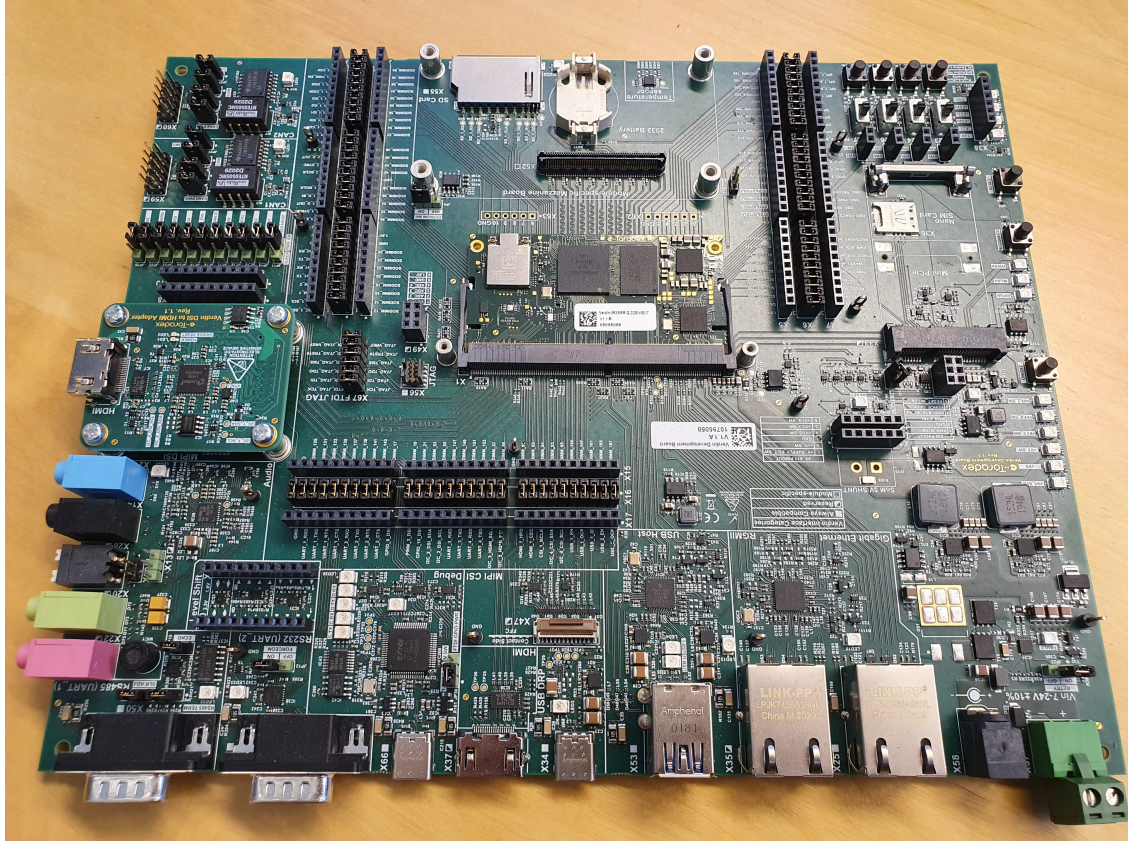




**CHALMERS**  
UNIVERSITY OF TECHNOLOGY



# Cyber security and IoT, exploiting hardware

Master's thesis in Communication Engineering

Hampus Lidén Martinsson

---

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden 2022

[www.chalmers.se](http://www.chalmers.se)



MASTER'S THESIS 2022

# Cyber security and IoT, exploiting hardware

Hampus Lidén Martinsson



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY  
Gothenburg, Sweden 2022

Cyber security and IoT, exploiting hardware  
HAMPUS LIDÉN MARTINSSON

© HAMPUS LIDÉN MARTINSSON, 2022.

Technical Advisor: Torbjörn Tjelldén, Consat Engineering AB  
Examiner: Henk Wymeersch, Department of Electrical Engineering

Master's Thesis 2022  
Department of Electrical Engineering  
Chalmers University of Technology  
SE-412 96 Gothenburg  
Telephone +46 31 772 1000

Cover: Toradex Verdin iMX8M Mini SoC, mounted onto its development board.

Typeset in L<sup>A</sup>T<sub>E</sub>X  
Printed by Chalmers Reproservice  
Gothenburg, Sweden 2022



Cyber security and IoT, exploiting hardware  
HAMPUS LIDÉN MARTINSSON  
Department of Electrical Engineering  
Chalmers University of Technology

## Abstract

Cyber security (**CS**) is a well known and broad subject. However, the same cannot be said for the Internet of Things (**IoT**). While the development of IoT is still in its infancy, CS within this sector has up until now mostly been neglected. This paper aims to research and analyze the Toradex hardware platform for security weaknesses. The hardware is a System on Chip that is commonly used for IoT implementations. The Toradex is a common IoT solution that shares much of its computer on module structure with other similar companies. It therefore represents the overall security of this types of devices quite well. Security in IoT is different from traditional CS due to the larger attack surface. The focus of this thesis is firmly on one part of the surface, the hardware. The Toradex is a Linux based (Debian), embedded system. The hardware is used in IoT solutions and is therefore highly relevant in the IoT sector. The weaknesses found are highlighted and possible solutions are proposed. The problems are an exploitation of the bootloader (Uboot) and the recovery boot. The research has found these to be possible entryways to the system. The bootloader can be flashed, allowing a new more open loader to be installed, which in turn will open up the system. The recovery mode, hides its recovery port by using its own USB driver on the host computer. This port does not require password to install software onto the flash memory. This is exploited by using a malicious USB driver to open the port. These are problems that needs to be taken into account when using the hardware for IoT solutions.

**Keywords:** IoT, Cyber, Security, exploits, hacking, hardware, Uboot, Debian, SoC.



# Acknowledgements

I would like to thank everyone at Consat Engineering that helped me through this project. In particular, I would like to thank Torbjörn Tjelldén, whose technical expertise was always available no matter the day. I would like to thank Daniel Skatt, whose coding know-how enabled me to build my driver. Lastly, I would like to thank Frida Williamsson for being the connecting web in between.

At Chalmers I would like to thank my examiner Henk Wymeersch and Yingqi Zhang for helping me achieve the Chalmers standard in this thesis.

Hampus Lidén Martinsson, Gothenburg, June 2022



# List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

CS	Cyber security
CoM	Computer on Module
CPU	Central Processing Unit
EMI	Electro Magnetic Interference
eMMC	embedded MultiMediaCard
HDD	Hard Disk Drive
IoT	Internet of Things
JTAG	Joint Test Action Group
OS	Operating System
RAM	Random Access Memory
SoM	System on Module
SSD	Solid State Drive
TFTP	Trivial File Transfer Protocol
UART	Universal Asynchronous Receiver-Transmitter





# Contents

<b>List of Acronyms</b>	<b>ix</b>
<b>List of Figures</b>	<b>xiii</b>
<b>List of Tables</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.1.1 IoT and Cyber Security . . . . .	1
1.1.2 Embedded systems and Toradex . . . . .	3
1.2 Motivations . . . . .	5
1.3 Related work . . . . .	6
<b>2 Theory</b>	<b>7</b>
2.1 USB . . . . .	7
2.1.1 The protocol . . . . .	7
2.1.2 The hardware . . . . .	11
2.2 System booting process . . . . .	13
2.2.1 Post BIOS . . . . .	14
2.2.2 BIOS/Das Uboot . . . . .	14
2.2.3 Kernel . . . . .	15
2.2.4 Startup . . . . .	15
2.3 Containers . . . . .	15
2.4 Summary . . . . .	16
<b>3 Investigation and testing methodology</b>	<b>19</b>
3.1 Investigation methodology . . . . .	19
3.1.1 Background research . . . . .	19
3.1.2 Port evaluation . . . . .	20
3.1.3 Hypothesising solutions and evaluating threat . . . . .	20
3.2 Common hardware hacks . . . . .	20
3.3 The Debug port . . . . .	23
3.3.1 The hardware . . . . .	23
3.3.2 The software . . . . .	27
3.3.2.1 Test 1: Memory dumping . . . . .	27
3.3.2.2 Test 2: Bootloader scripting . . . . .	29
3.4 The Recovery port . . . . .	30

3.4.1	The hardware . . . . .	30
3.4.2	The software . . . . .	32
3.5	Summary . . . . .	34
<b>4</b>	<b>Results</b>	<b>37</b>
4.1	The Debug port . . . . .	37
4.1.1	Test 1: Memory Dumping . . . . .	37
4.1.2	Test 2: Bootloader scripting . . . . .	40
4.2	The Recovery port . . . . .	41
4.3	Summary . . . . .	47
<b>5</b>	<b>Discussion &amp; Conclusion</b>	<b>49</b>
5.1	Discussion . . . . .	49
5.1.1	Debugging port . . . . .	50
5.1.2	Recovery port . . . . .	51
5.2	Conclusion of investigation . . . . .	52
<b>6</b>	<b>Future Work</b>	<b>55</b>
	<b>Bibliography</b>	<b>57</b>
<b>A</b>	<b>Available U-boot commands</b>	<b>I</b>
<b>B</b>	<b>Boot dump</b>	<b>V</b>
<b>C</b>	<b>Code used in project</b>	<b>VII</b>

# List of Figures

1.1	Example of a classic IoT system [1]. . . . .	2
1.2	CoM Verdin imx8m mini, front view. . . . .	4
1.3	CoM Verdin imx8m mini, back view. . . . .	4
1.4	Development board for CoM. . . . .	5
2.1	Description tree [2] . . . . .	8
2.2	USB driver layers for a Linux system [3] . . . . .	10
2.3	Type-A connector [4] . . . . .	11
2.4	USB-C connector [4] . . . . .	12
2.5	Classic BPSK scheme in euclidean space . . . . .	12
2.6	Single vs Dual signalling [5]. . . . .	13
2.7	Software containers vs Virtual machines [6] . . . . .	16
3.1	An example of a flash clip used to attach to flash memory . . . . .	22
3.2	UART disabled by cutting the transmission line [7] . . . . .	22
3.3	Debug port X66 on development board . . . . .	23
3.4	Zoom on Debug port X66 . . . . .	24
3.5	Zoom on UART chip . . . . .	25
3.6	Zoom on LED indicators . . . . .	26
3.7	Zoom on UART socket X16 . . . . .	27
3.8	Uboot shell, running the "bdinfo" command . . . . .	28
3.9	Uboot shell, memory dumping from RAM . . . . .	28
3.10	Recovery port X34 on development board . . . . .	30
3.11	Buttons on the development board. 1: Power Switch, 2: Recovery mode access, 3: Hardware reset switch . . . . .	31
3.12	Recovery port X34 on the schematic . . . . .	31
3.13	Data lines from the recovery port to the DDR4 socket of the SoC . . . . .	32
3.14	Recommended schematic to force recovery mode [8] . . . . .	32
3.15	USBView on host, when the device is not in recovery mode . . . . .	33
3.16	USBView on host, when the device is in recovery mode . . . . .	33
4.1	Disassembled bootloader binary, code can be seen in appendix C . . . . .	41
4.2	Sniffed packets using Wireshark, before the easy installer has attached . . . . .	42
4.3	Sniffed packets using Wireshark, after the easy installer has attached . . . . .	43
4.4	Sniffed packets using Wireshark, endpoint 1 starts interrupting . . . . .	44
4.5	Sniffed packets using Wireshark, new handshake and then interrupts . . . . .	45
4.6	Sniffed packets using Wireshark, new handshake and then bulk transfers . . . . .	45

4.7	First set of Set_Reports sent by USB driver . . . . .	46
-----	---	----

# List of Tables

1.1	CoM components for figures 1.2 & 1.3 . . . . .	4
2.1	Transfer classes with their transfer characteristics [2] . . . . .	9
3.1	Connections of interest for the USB/UART interface . . . . .	25
4.1	U-boot commands of special interest . . . . .	38
5.1	Exploits tested for the debug port . . . . .	50
5.2	Exploits tested for recovery the port . . . . .	51
5.3	Security breach severity, low, medium, high . . . . .	53





# 1

## Introduction

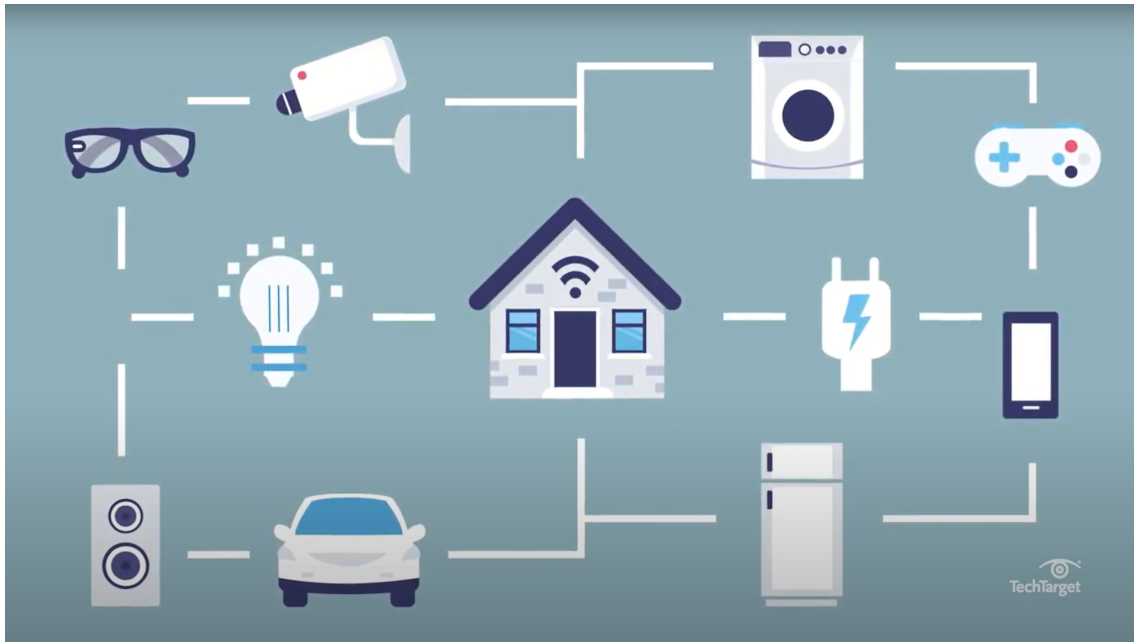
Your credit card information is encrypted, but is it safe? If information is encrypted, somewhere along the process a decryption have to occur. A chain is only as strong as its weakest link and the same can be said for the security of any system. If the online security of an IoT platform is impeccable, what if someone just stole it? What type of information can be obtained? Can it be replaced with the hackers own malicious hardware? These basic questions drive the research and analysis in this thesis.

### 1.1 Background

An overview of cyber security and IoT is introduced. It also contains information about the hardware used and a brief explanation as to what an embedded system is. A more in-depth information about these concepts can be found in the theory section.

#### 1.1.1 IoT and Cyber Security

The basis for the project is IoT-technology. As seen in the acronym list, IoT stands for Internet of Things. The fundamental idea is to move real-life data into the virtual world, process and execute tasks that impact the real world [9, 10]. By definition this means that any device that connects to the internet is a potential IoT device. This gives access to huge amounts of data that is especially needed when building AI applications. Usually these devices are embedded systems with multiple types of sensors. For example, the more common IoT devices today are smartwatches, smart TVs and cellphones. Slightly less obvious examples are IoT devices in the automation industry. These can be, but are not limited to, embedded systems sampling production speeds, production downtime and power consumption etc. These types of devices enable an extreme accuracy when calculating cost of production. IoT is not only the devices of course, it is also the cloud based platforms, algorithms and user input. As a consequence, there are usually a large amount of very different devices interconnected with each other. Which causes large issues when considering the security of the platform. Multiple platforms means multiple strategies are needed to guarantee the entire chains security.



**Figure 1.1:** Example of a classic IoT system [1].

As mentioned CS is a well known subject and the IoT situation differs due its devices unusually large heterogeneity. But what exactly is cyber security in its essence? The purpose of CS is to secure all computer systems and networks from security breaches. These breaches can lead to multiple consequences as a result, which include but are not limited to, theft of sensitive information, theft or damage of hardware and software, denial of service [11]. In general terms, the goal is to protect against any type of tampering be it on hardware or in software. The methods of attack are many, but some of the most common are [1, 11]:

- **Denial of Service.** As the name suggests its goal is to deny a user any types of internet service. This is done by bombarding the target servers with dummy requests until real user-requests cannot get through to the service. An example might be, bombarding a bank with requests until their internet service crashes.
- **Phishing.** This type of attack is focused on the users themselves. The goal of Phishing is to "fish" information such as passwords and usernames from the user by deceiving them. A classic fishing attempt is to use malicious links that moves the user to a dummy page shaped exactly like a trusted page, e.g twitter or facebook. When the user attempts to log in the malicious site saves the password and username, whilst simultaneously logging the user into the real site.
- **Spoofing.** Masquerading as a valid entity in order to get sensitive information. This might be a malicious device replacing an ordinary user on a private network, using faked IP addresses and MAC addresses. The network server, thinking the device can be trusted, unknowingly hands sensitive information

to the malicious device.

- **Malware.** The most obvious threat is malicious software. Any type of software that masquerades its true purpose is considered malware. This might be a file that pretends to only be an anti-virus program but simultaneously uses the users computer for mining cryptocurrency.

As can be glimpsed in the examples above, the types of attack can vary greatly and anything might be a weakness. This is the fundamental problem with IoT devices security. The devices on the network are drastically different from each other and therefore all face different types of attacks. This is what the unusually large attack surface refers to. An example of a IoT network is shown in figure 1.1, it gives a decent scope of device variety.

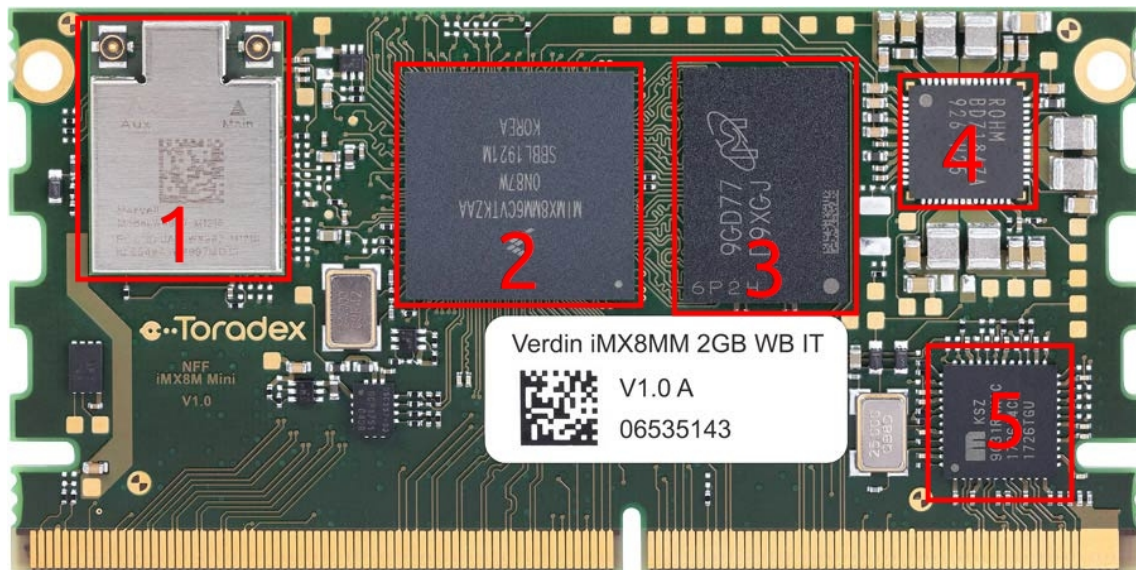
### 1.1.2 Embedded systems and Toradex

A normal computer system, e.g. a laptop, cellphone or iPad are all general purpose in their design. Referring to their multiple functions such as a camera, microphone, touch interface, network interface etc. An embedded system is designed to fulfill a particular function/functions [12] and they are often connected to electrical and/or mechanical hardware. The system is often limited in its power consumption, processing power and memory storage. The system is usually purposefully limited due to the well defined task: i.e, if the task is well defined, the same can be said for its requirements to function. Low power platforms are cheaper and therefore often used. A real-world example of an embedded system is a microwave. That contains a microcontroller which in turn controls the electrical hardware and timer. Another case can be a mouse or a keyboard, both peripherals are embedded systems that contain circuitry and software to fulfil a very specific purpose.

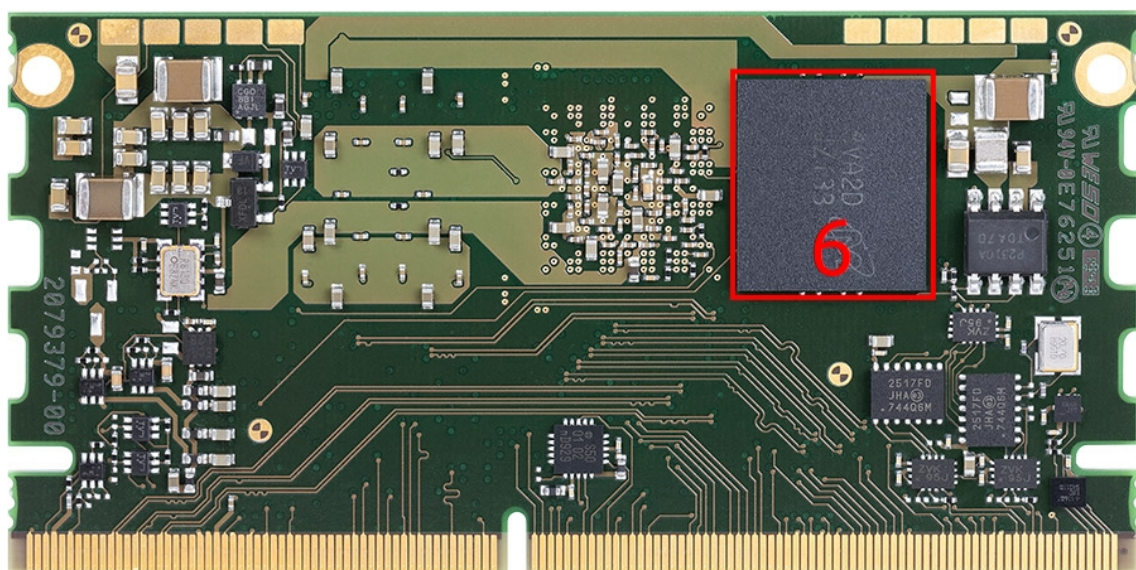
In order to test the CS capabilities of hardware, a fitting platform was chosen as a common implementation of IoT hardware. There are multiple companies that provide similar hardware solutions such as Toradex AG which is used here, Variscite or PHYTEC. The platform used for this research is the Verdin iMX8M Mini module (fig 1.2 & 1.3) and the Verdin Development Board (fig 1.4) by Toradex. Toradex is a company that originates from Switzerland. It focuses on easy user interface, commercial off the shelf embedded computing products with premium quality and long term availability. The module contains standard embedded hardware such as e-mmc flash memory, an ARM processor and RAM (fig 1.2). It is attached to peripheral circuitry using a standard SODIMM DDR4 socket. The platform supports common and standard software when developing embedded IoT platforms. These are for example Yocto project, Qt framework, Codesys, Linux embedded and Docker. Toradex offers standard blueprints for peripheral boards that can be used to tailor specific designs. The board used in this experiment is a standard development board (fig 1.4). The board contains all available I/O that the module supports. In reality, real designs used in finished products will not contain all of these I/Os, they will most likely also disable some of the ports such as JTAG and UART debugger.

Number	Component
1	Wifi module
2	ARM Cortex CPU
3	2GB DDR4 RAM
4	Integrated PMIC
5	Ethernet module
6	16GB e-MMC flash

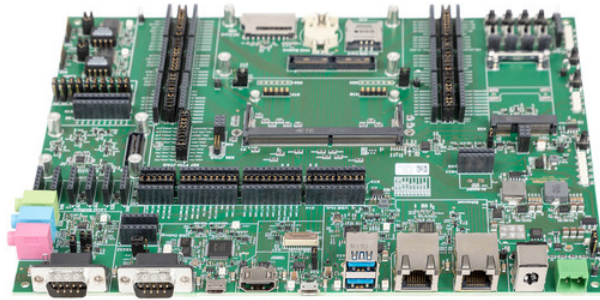
**Table 1.1:** CoM components for figures 1.2 & 1.3



**Figure 1.2:** CoM Verdin imx8m mini, front view.



**Figure 1.3:** CoM Verdin imx8m mini, back view.



**Figure 1.4:** Development board for CoM.

There are also multiple examples of the Toradex family in use. These are provided by the companies partners and are found on the Toradex website. One implementation uses cloud enabled AI to recognize and categorize objects on a conveyor belt [13]. Another features a rotary evaporator controller with a LCD screen and GUI [14]. Both are connected to the cloud and are examples of IoT and Industry 4.0.

## 1.2 Motivations

Even if cyber security is a well known subject and has been studied extensively, its implementation on IoT hardware remains new. The subject faces new challenges due to rapid generations of hardware and software [15]. IoT is often considered to be in its infancy, causing the industry to largely ignore its security in order to implement a platform as fast as possible [1, 9]. This is changing swiftly throughout the industry. However, as more and more attacks and breaches have occurred, the industry is starting to realize the problem which had previously been ignored [1]. As stated initially, any link in the chain of security is a possible access point and one of these links is the hardware itself. IoT solutions are usually placed out in the field and are rarely under surveillance. This thesis aims to explore to what extent a stolen platform can be used and what information can be accessed from it by the malicious party. This is especially true with the type of SoCs investigated in this project. These platforms are made to be as general purpose as possible and there is an extra amount of public information published about them.

The key questions that the thesis attempts to answer are:

1. What are some of the most common hardware hacks and how can they be mitigated?
2. What CS consideration does an IoT platform already have?
3. What weaknesses exists on the platform? How can they be exploited?
4. What can be done to remove these threats?

The contribution of this thesis is the hardware analysis of a common IoT platform called Toradex Verdin and the creation of basic security guidelines. CS is a broad subject and any part could be relevant for this platform. But in order to deepen the research, a specific area was chosen. Namely, the hardware. Due to limited research time, the analysis are limited to finding proof of concepts for the exploits. Assuming that any exploit is deemed possible during the course of the project. The hardware analysis will also be confined to the relevant chips on the CoM and a few of the I/O ports on the development board.

### 1.3 Related work

In previous work there has been multiple studies and surveys regarding the state of the IoT security. The survey by Mardiana et al. [15] gives an overview of the research trend, and by Vikas et al. [16], it specifies areas of IoT deemed critical for security. The work in [17] focuses on hardware Trojans as a type of tampering attack. This type of tampering occurs during design or production. This type of tampering is incredibly dangerous since its undetectable by any software countermeasures. The article suggests multiple countermeasures in both the design of the hardware and the software used for design. The countermeasures being Rots of Trust (RoT), Physical Unclonable Function (PUF) and Device Identifier Composition Engine (DICE). Similarly, this report will also focus on tampering attacks. But whereas this article focuses on solely hardware Trojans, this thesis will attempt all possible tampering available and define their efficiency.

Another article [18] studies side channel analysis attacks and hardware Trojans. The side-channel analysis is performed by measuring power to the cryptographic device. The hardware Trojans are commonly detected, in a non destructive way to the card by analysing the circuits voltage and current. The article suggest a different solution to both the side-channel hack and the hardware Trojan by dynamically randomising the order of information coming in from different sensors. This would make it impossible for the Trojan to activate on a predefined state and make the side-channel unreadable. This possibility will be explored on the IoT platform, if not the power supply to a cryptographic device than listening to I/O ports.

In a comprehensive study found in [19], tampering was listed as one of issues faced by edge nodes. These nodes are usually placed out of sight and out of mind, making their environment extremely hostile for tampering attacks. The suggested solutions for malicious firmware and hardware Trojans were doing side-channel analysis. For other tampering attacks, the study suggests self-destruct designs, electrical tamper proofing, improving memory management and shielding etc. This article lists possible solutions to the problem faced by nodes out in the field. This thesis will choose methods that might negate the successful exploits.



# 2

## Theory

This chapter contains more in-depth information to the technologies the project have touched upon. The goal of this chapter is to provide a strong theoretical base in order to fully grasp the project. The sections features USB, the linux system booting process, linux memory management and the portainer software.

### 2.1 USB

USB or Universal Serial Bus was created in order to allow for easier connections between the PC and its peripherals. The goal was to replace a multitude of different connectors with a single type of port. Hence the word "universal" in its name. The port was developed by seven collaborating companies, Compaq, DEC, IBM, Intel, Microsoft, NEC, and Nortel. The first port (USB 1.0) released in January 1996 with two signaling rates, 1.5 Mbit/s at low bandwidth and 12 Mbit/s at full speed. Today the protocol is still in use and its latest version USB 4, released 2019, based on the Thunderbolt 3 protocol [4]. The sections below will explain the hardware itself i.e the physical ports and the protocol they use. The latest versions of the USB protocol will not be touched upon however(USB 4 & thunderbolt), since they are not used with the Toradex. The physical ports will be limited to the relevant ones i.e **Type-A** and **USB-C**. Since they are the ones that exist on the board.

#### 2.1.1 The protocol

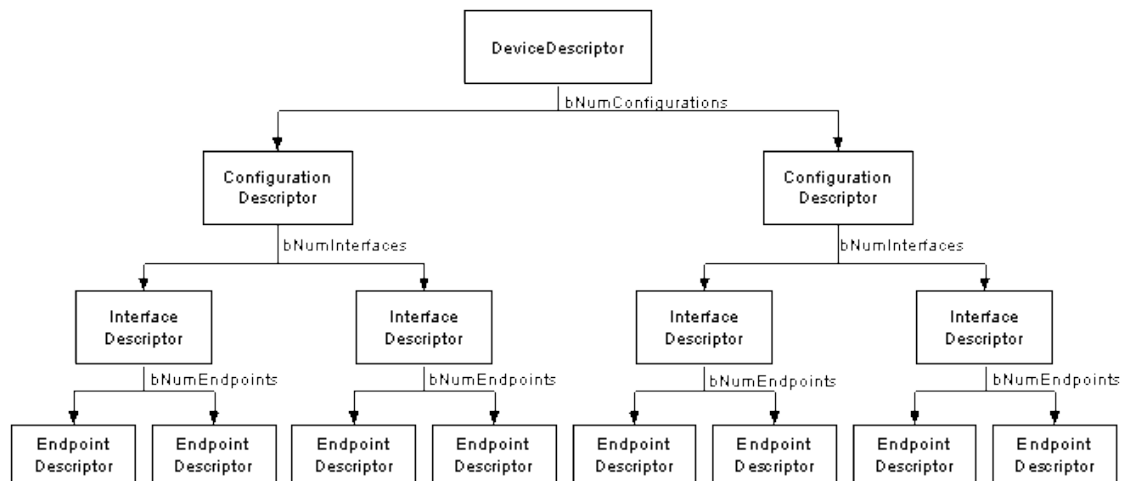
The USB protocol is the crucial foundation that enables the port its plug and play feature. Without it, the user experience would be much more complex. This section will summarize the USB 2.0 protocol so that the packet analysis in chapter 4 is understandable. The other protocols will not be mentioned since they are not deemed relevant to this project.

The protocol utilizes a master and slave structure, meaning that all communication is decided by the master and the slave always follows. So the master is always the one "asking" the slave for information and the slave answers. In this case the master is always referred as the **host** and the slave, the **device**. The host in most cases

including this project is the PC/laptop, whilst the device is the Toradex. However, usually the device is a **peripheral**, like a keyboard, mouse or a USB storage device. When first connecting the device to the host lots of things happens very quickly. First, the host sends GET requests to be able to understand what exactly the device is. These requests are usually of three types and are seen below:

1. GET Descriptor Device
2. GET Descriptor Configuration
3. GET Descriptor String

There are more GET types than seen in the list, but these are the three that typically begin the communication. First the host sends GET request no.1, this request essentially asks the device to identify itself. The response will contain its vendor ID, product ID, serial number and how many configurations it has. Then the host sends GET request no.2. This request asks about the devices configurations and the host dictates a standard length for the response. This is to first check if the full response is needed to save time. The response will say if the device needs power sent through the cable, how much amperage it tolerates, how long the full description is etc. If needed (and it mostly is), it sends the same request again but with the full length value it learned from the first response. The full length response contains the full description tree (see fig 2.1) of the device. It contains its number of configurations, its number of interfaces, the interface classes and the amount of endpoints each interface needs etc.



**Figure 2.1:** Description tree [2]

The description tree in figure 2.1 is quite descriptive of the USB topology albeit with some differences. Whereas a device might indeed have multiple configurations, only one can be running at a time. To switch configuration all interfaces and endpoints have to be terminated. This is why most devices usually only have one configuration. A configuration might have multiple interfaces running at once though. These interfaces are characterised by their class where some are listed below [4]:

- Human Interface Device, HID
- Mass Storage, MSC
- Physical Interface Device, PID
- Image, (PTP/MTP)
- Audio, (A)

The classes are generally used to assign higher level drivers. For example if a device is using an interface with the HID class, a driver that is responsible for that particular class is attached. Usually those drivers are for mouse, keyboard or similar peripherals. An interface is akin to a header which groups a number of endpoints together. Devices usually utilize multiple interfaces at once to fulfil a multi-functional device. For example, a webcam might have an interface for audio and another for video. Each interface can switch settings on the fly without needing to terminate other interfaces. Each interface uses their own type of endpoints. The endpoints are pipes used for data that uses a selected type of transfer. These pipes are always one way, either **IN** or **OUT**, from the hosts perspective. An interface uses at least two endpoints. Where endpoint 0 is the control endpoint, this pipe is setup immediately so that the device can send its settings. Even before the configuration has been fetched. The other endpoints are used to fulfil the devices function. The transaction types used by the endpoints are listed below [2]:

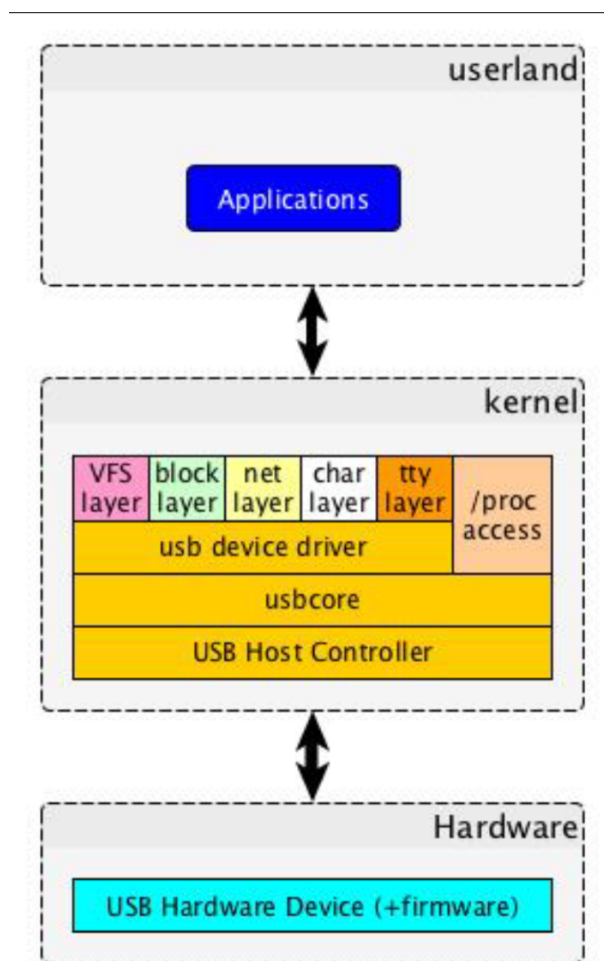
Transfer class	Latency	Bandwidth	Error detection	Packet retry
Control	Guaranteed	Minimum usage	No	Yes
Interrupt	Guaranteed	Minimum usage	Yes	Yes
Isochronous	Upper bounded	Guaranteed size	Yes	No
Bulk	No Guarantee	All available	Yes	Yes

**Table 2.1:** Transfer classes with their transfer characteristics [2]

The control transfer is typically used to setup the device. The data consists of instructions and status requests sent from host. Interrupt transfers are typically used by devices that require immediate attention whenever something happens. A great example here is peripherals like a mouse or keyboard. The data has to be processed immediately so the user does not experience lag when moving the mouse or clicking. The transfer type guarantees a set latency and utilizes error detection and next period retries, and dropped packets will be re-sent so that the user does not have to click twice. The amount of data that can be sent is limited though. Isochronous transfers are used for periodic and continuous transfers. They are usually time sensitive but they happen at scheduled intervals. This transfer type drops packets instead of retrying in order to keep the data stream on schedule. An example is a stream from a webcam or audio to speakers. If a frame is dropped, it is less likely to be noticed compared to if the audio suddenly got out of sync. Bulk transfers are as the name suggests used for large transfers. The goal of this type is to transfers as much data as possible whilst simultaneously guaranteeing data integrity.

That means full error detection with guaranteed packet retry until it arrives.

All of this is contained within the full device descriptor. The device essentially tells the host what it needs in order to fulfil its purpose. This is not verified by the host since the device is always considered trustworthy. The problem is also touched on in the result chapter, in section 4.2. The host reads the descriptor and sends a SET configuration request. This sets the entire tree of interfaces and endpoints. When the device sends its acknowledged response, the device is considered configured. At this point a higher level driver is attached, usually based on class as mentioned. Beforehand it was the host controller and usbcore that dealt with the device. There are usually multiple drivers on top of each other, each raising the USB transfer closer to the OS and application level. The layering can be seen in figure 2.2, where the different drivers are stacked on top of each other until they reach userland. This particular layout is of a linux system. This does not really matter however since all operating systems use basically the same type of layout.

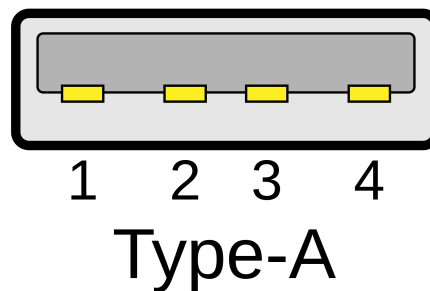


**Figure 2.2:** USB driver layers for a Linux system [3]

### 2.1.2 The hardware

The Toradex uses two types of USB ports. They are the Type-A port and USB-C port. Type-A is usually the port that the **host** end of the connection uses i.e a laptop or PC. Toradex in this case is the **device**, also have Type-A ports but they are mostly used for **peripherals** i.e mouse, keyboard and USB storage. The connection between the PC and Toradex is using a Type-A (host) to USB-C (device). The Type-A port seen in figure 2.3.

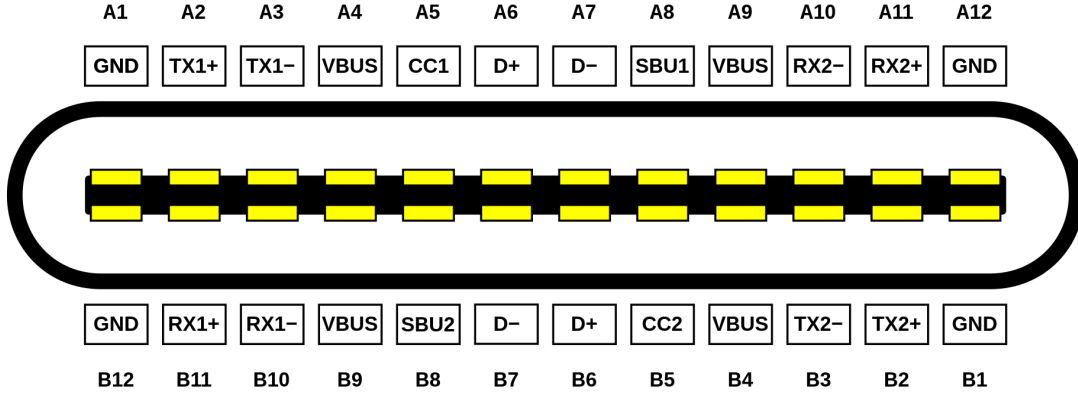
The Type-A connector supports all but the very latest USB protocols. It supports USB 1.0 up to 3.1, albeit with some added connectors for 3.0 and upwards. The USB connection between host and device is USB 2.0 which looks exactly like figure 2.3. The transfer rate with the connector using 2.0 is at maximum 480 Mbps. This does not necessarily mean that the actual speed used is at maximum. Usually when not performing standard bulk transfers maximum speed is not needed. As can be hinted below the USB-C port is much newer and more advanced.



**Figure 2.3:** Type-A connector [4]

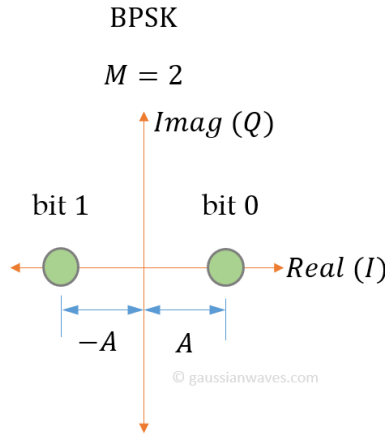
As can be seen in figure 2.4, the USB-C connector contains more connection points. These are the extra connection points needed for the newer protocols mentioned before. These are essentially just added data lines that can now run in parallel with the **D+** and **D-** lines. The aforementioned conductors are of particular interest in this project since the Toradex utilizes USB 2.0.

The first noticeable characteristic of USB is its twisted pair, differential signaling structure. Differential means that the data lines will always mirror each other. This is the reason for the  $\pm$  sign when reading USB related work. The data pair used in this project since its the 2.0 protocol, is the  $D\pm$ . The D lines utilizes half-duplex communication i.e the communication can only move one way at a time. The simplest method of transferring information is the single ended method, where one wire sends one signal. Commonly there are set voltage thresholds which are used to determine if a "one" or a "zero" is received. For example, a voltage below 0.3 is a zero and a voltage above 2.7 is a one. But with differential signaling, the two lines will send the same information but with opposite polarity. The difference between single-ended and differential is illustrated in figure 2.6. Since the signals are mirrored the amplitude will be the same but with opposite polarity. Therefore



**Figure 2.4:** USB-C connector [4]

the receiver will detect the polarity difference between the two signals instead of a voltage level, which is used in single-ended communication. The main benefit of this type of communication is shown below.



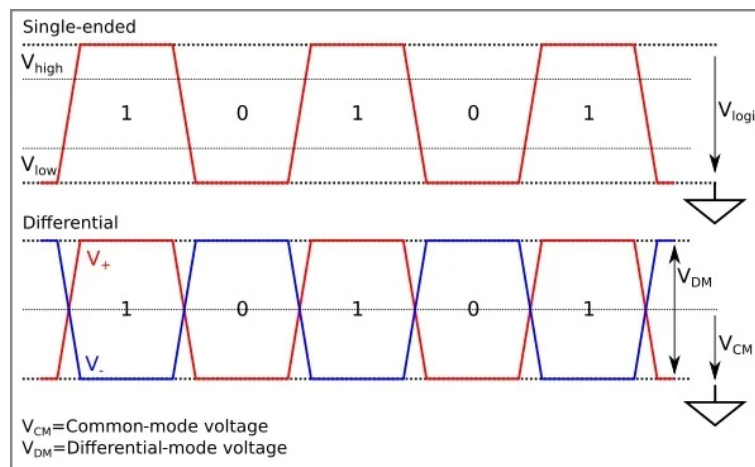
**Figure 2.5:** Classic BPSK scheme in euclidean space

Viewing the **1** and **0** as two symbols in a binary phase shift keying constellation. Binary meaning only two symbols, phase shift meaning they are shifted in phase of each other. So there are two symbols and they are opposites in the real plane. Such a constellation can be seen in figure 2.5, to the left a **1** and right a **0**. The x-axis is the real-axis and **y** the imaginary.

The dual signaling allows the euclidean distance to double between each symbol without increasing the noise factor, effectively doubling the SNR. The power of the signal  $P_{signal}$  is doubled whilst the noise  $P_{noise}$  is not. This is because the amplitude  $A$  is defined as  $\sqrt{E_s}$  and  $E_s$  in turn is defined as  $E = P_{signal} \times t = \frac{U^2}{I} \times t$ . Where  $t$  is time,  $U$  the signal voltage and  $I$  the signal current. Since its only the polarity difference that determines what symbol to choose, the amplitude of the difference will be the combined absolute amplitude of the received signals. This voltage can be seen in figure 2.6, where it is called  $V_{DM}$ . There is also the common mode voltage i.e the ground. To have a common ground will ensure that the zero point remains the



same to both conductors. This is not as important when using differential signaling however. Since we are only looking at polarity differences, some drift from either side will not impact the SNR to a noticeable effect. That is, until higher speeds are used. In USBs case the two conductors do share a ground. Combining the signal of multiple transmitters is a common method in communication, in order to increase SNR. The main drawback is of course that the system will require a factor of two conductors per communication line. The noise factor is slightly increased, but it is negligible. The main interference will be caused by EMI effects due to the current running through the connectors. But since the system uses two separate connectors, the current is not increased in either, keeping the EMI sensitivity the same. Larger currents mean larger EM-fields, which in turn will mean a larger sensitivity to EM-interference. [5]



**Figure 2.6:** Single vs Dual signalling [5].

## 2.2 System booting process

The project works closely with USB protocols since the relevant ports use this. These ports are also closely connected to the booting process of the platform. Since the project works closely with the boot loader of the Toradex, this section will summarize the booting process of a Linux system. The booting process of any PC system can be split into these steps [20, 21]:

1. Post BIOS
2. BIOS
3. Kernel
4. Startup

### 2.2.1 Post BIOS

Any PC enthusiast will have heard about the BIOS (Basic Input Output System) and most likely edit variables within its environment. But, in order to actually load BIOS, some hardware have to be initialized. This is where POST (Power On Self Test) comes in. As the name suggests the POST is a self test to ensure all hardware is functioning correctly. If this fails then the booting process is aborted and the computer will not start. When POST has been passed, an interrupt called INT13H is passed. This interrupt locates the boot sectors on any attached bootable device. This device is usually a hard drive such as a SSD or HDD but it can also be attached flash memories such as USB memories or a bootable DVD disk etc. The boot sector is then loaded into RAM and executed. This boot sector contains the larger part of the bootloader and after execution control is handed off to it. This stage is essentially only there to initialize the hardware enough so that the first stage of the bootloader can run. All computers do some form of this and the user can't really control this part. It is seen as a hardware process more than software.

### 2.2.2 BIOS/Das Uboot

The bootloaders main mission is always the same no matter the one the PC uses. The difference usually lies in their debugging, logging capabilities, boot features and some fine print steps in the booting process. These are not really of interest in this report. The bootloader used in Toradex is the Das Uboot, bootloader, and therefore the one that will be explained here.

The bootloaders purpose is to initialize the Kernel files, execute them and hand off control to the kernel. So in its essence the bootloaders purpose is to load files from hard memory i.e HDD, SSD or USB flash into RAM. The first stage of the bootloader is the U-boot SPL (Secondary Program Loader). This is essentially a bare bone version of U-boot that loads and initializes the rest of the hardware. Its purpose is to do just enough so that the rest can be run. This is also the first moment where the machine is running user-controlled code. Meaning that the code can be compiled and changed by the user. From here in the embedded systems world, the SPL might go and directly load the kernel. This will severely restrict the bootloader capabilities however, so usually a more feature filled bootloader is instead loaded before the kernel. This is seen as the second stage of the boot loader, even though it might be skipped.

The second stage is the full bootloader with all of its capabilities. Usually this means that there is an interactive shell with a plethora of different commands to interact with. The loader also contains debugging tools and logging. The logging for example is usually seen as a bunch of text that starts flowing on the screen as it attempts to boot. This is when the bootloader prepares the kernel files, loads them into memory and starts executing them. After execution is successful a handover process is started and the kernel takes over[20, 21].

### 2.2.3 Kernel

The kernel as the name suggests, is the core of the operating system. These are of course all different depending on the system used, but since the Toradex is an embedded system, the OS of choice is Linux. Most of the kernel is still in a compressed state to save space. The format of the files does allow for a self-extraction however, which is the first thing the kernel does. When that is done the kernel has full access to all of its files. With full access to all its boot files the machine is seen as online, i.e, that it is technically running. It will still have to be prepared so that more constructive work can be done in user space though. This is when the kernel launches the last steps of the booting process by opening the systemd folder [20, 21].

### 2.2.4 Startup

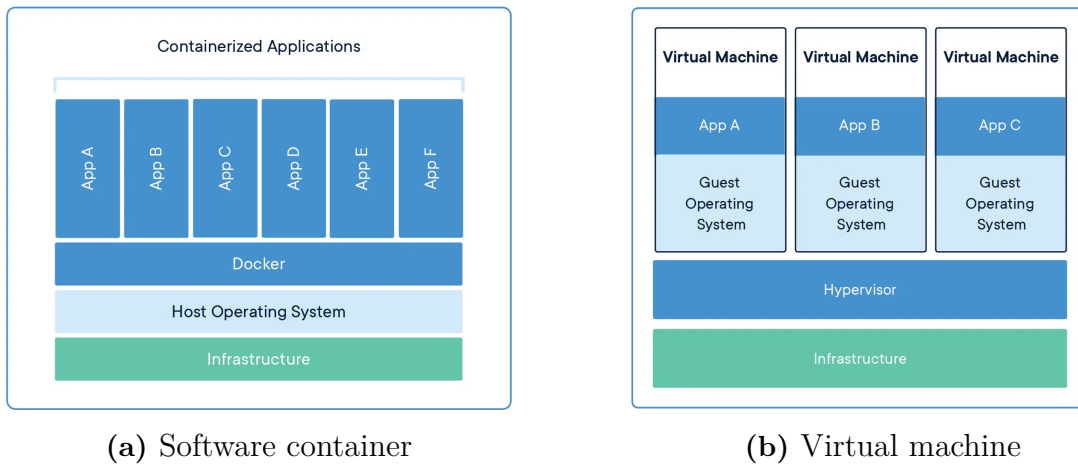
The folder include initializing secondary CPU cores and booting device drivers etc. It also decompresses the user space of the attached hard drives and mount its file system to them. This is when the well known file system structure of Linux can be accessed. Finally it starts booting the user space which will lead to full access to the user i.e the home screen of the operating system [20, 21]. Note that the startup process is very complex in its full step by step walkthrough, this is only a summary.

## 2.3 Containers

One of the strongest aspects of the Toradex and many other IoT platforms is the support of software containers. This section will summarize what a container is and how it is different from a comparable method, a virtual machine.

Software containers are code packages designed to frame an executable and isolate it from other applications (see fig 2.7a). The actual "frame" of the container can be tailored to fit the device it is supposed to be deployed onto. For example, if a container is to be deployed onto both an Arduino and a Raspberry pi, the frame can be set to include one set of dependencies for each. When set up correctly, this enables instant software deployment to devices that might be very different. Meaning that they run different hardware and applications. Since the containers isolates the application within, it is guaranteed that deploying a new container in parallel, will not interfere with other applications. The closest alternative to containers are virtual machines. A virtual machine attempts to simulate a separate device onto the host computer. This will require a separate operating system and virtual hard drive etc. Since that is needed for each separate instance, loading times, file sizes and hardware requirements are much higher. In short, a virtual machine simulates on the hardware level whereas a container simulates on top of the OS [6, 22].

As can be seen in figure 2.7a, since containers are layered on top of the OS, they



**Figure 2.7:** Software containers vs Virtual machines [6]

will require an engine on which to build the extra abstraction layer. In this case the engine is called "Docker Engine", there are many different engines, this one came recommended from Toradex and is therefore used here. The software using the engine is called Portainer supplied by Docker Inc, again recommended by Toradex.

## 2.4 Summary

There were three concepts introduced, the USB protocol, the system boot and containers. The USB protocol utilizes classes to define the device connected, the most important being Human Interface Device (HID). It also uses classes to define what type of transfer is needed. These transfers are the control, interrupt, isochronous and bulk. Control is used to setup the device so a data transfer can begin. Interrupts are used when host response time has to be guaranteed, e.g, a mouse click or movement. Isochronous transfers care more about timing than packets, it would rather drop a few packets than fall behind schedule. Typical of video and audio streaming. Bulk transfers usually for large transfers that are meant to happen as fast as possible. It will use ALL available bandwidth at any given time. Packet arrival is guaranteed by this class, it will retry until successful.

The booting process is quite similar for all computers. The process can be split into four steps [20, 21]:

1. Post BIOS
  - A simple hardware test and initialization of the hard drive.
  - Loads SPL into RAM and executes
2. BIOS/Das Uboot

- SPL loads first, which is an extremely simplified version of BIOS
- SPL usually loads full BIOS but can also run kernel instead
- Full BIOS has a command line and will execute the kernel

### 3. Kernel

- System core that is different for all OS (Linux)
- Extracts all boot files
- Executes Systemd folder

### 4. Startup

- All CPU cores and device drivers are initialized
- Decompresses user space and mounts file system
- User space is loaded and home screen appears

Furthermore, the container structure is one of the strongest aspects of an embedded system. A container is a framework of code that isolates applications from each other in user space. The containers framework can be configured so that they are adapted to each device, making it much easier to update large fleets of devices all running on different hardware and with different applications. Since the applications are isolated from each other, the chances of them interfering with each other is also very low. The closest comparison is a virtual machine (VM), the difference being that a VM runs multiple operating systems in parallel, whilst containers run on top of the OS [6, 22].



# 3

## Investigation and testing methodology

This section will begin by outlining what the process for the investigation is. Then there will be some information about the more common hardware hacks are and how they function. These common hacks are usually found on forums or hacking conventions. The most commonly used in this report is DEF CON. The conference is attended by security experts and engineers and it therefore an excellent source. The shown hacks are seen as known or common. Due to the aforementioned investigative process there will also be two ports that were selected. These ports will be explained and so will the tests that were developed to attempt to exploit them. The result of the tests regarding the ports can be found in 4.2 and 4.1.2.

### 3.1 Investigation methodology

The project will need a structure to its investigation in order to fulfill its purpose. This methodology is outlined below.

#### 3.1.1 Background research

The first step as with any project is the background research. The goal of this research was to familiarize with the platform. To understand how containers and applications are developed and deployed, how to setup the platform and how to enter Recovery mode. Also to find some of the more common hardware hacks and investigate if they can be done on this hardware. This is primarily to avoid researching exploits that might already be well known and patched. Obvious entryways should always be taken into account first. Below common hardware exploits are listed and how they function.

#### 3.1.2 Port evaluation

First the potential of a port will have to be discovered. This is where the background research stage is most important. The research will highlight what types of ports are usually targeted when hardware hacking. Ports with lots of access are of particular interest. The investigation of the ports will vary as to their function. But the common denominator for any port is to answer the following questions in no particular order:

1. What function does the port have and how does it work?
2. Can I control its input/output?
3. How can I exploit its original purpose?
4. What hardware does the port have access to and can it be sniffed?

Depending on the answer to these questions, evaluation might take a long time. The better the understanding about the ports and its related hardware, the more likely a conclusion with high accuracy is. The problem herein is the time consumption of the evaluation and the nature of the hack that is to be built. A proof of concept is much more important than a actual functioning hack.

#### 3.1.3 Hypothesising solutions and evaluating threat

The last stage strongly depends on the boards performance for each exploit and its I/O. The goal is to evaluate the threat and pinpoint exactly how the hack can be performed and what malicious activity can be done using the exploit. For each weakness, possible solutions are discussed.

### 3.2 Common hardware hacks

Most of the common hardware hacks are usually discussed in hacking and CS communities. One such community is the DEF CON convention which focuses on all kinds of hacking, including hardware hacking. Here the most common hardware hacks are openly discussed and explained. The presenters of these hacks are usually security researchers or security engineers, all there to showcase hacks that were found and how to do them. All hacks shown use almost exclusively open source tools and open resource software. The presentations usually contain a step by step explanation as to how the hack is performed and what damage it can do. This convention was a gold mine for this report so most of the hacks were actually taken from DEF CON:

- USB fuzzing [23]



- Firmware extraction [24, 25]
- Relay and replay [26]
- Finding UART and getting Shell [7, 24]
- Glitching [23, 27]

The first item called USB fuzzing, is a hack using USB traffic to cause a kernel panic. The general idea is to use trusted product/vendor IDs from interfaces previously connected to send junk data. Since the IDs are recognised by the OS, no notification will be sent to the user and can be connected freely. The IDs are sniffed using a packet sniffer like the one in section 3.4. This is generally how malicious USB devices disguises themselves. The device might disguise its intent by opening a second interface parallel to the devices shown interface. For example, a flash drive will use an interface of the bulk transfer class. But if the flash drive opens a second interface of HID class, the device might be able to access shell and write commands. The device will show up as a normal flash drive inside the OS and unless the user specifically checks the second interface will remain undetected. This principle is used to send traffic through to device drivers and hopefully causing them to crash. This will make the kernel panic which in turn will throw a crash dump. This dump can then be used to hack the system[23].

Firmware extraction, as the name suggests, aims to access the systems files in order to analyze them and find weaknesses. This weakness can then be used to gain access to the system, preferably a shell with root access or something similar. The easiest way to do this is by connecting to the flash memory using a flash clip (fig 3.1) and a host computer. This allows the host to communicate with the flash like it was apart of the same circuitry. All data that exists on the flash is then dumped as a binary "blob" and can be analyzed. Analyzing binary blobs is a complex science on its own, but there is multiple open source tools out there that can help. For example, one such tool is binwalk, this tool searches the binary blob for any recognizable data and labels it. This data might be an OS image or a bootloader image. There is also some more advanced tools that are not free but anyone can purchase a license, assuming someone does not simply pirate a copy. IDA pro is one such tool. IDA is an advanced disassembler that can break down almost any binary data into assembler code. The last step is then straightforward, if the flash can be connected to host in this way then the flash can be written to. Since the layout of all data on the flash is known, it would be easy to just inject a backdoor.[24, 25].

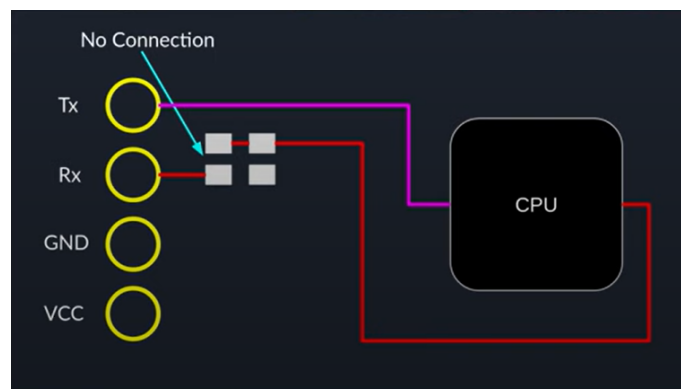
The relay and replay is in its essence a very basic hack. The example included [26], is a recording of a nfc transaction and then replaying it in order to spoof as a different card. This idea is generally the same across any type of device. Even if a transaction is encrypted it can still be mimicked due to not needing to know what exactly is being transmitted. The encryption can also be cracked by using known encrypted phrases. An example of this might be if the incorrect pin is entered to a payment terminal. The reply from the terminal, even if its encrypted, will most



**Figure 3.1:** An example of a flash clip used to attach to flash memory

likely be something similar to "denied", "error" or "wrong pin". These keywords can be searched and matched with the encryption to find the encryption key. This is assuming all communication is encrypted, if only passwords are encrypted for example, the encrypted phrase could still be sent in order to gain root privileges [26].

Finding UART and shell is by far the most common method that is always tried first. The port is almost always found as four pads on a line. These pads are the standard communication types, e.g, transmitted data (TxD), received data (RxD), power supply (VCC) and ground (GND). Depending on the type of device, these types of debugging ports are sometimes not even password protected [7]. The idea is quite simple, find the debugging port and connect to shell using any type of serial client. With full access to the command line any hack is possible. There have been multiple examples where the command line was simply disconnected inside the hardware, like fig 3.2.



**Figure 3.2:** UART disabled by cutting the transmission line [7]

Lastly is the method of glitching the hardware [27]. This is one step that might also go hand in hand with finding UART or firmware extraction. The general idea is to glitch the boot up process in order to panic the bootloader. This will force the bootloader to throw its command line even though the original OS shell was password protected. This method is also a common way of accessing the bootloader even though its disabled. The difficulty of this hack is the risk of destroying the hardware. In order to glitch the booting process, I/O pins of essential hardware is

grounded, shorted or set to high. Essential meaning chips required to boot the chip e.g, CPU, flash memory or RAM. In the example found in [27], pins between the flash and CPU was shorted using a small wire. This corrupted the data and the bootloader could not load the kernel, which in made the bootloader dump its shell. The following work using the bootloader in section 4.1.2, assumes that something like this was most likely tried.

### 3.3 The Debug port

One of the two ports investigated is the debugging port. The port has access to the OS and bootloader command line. It is commonly used when debugging and is therefore very hard for the developer to remove even after the product is finished. As outlined in the investigation methodology the behaviour of the port has to be observed, but also how the port is connected to the hardware and what protocol it uses. The debugging port is made as the name suggests for debugging and error finding. Meaning that the port will have access to both Linux command line and the bootloader. This is so that the port will work with or without the operating system. Otherwise it would not be possible to troubleshoot the hardware if the operating system stopped working.

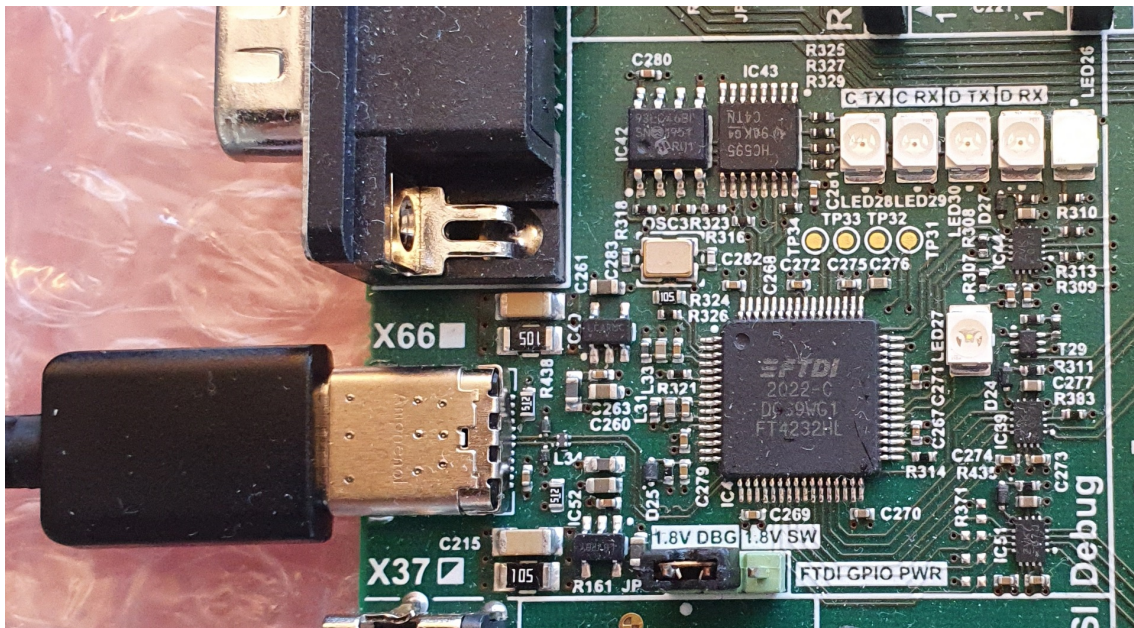


Figure 3.3: Debug port X66 on development board

#### 3.3.1 The hardware

Finding the debugging port on the schematic is trivial, in both the pdf version and altium project it is separately labeled. Below are some outtakes of the schematic, the figures 3.4-3.5 show the most important part of the schematic. View [28] for the

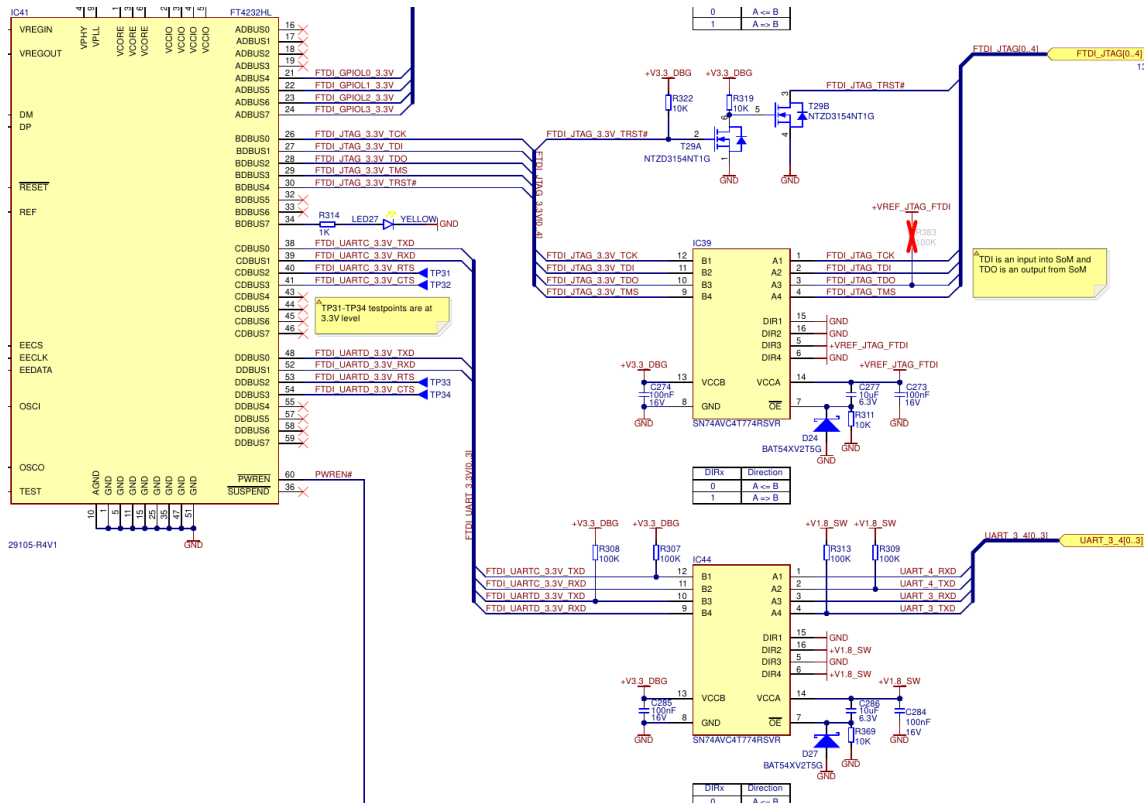


memory inside the interface chip. So the things that happen to the memory will show itself as a signal on the "EE" pins.

Following them will lead to figure 3.6, which clearly states that it is activity indicators. Using the real board for reference again, the real led lights can be located in order to confirm this is the correct connection. See figure 3.3, the LED lights are the five white blocks on a line. The top most LED light is the status light, it indicates if there is anything plugged into the port which is why it is currently on in the figure. Connected before the diodes is a bit shift register, it will take a serial input of 8 bits before outputting to its pins. The exact details of how the register works is not really important, the relevant part is that the register uses the clock signal from the interface and outputs the data bits to the LEDs. That means that the input and output to the interface from the debug port is directly displayed on the LED. By extension that also means that any debug input can be sniffed directly from the LEDs.

EECS	Chip Select signal
EECLK	Clock signal
EEDATA	DATA signal
DDBUS0	UART transmitted
DDBUS1	UART received

**Table 3.1:** Connections of interest for the USB/UART interface



**Figure 3.5:** Zoom on UART chip

### 3. Investigation and testing methodology

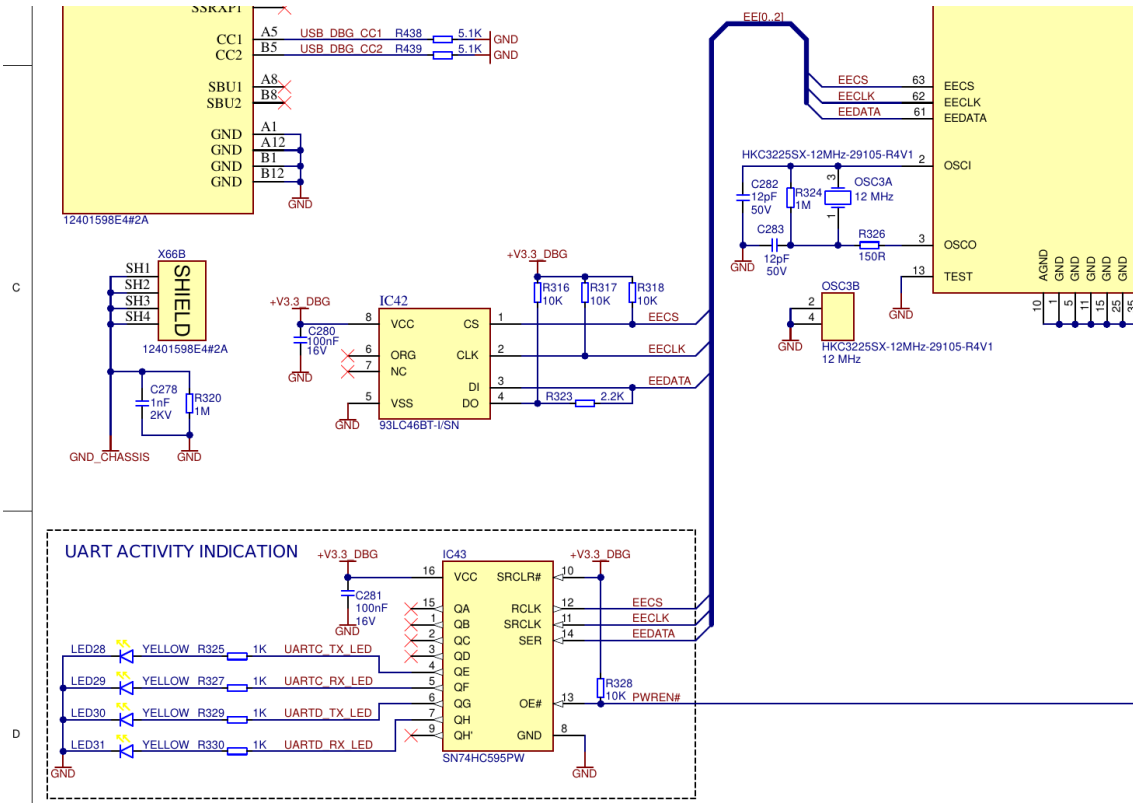


Figure 3.6: Zoom on LED indicators

Keeping the sniffing capabilities from the LEDs in mind, it is time to follow the UART signal after the interface. If the UART signal is in parallel with a socket there might be multiple places where the signal can be sniffed. In the table 3.1 there is also the "DDBUSx" signals. Also labeled in figure 3.5, with the prefix "FTDI\_UARTD\_3.3V". There is two other connections that is not mentioned in the table. These lead to test pads, used to test the chips functionality. These can also be seen on the real circuit board as two of the four golden dots to the left of the chip. Checking the blueprint also confirms that there is indeed two other test pads connected to the "CDBUS2-3", hence the grouping of four. These pads do not contain any of the data that is transmitted and is therefore not really of use here however. Following the data connections **T**ransmit x **D**ata (TxD) and **R**eceived x **D**ata (RxD) will lead to a voltage controller before continuing outside of the debug blueprint. The controller has multiple different functions but in this case it is basically used to lower the voltage from 3.3 V down to 1.8 V.

Following the connection even further means the end of this particular blueprint. Using the tag to find the next blueprint or simply double clicking it if viewed in Altium, opens the full blue print for the figure 3.7. Figure 3.7 is zoomed in on only the relevant part, there it shows multiple sockets. According to the blueprint the debug signal is attached to the socket X16 and in parallel with X15 and X17 before arriving at the "SODIMM" pins, which can also be seen in figure 3.3 as "UART\_3\_xx", albeit with some difficulty. The sockets means that it is even easier to sniff the debug I/O from here. In reality these sockets would most likely not exist,



but the same signal will still always go to the pins named to the left (fig 3.7), i.e "SODIMM\_147-153". These pins are a part of the DDR4 SODIMM socket which the SoC module uses. Since these pins are always exposed on the DDR4 socket the signal can also be sniffed here.

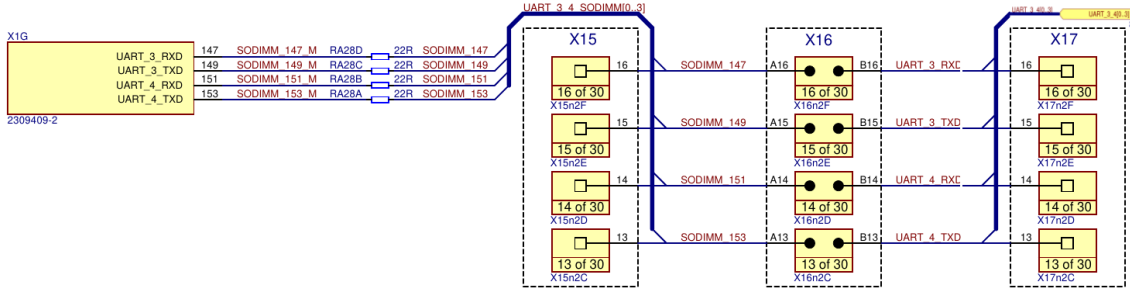


Figure 3.7: Zoom on UART socket X16

### 3.3.2 The software

The ports access to the bootloader command line gave rise to two tests. The goal of these are to find what type of information can be accessed or changed using the bootloader command line and also if it would be possible to run small scripts in order to search for sensitive data.

#### 3.3.2.1 Test 1: Memory dumping

This test was done in order to find the type of data that could be accessed inside the bootloader command line. Also if it would be possible to change the applications that were running at the time of the reset. The memory dump was done using the serial client Putty, python code (APPENDIX C) for parsing and a bat script. Using the hardware reset included on the SoC and every SoC made by Toradex, the RAM content can be kept whilst forcing the bootloader shell to appear. Even if the product uses a password protected shell, the containers that were running before the reset will still store unencrypted data on the RAM. This content can be dumped using the "md" command. Which is tested by building a container that stores some variables and continuously prints them using a while loop (Appendix C).

The issue with finding the container is the raw size of the RAM (2Gb) and what addresses to start from. This is solved by finding the start address of the RAM, which can be found by the commonly included command "bdfinfo". As can be seen in fig 3.8, the command states the start address (0x40000000) and size(0x80000000). Even though the size of the memory is relatively small, dumping 2 Gb over the 115200 baud serial-line would take over two days to dump fully. This is made even slower due to the format of the memory dump when using "md". The dump will contain symbols in both hex and ASCII, i.e the memory address, the memory data as hex and in ASCII the data once again. Effectively making the dump three times as large for each byte. See figure 3.9 for an example of a RAM dump. According

to the datasheet of the CPU [29], the lower addresses will always be allocated first. This means that starting the search from the lowest address first, guarantees that the majority of the memory will not have to be scanned.

```
Verdin iMX8MM # bdfinfo
arch_number = 0x0000000000000000
boot_params = 0x0000000000000000
DRAM bank   = 0x0000000000000000
-> start     = 0x0000000040000000
-> size      = 0x0000000080000000
baudrate     = 115200 bps
TLB addr     = 0x00000000bfff0000
relocaddr    = 0x00000000bfff1000
reloc off    = 0x000000007fd10000
irq_sp       = 0x00000000bd70fda0
sp start     = 0x00000000bd70fda0
FB base      = 0x0000000000000000
Early malloc usage: 8170 / 10000
fdt_blob     = 0x00000000bffa9f30
Verdin iMX8MM #
```

**Figure 3.8:** Uboot shell, running the "bdfinfo" command

```
Verdin iMX8MM # md 40000000 30
40000000: c9c5c5c9 a89494a8 26323226 fcfcfcfcf .....&22&....
40000010: edf5f5ed 94a8a894 cfe7e7cf 21111121 .....!...!
40000020: 928a8a92 a7b3b3a7 91898991 88848488 .....
40000030: 534b4b53 adb5b5ad 1a0e0e1a 15292915 SKKS.....) ) .
40000040: 381c1c38 d3cbcbdb cee6e6ce b5b9b9b5 8..8.....
40000050: c7e3e3c7 190d0d19 a89494a8 5d6d6d5d .....]mm]
40000060: 67737367 07232307 6e76766e 4a46464a gssg.##.nvvJFFJ
40000070: 05212105 47636347 725a5a72 3f3f3f3f .!!.GccGrZZr????
40000080: bdbdbdbd 84a0a084 b09898b0 80808080 .....
40000090: 63535363 afb7b7af d2cacad2 21111121 cSSc.....!...!
400000a0: 7b5f5f7b 938b8b93 d2cacad2 c0c0c0c0 {__ { .....
400000b0: 777b7b77 e2d2d2e2 584c4c58 20101020 w{ {w....XLLX ..
Verdin iMX8MM #
```

**Figure 3.9:** Uboot shell, memory dumping from RAM

In order to store and search this data for the container, two different methods can be used. The first method is to store the dump directly into a text file and analyze the file. A script is used to search for variable names, values and the name of the container (See appendix C) inside the text file. This is done by entering the dump command with a huge number of objects, closing Putty and running a MSDOS command to pipe the data into a text file. Due to the baud-rate of the communication line run time may well surpass eight hours.

For further clarification, the process was as follows:

1. In U-boot shell: md 40000000 ffffffff
2. Close Putty as fast as possible
3. In CMD (Windows): copy COMx myfile.txt



4. Leave overnight
5. Copy file using file explorer
6. Close CMD
7. Analyze using python script

This method is easier to implement, but it is not as fast as the second method. It also requires a .bat script to save the dumps.

The second method is very similar, but it pipes the data directly into the python script which scans continuously. This is much faster since each match is immediately discovered. The code is a bit more complex though since the piping is done using libUSB. The script found in Appendix C does not store any of the data, it only reads it and searches for the keywords. If a match is found the address is saved in a log and a notification is sent using a API. The API makes sure that the notification can be seen on any device. If the process was killed unexpectedly the latest searched address is saved and a notification is sent.

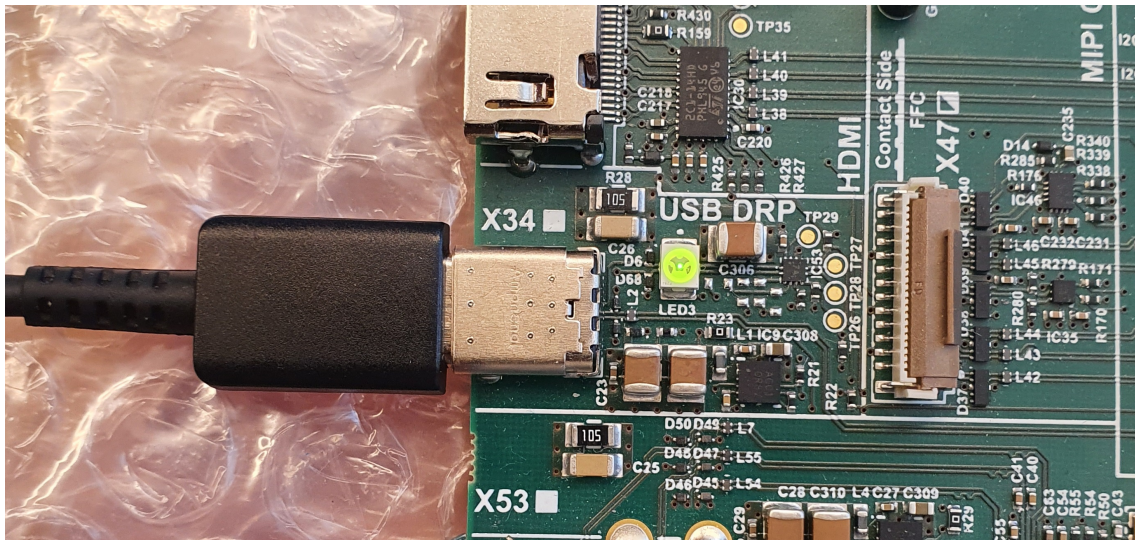
#### **3.3.2.2 Test 2: Bootloader scripting**

The point of this test is to investigate the scripting capabilities of the bootloader command line. The command line has a command called "go" which is capable of executing small binaries. For example if a script could be used to find usernames, password or credit card numbers this will prove to be a powerful exploit. Below are the three simple steps outlined.

1. Compile script
2. Load into RAM
3. execute with "go"

Bootloader scripts require very specific dependencies due to the bootloaders low abstraction layer. The dependencies vary depending on the U-boot version and the hardware itself. For the Toradex these project files are included on their website.

## 3.4 The Recovery port



**Figure 3.10:** Recovery port X34 on development board

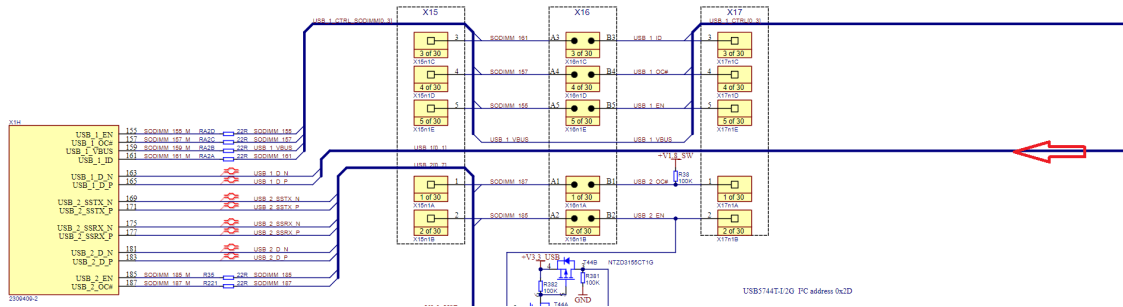
The behaviour of the Recovery port is very different from the debugging port. The port itself is not meant to be interacted with outside of the easy installer software that Toradex provides. The installer being the software that transfers the U-boot and OS image. In fact, the port remains completely undetectable by host OS until the recovery mode is activated. The actual purpose of the port is to enable the developer to install or re-install the bootloader and operating system. It does so by only opening when the SoC is set to something called recovery mode. To enter recovery mode the reset button is pressed (button 2) whilst holding the recovery button (button 3). The buttons are shown in figure 3.11. Otherwise it remains hidden as a USB hub. This enables the developer to save the platform even if the OS system or/and the bootloader files corrupts. Whilst also hiding the port from malicious parties. If the ports only means of security is hiding behind the USB protocol then its hacking potential might be immense. It is *strongly* recommended to read section 2.1 before continuing, the section will assume that the reader is already aware of how the USB protocol functions.

### 3.4.1 The hardware

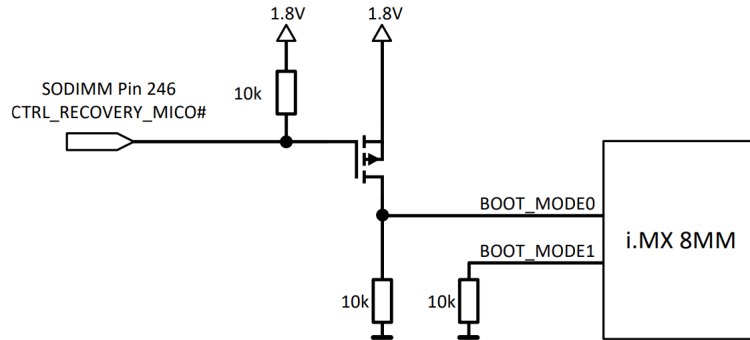
The behaviour of the recovery port is very different from the debugging port. This also applies to the hardware and how it is connected to the SoC. Using the blueprints supplied by Toradex, the first part of the connection between the port and the SoC can be seen in figure 3.12. Following the data lines DP 1-2 and DN 1-2 will lead to figure 3.13. According to the figure, the line completely bypasses any connection on the development board. Instead, it leads straight to the SoC on pins 181 and 183. This makes the port very hard to sniff, although not impossible. The SoC still uses the DDR4 socket, which means that the pins on the socket can be sniffed albeit with



### 3. Investigation and testing methodology



**Figure 3.13:** Data lines from the recovery port to the DDR4 socket of the SoC



**Figure 3.14:** Recommended schematic to force recovery mode [8]

#### 3.4.2 The software

As mentioned the port will be inactive at first, but a different behaviour can be observed when using the recovery mode and running the easy installer. At first glance the port cannot be found, using the windows debugging tool called USBView this is proven in figure 3.15. Pay special attention to "Port 1" under the second to last root hub. As can be seen, the system does not detect the port at all. The problem is that in order to communicate with the device, the port will have to open or at the very least show itself as a type of device. Enter the recovery mode, this mode is mentioned in multiple places in the Toradex documentation. The mode exists for all SoCs provided by Toradex. In order to activate the mode a special button is included on the development board figure 3.11, see the documentation in [8] for further information. This button makes it easier to activate but it is not required, see the hardware subsection. When the recovery mode is accessed the port changes class into a HID, and can be seen as such using the USBView debugging tool in figure 3.16.

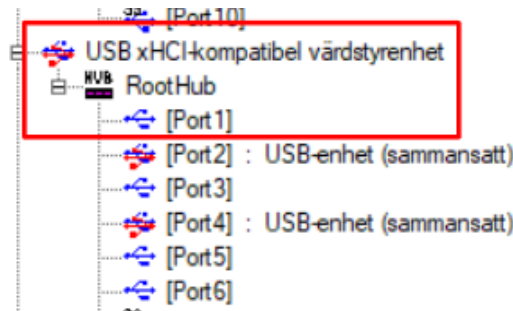


Figure 3.15: USBView on host, when the device is not in recovery mode

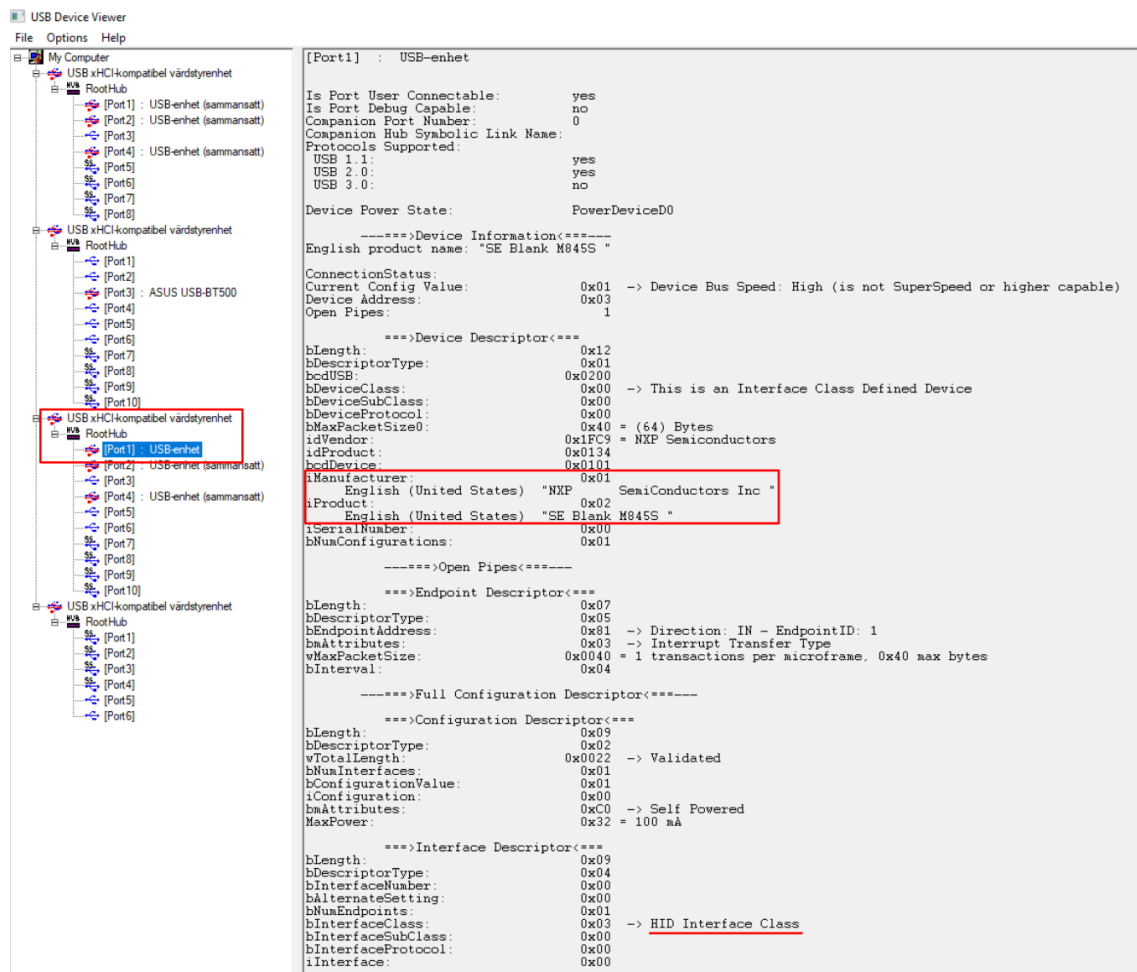


Figure 3.16: USBView on host, when the device is in recovery mode

In order to investigate the process of the port closer, especially how it opens up for flashing the U-boot and OS, a test was developed. The goal of this test is to analyse the process when the easy installer attaches itself to the port and transfers the two images. When this process is understood, it might be possible to hijack the transaction and gain full control of the port. As mentioned the port will remain inactive until the recovery mode is activated. This mode opens the recovery port and displays it as a HID device (more in 4.2). It is recommended to check the

Toradex documentation for how to run the easy installer correctly. The test had the following steps

1. Connect host PC to the recovery port
2. Activate sniffer
3. Activate recovery mode
4. Run easy installer and record transaction
5. Analyse transaction
6. Hijack process with a tailored USB driver

The easy installer will most likely look slightly different depending on the SoC in use. However, the process will remain the same and the documentation from Toradex is excellent. The sniffing is done with Wireshark and development of the USB driver was done using a C library called libUSB 1.0. See subsection 4.2 for the result and appendix C for the driver code.

## 3.5 Summary

As the section 3.1.1 suggests, the investigation begins with background research and setting up the system. The background research also aims to find simple and commonly known hardware exploits. Some of them might work on this platform. Then the investigation of two separate I/O ports is done. The investigation has multiple questions that needs to be kept in mind. These summarized, are the functions of the port, what hardware it is connected to and how this can be exploited. They are all answered under their respective sections 4.1.2 & 4.2 and discussed under 5.1.1 & 5.1.2.

In summary, there are multiple simple hacks that were found. These were as listed below:

- USB fuzzing [23]
  - Done by opening as many endpoints of as many different classes as possible
  - Can also be a way to disguise endpoints from user
  - Goal is to get a crash dump or any type of access
- Firmware extraction [24, 25]
  - Extracting the firmware code so it can be analysed and hacked



- Can be done through UART or hijacking the ROM for example.
- Relay and replay [26]
  - Record transfer transaction and replay it
  - Can pretend to be a different nfc device
- Finding UART and getting Shell [7, 24]
  - Many products hides UART inside the hardware so the debug port can remain active.
  - If found, root access is usually one standard password away.
- Glitching [23, 27]
  - An attempt to crash the boot sequence
  - Usually done by shorting or grounding flash, CPU or RAM pins.

Lastly there are multiple tests that were done, two for the debugging port and one for the recovery port. The first one was memory dumping the RAM. The test utilizes the Uboot command line to see if the containers image can be changed by resetting the device whilst running. This is done by memory dumping large amounts of data and then memory writing to RAM and proceeding with the boot process. The second test is essentially port sniffing. This can be done in multiple ways, the one used here is through Wireshark. The last test is trying to run the malicious driver developed. The point is to see if the device responds to the data in a similar fashions as the sniffed data. The script is launched and then the device put into recovery mode.





# 4

## Results

This chapter contains the result of the investigation. It will provide evidence of the security weaknesses and how these were found.

### 4.1 The Debug port

As mentioned in the methods subsection 3.3.2, two tests were developed to explore the vulnerabilities found in the bootloader command line. The result of these two can be found below.

#### 4.1.1 Test 1: Memory Dumping

To investigate the port, it is necessary to interact with it and observe its behaviour. The first step is then to plug into the port and observe its behaviour. As mentioned, the port is called X66 on the development board and utilizes a USB-C connector (fig.3.3). Since the port uses UART, putty was used to communicate with the device.

The startup text is very informative to the start of this port investigation. At the top of the text (appendix B), there is two mentions of the U-boot version (excluding the build info). This is due to the startup process. The system starts with a binary injection called the SPL which in its turn boots the larger version of U-boot. See 2.2 for further information on this process. The mentioned lines can also be seen below:

- "U-Boot SPL 2020.04-5.5.0+git.81bc8894031d (Jan 01 1970 - 00:00:00 +0000)"
- "U-Boot 2020.04-5.5.0+git.81bc8894031d (Jan 01 1970 - 00:00:00 +0000)"

The versions are important since they show how up to date the bootloader is. There is a high number of products that updates its operating system but not its bootloader. This is why the OS and bootloader versions both, are very important. In this case, the system is seemingly quite up to date. There are no known large security gaps in this version. This can be found by googling around the version or checking the Toradex errata and U-boot errata, both which are publicly known. Continuing

down the text, it displays the board version, its serial number and DRAM size. These can all be easily found on the Toradex website as well. The most interesting text can be seen in the list below:

- "NOTICE: BL31: v2.2(release):toradex\_imx\_5.4.70\_2.3.0-g835a8f67b2"
- "Hit any key to stop autoboot: 0"
- "Scanning mmc 0:1..."
- "Found U-Boot script /boot.scr"

The first item mentions the image file that is used to launch the bootloader from memory. The name of the file indicates that it is a tailored version of the bootloader. There is also a version number in the image files name. The version at this time is the most recent bootloader supplied by Toradex. The next item mentions the autoboot function. The autoboot is a timer that starts counting down until the kernel is opened. During this timer it is possible to abort the booting process and instead, open the boot shell. This should always be disabled, but in this case it was not. Autoboot enabled is very promising but not necessary since the shell can be forced to open, as mentioned in 3.2 and discussed in 6. For now, the next item "Scanning mmc 0:1...", means that the device is using a **M**ulti **M**edia **C**ard (**MMC**) when booting. The "0:1" says that the card is connected to socket 0 and the partition scanned on the card is partition 1. The last item on the list confirms that this is where the U-boot and therefore the operating system is located. This means that all data from containers and other software will be on this device and partition.

Since booting the system the normal way will throw a login screen if configured correctly, rest of the investigation focused on the bootloader. When in the bootloader shell, there is a help command that will display all available commands. See appendix A for the full list of available commands in this U-boot version. The amount of commands in the bootloader tend to vary depending on the product. Since it is open-source, the majority of commands are implemented in the code but deactivated by default when compiling. In the Toradex, the commands which had the most potential were "md", "mw" and "mmc". See table 4.1 for the synopsis and description of these commands.

md	md [.b, .w, .l, .q] address [# of objects]	Dump data from RAM adr
mw	mw [.b, .w, .l, .q] address value [count]	Write data to RAM adr
mmc read	mmc read addr blk# cnt	Read data from mmc adr
mmc write	mmc write addr blk# cnt	Write data to mmc adr

**Table 4.1:** U-boot commands of special interest

The commands are clearly designed to interact with either the RAM memory or flash memory. The obvious command of interest is the mmc command. The problem however, is that the command uses raw memory addressing when reading. This denotes that in order to access data on the flash, parts would systematically have

to be moved to RAM and then moved out to the host PC using a serial connection or tftp. Due to the size of the flash, this would take an incredible amount of time. Especially if using the serial connection. This is still noted as a possibility, system files can most likely be accessed this way. Proving this however, would leave no time for anything else.

The result of the RAM dump search can be seen as examples below. There are larger amounts of matches in total but these remain the most relevant:

```
5e4b2040: 00011040 00000000 736a6568 73206e61 @.....hejsan s
5e4b2050: 736a6576 000a6e61 00012fd1 3a434347 vejsan.../..GCC:
5e4b2060: 65442820 6e616962 2e303120 2d312e32 (Debian 10.2.1-
5e4b2070: 31202936 2e322e30 30322031 31303132 6) 10.2.1 202101
5e4b2080: 2c003031 02000000 00000000 00000800 10.,.....
```

**Match 1:** Match for a char array containing the string "hejsan svejsan". Proving that container string values can be found inside RAM.

```
85d9e240: 00000101 00000100 65480001 576f6c6c .....HelloW
85d9e250: 646c726f 0000632e 05000000 02090001 orld.c.....
85d9e260: 000007b4 00000000 4b090518 01040200 .....K....
85d9e270: 04020022 02005901 022d0104 01010001 "....Y....-....
85d9e280: 20554e47 20373143 322e3031 3220312e GNU C17 10.2.1 2
85d9e290: 30313230 20303131 696c6d2d 656c7474 0210110 -mlittle
85d9e2a0: 646e652d 206e6169 62616d2d 706c3d69 -endian -mabi=lp
85d9e2b0: 2d203436 662d2067 6e797361 6f726863 64 -g -fasynchro
85d9e2c0: 73756f6e 776e752d 2d646e69 6c626174 nous-unwind-tabl
85d9e2d0: 6d007365 006e6961 726f772f 6170736b es.main./workspa
85d9e2e0: 2f736563 6c6c6548 726f576f 6c00646c ces/HelloWorld.l
85d9e2f0: 20676e6f 69736e75 64656e67 746e6920 ong unsigned int
85d9e300: 736e7500 656e6769 68632064 6c007261 .unsigned char.l
85d9e310: 20676e6f 00746e69 6c6c6548 726f576f ong int.HelloWor
85d9e320: 632e646c 67726100 68730063 2074726f ld.c.argc.short
85d9e330: 69736e75 64656e67 746e6920 69687400 unsigned int.thi
85d9e340: 72747373 6f687300 69207472 6100746e sstr.short int.a
85d9e350: 00766772 00000000 00000000 00000000 rgv.....
```

**Match 2:** Left most column is the memory address, the rest is its content and the ASCII representation of it. Match containing variable names and the file name "HelloWorld.c". The file name is the main file from which the container was built from.

```
85bcf890: 01000001 726f772f 6170736b 2f736563 ....workspaces/
```

```
85bcf8a0: 7041796d 78652f70 752f0065 612f7273 myApp/exe./usr/a
85bcf8b0: 68637261 6c2d3436 78756e69 756e672d arch64-linux-gnu
85bcf8c0: 636e692f 6564756c 7469622f 65000073 /include/bits..e
85bcf8d0: 632e6578 01007070 6f630000 616e666e xe.cpp....confna
85bcf8e0: 682e656d 00000200 00010500 0a440209 me.h.....D.
```

**Match 3:** Match containing mentions of the workspace. The file path mentions the workspace of the container and the toolchain when processed.

```
6a9bdbb0: 20746e69 6e696f70 20726574 0a207825 int pointer %x .
6a9bdbbc0: 00000000 00000000 6e696f70 3a726574 .....pointer:
6a9bdbbd0: 20782520 0000000a 6c6c616d 2520636f %x ....malloc %
6a9bdbbe0: 00000070 3b031b01 00000040 00000007 p.....;@.....
```

**Match 4:** This match contains the declaration of some of the variables. Proving that variable declarations inside container images can be found inside RAM.

These matches will be further discussed in chapter 6, but the results here prove that contents of a container can be found unencrypted in RAM. These matches gave rise to another test as well, namely if changing any of the RAM data would impact the contents of the container. The testing method meant doing the same thing as before but adding a step at the end. For all matches the script found, the command "mw" was used to manipulate the content. Variable values were changed, e.g a char array containing "hello world!" was changed to "abcabcabcabc" or a integers number were changed from "77777" to "99999". The container always prints its values when launched, so if the change manifested itself within the container, the new values would be seen in the portainer console. This change did no impact the container at all however, why that is will again be discussed in chapter 6.

### 4.1.2 Test 2: Bootloader scripting

The result of this test was semi-successful, as can be seen in figure 4.1 the binary has compiled successfully. The binary did not run inside the command line however. This was due to missing dependencies that were hardware specific. The problem is therefore in the linking process before compilation.

```

.data:000001f3          f9 29 59 40 f9 20 01 1f d6 49 72 40 f9          .)Y@. ...Ir@.
.data:00000200 29 5d 40 f9 20 01 1f d6 49 72 40 f9 29 61 40 f9  )]|. ...Ir@.)a@.
.data:00000210 20 01 1f d6 49 72 40 f9 29 65 40 f9 20 01 1f d6  ...Ir@.)e@. ...
.data:00000220 49 72 40 f9 29 69 40 f9 20 01 1f d6 49 72 40 f9  Ir@.)i@. ...Ir@.
.data:00000230 29 6d 40 f9 20 01 1f d6 49 72 40 f9 29 71 40 f9  )m@. ...Ir@.)q@.
.data:00000240 20 01 1f d6 49 72 40 f9 29 75 40 f9 20 01 1f d6  ...Ir@.)u@. ...
.data:00000250 49 72 40 f9 29 79 40 f9 20 01 1f d6 49 72 40 f9  Ir@.)y@. ...Ir@.
.data:00000260 29 7d 40 f9 20 01 1f d6 49 72 40 f9 29 81 40 f9  )}|. ...Ir@.)@.
.data:00000270 20 01 1f d6 c0 03 5f d6 80 00 00 90 81 00 00 90  .....
.data:00000280 00 b0 3f 91 21 c0 3f 91 1f 00 01 eb 43 00 00 54  ..?.!?.....C..T
.data:00000290 c0 03 5f d6 1f 14 00 38 fc ff ff 17 3c 4e 55 4c  .._...8....<NUL
.data:000002a0 4c 3e 00 48 65 6c 6c 6f 20 74 68 65 72 65 00 45  L>.Hello there.E
.data:000002b0 78 61 6d 70 6c 65 20 65 78 70 65 63 74 73 20 41  xample expects A
.data:000002c0 42 49 20 76 65 72 73 69 6f 6e 20 25 64 0a 00 41  BI version %d..A
.data:000002d0 63 74 75 61 6c 20 55 2d 42 6f 6f 74 20 41 42 49  ctual U-Boot ABI
.data:000002e0 20 76 65 72 73 69 6f 6e 20 25 64 0a 00 48 65 6c  version %d..Hel
.data:000002f0 6c 6f 20 57 6f 72 6c 64 0a 00 61 72 67 63 20 3d  lo World..argc =
.data:00000300 20 25 64 0a 00 61 72 67 76 5b 25 64 5d 20 3d 20  %d..argv[%d] =
.data:00000310 22 25 73 22 0a 00 48 69 74 20 61 6e 79 20 6b 65  "%s"..Hit any ke
.data:00000320 79 20 74 6f 20 65 78 69 74 20 2e 2e 2e 20 00 0a  y to exit ... ..
.data:00000330 0a 00 00 00 00 00 00 00 10 00 00 00 00 00 00 00  .....
.data:00000340 01 7a 52 00 04 78 1e 01 1b 0c 1f 00 30 00 00 00  ..zR..X.....0...
.data:00000350 18 00 00 00 ac fc ff ff e8 00 00 00 00 41 0e 40  .....A.@
.data:00000360 9d 08 9e 07 42 93 06 94 05 44 95 04 96 03 42 97  ....B....D....B.
.data:00000370 02 69 0a de dd d7 d5 d6 d3 d4 0e 00 41 0b 00 00  ..i.....A...
.data:00000380 10 00 00 00 4c 00 00 00 60 fd ff ff 90 01 00 00  ....L.....
.data:00000390 00 00 00 00 10 00 00 00 60 00 00 00 dc fe ff ff  ....`.....
.data:000003a0 24 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  $.
.data:000003b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
.data:000003c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
.data:000003d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
.data:000003e0 00 00 00 00 00 00 00 00  .....

```

**Figure 4.1:** Disassembled bootloader binary, code can be seen in appendix C

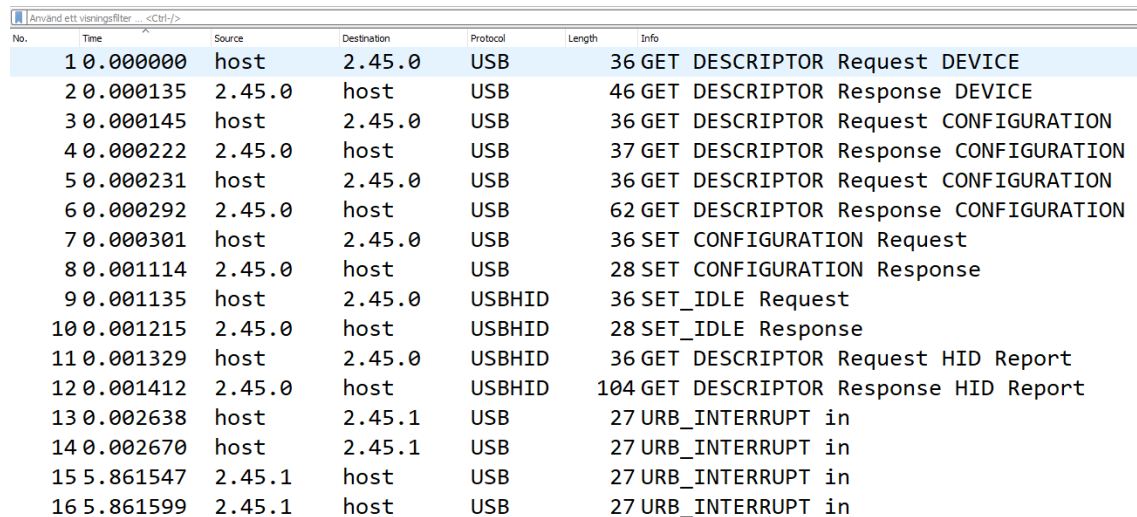
## 4.2 The Recovery port

In order to obtain more data about the recovery mode and how the easy installer functions, a USB sniffer was used. See the test in subsection 3.4.2. The result can be split into two different parts, the first being the communication when the recovery mode is activated (seen in fig 4.2) and the second when the easy installer takes over.

No communication will happen until the recovery mode is activated. This is so even though the USB hub it hides as can be seen in the device manager. When the mode is activated communication starts immediately, as can be seen in figure 4.2. This handshake is the first part of the communication. The port reveals itself to the operating system by opening up a HID interface. When opened, the device tells the operating system that it is a HID with a certain vendor and product ID. This is most likely where the easy installer comes in and sets up the communication by using an application layered driver.

Before showcasing the evidence for a second driver, the handshake and set reports needs to be shown. As can be alluded to in the figure 4.3, the communication line uses two endpoints. The endpoint number in Wireshark is displayed by the last digit under the "Destination" label. The second number corresponds to the assigned address and the first number the bus ID. For example, the first packets destination is from the host PC to 2.45.0, which means bus 2, address 45 and endpoint 0. For further clarification see subsection 2.1.1. The line clearly uses two points of communication, endpoint 0, which is the control endpoint and endpoint 1 which

## 4. Results



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	host	2.45.0	USB	36	GET_DESCRIPTOR Request DEVICE
2	0.000135	2.45.0	host	USB	46	GET_DESCRIPTOR Response DEVICE
3	0.000145	host	2.45.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
4	0.000222	2.45.0	host	USB	37	GET_DESCRIPTOR Response CONFIGURATION
5	0.000231	host	2.45.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
6	0.000292	2.45.0	host	USB	62	GET_DESCRIPTOR Response CONFIGURATION
7	0.000301	host	2.45.0	USB	36	SET_CONFIGURATION Request
8	0.001114	2.45.0	host	USB	28	SET_CONFIGURATION Response
9	0.001135	host	2.45.0	USBHID	36	SET_IDLE Request
10	0.001215	2.45.0	host	USBHID	28	SET_IDLE Response
11	0.001329	host	2.45.0	USBHID	36	GET_DESCRIPTOR Request HID Report
12	0.001412	2.45.0	host	USBHID	104	GET_DESCRIPTOR Response HID Report
13	0.002638	host	2.45.1	USB	27	URB_INTERRUPT in
14	0.002670	host	2.45.1	USB	27	URB_INTERRUPT in
15	5.861547	2.45.1	host	USB	27	URB_INTERRUPT in
16	5.861599	2.45.1	host	USB	27	URB_INTERRUPT in

**Figure 4.2:** Sniffed packets using Wireshark, before the easy installer has attached

is the point going from the device into the host. This is very common when the host wants to see further acknowledgement not coming from the USB protocol, but some other software running on the device (it is therefore technically data). This is where all communication ends until the installer is executed and the new driver takes over. When the new driver is attached it starts setting up the device. For any HID this is always done using the SET\_REPORT Requests which can be seen starting at the bottom of the figure 4.3. Due to the nature of the packet format, the number of endpoints and the purpose of the easy installer, this section is theorized to be the setup stage that unlocks the Toradex so that the bootloader and OS can be installed.

The reason for why a second driver is most likely used, can be insinuated from the nature of the communication. As can be seen in figure 4.3, endpoint 0 is set to idle twice. This happens first when the device changes endpoint class to HID with packets 9-10. At that point the installer is not running so it has to be the OS specified driver that handles the handshake. The moment the installer runs the second request is sent 17-18. This happened since the new driver does not know what mode the HID is running. The "IDLE" mode is not contained inside the descriptor objects, which any driver would have access to in order to find first time information about the connection. It could therefore not know the mode without setting it itself. When activating the driver, it searches for the device with the correct vendor and product ID contained in the descriptor objects. It then attaches itself to the device and sets it to idle, now sure about the mode of the device. This behaviour is made even more obvious when activating the easy installer before the device enters its recovery mode. The easy installer sits and waits for the correct user ID and no

traffic is observed beforehand.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	host	2.41.0	USB	36	GET_DESCRIPTOR Request DEVICE
2	0.000174	2.41.0	host	USB	46	GET_DESCRIPTOR Response DEVICE
3	0.000208	host	2.41.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
4	0.000291	2.41.0	host	USB	37	GET_DESCRIPTOR Response CONFIGURATION
5	0.000304	host	2.41.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
6	0.000395	2.41.0	host	USB	62	GET_DESCRIPTOR Response CONFIGURATION
7	0.000411	host	2.41.0	USB	36	SET_CONFIGURATION Request
8	0.001150	2.41.0	host	USB	28	SET_CONFIGURATION Response
9	0.001167	host	2.41.0	USBHID	36	SET_IDLE Request
10	0.001239	2.41.0	host	USBHID	28	SET_IDLE Response
11	0.001386	host	2.41.0	USBHID	36	GET_DESCRIPTOR Request HID Report
12	0.001462	2.41.0	host	USBHID	104	GET_DESCRIPTOR Response HID Report
13	0.002620	host	2.41.1	USB	27	URB_INTERRUPT in
14	0.002645	host	2.41.1	USB	27	URB_INTERRUPT in
15	6.085037	2.41.1	host	USB	27	URB_INTERRUPT in
16	6.085090	2.41.1	host	USB	27	URB_INTERRUPT in
17	18.467828	host	2.41.0	USBHID	36	SET_IDLE Request
18	18.467993	2.41.0	host	USBHID	28	SET_IDLE Response
19	18.468086	host	2.41.1	USB	27	URB_INTERRUPT in
20	18.468114	host	2.41.1	USB	27	URB_INTERRUPT in
21	18.468385	host	2.41.0	USB	36	GET_DESCRIPTOR Request STRING
22	18.468537	2.41.0	host	USB	86	GET_DESCRIPTOR Response STRING
23	18.468595	host	2.41.0	USB	36	GET_DESCRIPTOR Request STRING
24	18.468801	2.41.0	host	USB	60	GET_DESCRIPTOR Response STRING
25	18.475044	host	2.41.0	USBHID	53	SET_REPORT Request
26	18.475273	2.41.0	host	USBHID	28	SET_REPORT Response
27	18.475834	host	2.41.0	USBHID	1061	SET_REPORT Request
28	18.476102	2.41.0	host	USBHID	28	SET_REPORT Response

**Figure 4.3:** Sniffed packets using Wireshark, after the easy installer has attached

When the settings have been sent to the device using the set report format, the device start acknowledging changes in the system. This specifically happens through endpoint 1 which can be seen happening in figure 4.4. Shortly after, the communication is aborted and renewed as a different device. This can be seen in figure 4.5, where the handshake happens all over again. This time the descriptors are different and a different configuration is used, the same can be said for the interface. The new interface classes itself as a HID once more but instead uses three endpoints, including the control endpoint 0. This time the new endpoints (1 and 2), are used for sending commands in the form of interrupts. This is most likely due to the host needing more bandwidth to send data to the device. Using the set report on endpoint 0 also works but it limits bandwidth since it will have to be shared with controller packets. The acknowledgement outside the protocol happens the same as before through a separate endpoint, numbered 1 in this case. These are acknowledgements are small data packets and are assumed to be some sort of confirmation from the device software that the host can continue.

There is then a much more intuitive break in communication. This time the device uses a new interface that again uses three endpoints. This time the endpoints are defined as bulk transfer points however. The beginning of the packets can be seen in figure 4.6. This is clearly the point where the actual data transfer of the bootloader

## 4. Results

No.	Time	Source	Destination	Protocol	Length	Info
384	18.536864	2.41.0	host	USBHID	28	SET_REPORT Response
385	18.536953	host	2.41.0	USBHID	1061	SET_REPORT Request
386	18.537188	2.41.0	host	USBHID	28	SET_REPORT Response
387	18.537257	host	2.41.0	USBHID	1061	SET_REPORT Request
388	18.537478	2.41.0	host	USBHID	28	SET_REPORT Response
389	18.537543	host	2.41.0	USBHID	1061	SET_REPORT Request
390	18.537782	2.41.0	host	USBHID	28	SET_REPORT Response
391	18.537846	host	2.41.0	USBHID	1061	SET_REPORT Request
392	18.538072	2.41.0	host	USBHID	28	SET_REPORT Response
393	18.538137	host	2.41.0	USBHID	1061	SET_REPORT Request
394	18.538348	2.41.0	host	USBHID	28	SET_REPORT Response
395	18.538418	host	2.41.0	USBHID	1061	SET_REPORT Request
396	18.538658	2.41.0	host	USBHID	28	SET_REPORT Response
397	18.540232	2.41.1	host	USB	32	URB_INTERRUPT in
398	18.540274	host	2.41.1	USB	27	URB_INTERRUPT in
399	18.541212	2.41.1	host	USB	92	URB_INTERRUPT in
400	18.541252	host	2.41.1	USB	27	URB_INTERRUPT in
401	18.541465	host	2.41.0	USBHID	53	SET_REPORT Request
402	18.541636	2.41.0	host	USBHID	28	SET_REPORT Response
403	18.543231	2.41.1	host	USB	32	URB_INTERRUPT in
404	18.543266	host	2.41.1	USB	27	URB_INTERRUPT in
405	18.544217	2.41.1	host	USB	92	URB_INTERRUPT in
406	18.544260	host	2.41.1	USB	27	URB_INTERRUPT in
407	18.544333	host	2.41.0	USBHID	53	SET_REPORT Request
408	18.544498	2.41.0	host	USBHID	28	SET_REPORT Response
409	18.546211	2.41.1	host	USB	32	URB_INTERRUPT in
410	18.546250	host	2.41.1	USB	27	URB_INTERRUPT in
411	18.944159	2.41.1	host	USB	27	URB_INTERRUPT in
412	18.944208	2.41.1	host	USB	27	URB_INTERRUPT in
413	19.079670	host	2.41.1	USB	27	URB_FUNCTION_ABORT_PIPE
414	19.079703	2.41.1	host	USB	27	URB_FUNCTION_ABORT_PIPE

**Figure 4.4:** Sniffed packets using Wireshark, endpoint 1 starts interrupting

and OS begins. The bulk class is always used when a long stream of data is to be transmitted without a special need for guaranteed latency. This is also supported by some of the ascii data inside the bulk packets. Some of them are obvious commands such as "UCmd: setenv fastboot\_buffer 0x42e00000" or "download: 0000 0bb0". Right at the beginning of the maximum sized bulk transfers there is also the ascii, "U-Boot fitImage for Toradex Easy Installer...". These leave little doubt as to the nature of the data.



No.	Time	Source	Destination	Protocol	Length	Info
413	19.079670	host	2.41.1	USB	27	URB_FUNCTION_ABORT_PIPE
414	19.079703	2.41.1	host	USB	27	URB_FUNCTION_ABORT_PIPE
415	19.193862	host	2.42.0	USB	36	GET_DESCRIPTOR Request DEVICE
416	19.194058	2.42.0	host	USB	46	GET_DESCRIPTOR Response DEVICE
417	19.194086	host	2.42.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
418	19.194274	2.42.0	host	USB	37	GET_DESCRIPTOR Response CONFIGURATION
419	19.194322	host	2.42.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
420	19.194482	2.42.0	host	USB	69	GET_DESCRIPTOR Response CONFIGURATION
421	19.194496	host	2.42.0	USB	36	SET_CONFIGURATION Request
422	19.195604	2.42.0	host	USB	28	SET_CONFIGURATION Response
423	19.195623	host	2.42.0	USBHID	36	SET_IDLE Request
424	19.195730	2.42.0	host	USBHID	28	SET_IDLE Response
425	19.195863	host	2.42.0	USBHID	36	GET_DESCRIPTOR Request HID Report
426	19.198271	2.42.0	host	USBHID	104	GET_DESCRIPTOR Response HID Report
427	19.199373	host	2.42.1	USB	27	URB_INTERRUPT in
428	19.199399	host	2.42.1	USB	27	URB_INTERRUPT in
429	20.053195	host	2.42.0	USB	36	GET_DESCRIPTOR Request STRING
430	20.053702	2.42.0	host	USB	36	GET_DESCRIPTOR Response STRING
431	20.053752	host	2.42.0	USB	36	GET_DESCRIPTOR Request STRING
432	20.053933	2.42.0	host	USB	68	GET_DESCRIPTOR Response STRING
433	21.069208	host	2.42.2	USB	44	URB_INTERRUPT out
434	21.070201	2.42.2	host	USB	27	URB_INTERRUPT out
435	21.070280	host	2.42.2	USB	1052	URB_INTERRUPT out
436	21.075210	2.42.2	host	USB	27	URB_INTERRUPT out
437	21.075340	host	2.42.2	USB	1052	URB_INTERRUPT out
438	21.076685	2.42.2	host	USB	27	URB_INTERRUPT out
439	21.076808	host	2.42.2	USB	1052	URB_INTERRUPT out
440	21.078188	2.42.2	host	USB	27	URB_INTERRUPT out
441	21.078296	host	2.42.2	USB	1052	URB_INTERRUPT out
442	21.079684	2.42.2	host	USB	27	URB_INTERRUPT out
443	21.079765	host	2.42.2	USB	1052	URB_INTERRUPT out

**Figure 4.5:** Sniffed packets using Wireshark, new handshake and then interrupts

No.	Time	Source	Destination	Protocol	Length	Info
2539	23.027289	host	2.42.1	USB	27	URB_FUNCTION_ABORT_PIPE
2540	23.027321	host	2.42.1	USB	27	URB_FUNCTION_ABORT_PIPE
2541	24.422701	host	2.43.0	USB	36	GET_DESCRIPTOR Request DEVICE
2542	24.422898	2.43.0	host	USB	46	GET_DESCRIPTOR Response DEVICE
2543	24.422927	host	2.43.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
2544	24.423084	2.43.0	host	USB	37	GET_DESCRIPTOR Response CONFIGURATION
2545	24.423124	host	2.43.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
2546	24.423235	2.43.0	host	USB	60	GET_DESCRIPTOR Response CONFIGURATION
2547	24.423244	host	2.43.0	USB	36	GET STATUS Request
2548	24.423362	2.43.0	host	USB	30	GET STATUS Response
2549	24.423391	host	2.43.0	USB	36	SET_CONFIGURATION Request
2550	24.424150	2.43.0	host	USB	28	SET_CONFIGURATION Response
2551	24.709428	host	2.43.2	USB	65	URB_BULK out
2552	24.709518	2.43.2	host	USB	27	URB_BULK out
2553	24.709619	host	2.43.1	USB	27	URB_BULK in
2554	24.713505	2.43.1	host	USB	31	URB_BULK in
2555	24.715214	host	2.43.2	USB	44	URB_BULK out
2556	24.715302	2.43.2	host	USB	27	URB_BULK out
2557	24.715396	host	2.43.1	USB	27	URB_BULK in
2558	24.718177	2.43.1	host	USB	39	URB_BULK in
2559	24.718381	host	2.43.2	USB	3...	URB_BULK out
2560	24.718515	2.43.2	host	USB	27	URB_BULK out
2561	24.718590	host	2.43.1	USB	27	URB_BULK in

**Figure 4.6:** Sniffed packets using Wireshark, new handshake and then bulk transfers

Since the only security found on the recovery port is seemingly hiding behind the USB protocol, a separate driver was developed. The driver found in appendix C, is a

## 4. Results

proof of concept that the communication sent by the easy installer can be hijacked. The goal was to send the exact same communication, whilst also sending a hijacked version of U-boot. The resulting communication succeeded to a point as seen in figure 4.7.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	host	2.40.0	USB	36	GET_DESCRIPTOR Request DEVICE
2	0.000199	2.40.0	host	USB	46	GET_DESCRIPTOR Response DEVICE
3	0.000224	host	2.40.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
4	0.000367	2.40.0	host	USB	37	GET_DESCRIPTOR Response CONFIGURATION
5	0.000391	host	2.40.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
6	0.000516	2.40.0	host	USB	62	GET_DESCRIPTOR Response CONFIGURATION
7	0.000539	host	2.40.0	USB	36	GET_STATUS Request
8	0.000670	2.40.0	host	USB	30	GET_STATUS Response
9	0.000723	host	2.40.0	USB	36	SET_CONFIGURATION Request
10	0.001643	2.40.0	host	USB	28	SET_CONFIGURATION Response
11	9.380947	host	2.40.0	USBHID	53	SET_REPORT Request
12	9.381129	2.40.0	host	USBHID	28	SET_REPORT Response
13	9.381224	host	2.40.0	USBHID	1061	SET_REPORT Request
14	9.381453	2.40.0	host	USBHID	28	SET_REPORT Response
15	9.381590	host	2.40.0	USBHID	1061	SET_REPORT Request
16	9.381832	2.40.0	host	USBHID	28	SET_REPORT Response
17	9.381949	host	2.40.0	USBHID	1061	SET_REPORT Request
18	9.382135	2.40.0	host	USBHID	28	SET_REPORT Response
19	9.382287	host	2.40.0	USBHID	1061	SET_REPORT Request
20	9.382516	2.40.0	host	USBHID	28	SET_REPORT Response
21	9.382642	host	2.40.0	USBHID	1061	SET_REPORT Request
22	9.382861	2.40.0	host	USBHID	28	SET_REPORT Response
23	9.383103	host	2.40.0	USBHID	1061	SET_REPORT Request
24	9.383349	2.40.0	host	USBHID	28	SET_REPORT Response
25	9.383597	host	2.40.0	USBHID	1061	SET_REPORT Request
26	9.383852	2.40.0	host	USBHID	28	SET_REPORT Response
27	9.384172	host	2.40.0	USBHID	1061	SET_REPORT Request
28	9.384387	2.40.0	host	USBHID	28	SET_REPORT Response
29	9.384470	host	2.40.0	USBHID	1061	SET_REPORT Request
30	9.384680	2.40.0	host	USBHID	28	SET_REPORT Response
31	9.384773	host	2.40.0	USBHID	1061	SET_REPORT Request
32	9.384994	2.40.0	host	USBHID	28	SET_REPORT Response
33	9.385078	host	2.40.0	USBHID	1061	SET_REPORT Request
34	9.385281	2.40.0	host	USBHID	28	SET_REPORT Response
35	9.385381	host	2.40.0	USBHID	1061	SET_REPORT Request
36	9.385607	2.40.0	host	USBHID	28	SET_REPORT Response
37	9.385693	host	2.40.0	USBHID	1061	SET_REPORT Request
38	9.385912	2.40.0	host	USBHID	28	SET_REPORT Response

**Figure 4.7:** First set of Set\_Reports sent by USB driver

The driver is currently able to send all set reports as recorded by the sniffer. The device acknowledges and receives correctly without issue. The communication has some slight differences though, e.g., there is no string descriptors requested and endpoint 1 is not set to idle. These do not really matter since the actual data sent and received is the same. Sending the rest of the packets using the different USB classes, is only a matter of coding. The successful communication proves that there is no encrypted handshake that is blocking a file transfer from a malicious host. Even if there were, using a disassembler to inject malicious code inside the bulk transfer could be done. The only reason this was not fully completed is a lack of time, the USB library used to develop this communication is extremely work intensive and the documentation extremely poor. It is however beyond any reasonable doubt, possible. The last steps would also take too long, they would require a separate compilation of U-boot and intimate knowledge on how the flashing process works. Compiling U-boot is extremely complicated and was therefore dropped at first. However, during the later stages another entryway was found, this one without the driver. This caused a revisit to the issue of compiling U-boot.

As mentioned, there exists a different possible entryway that does not require a driver. When opening the easy installers system files and then the recovery folder,

a file with the name "UUU" can be found. The UUU files were first assumed to be an executable named strangely to obstruct hijacking the installer. Especially since Toradex recommends running the installer with a .bat file and the file itself only containing a command that runs the UUU.exe. The UUU turned out to be a tool called **Universal Update Utility**. The tool is the very script that the easy installer uses to flash eMMC memory with the OS and bootloader. The system files also contain the U-boot binary labeled clearly. This opens up many possibilities with which to hack the system.

The first approach is to compile a separate U-boot and replace the U-boot binary inside the installer folder. This was immediately tried but not completely finished. Since this was discovered so late, setting up the toolchain and files needed to compile took too much time. It is still very much a possible way in, in fact, Toradex offers a guide on how to replace the current U-boot using the easy installer. In order to replace the binary the file can be replaced but, there is also multiple steps in order to prepare the raw binary so it can be flashed using the UUU tool. For example it will need to be converted into a image file. Using a different compile for the U-boot also opens up possibilities to compile U-boot scripts either directly into the U-boot code or loading it as a binary into the U-boot shell. The U-boot repository has examples exactly for this, for this project a hello world file was compiled and transferred into to RAM using the bootloader. This proved semi-successful, the program will run only if the correct assembler command is found inside the binary. It will also not print the string since it requires the library "stdio.h", found in the toolchain. This will be discussed more in the conclusion chapter.

### 4.3 Summary

The two chosen ports also came up with interesting results. Summarized the recovery port showed the largest potential for hacking due to its easy installer. The test successfully mimicked the transaction and could potentially inject harmful code inside the Uboot. It would also be possible to just flash a separately compiled Uboot that contains more harmful commands inside its command line. The mimicked transaction would be make the code injection likelier to remain hidden, since it uses the recommended Toradex version. Compiling Uboot would also work but the versions signature could be checked and the code removed. The debugging ports potential lies within the success of a glitching attempt. If the bootloader panics, access to its command line would be granted. The debugging port would be able to dump RAM, run malicious code bits and read the content of some applications that were running on reset.



# 5

## Discussion & Conclusion

This chapter will discuss the results of the investigation and then conclude based on that. The first section will discuss the exploits found that are general to all hardware and also the ports of interest. There will also be some theorizing on how these exploits can be avoided (if possible). Lastly there will be a conclusion of the overarching questions found in section 1.2.

### 5.1 Discussion

The security state of the Toradex platform will heavily depend on the separately designed platform. The easiest exploits found can be mitigated by simply deactivating ports and setting passwords. As mentioned in introduction etc, the board all tests were performed on was a development board. This board contains all available I/O for this specific SoC. A finished product will most likely not contain as many ports as this board. This excludes the two chosen ports however, these are required in order to setup the system and debug. As for the more advanced threats such as firmware extracting, glitching and USB fuzzing, there is not much that can be done. For example, even if the pins to the flash memory is hidden using ball pins, the chip could still be removed completely and then all information dumped. The glitching issue is also incredibly hard to solve since the bootloader opens shell for a very good reason. If the autoboot remained deactivated and only showed an error dump on boot up failure, repairing the system might be impossible. It would be impossible to access any system files, memory or bootloader variables. Barring removing flash or RAM memory completely of course.

### 5.1.1 Debugging port

Exploit	Possible	Can be mitigated	Solution (if there is one)
Port sniffing	Yes	No	-
Flash bootloader (if glitched)	Yes	Somewhat	Disable port
Memory dumping (if glitched)	Yes	Somewhat	Disable port
Bootloader scripts (if glitched)	Yes	Somewhat	Disable port

**Table 5.1:** Exploits tested for the debug port

The exploits that were tried are displayed in table 5.1. The table summarizes the possibilities of attempting to access the port using the exploit and if it can be stopped. Following the evaluation template in 3.1.2, the following questions are answered:

1. **What is the ports intended purpose and how does it work?** The ports purpose is to access the linux and U-boot command line.
2. **Can I control its input/output?** The ports uses UART to communicate and can be communicated with freely using any serial client.
3. **How can I exploit its original purpose?** The port can be forced to shell if glitched. This will give access to the bootloader which allows for code injection, memory dumping etc.
4. **What hardware does the port have access to and can it be sniffed?** In essence, the port will have access to everything, assuming shell with root access or using the bootloader. Everything sent through the port can be sniffed either directly on the UART socket or on the SoCs DDR4 socket.

As highlighted by the table 5.1 and proven in section 4.2, the port will always be able to be sniffed. There are even hardware sniffers that can be plugged directly onto the wire without the host or device knowing. The port uses simple UART so encryption isn't really possible. The rest of the exploits assumes that the bootloader shell was accessed by glitching the CPU, flash or RAM. It is still unclear how effective glitching would be on this platform. The risk of breaking the platform made it impossible to try the method. As seen in subsection 4.1.2, flashing the bootloader would be simple from the bootloader shell, it would just need a new Uboot image easily downloaded by a USB flashdrive. Memory dumping was done extensively (again in subsection 4.1.2) and can be seen containing information from the container code. Even if the container image could not be changed, access to parts of the container variable values and names is quite distressing. Especially if they were credit card numbers or passwords. Bootloader scripts would be able to do anything within the processing capabilities of the bootloader environment. It could download all of the user space as a binary blob for example.

The largest security issue with this port is that even if the port is password protected,

software deactivated and hardware deactivated, some way of activating it again will most likely be needed. Without the port, no debugging will be possible on system failure. Therefore the hacker would probably be able to find any activation sequence on the hardware and then glitch through the command line password. Any security measure on this port should therefore be seen as a deterrent more then anything.

### 5.1.2 Recovery port

Exploit	Possible	Can be mitigated	Solution (if there is one)
Port sniffing	Yes	No	-
Flash bootloader	Yes	Yes	Encrypt transaction
Bootloader code injection	Yes	Yes	Software signature

**Table 5.2:** Exploits tested for recovery the port

When answering the question it became increasingly clear that the port itself is barely protected at all. As can be seen in table 5.2, all tried exploits would work and can only be patched by trading functionality. The answers to the questions from the section 3.1.1 are as follows:

1. **What is the ports intended purpose and how does it work?** The ports purpose is to allow the host to recover the device by flashing a new OS or bootloader. This is done by using the easy installer.
2. **Can I control its input/output?** As seen in subsection 4.2, communicating with the hardware by mimicking the communication done by the easy installer is possible.
3. **How can I exploit its original purpose?** The communication could be mimicked down to the very bulk transfer that contains the U-boot and then simply injected with a backdoor. Only flashing the U-boot would allow the hacker to keep the data on the flash and then dump it as a bulk transfer.
4. **What hardware does the port have access to and can it be sniffed?** The port has access to the CPU, RAM and flash memory. As mentioned in the section 4.2, sniffing the port can be done using either the DDR4 socket or Wireshark. Sniffing can essentially not be stopped which means that injecting malicious binary into the bit stream will always be possible.

Port sniffing will always be possible on this port similar to the debugging port. There might be room for encrypting the system files transfer however. Since the port uses a personal driver through the UUU tool, some form of encryption could deter sniffing. This would require extra hardware though since the tool is downloading the bootloader straight down to its RAM and the CPU is not yet setup. Flasing the bootloader or injecting code to a existing booloader will always be possible as long as the port is activated. There was an attempt to prove this further by flashing the U-boot with a personalized version. Even if there was no time to finish this

process, the possibility is guaranteed. The binary was compiled and loaded into RAM. The only issue that could not be fixed in time was removing the header that the binary has, so that the "go" command could execute directly from the assembler commands. The issue is within the compilation process and requires some intimate knowledge on how to cross compile for U-boot. This could obviously be fixed given enough time and is therefore a very strong way into the system. Disabling the port is therefore the only option, which would trade away the ability to recover the device if the bootloader corrupts.

There was an attempt by Toradex to obstruct this by using the easy installer as an application layered driver and hiding the port as a HID. This is only a very basic deterrent however. Since by simply using a different driver that can be controlled by the hacker, this attempt is completely thwarted. There is no verification process to speak of, neither is there a password.

### 5.2 Conclusion of investigation

It has become increasingly clear that the forensic toolbox, available to all, has become more and more advanced. There exists USB sniffers, bus pirates and binary analysis tools, either completely for free or for incredibly cheap prices. The security of hardware is therefore increasingly impossible to safe-guard. In fact, the only way to fully guarantee tamper-proof hardware is to secure it on site. That means camera surveillance, guarding it and locking it up. This theme can be hinted at throughout the discussion of the ports and general hardware hacks. There is always a way in as long as the hardware can be accessed and there is enough time. Therefore the security of hardware is a surveillance issue more than anything.

Referring back to the questions in section 1.2. The most common hardware hacks are always the most obvious exploits. If the product has a debugging port it will be targeted first since it has the most obvious access. There is also sniffing, which would probably be done on-site. Whereas the platform might be protected from wifi sniffing, simply plugging in a sniffer onto the chip would render it useless. As mentioned this follows the trend that hardware hacks cannot be mitigated but they can be deterred.

When investigating the platform it became clear that some CS consideration existed for the hardware. Whilst for example, the easy installer is a clear security breach that anyone with some embedded understanding could exploit, the actual SoC uses hidden conductors and ball point pins. Likely, Toradex faced the issue of obstructing developers if they were to do more. Therefore any higher consideration for CS would have to come from the developer and the finished product.

The overall weaknesses found were as seen below:

The first items dealing with the ports are already discussed in their separate sections.



Weaknesses	Security risk	Exposed data
Debugging port	High	Coredump (Full exposure if glitching succeeds)
Recovery port	High	Full exposure
DDR4 socket	Medium	All I/O
Easy installer	High	Full exposure

**Table 5.3:** Security breach severity, low, medium, high

But the DDR4 socket poses a real security issue for the hardware. Since the socket has exposed pins, any information going from any I/O into the SoC, will be exposed. Which means that basically any information sent to the platform can be sniffed. It is unclear what can be done about this, even if the DDR4 socket were to hide its pins, a secondary socket could still be place in between allowing for sniffing anyways. It is possible to hide the pins and then solder the SoC onto the card of course, but the obvious problem would then be that it is attached to the card no matter what. Replacing the SoC if it were to break would be much harder and the cost for soldering it would be more expensive. As mentioned, the bootloader problem assumes that glitching is achieved. Doing so is always a bit risky since the platform might just break instead. There is not much that can be done here as well sadly. The easy installer could use some more work though. The installer is essentially just the UUU tool (see 4.2) with some settings and then a image file and the U-boot binary. Perhaps it would be better if a compiled executable was created instead. It could then in turn check the integrity of its own system files before continuing. This could of course be disassembled and hacked that way, but it would be much more difficult and therefore a good deterrent. As it stands now, the U-boot binary could just be overwritten and then installed the normal way.



# 6

## Future Work

There are several areas where the project will be able to continue. These are as an example and in no particular order as seen below:

1. Investigate Glitching vulnerability
2. Investigate all remaining I/O
3. Research if there is an other solution to the man in the middle problem
4. Research if there is a way to detect a malicious USB device

As mentioned in the discussion, the glitching vulnerability was never tested. This is due to it potentially breaking the platform. These particular types of SoCs share similar designs, if the vulnerability is true for one it is very possible that all of them are exposed. For the second item, many of the remaining I/O were never investigated. There is for example a very similar port to the two researched here that was never touched, namely the JTAG port. The second to last item is the man in the middle issue. This particular problem has existed for a while and encryption is not always possible when dealing with hardware. The last item is a problem that many hackers exploit daily. The USB protocol is inherently designed to trust the user, this allows many exploits that allows the hacker to crash the system or gain access to the computer unbeknownst to the user.



# Bibliography

- [1] S. Shea and I. Wigmore, “What is iot security? - definition from techtarget.com,” Mar 2022. [Online]. Available: <https://internetofthingsagenda.techtarget.com/definition/IoT-security-Internet-of-Things-security>
- [2] C. Peacock, “Usb in a nutshell,” Apr 2018. [Online]. Available: <https://www.beyondlogic.org/usbnutshell/usb1.shtml>
- [3] J. Corbet, A. Rubini, and G. Kroah-Hartman, *USB Drivers*, 3rd ed. O’Reilly, 2005, p. 327–361.
- [4] Wikipedia, “Usb,” Mar 2022. [Online]. Available: <https://en.wikipedia.org/wiki/USB>
- [5] C. Pinkle, “The why and how of differential signaling - technical articles,” Nov 2016. [Online]. Available: <https://www.allaboutcircuits.com/technical-articles/the-why-and-how-of-differential-signaling/>
- [6] Docker, “What is a container?” Apr 2022. [Online]. Available: <https://www.docker.com/resources/what-container/>
- [7] F. Team, *Hacker’s Guide to UART Root Shells*, 2021. [Online]. Available: <https://www.youtube.com/watch?v=01mw0oTHwxg>
- [8] Toradex, “Verdin imx8m mini datasheet v1.1,” Jun 2021. [Online]. Available: <https://docs.toradex.com/108681-verdin-imx8m-mini-datasheet-v1.1>
- [9] M. Burgess, “What is the internet of things? wired explains,” Feb 2018. [Online]. Available: <https://www.wired.co.uk/article/internet-of-things-what-is-explained-iot>
- [10] Oracle, “What is the internet of things (iot)?” Mar 2022. [Online]. Available: <https://www.oracle.com/se/internet-of-things/what-is-iot/>
- [11] Wikipedia, “Computer security,” Mar 2022. [Online]. Available: [https://en.wikipedia.org/wiki/Computer\\_security](https://en.wikipedia.org/wiki/Computer_security)
- [12] M. Barr and A. Massa, *Programming embedded systems: with C and GNU*

- development tools*. " O'Reilly Media, Inc.", 2006.
- [13] K. Ranjan, "Cloud-enabled ai demonstration - toradex: Amazon web services: Nxp," Oct 2019. [Online]. Available: <https://www.toradex.com/videos/cloud-enabled-ai-demonstration>
  - [14] A. Mhatre, "Industrial applications by gusto controls featuring custom carrier boards for toradex soms," May 2019. [Online]. Available: <https://www.toradex.com/videos/gusto-controls-industrial-applications-toradex-modules>
  - [15] M. binti Mohamad Noor and W. H. Hassan, "Current research on internet of things (iot) security: A survey," *Computer Networks*, vol. 148, pp. 283–294, 2019. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1389128618307035>
  - [16] V. Hassija, V. Chamola, V. Saxena, D. Jain, P. Goyal, and B. Sikdar, "A survey on iot security: Application areas, security threats, and solution architectures," *IEEE Access*, vol. 7, pp. 82 721–82 743, 2019.
  - [17] S. Sidhu, B. J. Mohd, and T. Hayajneh, "Hardware security in iot devices with emphasis on hardware trojans," *Journal of Sensor and Actuator Networks*, vol. 8, no. 3, 2019. [Online]. Available: <https://www.mdpi.com/2224-2708/8/3/42>
  - [18] J. Dofe, J. Frey, and Q. Yu, "Hardware security assurance in emerging iot applications," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 2050–2053.
  - [19] A. Mosenia and N. K. Jha, "A comprehensive study of security of internet-of-things," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 4, pp. 586–602, 2017.
  - [20] D. Both, "An introduction to the linux boot and startup processes," Feb 2017. [Online]. Available: <https://opensource.com/article/17/2/linux-boot-and-startup>
  - [21] A. Holt and C.-Y. Huang, *Overview of GNU/Linux*. Springer International Publishing, 2018, p. 11–25.
  - [22] G. Cloud, 2022. [Online]. Available: <https://cloud.google.com/learn/what-are-containers>
  - [23] M. DuHarte, *DEF CON 23 - Hardware Hacking Village - Matt DuHarte - Introduction to USB and Fuzzing*, 2015. [Online]. Available: <https://www.youtube.com/watch?v=KWOTXypBt4E>
  - [24] —, *DEF CON 24 - Hardware Hacking Village - Matt DuHarte - Basic Firmware Extraction*, 2018. [Online]. Available: <https://www.youtube.com/>

watch?v=Kxvpbu9STU4

- [25] F. Team, *How We Hacked a TP-Link Router and Took Home \$55,000 in Pwn2Own*, 2021. [Online]. Available: <https://www.youtube.com/watch?v=zjafMP7EgEA>
- [26] S. Mendoza, *NFC Payments The Art of Relay & Replay*, 2018. [Online]. Available: <https://www.youtube.com/watch?v=MVU3gbPnk0g>
- [27] T. Rigas, “Iot hacking field notes #1: Intro to glitching attacks,” 2020. [Online]. Available: <https://blog.nviso.eu/2020/02/21/iot-hacking-field-notes-1-intro-to-glitching-attacks/>
- [28] Toradex, “Verdin development board design data v1.1,” 2022. [Online]. Available: <https://developer.toradex.com/hardware/verdin-som-family/carrier-boards/verdin-development-board/#designresources>
- [29] “Mimx8mm6cvtkzaa product information,” 2022. [Online]. Available: <https://www.nxp.com/part/MIMX8MM6CVTKZAA#/>





# A

## Available U-boot commands

? - alias for 'help'  
askenv - get environment variables from stdin  
base - print or set address offset  
bdinfo - print Board Info structure  
blkcache - block cache diagnostics and control  
boot - boot default, i.e., run 'bootcmd'  
bootaux - Start auxiliary core  
bootd - boot default, i.e., run 'bootcmd'  
bootefi - Boots an EFI payload from memory  
bootelf - Boot from an ELF image in memory  
booti - boot Linux kernel 'Image' format from memory  
bootm - boot application image from memory  
bootp - boot image via network using BOOTP/TFTP protocol  
bootvx - Boot vxWorks from an ELF image  
cftgblock - Toradex config block handling commands  
clk - CLK sub-system  
clocks - display clocks  
cmp - memory compare  
coninfo - print console devices and information  
cp - memory copy  
dcache - enable or disable data cache  
dhcp - boot image via network using DHCP/TFTP protocol  
dm - Driver model low level access  
echo - echo args to console  
editenv - edit environment variable  
env - environment handling commands  
exit - exit script  
ext2load - load binary file from a Ext2 filesystem  
ext2ls - list files in a directory (default /)  
ext4load - load binary file from a Ext4 filesystem  
ext4ls - list files in a directory (default /)  
ext4size - determine a file's size  
ext4write - create a file in the root directory  
false - do nothing, unsuccessfully

## A. Available U-boot commands

---

fastboot - run as a fastboot usb or udp device  
fatinfo - print information about filesystem  
fatload - load binary file from a dos filesystem  
fatls - list files in a directory (default /)  
fatmkdir - create a directory  
fatrm - delete a file  
fatsize - determine a file's size  
fatwrite - write file into a dos filesystem  
fdt - flattened device tree utility commands  
fstype - Look up a filesystem type  
fuse - Fuse sub-system  
go - start application at address 'addr'  
gpio - query and control gpio pins  
gpt - GUID Partition Table  
gzwrite - unzip and write memory to block device  
help - print command description/usage  
i2c - I2C sub-system  
icache - enable or disable instruction cache  
iminfo - print header information for application image  
imxtract - extract a part of a multi-image  
itest - return true/false on integer compare  
lcdputs - print string on video framebuffer  
ln - Create a symbolic link  
load - load binary file from a filesystem  
loadb - load binary file over serial line (kermit mode)  
loads - load S-Record file over serial line  
loadx - load binary file over serial line (xmodem mode)  
loady - load binary file over serial line (ymodem mode)  
loop - infinite loop on address range  
ls - list files in a directory (default /)  
lzmdec - lzma uncompress a memory region  
md - memory display  
mdio - MDIO utility commands  
mii - MII utility commands  
mm - memory modify (auto-incrementing address) mmc - MMC sub system  
mmcinfo - display MMC info  
mtest - simple RAM read/write test  
mw - memory write (fill)  
nfs - boot image via network using NFS protocol  
nm - memory modify (constant address)  
part - disk partition related commands  
ping - send ICMP ECHO\_REQUEST to network host  
pinmux - show pin-controller muxing  
printenv - print environment variables  
pxe - commands to get and boot from pxe files  
random - fill memory with random pattern

regulator - uclass operations  
reset - Perform RESET of the CPU  
run - run commands in an environment variable  
save - save file to a filesystem  
saveenv - save environment variables to persistent storage  
setcurs - set cursor position within screen  
setenv - set environment variables  
setexpr - set environment variable as the result of eval expression  
showvar - print local hushshell variables  
size - determine a file's size  
sleep - delay execution for some time  
source - run script from memory  
sysboot - command to get and boot from syslinux files  
test - minimal test like /bin/sh  
tftpboot - boot image via network using TFTP protocol  
true - do nothing, successfully  
ums - Use the UMS [USB Mass Storage]  
unzip - unzip a memory region  
usb - USB sub-system  
usbboot - boot from USB device  
version - print monitor, compiler and linker version  
videolink - list and select video link



# B

## Boot dump

```
/dev/ttyUSB3 - PuTTY

U-Boot SPL 2020.04-5.5.0+git.81bc8894031d (Jan 01 1970 - 00:00:00 +0000)
DDRINFO: start DRAM init
DDRINFO: DRAM rate 3000MTS
DDRINFO:ddrphy calibration done
DDRINFO: ddrmix config done
Normal Boot
Trying to boot from MMC1
NOTICE: BL31: v2.2(release):toradex_imx_5.4.70_2.3.0-g835a8f67b2
NOTICE: BL31: Built : 00:00:00, Jan 1 1970

U-Boot 2020.04-5.5.0+git.81bc8894031d (Jan 01 1970 - 00:00:00 +0000)

CPU:   i.MX8MMQ rev1.0 1600 MHz (running at 1200 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 54C
Reset cause: POR
DRAM:   2 GiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
Loading Environment from MMC... OK
Fail to setup video link
In:     serial
Out:    serial
Err:    serial
Model: Toradex Verdin iMX8M Mini Quad 2GB Wi-Fi / BT IT V1.1B, Serial# 06895056
Carrier: Toradex Verdin Development Board V1.1A, Serial# 10795050

BuildInfo:
- ATF 835a8f6
- U-Boot 2020.04-5.5.0+git.81bc8894031d

Flash target is MMC:0
Net:    eth0: ethernet@30be0000
Fastboot: Normal
Normal Boot
Hit any key to stop autoboot: 0
switch to partitions #0, OK
mmc0(part 0) is current device
Scanning mmc 0:1...
Found U-Boot script /boot.scr
1182 bytes read in 13 ms (87.9 KiB/s)
## Executing script at 46000000
5376 bytes read in 22 ms (238.3 KiB/s)
64786 bytes read in 29 ms (2.1 MiB/s)
47 bytes read in 21 ms (2 KiB/s)
Applying Overlay: verdin-imx8mm_lt8912_overlay.dtbo
1620 bytes read in 31 ms (50.8 KiB/s)
12177067 bytes read in 288 ms (40.3 MiB/s)
Uncompressed size: 30593536 = 0x1020200
9176227 bytes read in 227 ms (39.6 MiB/s)
## Flattened Device Tree blob at 43000000
   Booting using the fdt blob at 0x43000000
   Loading Device Tree to 00000000bd5d9000, end 00000000bd70bfff ... OK

Starting kernel ...

[ 0.079078] No EMan portals available!
[ 0.079640] No QMan portals available!
[ 1.217903] imx_sec_dsim_drv 32e10000.mipi_dsi: Failed to attach bridge: 32e10000.mipi_dsi
[ 1.226211] imx_sec_dsim_drv 32e10000.mipi_dsi: failed to bind sec dsim bridge: -517, retry 0
[ 1.986149] rtc-ds1307 0-0032: hctosys: unable to read the hardware clock
[ 2.772448] imx6q-pcie 33800000.pcie: failed to initialize host
[ 2.778381] imx6q-pcie 33800000.pcie: unable to add pcie port.
Starting version 244.5+
[ 6.169947] debugfs: Directory '30020000.sai' with parent 'imx8mm-nau8822' already present!

TorizonCore 5.5.0+build.11 verdin-imx8mm-06895056 ttyMXC0
verdin-imx8mm-06895056 login: █
```



# C

## Code used in project

```
1 //////////////////////////////////////////////////
2 // Container used for test //
3 //////////////////////////////////////////////////
4 #include <lib.h>
5 #include <stdio.h>
6 #include <malloc.h>
7 #include <unistd.h>
8 static int var = 77777; //Value to be printed and identified
9 static char mystr[] = "general kenobi\n"; //String to be printed
10 static int *ip; //Integer pointer to the value
11
12 int main(int argc, char const *argv[])
13 {
14     ip = &var;
15     void *p=malloc(1); // Trying to move outside of process memory
16     // space
17     while(1){
18         for(int i = 50; i>0;i--){ //Print address: symbol, prints the
19             // string backwards
20             int * address = (int *)&mystr-i;
21             printf("%x: %x\n",&mystr-i, *address);
22         }
23         printf("string: %s\n", mystr); //Print address: symbol, prints
24         // the string forwards
25         for(int i = 0; i<50;i++){
26             int * address = (int *)&mystr+i;
27             printf("%x: %x\n",&mystr+i, *address);
28         }
29         printf("int pointer %x \n", ip);
30         printf("pointer: %x \n", &mystr);
31         printf("malloc %p\n",p);
32         sleep(10); //Sleep for 10s then print again
33     }
34 }
```

C.1 Container code

```
1 #include <stdio.h>
2 #include "libusb.h"
3 #include <unistd.h>
4 #include <time.h>
5
6
7 #define PRODUCT 0x0134 //Toradex product id
8 #define VENDOR 0x1fc9 //Toradex vendor id
9 #define TIMEOUT 10 //Time until timeout in seconds
10
11 // HID Class-Specific Requests values. See section 7.2 of the HID
12 // specifications
13 #define HID_GET_REPORT 0x01
14 #define HID_GET_IDLE 0x02
15 #define HID_GET_PROTOCOL 0x03
16 #define HID_SET_REPORT 0x09
17 #define HID_SET_IDLE 0x0A
18 #define HID_SET_PROTOCOL 0x0B
19 #define HID_REPORT_TYPE_INPUT 0x01
20 #define HID_REPORT_TYPE_OUTPUT 0x02
21 #define HID_REPORT_TYPE_FEATURE 0x03
22
23 //
24 ///////////////////////////////////////////////////////////////////
25
26 ///////////////////////////////////////////////////////////////////DEBUG SETTINGS
27 ///////////////////////////////////////////////////////////////////
28 #define debug 0 //Activate debug settings
29
30 #if debug
31 #define USBSKIP 1 // Activate or deactivate USB code
32 #define mouse 1 //Activate mouse test
33 #define PRODUCT 0X6011
34 #define VENDOR 0x0403
35 #if mouse //debug mode test with mouse
36 #define PRODUCT 0xC534
37 #define VENDOR 0x046D
38 #endif
39
40 //etc..
41 #endif
42 //
43 ///////////////////////////////////////////////////////////////////
44
45 //
46 ///////////////////////////////////////////////////////////////////
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```



```

49     clock_t start_time = clock();
50
51     // looping till required time is not achieved
52     while (clock() < start_time + milli_seconds);
53 }
54
55 unsigned char char_2byte(char arr[]){ //Converts char array to their
    hex i.e '0','f' to 0xf
56     unsigned char outChar;
57     if (arr[0] >= '0' && arr[0] <= '9'){
58         outChar = (arr[0] - '0') * 16;
59     }
60     else {
61         outChar = (arr[0] - 'a' + 10) * 16;
62     }
63     if (arr[1] >= '0' && arr[1] <= '9') {
64         outChar += (arr[1] - '0');
65     }
66     else {
67         outChar += (arr[1] - 'a' + 10);
68     }
69     return outChar;
70 }
71
72 int port_open (libusb_device* dev, libusb_device_handle **dev_handle){
    //Homebrew func. Retries after failed libusb_open
73     int count = 0;
74     int err = 0;
75     printf("Attempting to open port...\n");
76     while (1){
77         printf("...\n");
78         err = libusb_open(dev, dev_handle); //Attempts to open port
79         if (err == 0){
80             break;
81         }
82         if(count>TIMEOUT){ //5s without the port opening and it will
            throw error
83             printf(libusb_error_name(err));
84             return 1;
85         }
86         count++;
87         delay(1);
88     }
89     return 0;
90 }
91
92
93 static int target_find(libusb_device *dev,uint16_t vendor_id,uint16_t
    product_id){ //Check for device. Libusb already had this but it was
    done before I saw it.
94     struct libusb_device_descriptor my_dev;
95     libusb_get_device_descriptor(dev,&my_dev);
96     if ((my_dev.idVendor == vendor_id) && (my_dev.idProduct == product_id
97         )){
98         return 1;
99     }

```

```
99     return 0;
100 }
101
102 int fragment_reader(FILE* ptr, unsigned char *data){ //Read data
103     fragment (Sniffed from Wireshark)
104     unsigned char ch[2] = {0};
105
106     if (NULL == ptr) {
107         printf("USB DATA ERROR: File can't be opened \n");
108         return 1;
109     }
110
111     else {
112         int index = 0;
113         while (!feof(ptr)){ //End of data fragment
114             ch[0] = fgetc(ptr);
115             if ((ch[0] == '\n' || ch[0] == '\r')){
116                 break;
117             }
118
119             else{
120                 ch[1] = fgetc(ptr);
121                 data[index] = char_2byte(ch); //Save one byte per index
122             }
123
124             index++;
125             if(debug){
126                 printf("%c%c", ch[0], ch[1]);
127             }
128         }
129         if (debug){
130             printf("\n");
131             for (int i = 0; i<index;i++){
132                 printf("%x", data[i]);
133             }
134             printf("\n");
135         }
136
137         return index;
138     }
139 }
140 }
141
142 int device_opener(struct libusb_device_handle *handle, struct
143     libusb_device_descriptor desc, struct libusb_config_descriptor *
144     config, unsigned char devname[2][256], uint16_t vendor_id, uint16_t
145     product_id){
146     libusb_device **list;
147     libusb_device *found = NULL;
148     while(1){ //Finds device and opens port
149         ssize_t cnt = libusb_get_device_list(NULL, &list);
150         ssize_t i = 0;
151         if (cnt < 0){
152             printf("USB ERROR: Could not fetch device list\n");
153             return 1;
154         }
155     }
```

```

151     }
152     for (i = 0; i < cnt; i++) {
153         libusb_device *device = list[i];
154         if (target_find(device, vendor_id, product_id)) {
155             found = device;
156             break;
157         }
158     }
159
160     if (found) {
161         printf("Port found!\n");
162         if (libusb_get_device_descriptor(found, &desc)){printf("USB ERROR
: Unable to fetch device descriptor\n"); return 1;}
163         if (libusb_get_config_descriptor(found,0,&config)){printf("USB
ERROR: Unable to fetch config descriptor\n"); return 1;}
164         if (port_open(found, &handle)){ printf("\n Timeout ERROR: Could
not open port\n"); return -1; }
165         libusb_get_string_descriptor_ascii(handle,1,devname[0],255);
166         libusb_get_string_descriptor_ascii(handle,2,devname[1],255);
167         printf("Device: \"%s\" \"%s\" Opened!\n",devname[0], devname[1]);
168         break;
169     }
170
171     libusb_free_device_list(list, 1); //Free device list so I can
refresh. Unclear if this is needed though.
172 }
173 return 0;
174 }
175 //Supposed to open the device with selected ID
176
177 int main(void)
178 {
179
180     uint16_t vendor[2] = {0x1fc9,0x0525};
181     uint16_t product[2] = {0x0134,0xb4a4};
182     #ifdef USBSKIP
183         if(USBSKIP){
184             goto usb_skip;
185         }
186     #endif
187
188     int err = 0;
189     err = libusb_init(NULL); //libusb setup
190     if (err != 0) {
191         printf("LIBUSB ERROR: Could not initialize\n");
192
193         return 1;
194     }
195
196     //Define libusb types
197     unsigned char devname[2][256] = {0};
198     unsigned char buf [140] = {0};
199     struct libusb_device_descriptor desc;
200     struct libusb_config_descriptor *config;
201     struct libusb_device_handle *handle;

```

```
201 unsigned char data[2048] = {0};
202
203
204 err = device_opener(handle, desc, config, devname, vendor[0], product[0]);
205 if(libusb_set_configuration(handle, 1)){printf("USB ERROR: Could not
    set configuration\n"); return 1;} //Set config 1 for device, should
    contain 1 interface with two endpoints. I think.
206 if(libusb_claim_interface(handle, 0)){printf("USB ERROR: Could not
    claim interface\n"); return 1;}
207
208 //
    ///////////////////////////////////////////////////////////////////
209 usb_skip:///Skips the usb interface completely. ONLY FOR DEBUGGING
    !/////////////////////////////////////////////////////////////////
210
    //
    ///////////////////////////////////////////////////////////////////
211 printf("Sending traffic..\n");//Send set_reports
212 FILE* ptr;
213 ptr = fopen("C:/Users/hampus.martinsson/Min enhet/Exjobb - Master/
    Recovery search/data.txt", "r");
214 if (NULL == ptr) {
215     printf("USB DATA ERROR: File can't be opened \n");
216 }
217
218
219 int numb_packet = 0;
220 while (!feof(ptr)){
221     int wlen = fragment_reader(ptr, data);
222     //err = libusb_interrupt_transfer(handle, config->interface->
    altsetting->endpoint->bEndpointAddress, data, wlen, &wlen, 1000);
223     //Interrupt transfer seems to be incorrect
224
225     //Set_Report Packets
226     if(numb_packet == 0){
227         err = libusb_control_transfer(handle, 0x21, HID_SET_IDLE, 0, 0, NULL
    , 0, 0);
228         /*
229         err = libusb_control_transfer(handle, LIBUSB_ENDPOINT_IN |
    //GET_DESCRIPTOR HID Report, not needed though
230         LIBUSB_RECIPIENT_INTERFACE,
231         LIBUSB_REQUEST_GET_DESCRIPTOR, (LIBUSB_DT_REPORT << 8) | 0,
    0, buf,
232         sizeof(buf), 1000);*/
233         err = libusb_control_transfer(handle, 0x21, HID_SET_REPORT, 0x201, 0,
    data, wlen, 0); // Settings stolen from wireshark packets sent here
234         //status = libusb_control_transfer(handle, LIBUSB_ENDPOINT_IN |
    LIBUSB_REQUEST_TYPE_VENDOR | LIBUSB_RECIPIENT_DEVICE, 0x09, addr & 0
    xFFFF, addr >> 16, (unsigned char*)data, (uint16_t)len, 1000);
235         numb_packet++;
236     }
237     else {
238         err = libusb_control_transfer(handle, 0x21, HID_SET_REPORT, 0x202, 0,
    data, wlen, 0); // Settings stolen from wireshark packets
```

```
239     numb_packet++;
240 }
241
242 if (err < 0){
243     printf(libusb_error_name(err));
244     printf("\nINTERRUPT TRANSFER ERROR: Could not send packets\n");
245     return 1;}
246 }
247 libusb_close(handle);
248 fclose(ptr);
249
250 err = device_opener(handle, desc, config, devname, vendor[1], product[1])
    ;
251 libusb_exit(NULL);
252 return 0;
253 }
```

## C.2 Tailored USB Driver

```
1 from cgitb import reset
2 import serial
3 import sys, getopt
4 import os
5 import time
6
7 notice = "curl https://notify.run/YIbIqVmnzmKmcR3iNsm1 -d" #Adress to
      notification API, notify in this case
8 DEBUG = False
9 dump = []
10 START = "50000000";
11 SIZE = "0xffffffff";
12
13
14 def pat_search(_list, pattern): #Search for pattern in list
15     _str = ""
16
17     for val in _list:
18         tmp=val.rstrip("\r\n")
19         tmp=tmp.split(" ")
20         _str=_str+ " ".join(tmp[8:])
21     #Below are strings to look for
22     if pattern.lower() in _str.lower(): #Takes input arguments and looks
        for match
23         print("Match found!")
24         return True
25     #Below are used if more strings are searched at once
26     if "hello world" in _str.lower():
27         print("Match found!")
28         return True
29     if "helloworld" in _str.lower():
30         print("Match found!")
31         return True
32     #Add more depending on what to search for.
33     else:
34         return False
35
36 def main(argv):
37     global dump
38
39     if DEBUG == False: #Looking for a real match
40         try:
41             opts, args = getopt.getopt(argv[1:], "hi:s:", ["input=", "search
42             ="]) #Black magic, uses input args
43             except getopt.GetoptError:
44                 print ('USB_sniffer.py -i <inputport> -s <search pattern>')
45                 sys.exit(2)
46             for opt, arg in opts:
47                 if opt == '-h':
48                     print ('USB_sniffer.py -i <inputport> -s <search pattern>')
49                     sys.exit()
50                 elif opt in ("-i", "--input"):
51                     input = arg
52                 elif opt in ("-s", "--search"):
53                     _search = arg
54                     if args:
```

```

54     _search= _search+" "+ ".join(args)
55     print ("Starting search for '" + _search + "' in " + input)
56
57     if DEBUG == True: #Simple debug tool, checks a normal text file for
58         a known value
59         input = "COM6"
60         _search = "nasjjsan"
61
62     ###USB begins here###
63
64     serialPort = serial.Serial(port=input, baudrate=115200, bytesize=8,
65         timeout=2, stopbits=serial.STOPBITS_ONE)
66     serialString = "" # Used to hold data coming over UART
67     counter = 0
68     timeout_start = time.time()
69     serialPort.reset_input_buffer()
70
71     serialPort.write(("md " + START + " " + SIZE + "\r").encode('utf-8'))
72     #Initiate mem dump from START to START+SIZE
73
74     while 1:
75         try:
76             timeout = time.time()
77             if timeout-timeout_start>5: #If input is idle for 5s close the
78                 program and notify
79                 tmp_str = ''
80                 if pat_search(dump,_search):
81                     log = open("sniff_log.txt","a") #Append text to file
82                     log.write("Match found!\n" + ''.join(dump))
83                     log.close()
84                     os.system("cmd /c " + notice + ' "Match found!"') #
85                     Notify that a match has been found
86                     for val in dump:
87                         tmp_str=tmp_str + val
88                     log = open("sniff_log.txt","a") #Append text to file
89                     log.write("latest search: \n'+tmp_str + "'")
90                     log.close()
91                     serialPort.close()
92                     sys.exit()
93             # Wait until there is data waiting in the serial buffer
94             if serialPort.in_waiting > 0:
95                 timeout_start = time.time()
96
97                 # Read data out of the buffer until a carriage return /
98                 new line is found
99                 serialString = serialPort.readline()
100
101                 # Print the contents of the serial data
102                 try:
103                     _str = serialString.decode("Ascii")
104                     if len(_str) == 67:
105                         dump.append(_str)
106                         counter = counter+1
107                     else:
108                         log = open("sniff_log.txt","a") #Append text to file
109                         log.write("Buffer rest: '"+_str + "'\n")

```

```

104         log.close()
105         if counter > 2:
106             counter = 0
107             if pat_search(dump, _search):
108                 log = open("sniff_log.txt", "a") #Append text to
file
109                 log.write("Match found!\n" + '\n'.join(dump))
110                 log.close()
111                 os.system("cmd /c " + notice + ' "Match found!"')
#Notify that a match has been found
112                 del dump[0:-1]
113
114             except:
115                 pass
116         except KeyboardInterrupt:
117             tmp_str = ''
118             for val in dump:
119                 tmp_str=tmp_str + val + "\n"
120             log = open("sniff_log.txt", "a") #Append text to file
121             log.write("latest search: \n"+tmp_str + "\n")
122             log.close()
123             serialPort.close()
124
125
126
127
128 if __name__ == "__main__":
129
130     try:
131         main(sys.argv)
132
133     finally:
134         os.system("cmd /c " + notice + ' "Script shutting down"')

```

### C.3 Searches for target match in memory dump



```

1 //Standard standalone script, prints a simple hello world//
2
3
4 #include <common.h>
5 #include <exports.h>
6
7
8 /*
9 * add at least one variable. Without this at least in some cases the
10 * standalone application sometimes freezes, sometimes it prints 'random
11 * stuff with printf ...
12 * I assume that this forces alignment of some linker sections.
13 */
14 int dummy_var_in_text = 1;
15
16 #ifdef __thumb__
17 /*
18 * make thumb work by providing a forwarder to the (thumb) entry point
19 * compiled for arm instruction set. Note that while not needed this
20 * works for arm instruction set too.
21 */
22 void __attribute__((unused)) __attribute__((naked)) dummy2 (void)
23 {
24     asm volatile ( \
25         ".code 32\n" \
26         ".arm\n" \
27         "ldr pc,=hello_world\n" );
28 }
29 #endif
30
31
32 int hello_world (int argc, char * const argv[])
33 {
34
35     int i;
36     printf("Hello there");
37     // Print the ABI version
38     app_startup(argv);
39     printf ("Example expects ABI version %d\n", XF_VERSION);
40     printf ("Actual U-Boot ABI version %d\n", (int)get_version());
41
42     printf ("Hello World\n");
43
44     printf ("argc = %d\n", argc);
45
46     for (i=0; i<=argc; ++i) {
47         printf ("argv[%d] = \"%s\"\n",
48             i,
49             argv[i] ? argv[i] : "<NULL>");
50     }
51
52     printf ("Hit any key to exit ... ");
53     while (!tstc())
54         ;
55     // consume input

```

```
56  (void) getc();  
57  
58  printf ("\n\n");  
59  return (0);  
60 }
```

C.4 U-boot script that prints hello world

DEPARTMENT OF ELECTRICAL ENGINEERING  
CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden

[www.chalmers.se](http://www.chalmers.se)



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY