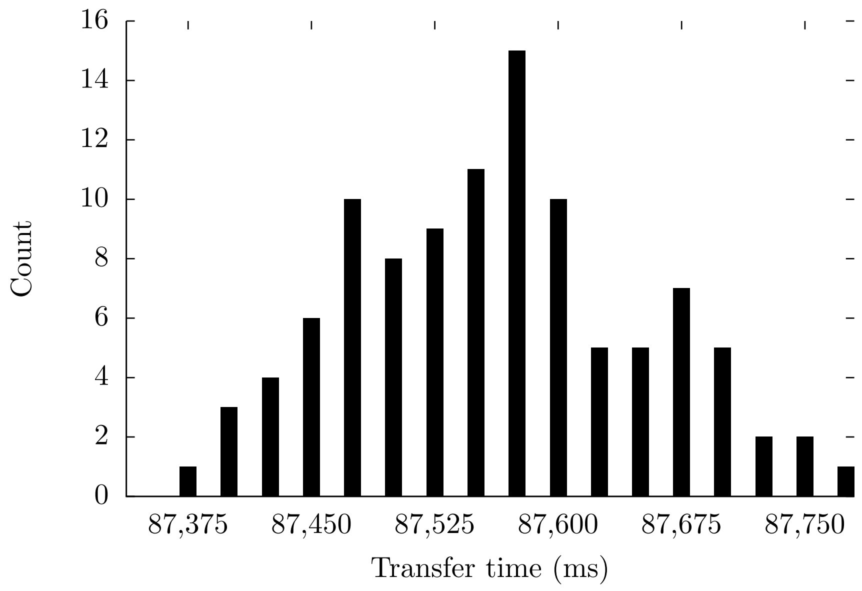


Figure 5.4: Histogram of file transfer durations of an 896 kB file over UDS. The file transfer time is the time between the first UDS *RequestDownload* transaction and the final UDS *RequestTransferExit* transaction. Of the 104 file transfers, the average transfer time is 87.449 seconds and the standard deviation is 89 milliseconds.

scheduled by the operating system, which feeds the watchdog, preventing the watchdog from timing out.

For this test, a UDS transaction is used to trigger the consistency check to be executed. An 896 kB firmware package containing random data is transferred to the Interface ECU, and the consistency check is triggered. The time between the UDS request and UDS response CAN frames is considered the consistency check duration.

For 100 samples, the average duration is 2.110 seconds with a standard deviation of 21 milliseconds. An 896 kB file is streamed in 1,836 chunks, each adding 1 millisecond to the total duration. Without this intentional delay, about 274 milliseconds is actually spend in verifying the checksum. Observe that during this computation, other higher-priority RTOS tasks can be scheduled.

5.3 Update of Interface ECU

As discussed in Section 4.3.2, updating of the Interface ECU consists of the Interface ECU's application to place a new binary encoded firmware file in a secondary memory location in the Interface ECU's external flash memory, and subsequently the bootloader reads this new image and writes it over the microcontroller's flash memory. This section evaluates the steps that make up for almost all the time of an Interface ECU firmware update.

5.3.1 Interface ECU Secondary Memory Interaction

For this test, the external memory are repeatedly populated with random data and a custom flash erase command, using a UDS *RoutineControl* transaction, is send over UDS to the Interface ECU. Upon receiving this command, the Interface ECU erases the memory location designated for the Interface ECU's binary firmware image. The time between the UDS request message and the UDS response message of the UDS transaction is considered the flash erasure time. Similar to the previous tests, the exact timing is determined with a logic analyzer.

Over 100 samples, the average memory erase time is 1.948 seconds, with a standard deviation of 9.1 milliseconds. The duration of this erase operation is fully determined by the internal workings of the external memory's controller.

5.4 Update of Sub-ECUs

As described in Section 4.3.3, the updating procedure for updating the firmware of a sub-ECU consists of erasing flash memory, transferring a firmware file over CAN to the sub-ECU and validation checks by the sub-ECU. This section depicts the duration of each step and discusses how the results relate to the expected results.

5.4.1 Sub-ECU Flash Memory Erase Duration

To test the duration for a sub-ECU to erase its flash memory, the flash erase command is repeatedly send to the sub-ECU. The duration between the UDS request and UDS response of this UDS transaction, measured with a logic analyzer, is considered the flash erasure time. In order for the flash controller to not optimize flash erasures, the entire flash memory region dedicated to application code is reinitialized to random values prior to issuing an erasure.

Over 100 test samples, the flash erasure time is measured to have a mean duration of 943.2 milliseconds and a standard deviation of 0.4 milliseconds. Similar to flash erasure of the Interface ECU's external memory, the duration the erase operation is fully determined by the internal workings of the memory controller.

5.4.2 Sub-ECU File Transfer Duration

To evaluate the transfer speed of firmware images from a Powerwall Interface ECU to a sub-ECU, similar tests are conducted as for firmware package transfers over CAN, as described in Section 5.1.2. Now, randomly generated files of 448 kB in size are send to a sub-ECU. Using (4.6), 448 kB of data takes 35.8 seconds to be transferred to a sub-ECU over the CAN bus. Again, the file transfer duration is considered the duration between the first UDS *RequestDownload* transaction and the final UDS *RequestTransferExit* transaction.

Figure 5.5 depicts the histogram of durations of 100 file transfers over UDS to a sub-ECU. The mean duration is 42.042 milliseconds with a standard deviation of 49 milliseconds. Similar to firmware package transfers over UDS to the Interface ECU as depicted in Figure 5.4, the duration follows a normal distribution.

The obtained file transfer rate is faster than the firmware package transfer speed to the Interface ECU, as discussed in Section 5.1.2. The difference in transfer rate is mainly due to the relatively little time spend in writing to the sub-ECU's flash memory. Less than a millisecond is spend in between consecutive UDS transactions, indicating that each write operation to the sub-ECU's flash memory takes less than a millisecond.

5.4.3 Sub-ECU Validation Check Duration

As discussed in Section 4.3.3, the sub-ECU's bootloader performs two checks to verify a firmware file: one consistency check and validation check. In order to test the duration of these steps, the largest possible firmware file is generated and transferred to a sub-ECU, and a PC initiates both checks by sending the appropriate UDS commands. The duration between the UDS request and response messages of both UDS transactions are considered the duration of the firmware file checks.

Over 100 samples, the mean duration for the firmware file consistency check is 32.1 milliseconds with a standard deviation of 0.03 milliseconds. These consistent numbers can be explained by the fact that the sub-ECU's bootloader does not consist of an RTOS, and thus all bootloader code runs in one single thread. As such, no other tasks are executed while a validation check is initiated.

Over 100 samples, the sub-ECU's firmware file validation check's duration always is less than 1 millisecond, and thus this check is not accounted for. The duration for this check is so little since it consists of a couple lookups of the bootloader's ECU identifier and the firmware file's ECU identifier.

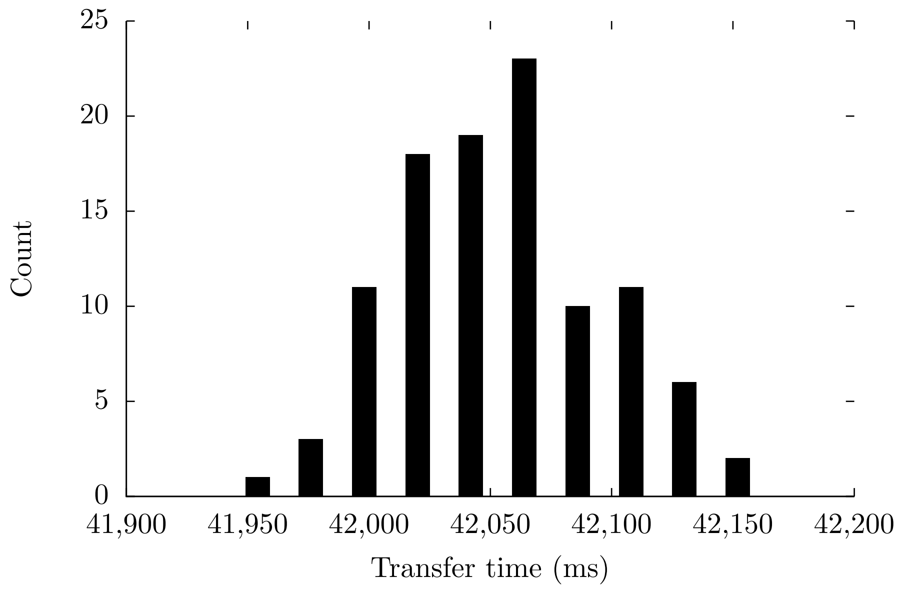


Figure 5.5: Histogram of durations of 100 file transfers of an 448 kB file over UDS to a sub-ECU. The file transfer time is the time between the first UDS *RequestDownload* transaction and the final UDS *RequestTransferExit* transaction. Of the 100 file transfers, the average transfer time is 42.042 milliseconds and the standard deviation 49 milliseconds.

5.5 Fault Tolerance

To analyze the fault tolerance of the firmware update mechanism, a Powerwall has been subjected to non-nominal scenarios. The non-nominal scenarios are categorized into the following three groups:

- Firmware package-level errors.
- Interface ECU-level errors.
- Sub-ECU-level errors.

5.5.1 Firmware Package-Level Errors

The following non-nominal scenarios have been identified regarding firmware package-level errors:

- Corrupt protobuf data.
- Corrupt tar archive.
- Firmware package checksum error.

The effect of all three errors are checked for in unit tests of the Interface ECU's application firmware. The entire firmware package decoding logic (including tar file decoding and nanopb decoding) is written such that it can run on x86 machines.

To mimic corrupt protobuf data, randomly generated data is served as manifest data to the `nanopb` library. In a similar fashion, randomly generated data is served to the `tar` library. To test a firmware package checksum error, a firmware package is generated and manually the checksum bytes are overwritten with the checksum's inverse.

All three scenarios are correctly identified by the Interface ECU as failure cases, and the firmware update is subsequently aborted.

5.5.2 Interface ECU-Level Errors

The non-nominal scenarios for Interface ECU-level errors are described in Figure 4.1. The effect of all listed scenarios are checked for in unit tests of the Interface ECU's bootloader firmware. Power cuts are simulated by providing corrupt images in certain parts of the firmware, e.g. a corrupt image in primary or secondary application space.

All scenarios are correctly handled by the bootloader firmware. In the case of a corruption, there is always one firmware image available to fall back on.

5.5.3 Sub-ECU-Level Errors

The following non-nominal scenarios have been identified regarding sub-ECU-level errors:

- Firmware file checksum error.
- Firmware file loaded with incorrect ECU identifier.

The effect of the two scenarios are checked for in unit tests of the sub-ECU's bootloader firmware. Similarly to Interface ECUs, a power cut during the transfer of a firmware file is simulated by providing a corrupt firmware file to the sub-ECU.

All scenarios are correctly handled by the bootloader firmware. When a corrupt firmware file or a firmware file with an incorrect ECU identifier is loaded, the bootloader will not start this firmware and will remain in the bootloader instead.

5.6 Discussion

Figure 5.1 depicts the summary of the results obtained in the previous sections of this chapter. This figure highlights how each step in the update

Step	**Mean time (s)**	**Powerwall operational?**
Firmware package transfer (Modbus)	2,173.0	Yes
Firmware package transfer (UDS)	87.449	Yes
Firmware package checks	2.110	Yes
Erase sub-ECU memory	0.9432	No
Transfer firmware to sub-ECU	42.042	No
Firmware file test on sub-ECU	0.00321	No

Table 5.1: Mean durations of different steps in a Powerwall's firmware update process as empirically observed. All data is observed from the tests as discussed in the previous sections of this chapter. For each step of the firmware update process, the right-most column depicts whether the Powerwall is in an operational state.

process contributes to the overall firmware update duration of a Powerwall with a single sub-ECU. The cumulated duration of all steps for such an update over Modbus is 2,235 seconds and over UDS is 149 seconds.

As can be seen, the transfer of data from an inverter to the Powerwall take a very significant time in the Powerwall's update process. For Modbus, this step accounts for 97% and 59% of the total update procedure for Modbus and UDS, respectively. This shows the importance that local storage at the Powerwall of firmware packages is performed, since the non-operational time otherwise would be greatly increased.

The cumulative non-operational time is 43.99 seconds, of which 98% is due to the firmware file transfer from Interface ECU to the sub-ECU. Any optimization in this step would equally decrease the non-operational time. Examples of such optimizations are:

- Local storage at sub-ECU. The Interface ECU transfers a firmware file onto the sub-ECU's external memory (which currently does not exist). The sub-ECU's bootloader transfers the firmware file from external memory onto microcontroller memory. The non-operational time is limited since the sub-ECU is running firmware code as it receives a firmware file over the CAN bus.
- Increase CAN bus baudrate, which increases the number of UDS transactions per second on the CAN bus.
- Compress firmware files, which decreases the number of UDS transactions over the CAN bus.

Chapter 6

Conclusions

The goal of this thesis is to design, implement and evaluate a fault-tolerant firmware update method for a time-critical battery product, specifically for the Tesla Powerwall product. In addition to being fault-tolerant, requirements as described in Section 2.2 have been taken into the design phase to lead to a product that satisfies all requirements. In particular, the firmware update process should have minimal impact on the customer of the battery product.

The design and implementation as described in Section 4 result in a firmware update strategy that is recoverable in every scenario. Safeguards are put in those places where non-nominal scenarios occur, such as power interruption during an update and corrupt images. The design is such that in the event of a failed firmware update of a certain ECUs, the ECU remains in such a state that another firmware update can be attempted.

The only scenario in which a Powerwall cannot be updated is when erroneous firmware is updated onto the Powerwall's Interface ECU. More precisely, bugs in the Interface ECU's firmware updating logic can cause the Interface ECU to not being able to update itself. Since the firmware update process cannot identify bugs in firmware, safeguards at the firmware validation level are to be placed. These safeguards are out of the scope of this thesis, and therefore not further discussed.

As discussed in Section 4.4.3, the designed firmware package format is designed in such a way that it can be used for the updating of any embedded system, even microcontrollers constrained by RAM and flash. The format allows for any type of ECU to be packaged, and the format of the firmware file can be of any type. Supporting the packaging of multiple firmware file format allows for the updating of ECUs that don't follow the default update method, e.g. in the case of the updating of third-party devices.

As discussed in Section 5.6, the firmware update method performs very well in regards to the impact to the customer. Appropriate design decisions were made to keep the product operational for as long as possible. Moreover, most of the contributing factors to the time of a firmware update are due to hardware limitations, such as data communication over a fixed-speed CAN bus and write operations to memory. Hence, with the current hardware and communications interfaces, not much improvement can be made to further limit the duration of a firmware update.

One of the points of improvement is to use a multicast-like transport protocol to allow parallel file transfer to multiple sub-ECUs. The current design transfers a firmware image to each sub-ECU separately, leading to increased firmware update durations as the number of sub-ECUs increases. However, these sub-ECUs normally need different firmware files, which still leads to a sequential file transfer.

To date, tens of thousands of firmware updates have been performed on Powerwall test setups. The vast majority of these update have been successful, while several firmware updates have failed, mostly due to errors within the checksum calculation. However, these failed updates were all recoverable, proofing the recoverability of the design.

Bibliography

[1] KC Divya and Jacob Østergaard. Battery energy storage technology for power systemsan overview. *Electric Power Systems Research*, 79(4):511–520, 2009.

[2] Modbus.org. *MODBUS over serial line specification and implementation guide.* http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf, v1.02 edition, December 2006.

[3] ISO/TC 22/SC 31. Iso 11898-1:2015. Standard, International Organization for Standardization, Geneva, CH, December 2015.

[4] Masayuki Kitagawa. Method and system for performing a peripheral firmware update, March 16 2004. US Patent 6,708,231.

[5] B. Moran and F. Mayer. System and method for remotely updating control software in a vehicle with an electric drive system, December 3 2009. US Patent App. 12/130,834.

[6] Michael Hicks, Jonathan T Moore, and Scott Nettles. *Dynamic software updating*, volume 36. ACM, 2001.

[7] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. *SIGPLAN Not.*, 36(5):13–23, May 2001.

[8] Meik Felser, Rüdiger Kapitza, Jürgen Kleinöder, and Wolfgang Schröder-Preikschat. Dynamic software update of resource-constrained distributed embedded systems. In *Embedded System Design: Topics, Techniques and Trends*, pages 387–400. Springer, 2007.

[9] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. *Practical dynamic software updating for C*, volume 41. ACM, 2006.

[10] Raghuveer Tarra and Harsha Saagi. Firmware update for consumer electronic device, October 18 2011. US Patent 8,041,988.

[11] Chromium Project authors. The chromium projects - file system/autoupdate. (visited 11/21/2016).

[12] Tun-Hsing Liu and Yuan-Ting Wu. Firmware updating method and related apparatus for checking content of replacing firmware before firmware updating, October 8 2003. US Patent App. 10/605,560.

[13] Andrew Ferlitsch. Methods and systems for managing firmware, March 27 2006. US Patent App. 11/277,551.

[14] Modbus.org. *Modbus Application Protocol Specification v1.1b*. http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf, v1.1b edition, April 2012.

[15] ISO/TC 22/SC 31. Iso 14229-1:2013. Standard, International Organization for Standardization, Geneva, CH, March 2013.

[16] Road vehicles diagnostics on controller area networks (can) part 2: Network layer services. Standard, International Organization for Standardization, Geneva, CH, October 2004.

[17] Philip Koopman and Tridib Chakravarty. Cyclic redundancy code (crc) polynomial selection for embedded networks. In *Dependable Systems and Networks, 2004 International Conference on*, pages 145–154. IEEE, 2004.

[18] Intel Corporation, http://microsym.com/editor/assets/intelhex.pdf. *Hexadecimal Object File Format Specification*, January 6 1988.

[19] Heinrich Hawig and Ulfert Ulken. Method for programming flash eeproms in microprocessor-equipped vehicle control electronics, September 28 2004. US Patent 6,799,101.

[20] Jaein Jeong. Node-level representation and system support for network programming. *University of California, Berkeley*, 2003.

Appendix A

UDS TransferData service

A UDS *TransferData* transaction request consists of the following elements:

1. 1 byte indicating UDS *TransferData* service (`0x36`)
2. 1 byte indicating the block sequence counter
3. n bytes of data to transfer

A UDS *TransferData* transaction response consists of the following elements:

1. 1 byte indicating UDS *TransferData* response service (`0x76`)
2. 1 byte indicating the echoed block sequence counter from the request
3. Optional data as specified by the user

The optional data field in the UDS response can be used to transport a checksum of the data as received by the UDS server. With this checksum, the UDS client can verify that the UDS has successfully received the data as sent by the UDS client.

An example UDS *TransferData* transaction as transported over a CAN bus is shown in Table A.1. This example shows how 254 data bytes (green) are transported over three protocols: UDS, ISO-TP and CAN.

	CAN data byte #							
Message	**B0**	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**
UDS Request	`\11`	`\00`	`\36`	`\02`	`\16`	`\84`	`\A9`	`\E3`
UDS Response	`\30`	`\00`	`\00`	`\00`	`\00`	`\00`	`\00`	`\00`
UDS Request	`\21`	`\00`	`\00`	`\7F`	`\A0`	`\00`	`\00`	`\78`
UDS Request	`\22`	`\42`	`\00`	`\01`	`\78`	`\82`	`\14`	`\AA`
. . . 35 UDS Requests . . .								
UDS Request	`\24`	`\80`	`\64`	`\1F`	`\70`	`\00`	`\00`	`\00`
UDS Response	`\04`	`\76`	`\02`	`\F3`	`\44`	`\00`	`\00`	`\00`

Table A.1: Example UDS *TransferData* transaction, transferring a total of 254 data bytes using 39 CAN frames. The UDS transaction request consists of the UDS *TransferData* service identifier (orange), followed by the current block number (purple), followed by 254 data bytes (green). The ISO-TP bytes (grey) add functionality such as sequence numbers (e.g. `0x21`, `0x22`), total request length (`0x100`) and control flow (`0x30`). The UDS transaction response consists of the UDS *TransferData* response service identifier (orange), the echoed block number (purple) and a checksum of the received 254 data bytes (yellow).