Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering

Master’s Thesis

Industrial graphical terminal based on Linux

Bc. Jiří Matěják

Supervisor: Ing. Michal Sojka, Ph.D.

Study Programme: Otevřená informatika

Field of Study: Počítačové inženýrství

25th May 2018

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Matěják**	Jméno: **Jiří**	Osobní číslo: **420215**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra řídicí techniky**		
Studijní program:	**Otevřená informatika**		
Studijní obor:	**Počítačové inženýrství**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Průmyslový grafický terminál založený na OS Linux

Název diplomové práce anglicky:

Idustrial graphical terminal based on Linux

Pokyny pro vypracování:

Cílem této práce je vytvořit grafický terminál pro vestavné průmyslové aplikace. Předpokládá se nasazení ve výrobních závodech, kde bude terminál sloužit k monitorování či ovládání různých zařízení a jejich napojení na cloudové služby.
1. Vyberte vhodný hardware dle následujících požadavků: Dotykový full HD displej s úhlopříčkou cca 10', čtečka RFID čipů, CPU a GPU s dobrou podporou Linuxu, rozmanité možnosti vstupů a výstupů (Ethernet, Wi-Fi, RS232, SPI, I2C, GPIO, USB).
2. Sestavte Linuxový systém (jádro, uživatelské prostředí) pro vybraný hardware. Sledujte následující cíle: a) na terminálu bude běžet webový prohlížeč s HW akcelerovanou grafikou, b) výsledný systém bude flexibilní kvůli potřebě nasazování do různých aplikací a c) systém bude možné snadno a vzdáleně aktualizovat, zejména kvůli aplikaci bezpečnostních záplat.
3. Systém otestuje na aplikaci, ve které bude terminál připojen ke kávovaru, bude ukládat data o jeho provozu a umožní zobrazovat grafické statistiky jeho používání jednotlivými uživateli.
4. Výsledek pečlivě zdokumentujte a zdůvodněte rozhodnutí činěná při návrhu.

Seznam doporučené literatury:

https://buildroot.org/downloads/manual/manual.html
https://www.yoctoproject.org/

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Michal Sojka, Ph.D., katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **30.01.2018** Termín odevzdání diplomové práce: ______________

Platnost zadání diplomové práce: **30.09.2019**

Ing. Michal Sojka, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání Podpis studenta

Aknowledgements

I thank prof. Dr. Ing. Zdeněk Hanzálek for providing advice and a nice working environment. I thank Ing. Jan Dvořák from Vozerovice for making me the first debugging cable, Bc. Pavel Bezstarosti for insights into hardware, Ing. Martin Vajnar for laser-cutting and Ing. Martin Gurtner for 3D-printing. I thank my supervisor, Ing. Michal Sojka, Ph.D., for advice and guidance throughout the project.

Declaration

Abstract

This thesis presents a prototype of an industrial graphical terminal and describes its development. The terminal consists of a WUXGA touchscreen display, an RFID reader and a single-board computer featuring the i.MX 6Quad processor. It runs a GPU-accelerated web browser on a Linux operating system. The browser receives and displays data from the RFID reader and GPIO pins through an application developed specifically for the terminal. If connected to a network (e.g. via WiFi), the browser can send the data to the cloud and download remote content. The functions are demonstrated in a testing scenario with the terminal connected to a coffee machine.

Abstrakt

Tato práce představuje prototyp průmyslového grafického terminálu a popisuje jeho vývoj. Terminál se skládá z dotykového WUXGA displeje, RFID čtečky a jednodeskového počítače s procesorem i.MX 6Quad. Na terminálu běží pod operačním systémem Linux webový prohlížeč s hardwarově akcelerovanou grafikou. Prohlížeč přijímá a zobrazuje data z RFID čtečky a GPIO pinů skrz aplikaci vyvinutou speciálně pro tento terminál. Pokud je připojen k síti (třeba pomocí WiFi), může prohlížeč odesílat tato data do cloudu a stahovat vzdálený obsah. Funkce jsou demonstrovány na testovacím scénáři s terminálem připojeným ke kávovaru.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the age of Industry 4.0, when automation of manufacturing technologies spreads, it is desirable to have a central point for controlling and monitoring production. Moreover, as cloud computing becomes more and more common, this point should be accessible remotely from the cloud. A graphical terminal, connected to the factory network, is an elegant solution to the problem. Imagine a device that collects data from all manufacturing machines and immediately uploads them to the cloud, while allowing factory personnel to monitor and control the machines on-site.

I was tasked to develop a prototype of such a terminal for Merica s.r.o., a company that specializes in production optimization and embedded systems. According to our plan, the terminal would be light, portable and yet capable of displaying high-resolution graphics. It would feature a touchscreen display, provide a variety of connector interfaces and enable users to log in using RFID tags. All components of the terminal would have to be able to endure possibly harsh conditions of an industrial environment.

The terminal would run a GPU-accelerated web browser on a Linux operating system. It would utilize cloud services but work off-line as well. Data from peripherals would be sent to the browser, displayed and possibly uploaded to the cloud. The browser would download and display remote content from the cloud, too.

This thesis describes the prototype terminal and its development. It consists of three parts: First, all hardware components are introduced, then software is discussed and, finally, a testing scenario with a coffee machine is presented.

Chapter 2

Hardware

We designed the terminal to consist of three main components: a processor board, a touchscreen display and an RFID reader. The first task for me was to choose the right hardware.

2.1 Processor board

There are many embedded systems on the market. A single-board computer (SBC), an all-in-one solution with a processor, memory and connectors, came out the most suitable type. An alternative approach could be to use a computer on module, a system that lacks standard peripheral connectors but can be plugged into a carrier board.

I was given the following list of desirable properties:

- guaranteed longevity of supply
- wide operating temperature range (−30 to 65 °C)
- power supply voltage 6 to 24 V
- Full HD graphics, touchscreen support
- serial port (TTL, RS-232, RS-485), Ethernet (1 Gbps), USB, SPI, I^2C
- serial console (TTL, USB), JTAG connector
- GPIO, PWM, ADC
- hardware accelerated video decoding and user interface
- Linux support

Because of the hardware accelerated Full HD graphics requirement, I could rule out all lower-end solutions. The longevity of supply we needed then, along with the requirement of almost industrial-grade operating temperature range, narrowed down the results to just a few products. Soon it became clear that no solution would provide everything.

Many industrial SBCs incorporate Intel CPUs (see e.g. [1]). While their performance would be sufficient, I/O peripherals of these systems are often limited. SBCs based on ARM processors, widely used in cheap microcontrollers as well as high-end smartphones, present an alternative. A prominent place among industrial ARM-architecture CPUs is held by Freescale’s i.MX series. An i.MX 6-based solution, providing both hardware accelerated graphics and extensive assortment of I/O interfaces, seemed the best choice in the end.

2.1.1 i.MX 6Quad processor

The i.MX 6 series are multimedia-focused single-, dual- and quad-core processors optimized for low power consumption. They were launched in November 2012 and are guaranteed for a production longevity of 15 years (10 years for consumer models). The i.MX 7 series are newer but only low-power processors without accelerated graphics and the i.MX 8 processor, a powerful successor, was introduced just this year so no boards are available yet.

The i.MX 6Quad processors 'feature implementation of the quad ARM Cortex-A9 core, which operates at speeds up to 1.2 GHz', says [2]. I have already mentioned consumer models of the processors, the other ones are industrial and automotive. Consumer models provide operating temperature range −20 to 105 °C, industrial models can endure even −40 °C but at the cost of lower speed (800 MHz).

The i.MX 6Quad 'include 2D and 3D graphics processors, 1080p video processing, and integrated power management', continues [2]. The following graphics accelerators are available:

- **Video Processing Unit (VPU)** is able to decode or encode multiple video streams simultaneously by time multiplexing. Optional video processing such as rotation or mirroring is also covered. VPU is capable of 1080p30 H.264 video encoding and supports decoding of MPEG-2, MPEG-4, H.264, VC-1 or VP8.
- Two **Image Processing Units (IPUs)** provide connectivity to displays and cameras, related image processing (resizing, rotations, colour adjustments, ...), synchronisation and control. Each IPU supports concurrent connection to two displays and two cameras. Four display interface types are available:
  - two parallel display ports (TTL, 24 bpp, up to 200 MHz)
  - MIPI Display Serial Interface (2 lanes, 1 GHz)
  - HDMI 1.4
  - two LVDS channels (up to 170 MHz, 1920×1200 at 60 Hz)

  The parallel display ports are driven directly by the IPUs, the other interfaces require corresponding bridges. Specifically, the LVDS channels are driven by the LVDS Display Bridge (LDB).
- **3D Graphics Processing Unit (GPU3D)** possesses sufficient power to accelerate 3D graphics; including graphical user interface, animations and gaming; on a Full HD display. According to [3], it supports the following APIs:
  - OpenGL ES 2.0
  - OpenGL ES 1.1
  - OpenVG 1.1
  - DirectX 11
  - OpenGL 2.1 and 3.0
  - OpenCL 1.1 E
  - EGL 1.4
- **2D Graphics Processing Unit (GPU2D)** is able to accelerate many 2D graphics algorithms such as line drawing, image scaling and colour space conversion.
- **Vector Graphics Processing Unit (GPUVG)** supports OpenVG 1.1 as well as other vector graphics accelerations.

All the graphics accelerators constitute the Video-Graphics Subsystem. Figure 2.1, inspired by [3], shows a scheme thereof.

The i.MX 6Quad processors provide 1 Gbps Ethernet, High Speed USB 2.0 (480 Mbps), five serial ports (up to 5 Mbps each), SPI, I^{2}C, GPIO with interrupt capabilities, PWM, JTAG and many other

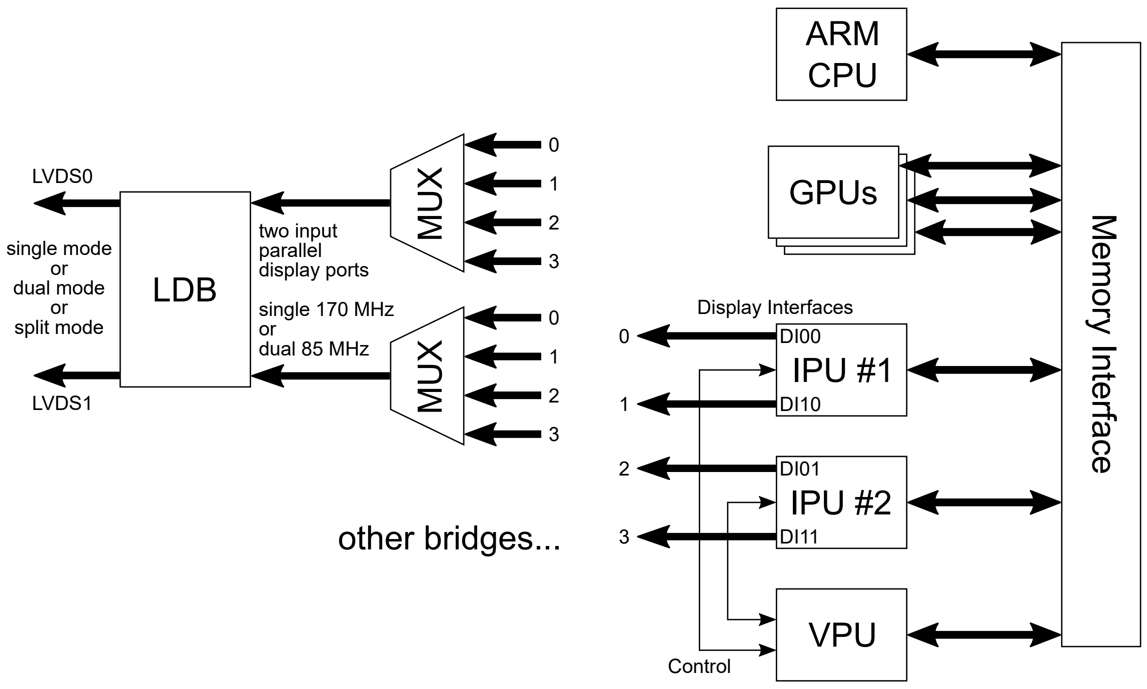


Figure 2.1: Simplified schemes of the i.MX 6Quad Video-Graphics Subsystem and the LDB

peripheral interfaces. The actual performance of the 1 Gbps Ethernet is up to 400 Mbps due to internal bus throughput limitations.

2.1.2 Candidates

This subsection sums up my search for an industrial SBC and presents three products that I find to be on top. All of them incorporate the i.MX 6Quad processor.

- **Technologic Systems TS-7970** was released in 2017 and the company guarantees 10-year availability. The quad-core version costs $382. The product provides 34 GPIO pins (two are 30 V tolerant) and 3× analog input. I did not find any other SBC that featured an ADC. Unfortunately, this model uses only one LVDS channel and thus it cannot be connected to a Full HD display via this interface (see subsection 2.2.1).

- **WinSystems SBC35-C398Q** is a solution whose chief asset are probably its 24 GPIO pins tolerant to 30 V . It features dual-channel LVDS, too. The company provides lifetime support and they were helpful even during our email communication: it was them who warned me about the Ethernet limitation (see subsection 2.1.1) of the i.MX 6Quad. This is, in my opinion, the best SBC money can buy; however, $495 was well beyond our budget.

- **Advantech RSB-4411** supports dual-channel LVDS and costs ‘only’ €237. This was the winner in the end.

2.1.3 Advantech RSB-4411

Advantech RSB-4411 is a new 3.5-inch SBC featuring the i.MX 6 processor. One can choose from dual- and quad-core models for either consumer or industrial applications. Main RSB-4411 features, given by [4, 1], follow:

- i.MX 6Quad, up to 1 GHz (consumer) or 800 MHz (industrial)
- 1 GB memory, 4 GB eMMC storage, SD card slot
- HDMI, VGA, dual-channel LVDS (all 1920×1080 at 60 Hz)
- 3× serial port (UART), 2× I²C, 1× SPI, 20× GPIO (3.3 V, without isolation)
- 5× USB 2.0, 1 Gbps Ethernet, Mini PCIe, M.2
- operating temperature range 0–60 °C (consumer) or −40–85 °C (industrial)
- power supply voltage 12 V, 19 V or 24 V

Except for PWM, an ADC and a JTAG connector RSB-4411 matches the desirable properties on page 2 quite well. Because the official online shop at buy.advantech.eu offered only the consumer quad-core model, we purchased that one; we decided it was sufficient for the prototype. Possible future production of the terminal will, of course, use the industrial variant.

I said RSB-4411 was new. In fact it is so new that at the time of the purchase there wasn't any manual available to download. I had to request [5] by email. Sadly, information in the manual still wasn't enough and it took another round of emails to acquire everything I needed to buy the right connectors. Every connector I didn't know is listed in Table 2.1.

Function	**Connector**	**Mating part**
LVDS	CN30	Hirose DF13-40DS-1.25C
LVDS Backlight	LVDS_BKLT_PWR	JST PHR-5
GPIO	CN31	Molex Series Milli-Grid 51110
USB	CN12, CN29	
RS-232	CN21	
CAN	CN32	
SPI	CN35	
Debug Port	CN5	Molex Series PicoBlade 51021
I²C	CN33, CN34	
MIC in	CN23	
Line out	CN14	
DC Power Jack	CN10	5.5 × 2.5 mm DC plug

Table 2.1: List of lesser-known connectors on RSB-4411 including mating parts

pin	21	19	17	15	13	11	9	7	5	3	1
chip	1	5	1	1	2	4	2	5	0	0	3.3 V
line	0	7	3	2	23	2	31	31	25	27	
chip	5	1	5	5	5	2	2	2	0	0	GND
line	8	4	16	9	11	20	21	30	30	29	
pin	22	20	18	16	14	12	10	8	9	4	2

Table 2.2: Scheme of the GPIO pin header on RSB-4411 including chip and line numbers

Another piece of information the manual doesn't provide concerns the GPIO pin header. The i.MX 6Quad features 6 GPIO chips, each consisting of 32 lines. The 20 lines connected to the pins of

CN31 are scattered throughout the chips and [5] is of no help in identifying them. Therefore I include Table 2.2 which shows each pin's chip-line location. See subsection 3.2.2 for further explanation.

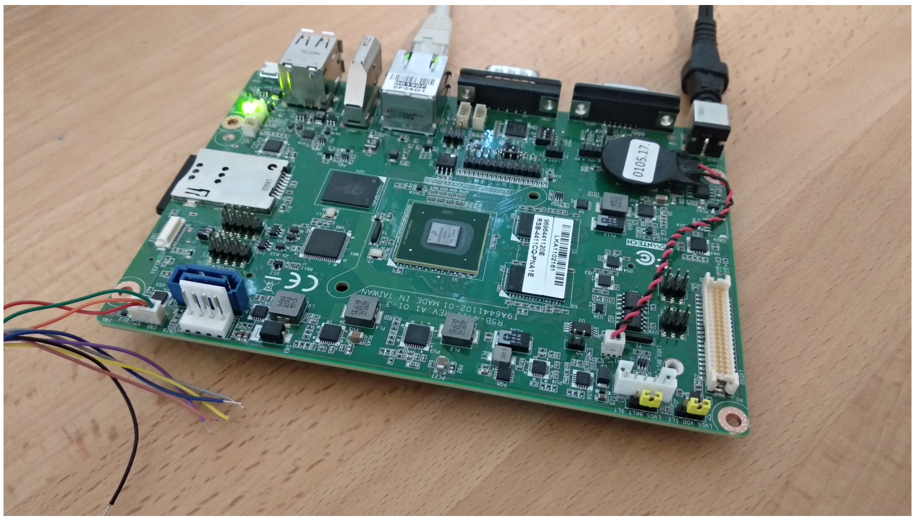


Figure 2.2: Advantech RSB-4411

2.2 Display

Finding the display was the most complicated part. I was given the following requirements:

- screen size about 10 in
- Full HD resolution
- capacitive touchscreen

That is not a common combination at all: most 10-inch displays have a lower resolution. I wanted to buy a display with an embedded touch panel so I wouldn't have to install it myself. Our options were therefore very limited.

My first idea was to find a full-featured monitor and connect it via HDMI. However, such a solution would have been too heavy; and the goal was to keep the terminal portable. I started searching for a standalone display panel. Panelook.com proved to be an excellent source of information here.

2.2.1 Display interfaces

I had to choose a display with a compatible interface. High-resolution display panels usually incorporate one of these:

- **LVDS** is an old but still widely used standard for transferring data at high rates. 'There is no standardized connector defined', says [6]. 'Signals consist of dedicated data pin pairs, clocking, backlight control, power panel sequencing, and I^{2}C signals'. Because a single LVDS channel is not enough, Full HD displays use dual-channel interfaces, which suffice for resolutions of up to 2048×1536 at 60 Hz. As I said in subsection 2.1.1, dual-channel LVDS is supported by the i.MX6Quad.
- **MIPI**, which refers to Display Serial Interface by the MIPI Alliance, was designed for battery-powered mobile devices. The i.MX6Quad processors provide this interface, as I have already mentioned, but support only resolutions of up to 1280×720.

- **eDP** is a next-generation standard defined by the Video Electronics Standards Association. It is based on packetized data transmission (similarly to USB) and requires fewer signals than LVDS. Alas, eDP is not supported by the i.MX 6Quad.

Because we wanted to connect the display directly to the SBC, the only acceptable interface was dual-channel LVDS. Touchscreen digitizers use either I²C or USB. Though USB was preferable, this wasn't a limiting factor.

2.2.2 Candidates

Here I present three display solutions my search came up with. Both the display panels listed below feature dual-channel LVDS so they can be connected to RSB-4411 directly.

- **Lilliput TK1330-NP/C/T** is a full-featured computer monitor, but I list it anyway. It features a Full HD 13.3-inch display panel, a capacitive touchscreen and HDMI. It costs €279.
- **Ampire AM-19201200B1TZQW-T51** is an industrial 10.1-inch display with the resolution of 1920×1200 and an embedded USB touchscreen digitizer. It is by all means the ideal candidate. I contacted datadisplay-group.de about it and they quoted a hilarious price of €500.
- **AUO B101UAN02.1** isn't really a touchscreen display and has been discontinued. However, this model constitutes a part of many do-it-yourself touchscreen display sets one can buy from China. B101UAN02.1 is, similarly to the Ampire, a 10.1-inch WUXGA display. The Chinese sets add a USB touchscreen digitizer, a controller board and all necessary cables. The controller board provides HDMI, which is a plus. We figured this would be sufficient for the prototype and bought one set at [7] for £105.08.

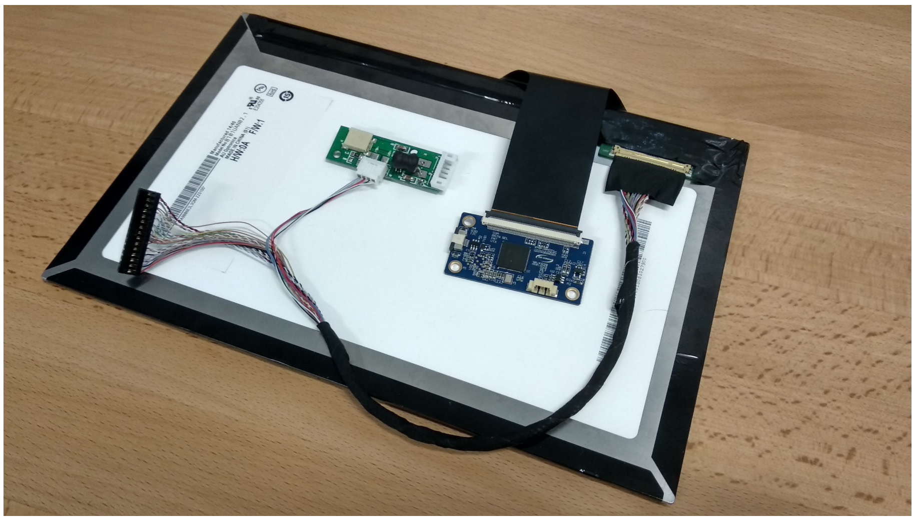


Figure 2.3: AUO B101UAN02.1

2.2.3 AUO B101UAN02.1 set

Detailed description of the Chinese set's components follows.

As I have said, the AUO B101UAN02.1 is a 10.1-inch WUXGA display. According to [8], it features a 50-wire I-PEX 20455-050E-12 connector. Its driving logic requires 3.3 V power supply, the backlight unit needs 24 V power supply. The timing characteristics provided by [8] do not match the given EDID values. The right timing values are shown in Figure 2.4.

The touchscreen is a capacitive model by ILITEK. It is able to track at least 10 fingers simultaneously. It comes with a driver board which can be connected to USB using a PicoBlade 51021 connector.

The control board is a M.NT68676.2A model. See [9] for more information. In addition to the control board, the set includes a 12 V-to-24 V level shifter, a button board and necessary cables.

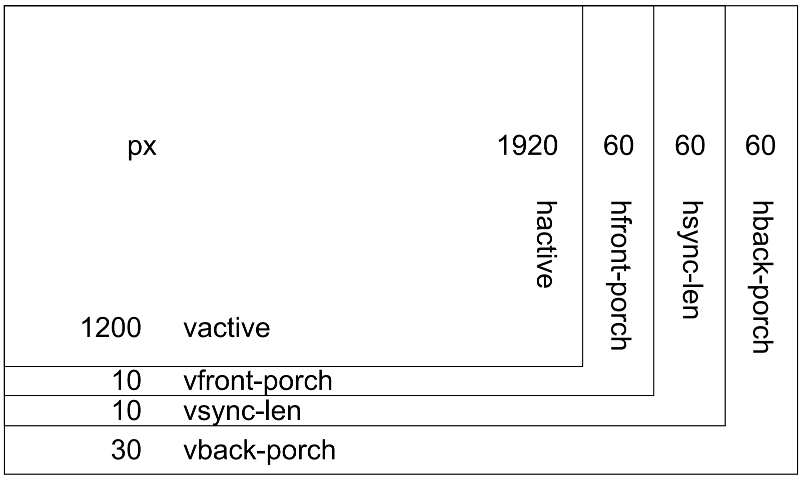


Figure 2.4: Display timings

2.3 RFID reader

The last piece for me to find was an RFID reader. It had to be compatible with my ISIC.

2.3.1 Technology and standards

‘Radio-frequency identification uses electromagnetic fields to automatically identify and track tags attached to objects’, says [10]. The tags used in access cards are the passive ones: they ‘collect energy from a nearby reader’s interrogating radio waves’. Each RFID tag contains an integrated circuit for storing and processing data, and an antenna. When the tag receives the reader’s interrogating radio signal, it responds with its UID. ‘Unlike a barcode, the tag need not be within the line of sight of the reader.’

Radio frequencies used by RFID systems range from 125 kHz (low frequency) to 2.45 GHz (super high frequency) and higher. Most smart cards, including my ISIC, operate at 13.56 MHz (high frequency, HF).

I learned that my ISIC is a MIFARE DESFire card. Smart RFID cards are defined by an international standard ISO/IEC 14443 and MIFARE is one of its most widely used proprietary implementations. NFC protocols, supported by many smartphones, are also based on said standard.

So, I had to find an active RFID reader that operated at 13.56 MHz and supported MIFARE smart cards.

2.3.2 Digital Logic μFR Nano

I call it the marvel from Serbia. Digital Logic's μFR Nano is a compact RFID and NFC reader writer supporting a wide range of smart cards. It is available as a printed circuit board which can be easily mounted on a larger system. I wasn't able to find any alternative candidates; μFR Nano is quite unique. Some notable features, given by [11, 12], follow:

- operating frequency 13.56 MHz
- implementation of the ISO/IEC 14443 A & B and ISO/IEC 18092 (NFC) standards
- support of all variants of the MIFARE family
- asynchronous UID sending (no need to manually poll the reader)
- connected via Micro-USB 2.0, visible as a serial device
- operating temperature range −10 to 60 °C

The company provides free libraries for all major platforms as well as source code examples in many languages.

I bought μFR Nano at webshop.d-logic.net for €56.

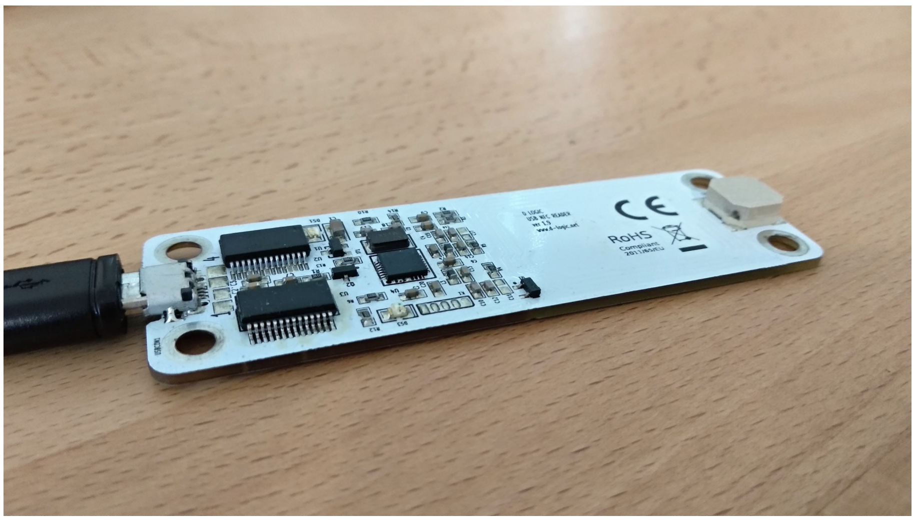


Figure 2.5: Digital Logic μFR Nano

2.4 Accessories

In addition to the main three components, I had to acquire or make a few other things. I have already mentioned some necessary connectors (see Table 2.1); those I purchased at farnell.com, along with compatible crimp terminals. I also bought an RS-232 serial cable for debugging. In this section, I will discuss the other accessories in detail.

2.4.1 WiFi card

Unlike some other SBCs, RSB-4411 doesn't integrate a WiFi module. Such a module, however, can be connected via the Mini PCIe connector. There are some WiFi modules using the M.2 connector, but these modules utilize the PCIe interface nevertheless and, as one can find in [5], RSB-4411's M.2 connector doesn't provide necessary PCIe pins.

My supervisor was able to obtain a Wireless-N 7260 module by Intel. It has the following features:

- Mini PCIe connector
- radio band 2.4 GHz, up to 300 Mbps
- WiFi 802.11bgn (via PCIe), Bluetooth 4.0 (via USB)

The only problem was the size of module. RSB-4411 only supports full-size Mini PCIe cards, so I needed an extending bracket. I considered buying it, but then I had the most excellent idea: Could it be 3D printed? I found a schematic at [13] and took it to a school 3D-printing lab, where a colleague kindly printed the bracket for me. With the extending bracket, I was able to attach the WiFi module to the SBC.

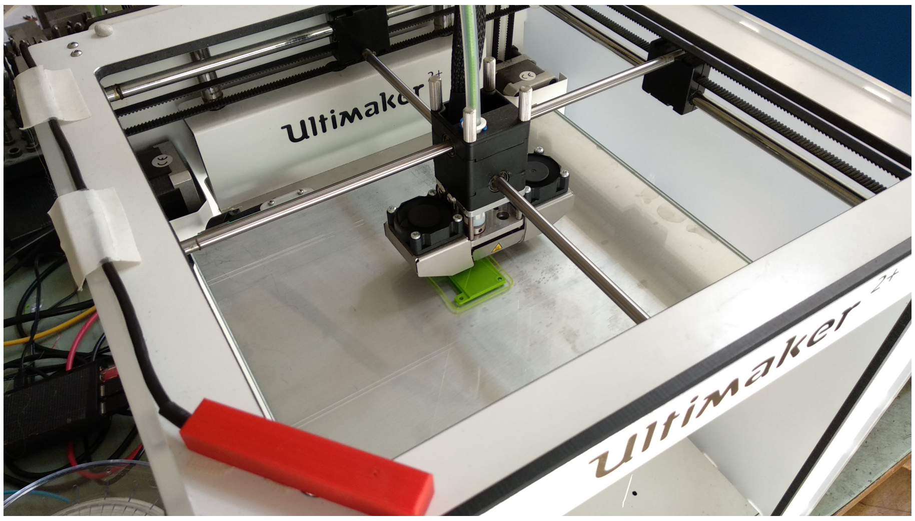


Figure 2.6: PCIe extending bracket in the making

2.4.2 Level shifter

As I said in subsection 2.1.3, RSB-4411 features 3.3 V GPIO pins without isolation. In order to enable 5 V I/O signals, we decided to connect the pins to a level shifter. I found a suitable model by Texas Instruments, SN74LVC245A, and purchased it at farnell.com. I also bought a set of flat cable connectors, specific models follow: MLW10G, PSLV10 and PFL10. My supervisor soldered the level shifter to the connectors according to the diagram in Figure 2.7.

2.4.3 Cables

This section describes cables I made for the terminal.

I wanted to spare RSB-4411's **USB** connectors and use the CN12 pin header instead. I took a USB-to-Micro-USB cable, cut it in half, crimped the wires and attached them to the right connectors. Table 2.3 shows a scheme of the cable. A photo of the cable is shown in Figure 2.8c.

Devices	**Connectors**	**Connector**	**Device**
RFID reader	Micro-USB	Milli-Grid 51110	SBC
touchscreen	PicoBlade 51021		

Table 2.3: USB cable scheme

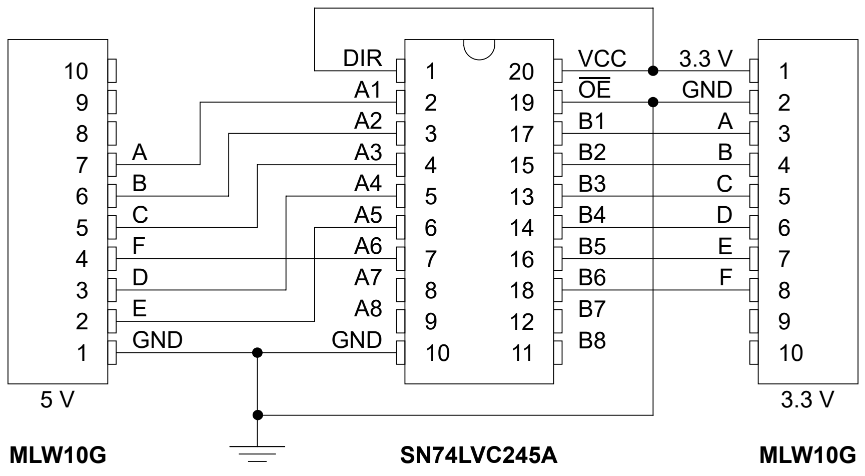


Figure 2.7: Level shifter circuit diagram

GPIO pins had to be connected to the level shifter. This was a simple matter of crimping flat cable. A photo of the cable is shown in Figure 2.8b. Table 2.4 depicts a scheme.

Device	Connector	Connector	Device
level shifter	PFL10	Milli-Grid 51110	SBC

Table 2.4: GPIO cable scheme

First, I didn't want to make the **LVDS** cable at all. I contacted several companies and asked them to make it for me. Some said they didn't do 50-wire cables, because it was 'a difficult connector system to work with'. Others refused to deal with 'individuals or educational institutes'. One company offered they would make it in three months. In the end, I decided to make the cable myself. I took the cable from the display set, but I had to replace the board-side connectors. Doing that was a real pain in the neck. See a photo of the cable in Figure 2.8a and a scheme in Table 2.5. I also include the pinouts of both the original and the new connectors in Figure 2.9.

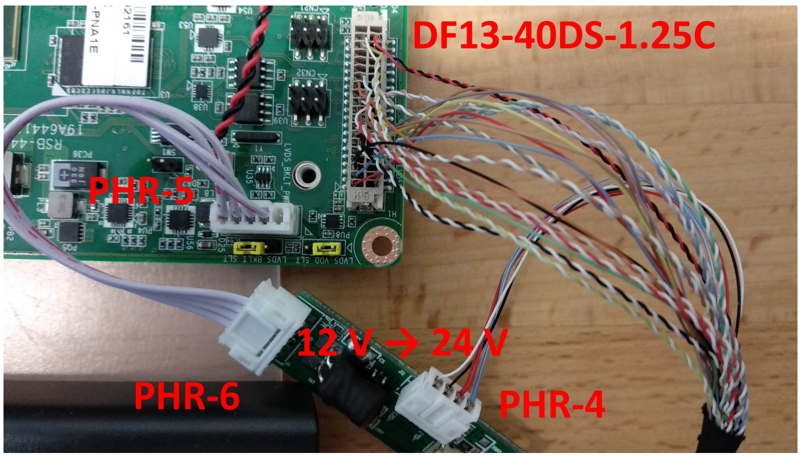
Device	Connector	Connector	Device
display	I-PEX 20453-050T-11	DF13-40DS-1.25C	SBC
		PHR-4	level shifter
level shifter	PHR-6	PHR-5	SBC

Table 2.5: LVDS cable scheme

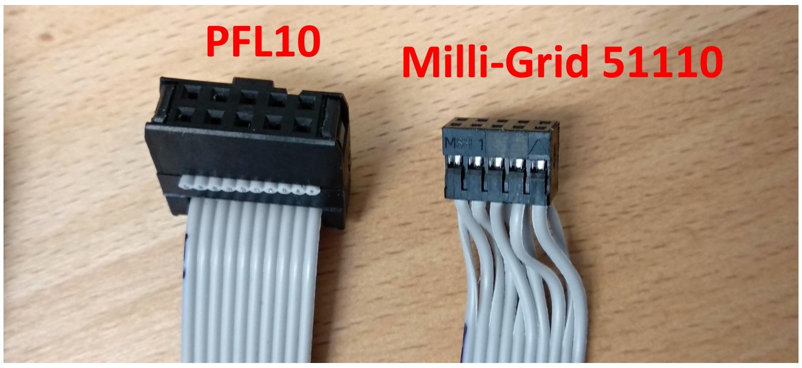
2.5 Body

Finally, I had to design a body that would hold all the components of the terminal. The idea was to make the body from two separate acrylic sheets joined with standoffs.

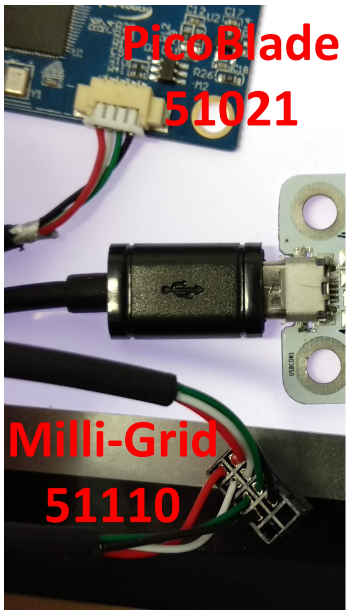
I designed the body focusing on two goals: to place all connectors on one side and to keep the dimensions as small as possible. Both sheets are shown in Figure 2.10. The top sheet holds the display and the RFID reader from above, while the SBC and the level shifter are mounted on



(a) LVDS



(b) GPIO



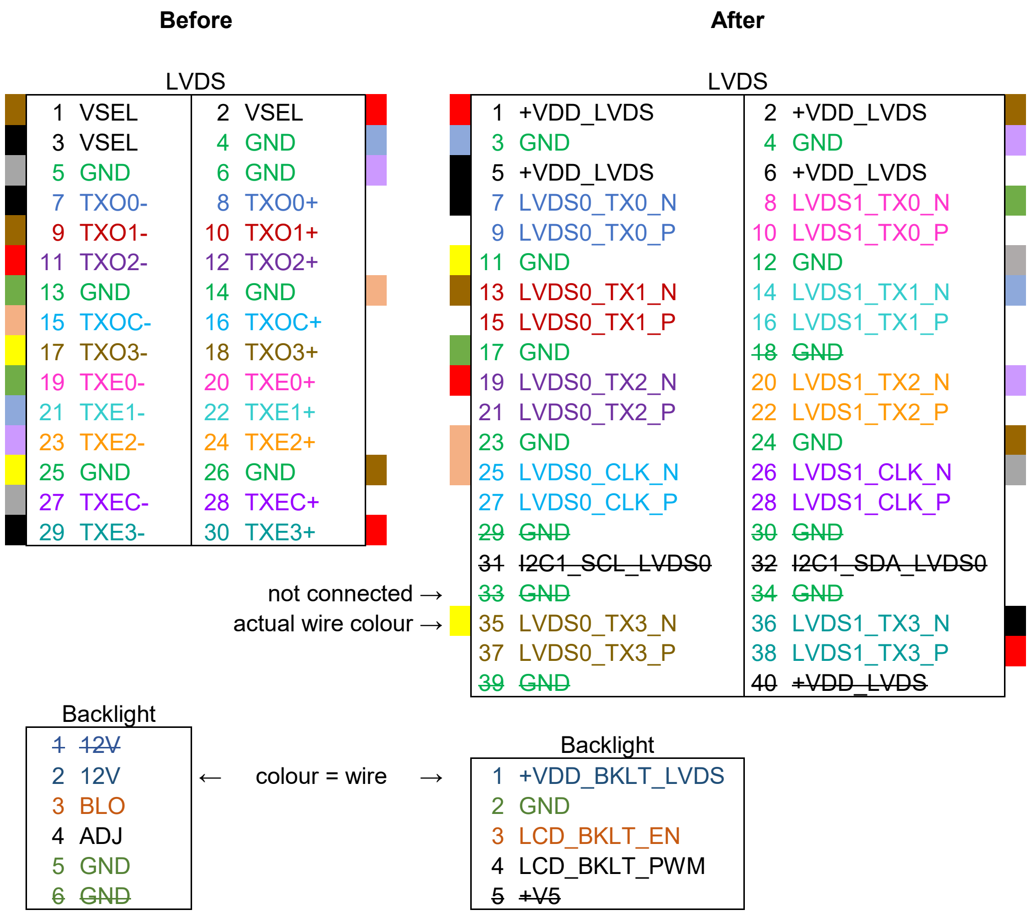
(c) USB

Figure 2.8: Cables

top of the bottom sheet. The top sheet has to be sufficiently thick because of the big hole for the touchscreen. The bottom sheet also contains eight holes that might be used for cable fixing.

I drew the body in the SolveSpace CAD program, which was recommended to me by my supervisor, and took the schematics to a laser cutter in a school lab. (In fact, it was the same lab the PCIe extending bracket was made in.) An experienced colleague cut the sheets for me.

The terminal was ready for assembly.



Before

LVDS

🟫	1	VSEL	2	VSEL	🟥
⬛	3	VSEL	4	GND	🟦
▨	5	GND	6	GND	🟪
⬛	7	TXO0-	8	TXO0+	
🟫	9	TXO1-	10	TXO1+	
🟥	11	TXO2-	12	TXO2+	
🟩	13	GND	14	GND	🟧
🟧	15	TXOC-	16	TXOC+	
🟨	17	TXO3-	18	TXO3+	
🟩	19	TXE0-	20	TXE0+	
🟦	21	TXE1-	22	TXE1+	
🟪	23	TXE2-	24	TXE2+	
🟨	25	GND	26	GND	🟫
▨	27	TXEC-	28	TXEC+	
⬛	29	TXE3-	30	TXE3+	🟥

After

LVDS

🟥	1	+VDD_LVDS	2	+VDD_LVDS	🟫
🟦	3	GND	4	GND	🟪
⬛	5	+VDD_LVDS	6	+VDD_LVDS	
⬛	7	LVDS0_TX0_N	8	LVDS1_TX0_N	🟩
	9	LVDS0_TX0_P	10	LVDS1_TX0_P	
🟨	11	GND	12	GND	▨
🟫	13	LVDS0_TX1_N	14	LVDS1_TX1_N	🟦
	15	LVDS0_TX1_P	16	LVDS1_TX1_P	
🟩	17	GND	~~18~~	~~GND~~	
🟥	19	LVDS0_TX2_N	20	LVDS1_TX2_N	🟪
	21	LVDS0_TX2_P	22	LVDS1_TX2_P	
🟧	23	GND	24	GND	🟫
🟧	25	LVDS0_CLK_N	26	LVDS1_CLK_N	▨
	27	LVDS0_CLK_P	28	LVDS1_CLK_P	
	~~29~~	~~GND~~	~~30~~	~~GND~~	
	~~31~~	~~I2C1_SCL_LVDS0~~	~~32~~	~~I2C1_SDA_LVDS0~~	
not connected →	~~33~~	~~GND~~	~~34~~	~~GND~~	
actual wire colour → 🟨	35	LVDS0_TX3_N	36	LVDS1_TX3_N	⬛
	37	LVDS0_TX3_P	38	LVDS1_TX3_P	🟥
	~~39~~	~~GND~~	40	~~+VDD_LVDS~~	

← colour = wire →

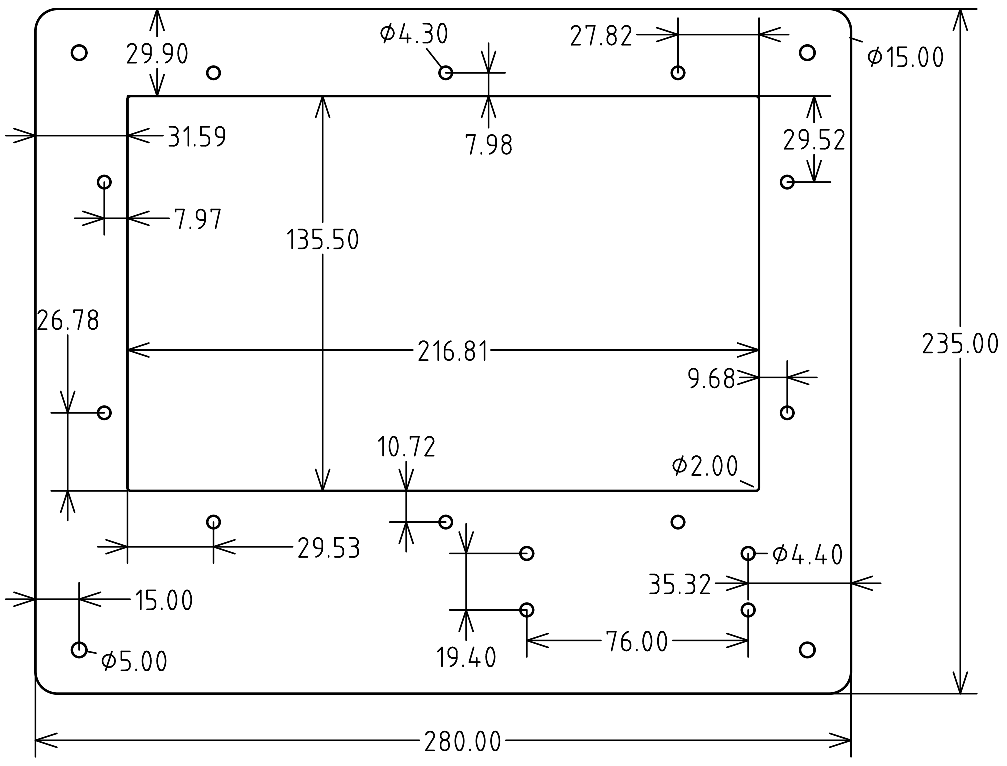
Backlight (Before)

~~1~~	~~12V~~
2	12V
3	BLO
4	ADJ
5	GND
~~6~~	~~GND~~

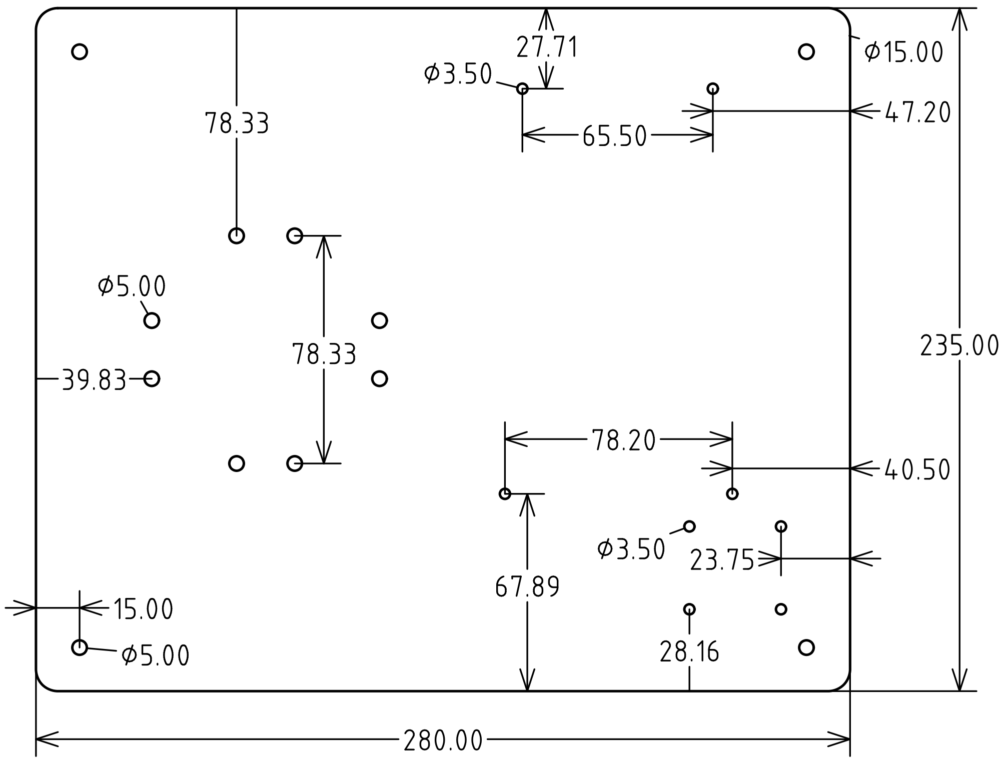
Backlight (After)

1	+VDD_BKLT_LVDS
2	GND
3	LCD_BKLT_EN
4	LCD_BKLT_PWM
5	~~+V5~~

Figure 2.9: Display connector pinout



(a) Top sheet



(b) Bottom sheet

Figure 2.10: Drawing of the terminal's body

Chapter 3

Software

When all the chosen hardware lay on my table and the devices were successfully connected, I started to build a Linux operating system. The ultimate goal was to deploy a web browser that would allow users to view remote web pages as well as local data from the peripherals. To achieve this goal I had to build many underlying layers of software first.

3.1 Linux system

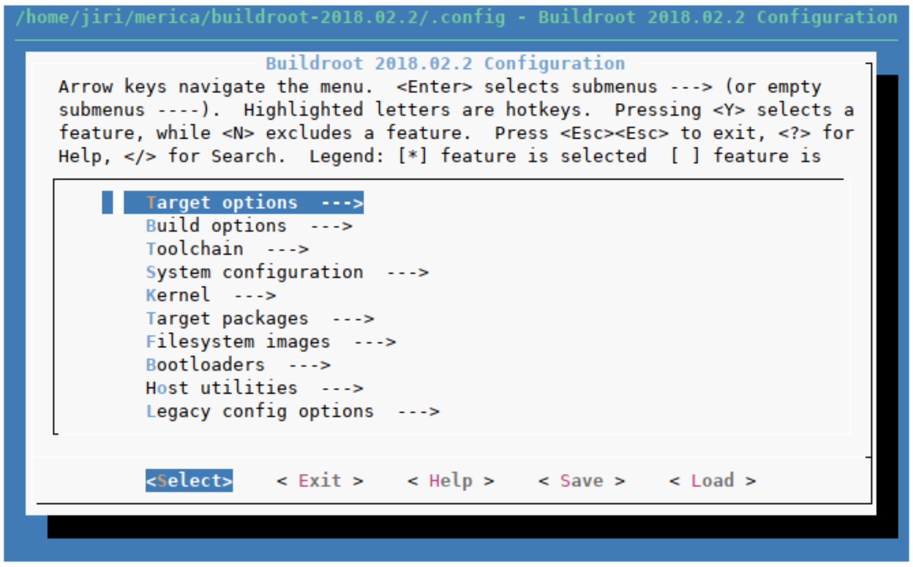
Building a Linux operating system from the source code to a single SD-card image is a complex task. Luckily, there are tools that simplify and automate the process. In terms of embedded Linux systems two major tools are available: Buildroot and the Yocto Project. Both are capable of cross-compiling the system for ARM and provide hundreds of packages.

- **The Yocto Project** is supported by many manufactures including Advantech. It uses its own BitBake tool to build software from so-called ‘recipes’. Yocto is more complex and more difficult to learn than Buildroot.
- **Buildroot** strives to be simple and easy to use. It reuses existing technologies: a Buildroot system is configured similarly to the Linux kernel (via Kconfig files) and built using the Make utility. See the configuration tool in Figure 3.1. Compared to Yocto, Buildroot is less supported by manufactures.

So I had two options. On the one hand, Advantech provided Yocto-based BSP for RSB-4411 at [14]. On the other hand, Buildroot had been recommended to me by my supervisor and colleagues at Merica. I tried to follow the Yocto user guide at [15] and build the Advantech image. My attempts to create the right BitBake environment in my system were not successful so I downloaded the Docker image. The build failed after a few hours due to a dead package source. I had enough of Docker and Advantech’s outdated BSP. I switched to Buildroot and started to create a configuration for RSB-4411 from scratch.

3.1.1 Buildroot

Buildroot has very few requirements and no system restrictions (as long as it is a Linux). One can just download a 5-megabyte archive, and build a Linux system using two `make` commands. In case of more complex scenarios there is an extensive user manual available at [16].



```
/home/jiri/merica/buildroot-2018.02.2/.config - Buildroot 2018.02.2 Configuration
                    Buildroot 2018.02.2 Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> selects a
  feature, while <N> excludes a feature.  Press <Esc><Esc> to exit, <?> for
  Help, </> for Search.  Legend: [*] feature is selected  [ ] feature is

          Target options  --->
          Build options  --->
          Toolchain  --->
          System configuration  --->
          Kernel  --->
          Target packages  --->
          Filesystem images  --->
          Bootloaders  --->
          Host utilities  --->
          Legacy config options  --->

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 3.1: Buildroot configuration menu

Buildroot is very actively developed. A new version is released every three months; always featuring new packages, improvements and bug fixes. The biggest problem I have encountered was solved simply by updating Buildroot. Therefore it is very desirable to keep the whole system easily maintainable and updatable. In order to achieve that, I used Git and tried to stick to the following rules:

1. Always use the latest stable version.
2. Maintain a separate Git branch.
3. Do not change Buildroot files, only add new ones.

See a simple setup-update scenario below.

```
git clone git://git.buildroot.net/buildroot  #download Buildroot
cd buildroot
git checkout -b my_branch 2018.02.1          #start a new branch at
                                             #the latest stable release
#add some files and commit
                                             #later...
git rebase --onto 2018.02.2 2018.02.1        #rebase onto an update
```

I didn't have to create the configuration entirely from scratch. Buildroot's `configs` directory contains config files for many devices and even though RSB-4411 is not one of them, some other i.MX 6Quad-based SBCs are supported. The following commands can be used to start a system configuration:

```
make imx6-sabresd_defconfig O=../build       #a new out-of-tree build
cd ../build
make menuconfig                              #run the configuration tool
```

Note the 'O=' argument used to start the build out of tree. Since it instructs Buildroot to put all generated files into the given directory, this option is quite useful when testing multiple configurations simultaneously. In the rest of this section I will describe the configuration steps. I'll provide details about each menuconfig entry shown in Figure 3.1.

Target options must match the CPU's properties. Because I used existing i.MX 6Quad defconfig, there was no need to change these. As I mentioned in subsection 2.1.1, the i.MX6Quad is based on Cortex-A9. VFP and NEON are also implemented, says [3]. I only switched the instruction set to Thumb2 which, according to [17], results in about 20–30 % smaller binaries and is only slightly slower than ARM. Target options thus look as follows:

```
    Target Architecture (ARM (little endian))  --->
    Target Binary Format (ELF)  --->
    Target Architecture Variant (cortex-A9)  --->
[*] Enable NEON SIMD extension support
[*] Enable VFP extension support
    Target ABI (EABIhf)  --->
    Floating point strategy (VFPv3)  --->
    ARM instruction set (Thumb2)  --->
```

In **Build options** I renamed the config file `imx6q-rsb4411-a1_defconfig` and enabled a compiler cache to speed up the build process. I left the rest of the options untouched. **Filesystem images** didn't require many changes either: with the ext4 option enabled I only had to adjust the exact size as the system expanded. I strictly avoided **Legacy config options**.

The next step was to configure the **Toolchain** options. A toolchain is a set of software tools and libraries that as a whole enable building software from source. Buildroot provides two main options here: One can either use Buildroot's own toolchain, which is well integrated but has to be compiled first, or download an external one (Linaro by default). I used Linaro in an early stage of development (for the same reason I enabled the compiler cache) but switched to Buildroot toolchain later. The toolchain configuration changes follow:

```
    C library (glibc)  --->
    Kernel Headers (Same as kernel being built)  --->
    Custom kernel headers series (4.9.x)  --->
    Binutils Version (binutils 2.30)  --->
    GCC compiler Version (gcc 7.x)  --->
[*] Enable C++ support
```

Due to requirements of the packages described later I had to enable the glibc library and C++ support. I set the kernel headers series to match the kernel version (see subsection 3.1.3).

The most important OS properties are configured in **System configuration**. Especially an init system; the first process to start on a Linux OS which is responsible for launching everything else. The default option is Busybox and [16] says it 'is sufficient for most embedded systems'. I made some experiments with systemd, but due to its extensive logging the system soon ran out of free space. See an excerpt from the system configuration below.

```
    Init system (BusyBox)  --->
    /dev management (Dynamic using devtmpfs + eudev)  --->
[*] Use symlinks to /usr for /bin, /sbin and /lib
(board/advantech/imx6q-rsb-4411-a1/rootfs_overlay) Root filesystem
()  Custom scripts to run before creating filesystem images
()  Custom scripts to run inside the fakeroot environment
(board/advantech/imx6q-rsb4411-a1/post-image.sh) Custom scripts to
```

The /dev management option determines how the kernel handles hardware devices. I selected 'devtmpfs + eudev' to satisfy requirements of the web browser. Note that I changed the location of the `post-image.sh` script used for creating SD-card images; in subsection 3.1.2, I will explain why. In order to decrease confusion, I enabled filesystem symlinks.

Target packages provide access to more than two thousand programs and libraries. This is where the system is actually defined. **Host utilities** list packages that might be necessary to build the system. These are enabled mostly automatically as dependencies. The manually selected utilities; dosfstools, genimage and mtools; are required for post-build image generating.

The remaining two entries will be discussed later. Now, to save the configuration back to Buildroot's `configs` directory and start the build, I used the followings commands:

```
make savedefconfig                  #save the configuration globally
make                                #wait and hope
```

3.1.2 Boot loader

A boot loader is a small program responsible for launching the kernel. There are many boot loaders available, but Linux-based embedded systems mostly use Das U-Boot. There was no reason for me to look for alternatives.

RSB-4411 features something [4] calls 'Advantech's boot loader' (which is probably a customized version of U-Boot 2015.04). This boot loader is stored on RSB-4411's integrated SPI NOR flash and it is the first program the SBC runs on power-up. When the SW2 switches are in the right positions (see [5]), it tries to start U-Boot from an SD card:

```
Adv-Boot SPL 2015.04-advantech_rsb4411a1_V6.150_svn1991+g6cf684a ...
BOOT_DEVICE_AUTO
MMC read: dev # 0, block # 3,count 1200 ...
booting from SD
Jumping to U-Boot
```

Unfortunately, Adv-Boot has some peculiar properties that preclude using it with Buildroot's default (upstream) version of U-Boot. Before the booting process is allowed to start, a CRC of the U-Boot binary is calculated. RSB-4411 refuses to boot from the card unless it stores the same checksum alongside U-Boot. Figure 3.2 attempts to explain the behaviour further. Rather than to risk bricking the SBC by overwriting the SPI NOR flash, I decided to adapt.

Advantech provides its own U-Boot at [14]. Though it's intended for Yocto, it is perfectly compatible with Buildroot, too. There are several versions (branches) available; I, of course, chose the latest: 2017.03. Advantech based these on Freescale's static releases; there is a similar situation with the kernel, so I'll provide some context later in subsection 3.1.3. They have applied several patches, adding custom board configuration files as well as checksum-making tools. Here, the standard `uboot.bin` binary is first padded with zeros and saved as `uboot_crc.bin`; then its CRC is calculated and saved into `uboot_crc.bin.crc`. These two files need to be put onto the SD card.

To incorporate Advantech's U-Boot into Buildroot, I set the **Bootloaders** entry as follows:

```
[*] U-Boot
(mx6qrsb4411a1_1G) U-Boot board name
      U-Boot Version (Custom Git repository)  --->
(https://github.com/ADVANTECH-Corp/uboot-imx6.git) URL of custom repo
(imx_v2017.03_4.9.11_1.0.0_ga) Custom repository version
```

```
        U-Boot binary format  --->
          [*] u-boot.bin
          [*] Custom (specify below)
          (u-boot_crc.bin u-boot_crc.bin.crc) U-Boot binary format: custom
  [ ]   Install U-Boot SPL binary image
```

To generate SD-card images correctly, I also had to adjust `genimage.cfg.template` as well as `post-image.sh` accordingly. I copied both files and made the following changes to the former:

```
@@ -20,12 +20,18 @@
   hdimage {
   }

-  partition u-boot {
+  partition u-boot-crc {
     in-partition-table = "no"
-    image = "u-boot.imx"
+    image = "u-boot_crc.bin.crc"
     offset = 1024
   }

+  partition u-boot {
+    in-partition-table = "no"
+    image = "u-boot_crc.bin"
+    offset = 1536
+  }
+
   partition boot {
     partition-type = 0xC
     bootable = "true"
```

Then the SBC finally booted from the card. Note that `crc32` from `perl-archive-zip` is required to successfully build Advantech's version of U-Boot. Also, the build fails if it has access to the latest version of `dtc`.

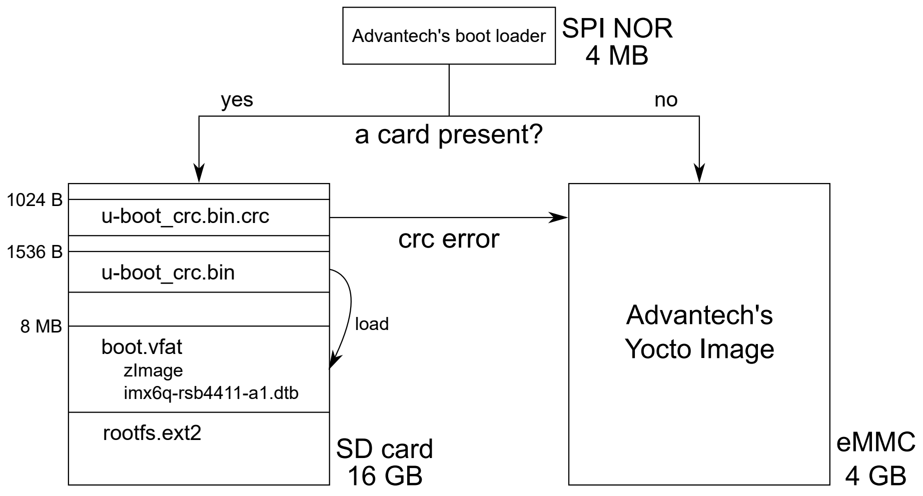


Figure 3.2: SD-card boot process of RSB-4411

3.1.3 Kernel

The Linux kernel is a central part of any Linux OS. It can be built by Buildroot or separately; I decided not to complicate things unnecessarily and chose the former option. Unfortunately, Buildroot's official Linux kernel did not work with RSB-4411.

As I said before, the kernel is configured the same way Buildroot is. There is a configuration file for each supported CPU. Moreover, the kernel utilizes Device Tree files. A board's Device Tree file (.dts) is used to exactly describe and configure all its hardware components. When the kernel is being built, the .dts files are compiled into binary .dtb files which are then stored alongside the kernel image (see 3.2). The kernel loads the hardware configuration from these .dtb files during the boot process; a single kernel binary can thus support multiple board models.

The situation around i.MX-targeted Linux kernel releases is quite complicated. I try to show it in Figure 3.3. There is the official mainline **linux** kernel from `kernel.org`. This version, Buildroot's default, is often updated and even includes some config files for generic i.MX 6 boards. However, RSB-4411 is anything but generic. Every once in a while, Freescale releases its internal **linux-imx** to test new evaluation boards. These releases are static, meaning they receive neither updates nor bug fixes. So, on the one hand there is the mainline kernel with its latest features and bug fixes, on the other hand there is the Freescale release which is designed to perfectly support the i.MX processors. Somewhere in between lie the FSL Community kernels which try to add board support to the the former while update the latter. The situation with U-Boot is the same.

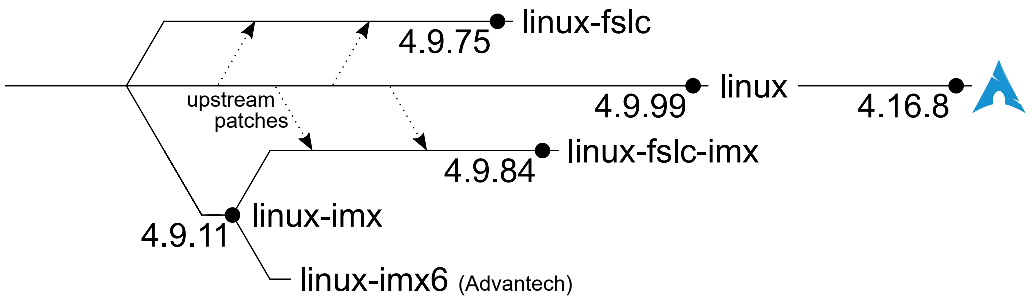


Figure 3.3: Linux kernel forks

As shown in Figure 3.3, Advantech based its Linux kernel on Freescale's static release (see [14]). The kernel is heavily patched: Along with new config and Device Tree files, Advantech has introduced the `CONFIG_ARCH_ADVANTECH` configuration option and many **`#ifdef`**'s throughout the code. Other than that, Advantech 'honours' Freescale's policy and provides no updates whatsoever.

Just like the U-Boot from Advantech, this kernel is intended for Yocto, but it is compatible with Buildroot, too. In order to include the kernel into the system, I set the **Kernel** menu entry as follows:

```
[*] Linux Kernel
      Kernel version (Custom Git repository)  --->
(https://github.com/ADVANTECH-Corp/linux-imx6.git) URL of custom repo
(imx_4.9.11_1.0.0_ga) Custom repository version
(board/merica/imx6q-rsb4411-a1/patches/kernel) Custom kernel patches
      Kernel configuration (Using a custom (def)config file)  --->
(board/merica/imx6q-rsb4411-a1/linux_defconfig) Configuration file path
()    Additional configuration fragment files
      Kernel binary format (zImage)  --->
      Kernel compression format (gzip compression)  --->
[*]   Build a Device Tree Blob (DTB)
(imx6q-rsb4411-a1) In-tree Device Tree Source file names
```

RSB-4411's default configuration file in the kernel tree (`imx_v7_adv_defconfig`) did not suffice our needs; some additional kernel components had to be enabled, so I decided to make a copy. I also created a patch to modify the Device Tree file (`imx6q-rsb4411-a1.dts`). I configured the kernel using these commands:

```
make linux-menuconfig                 #configure the kernel
make linux-update-defconfig           #save the configuration globally
```

3.1.4 Bottom line

All custom packages I have added to Buildroot are located in `package/merica`. In order to display them in the configuration menu I modified `package/Config.in` as follows:

```
@@ -2012,4 +2012,8 @@ menu "Text editors and viewers"
        source "package/vim/Config.in"
 endmenu

+menu "Merica packages"
+    source "package/merica/Config.in"
+endmenu
+
 endmenu
```

This is the only Buildroot file I have altered.

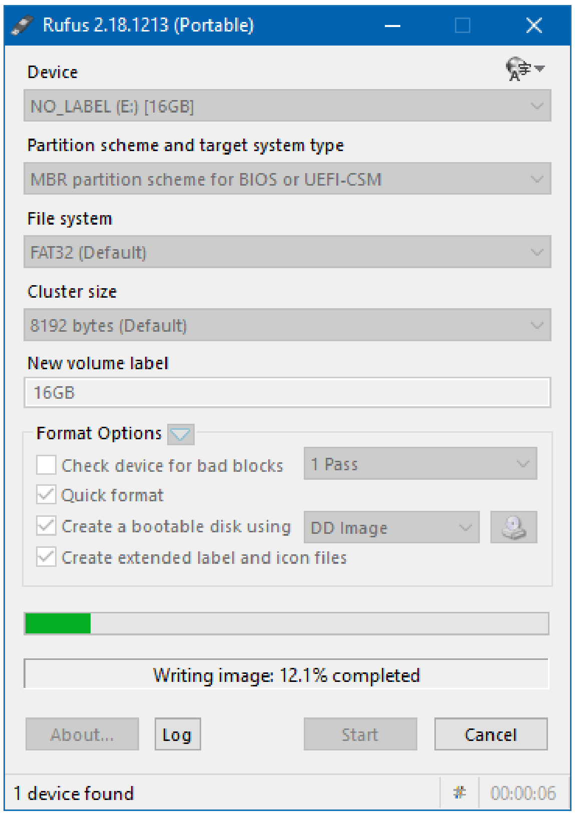
I respected the recommended directory structure one can find at [16]. The following scheme shows all files I have added into the Buildroot tree.

```
|-- board
|   |-- advantech
|   |   `-- imx6q-rsb4411-a1
|   |       |-- genimage.cfg.template
|   |       `-- post-image.sh
|   `-- merica
|       `-- imx6q-rsb4411-a1
|           |-- patches/
|           |-- rootfs_overlay/
|           `-- linux_defconfig
|-- configs
|   |-- imx6q-rsb4411-a1_defconfig
|   `-- merica-terminal_defconfig
`-- package
    |-- merica
    |   |-- <custom packages>
    |   |-- Config.in
    |   `-- merica.mk
    `-- Config.in
```

The complete system can be built as follows:

```
make merica-terminal_defconfig O=../build
cd ../build && make
```

If successful, these commands will result in crating `images/sdcard.img`; a disk image which can be copied onto an actual SD card. Linux users will of course use the dd command; on Windows, I recommend Rufus (`https://rufus.akeo.ie/`).



Rufus 2.18.1213 (Portable)

Device
NO_LABEL (E:) [16GB]

Partition scheme and target system type
MBR partition scheme for BIOS or UEFI-CSM

File system
FAT32 (Default)

Cluster size
8192 bytes (Default)

New volume label
16GB

Format Options
Check device for bad blocks: 1 Pass
[✓] Quick format
[✓] Create a bootable disk using: DD Image
[✓] Create extended label and icon files

Writing image: 12.1% completed

1 device found 00:00:06

Figure 3.4: Rufus

3.2 Peripherals

When the basic OS was up and running, I started to focus on the peripherals: the RFID reader and GPIO. The goal was to somehow deliver data from the peripherals to a web browser.

Both these peripherals, as well as many others, have one thing in common: they are sources of asynchronous events, meaning input data can arrive at any time. Luckily, The Linux kernel provides event notification mechanisms such as select, poll or epoll. My supervisor pointed me towards a library called libev that wraps these mechanisms. The library is really well documented (`man libev`) and simple examples are available, too. There are some alternatives, e.g. libevent and libuv, but I found libev the easiest to learn. With libev, all I had to do was add event watchers into a libev loop and write callback functions.

I designed the peripheral-event handlers to be modular. Each of them exists as a standalone binary, but they can be easily integrated into a single libev loop. The all-handling application is depicted in Figure 3.5. The following subsections discuss each of the components. All of them can be compiled and installed through a Buildroot package called `mt-apps`.

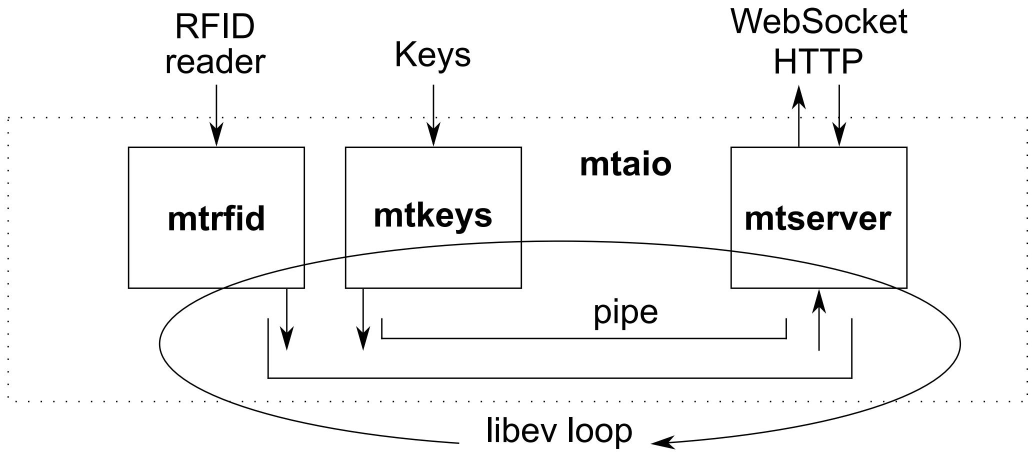


Figure 3.5: Scheme of the complete I/O handler

3.2.1 RFID

Digital Logic's software support I mentioned in subsection 2.3.2 is excellent. One can download the uFCoder library (also called ufr-lib), documentation and code examples at [18] for free. The repository is often updated, too. I started with the Python examples and in a few minutes had µFR Nano up and running.

The reader incorporates FTDI's USB-to-serial chip and thus shows up as a serial device when connected: /dev/ttyUSB0, COM9, etc. It communicates via D-Logic's proprietary protocol; complete documentation thereof is also available at [18]. The uFCoder library implements the protocol and provides an easy access to all µFR Nano's features. The library itself is not an open-source software, but it comes with a C header file and provides a well-described API. I decided not to waste time reimplementing the protocol and based my program on uFCoder.

As stated by the API reference from [18], the library is able to communicate with the reader either via FTDI's D2XX driver calls or directly via a serial port. The former option, though successful on Windows, failed on the embedded Linux system. The library did not find files from some archaic version of the driver. The latter option worked out of the box; to enable it I called the advanced opening function as follows:

```
ReaderOpenEx(
    1,              //uFR type (1 Mbps)
    "/dev/ttyUSB0",
    1,              //serial port interface
    NULL            //reserved
);
```

One of the reader advantages I listed in subsection 2.3.2 was the ability to send card UID's asynchronously. If this feature is enabled, whenever the reader detects a new card, it sends the card's UID and thus notifies the host. Each UID is sent as a hexadecimal string with a suffix and optionally a prefix. The API reference explains the feature further and gives the following example: Assuming `'X'` is the prefix and `'Y'` is the suffix, the 4-byte UID 0xA103C256 will be sent as `"XA103C256Y"`. To enable the feature I called the following function:

```
SetAsyncCardIdSendConfig(
     1,             //enable asynchronous UID sending
```

```
        0,             //disable prefix
        0,             //prefix
        0,             //suffix
        0,             //disable sending on card removals
        1000000        //baud rate
    );
```

I disabled the prefixing and set the suffix to zero so the reader would send proper C strings. Unfortunately, uFCoder does not provide any means to receive these asynchronous events (yet). I had to come up with a workaround.

This is when libev comes on the scene. In order to employ a libev watcher, however, I needed a file descriptor. Since uFCoder hides its internals, I decided to open the serial device once more, this time using standard Linux functions. The linux-uart-keyboard-emualation project at [18] was helpful here. The following code is heavily simplified to show just the most important lines.

```
int fd = open("/dev/ttyUSB0", O_RDONLY | O_NOCTTY);

//disable rts
int uart_status;
ioctl(fd, TIOCMGET, &uart_status);
uart_status &= ~TIOCM_RTS;
ioctl(fd, TIOCMSET, &uart_status);

//set baud rate
unsigned br = B1000000;
struct termios options;
tcgetattr(fd, &options);
cfsetispeed(&options, br); //input
cfsetospeed(&options, br); //output
tcsetattr(fd, TCSANOW, &options);

usleep(1200000);
tcflush(fd, TCIFLUSH);
```

With a file descriptor open and configured, it was an easy task to define a callback function and setup a watcher. In the callback, the program reads bytes from `fd` using the `read` function. It stores characters until the suffix (zero) is read. After reading the whole UID, the program begins communicating with the reader using uFCoder functions and outputs card information. For our purposes, it would be sufficient to output the UID alone; however, I wanted to be prepared for more complex scenarios.

The program outputs card data as JSON, so they are easily parsable in a web browser. The JSON string always contains the `"type":"rfid"` pair which allows the browser to identify the message. The `"uid"` attribute stores a card's UID as a string. The output thus may look as follows:

```
{"type": "rfid", "uid": "A103C256"}
```

I created a standalone version of the program, which prints card information to the standard output. Figure 3.6 presents it in a graphical form. This version is compiled as a part of the `mt-apps` package I've added into the system. It can be launched using the following command:

```
mtrfid                  #open the RFID reader and print card info to stdout
```

One can also include the header file, `mt_rfid.h`, and use the functions below to integrate the program into one's own libev loop and set an output file descriptor.

```
int mt_rfid_init(mt_rfid_t *self, struct ev_loop *loop, int fd);
void mt_rfid_deinit(mt_rfid_t *self);
```

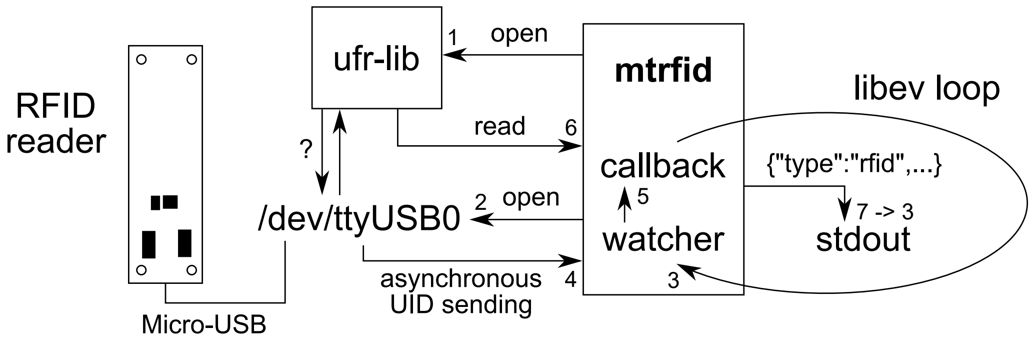


Figure 3.6: Scheme of the RFID handler

3.2.2 GPIO

The Linux kernel provides two GPIO user APIs. The Internet is full of tutorials for the integer-based `/sys/class/gpio` interface. However, this API has been deprecated since Linux 4.8. This kernel version introduced a new descriptor-based interface. Each GPIO chip, consisting of several lines, appears in the system as a character device `/dev/gpiochipN`. Materials concerning the new interface are scarce, but [19] provides a nice introduction and shows many usage examples. It also presents the libgpiod library, which wraps the new interface and comes with handy utilities like `gpioinfo`, `gpioget` or `gpioset`.

Libgpiod was available as a Buildroot package, so I decided to try it. The simplified code below shows an example of how to set up reading of line events. The code works with libgpiod v0.3.2, which is included in Buildroot 2018.02.x; the latest version of the library (v1.0.1) renamed the request function.

```
//open a GPIO line
unsigned chip_number = 0, line_offset = 27;
struct gpiod_chip *chip = gpiod_chip_open_by_number(chip_number);
struct gpiod_line *line = gpiod_chip_get_line(chip, line_offset);

//request button push events, get a file descriptor
gpiod_line_event_request_rising(line, "mtgpio", GPIO_ACTIVE_LOW);
int fd = gpiod_line_event_get_fd(line);

//read the line event, e.g. in a libev callback
struct gpiod_line_event e;
gpiod_line_event_read_fd(fd, &e);

gpiod_chip_close(chip);
```

However, one does not simply read button events. Due to a problem called contact bounce, a single button press often results in several signal changes. Unless the problem is solved in hardware, a button-handling software must support debouncing. That usually means reading the signal multiple times over a large enough time interval. I really didn't like the idea of combining debouncing with libev. Luckily, there was another option available.

Instead of using the user-space API, I decided to follow the 'you do not access GPIOs from userspace' rule and employ one of the kernel GPIO drivers. The driver is called gpio-keys, supports debounce and practically transforms GPIO lines into a keyboard (i.e. an event device in `/dev/input`). All I had to do to enable the driver was adding a gpio-keys block into the board's Device Tree file. An example follows:

```
gpio-keys {
    compatible = "gpio-keys";
    #address-cells = <1>;
    #size-cells = <0>;
    label = "gpio-keys";
    gpio01 {
        label = "gpio01";
        gpios = <&gpio1 27 GPIO_ACTIVE_LOW>;
        linux,code = <BTN_TRIGGER_HAPPY1>;
        gpio-key,wakeup;
    };
};
```

The example shows only the first GPIO pin (`gpio01`); the rest is added the same way. Note that `&gpio1` refers to `/dev/gpiochip0` etc. Since the driver makes the buttons act as an actual keyboard, there might be a problem with other applications reading it. That is why I chose the `BTN_TRIGGER_HAPPY` codes; there are 40 of these in `linux/input-event-codes.h` and I didn't observe any undesired reactions.

As I have said, the driver creates an event device, so it is quite easy to set up a libev watcher. An example of opening and reading the device follows.

```
int fd = open("/dev/input/by-path/platform-gpio-keys-event", O_RDONLY);

//in a callback
struct input_event e;
read(fd, &e, sizeof(e));
```

I created a libev-based program that reads the gpio-keys events and outputs JSON-formatted strings. The strings can be identified by the `"type":"keys"` pair. Each `BTN_TRIGGER_HAPPY` code is converted to an upper-case character and stored in the `"key"` attribute. For example, pressing the `gpio01` key results in the following output:

```
{"type": "keys", "key": "A"}
```

The standalone version of the program, that prints the strings to the standard output, is shown in Figure 3.7. It can be launched as follows:

```
mtrfid                        #watch GPIO key-presses and print info to stdout
```

Similarly to the RFID watcher discussed in subsection 3.2.1, this program can be easily integrated into a foreign code by including `mt_keys.h` and using the functions below.

```
int mt_keys_init(mt_keys_t *self, struct ev_loop *loop, int fd);
```

```
void mt_keys_deinit(mt_keys_t *self);
```

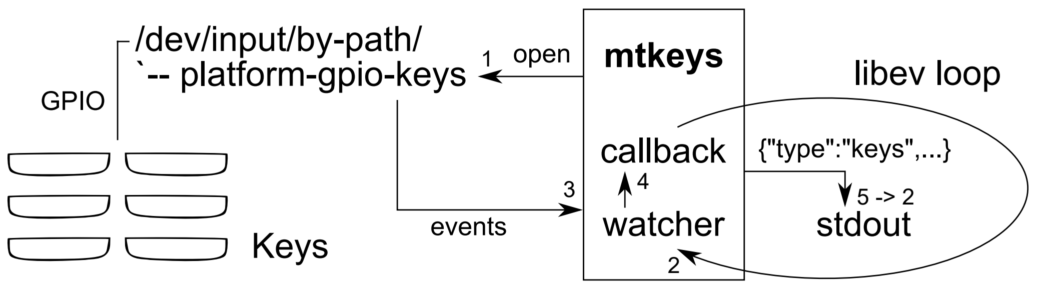


Figure 3.7: Scheme of the GPIO handler

3.2.3 WebSocket

My supervisor recommended sending data to a web browser via the WebSocket protocol. WebSocket provides real-time full-duplex communication over a single persistent TCP connection. It seemed like the right choice. The protocol is implemented by a big number of libraries (see [20] for a comprehensive list). I chose **libwebsockets** ([21]) for the following reasons:

- lightweight and portable library written in C
- supports HTTP, WebSocket, TLS and other protocols
- compatible with libev
- active development, cutting-edge technologies
- available as a Buildroot package
- popular among my colleagues at Merica

Sadly, the library's Doxygen-generated documentation is quite complicated and, due to the active development, most tutorials one can find on the Internet are outdated and useless. Perhaps the best starting point is the series of examples that comes with the latest version of the library (3.0). However, at the time of writing this, that version is just one week old and the previous release, 2.4, which is included in Buildroot 2018.02, is not fully compatible. Using the library can thus be challenging.

When deploying a libwebsockets server, one has to define some protocols. The first protocol must always be HTTP. The minimal-ws-server example from [21] was my inspiration here. I defined the protocols as follows:

```
enum protocols {
    PROTOCOL_HTTP = 0, // always first
    PROTOCOL_MERICA_TERMINAL
};

// list of supported protocols
static struct lws_protocols protocols[] = {
    {"http", lws_callback_http_dummy, 0, 0},
    {
        "merica-terminal-protocol", //protocol name
        callback_merica_terminal,   //callback function
        0,                          //per-session data size
        128                         //sufficient size to store messages
    },
    {NULL, NULL, 0, 0}              //terminator
};
```

Each protocol requires a callback. The library already implements an HTTP callback function called `lws_callback_http_dummy`, so I took advantage of that. I only had to provide some parameters through a **`struct`** `lws_http_mount` constant. I set it as follows to make the server respond to HTTP requests with `/usr/share/mtserver/index.html`.

```
const struct lws_http_mount mount = {0};
mount.mountpoint      = "/";
mount.origin          = "/usr/share/mtserver";
mount.def             = "index.html";
mount.origin_protocol = LWSMPRO_FILE; //serve files from origin
mount.mountpoint_len  = 1;
```

The `callback_merica_terminal` function will be described later. Now, the following code example shows how to run a libwebsockets server from a libev loop.

```
struct lws_context_creation_info info;
memset(&info, 0, sizeof(info));

info.port = 80;
info.mounts = &mount;
info.protocols = protocols;
info.max_http_header_pool = 1;
info.options |= LWS_SERVER_OPTION_LIBEV;

struct lws_context *context = lws_create_context(&info);
struct ev_loop *loop = EV_DEFAULT;
lws_ev_initloop(context, loop, 0);
```

My goal was to read data from a file descriptor and send them via merica-terminal-protocol. Since I'd already had a libev loop running, it was no problam to add a file descriptor watcher. However, it had to somehow share data with the server. I chose the following approach:

```
//initialization, info set
queue *q = empty_queue();
info.user = q;
//create context from info
fd_watcher->context = context;
fd_watcher->line =
    (char *)malloc((LWS_PRE + INPUT_LINE_LENGTH)*sizeof(char));
fd_watcher->line += LWS_PRE;
fd_watcher->queue = q;

//fd_watcher callback, after reading data to fd_watcher->line
queue_push(fd_watcher->queue, fd_watcher->line);
//allocate a new line
lws_callback_on_writable_all_protocol(
    fd_watcher->context,
    &protocols[PROTOCOL_MERICA_TERMINAL]);

//deinitialization
free(fd_watcher->line - LWS_PRE);
queue_destroy(fd_watcher->queue);
```

This way, the input watcher is able to notify the server and the data can be accessed from the protocol callback. Moreover, the queue prevents overwriting unsent data.

The last missing piece was the WebSocket protocol callback. Once again, I took inspiration from the minimal-ws-server example and just adjusted important parts. An excerpt from the code of the `callback_merica_terminal` function follows.

```c
static int callback_merica_terminal(struct lws *wsi,
                                    enum lws_callback_reasons reason,
                                    void *user, void *in, size_t len)
{
    struct lws_context *context = lws_get_context(wsi);
    queue *q = (queue *)lws_context_user(context);
    char *line;

    switch (reason) {
        case LWS_CALLBACK_PROTOCOL_INIT:
            //copied from the example

        case LWS_CALLBACK_SERVER_WRITEABLE:
            //send the input data
            line = queue_pop(q);
            lws_write(wsi, (unsigned char *) line,
                      strlen(line), LWS_WRITE_TEXT);
            break;

        case LWS_CALLBACK_RECEIVE:
            //parse client's response
            if (strcmp((const char *)in, "reset") == 0) {
                queue_push_copy(q, JSON_EMPTY);
                lws_callback_on_writable_all_protocol(context,
                    &protocols[PROTOCOL_MERICA_TERMINAL]);
            } else if (strcmp((const char *)in, "close") == 0) {
                lws_close_reason(wsi, LWS_CLOSE_STATUS_GOINGAWAY,
                                 (unsigned char *)"seeya", 5);
                return -1;
            }
            break;
    }

    return 0;
}
```

As I have already mentioned, the HTTP server responds with an `index.html` file. This HTML page has two purposes. First, it enables browsers to open a WebSocket connection to the server. This is achieved by defining a WebSocket variable called `socket` as well as its `onmessage` function. Secondly, the file provides a link to a remote script. The script is expected to define the `updateRemote` function, which is then called from `socket.onmessage` to handle incoming WebSocket data. Whenever the `updateRemote` function is not defined, `socket.onmessage` stores incoming strings into a persistent localStorage. The HTML document also includes the `<div id="remote"></div>` section for the remote script to modify. But of course, the script can do whatever it wants; it's JavaScript. A simplified version of `index.html` follows.

```html
<!doctype html>
<html>
<script type="text/javascript"
```

```
        src="//dejvice.merica.cz:5000/js" async></script>
<script>
    var socket = new WebSocket(
        "ws://" + document.domain + ':' + location.port,
        "merica-terminal-protocol"
    );

    socket.onopen    = function() {console.log("socket open");}
    socket.onclose   = function() {console.log("socket closed");}
    socket.onmessage = function(msg) {
        if (typeof updateRemote === "function") {
            updateRemote(msg.data);
        } else if (localStorage) {
            localStorage.setItem(Date.now(), msg.data);
        }
    }

    function sendReset() {socket.send("reset");}
    function sendClose() {socket.send("close");}
</script>

<body>
<div id="local">
    <button onclick="sendReset()">reset</button>
    <button onclick="sendClose()">close</button>
</div><hr>
<div id="remote">all alone in a danger zone...</div>
</body>
</html>
```

Just like the previous programs (subsections 3.2.1 and 3.2.2), the server is available in the `mt-apps` package as a standalone binary. It can also be integrated into a foreign code: by including `mt_server.h`. See below. A graphical representation of the server is shown in Figure 3.8.

```
mtserver        #start an HTTP server and send data from stdin via WebSocket
```

```
int mt_server_init(mt_server_t *self, struct ev_loop *loop, int fd);
void mt_server_deinit(mt_server_t *self);
```

3.2.4 All in one

As shown in Figure 3.5, all the previously discussed programs are integrated into a single binary. The all-in-one program thus handles HTTP and WebSocket connections as well as RFID and GPIO events. Integrating the components is really easy; the following code is only slightly simplified.

```
struct ev_loop *loop = EV_DEFAULT;
int pipefd[2]; //read <- write
mt_rfid_t rfid;
mt_keys_t keys;
mt_server_t server;

pipe(pipefd);
```

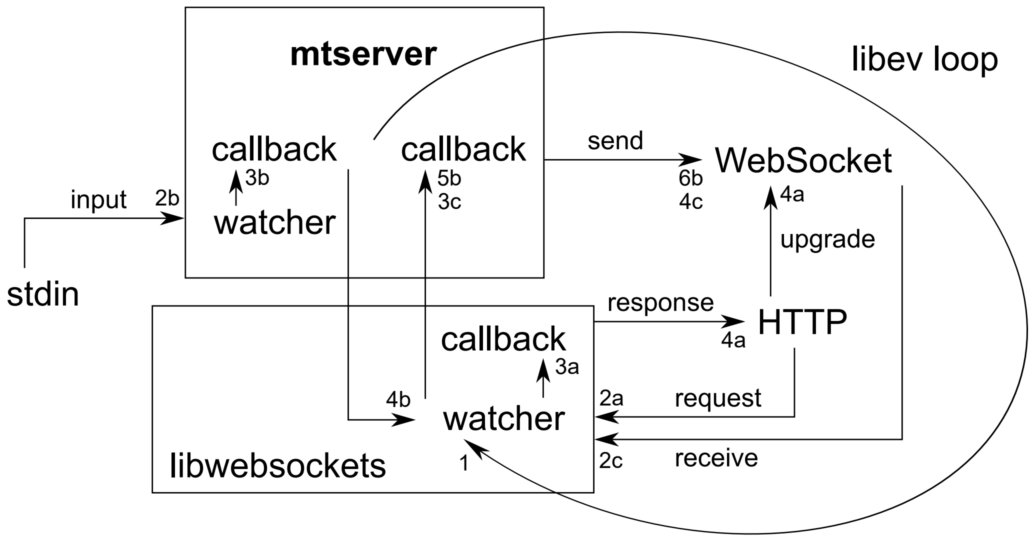


Figure 3.8: Scheme of the WebSocket server

```
mt_rfid_init(&rfid, loop, pipefd[1]);
mt_keys_init(&keys, loop, pipefd[1]);
mt_server_init(&server, loop, pipefd[0]);

ev_run(loop, 0);

mt_server_deinit(&server);
mt_keys_deinit(&keys);
mt_rfid_deinit(&rfid);
```

The binary is included in the `mt-apps` package. It is also launched as a service on system start-up and automatically restarted on failures (not that I observed any). I achieved this by appending the line below to `/etc/inittab`.

```
::respawn:/usr/bin/mtaio
```

The extended file is located in the `rootfs_overlay` directory.

3.3 Networking

Another important task was connecting the SBC to the Internet. The Ethernet connection worked out of the box; I only acquired an IP address via DHCP as follows:

```
udhcpc -i eth0                                    #acquire an IP address
```

Buildroot also provides a way to automate the process by setting the following `menuconfig` option:

```
System configuration
(eth0) Network interface to configure through DHCP
```

However, I knew the portable terminal would have to be connected via WiFi. And configuring WiFi is always a little more difficult.

3.3.1 WiFi

In order to use a PCIe WiFi card, some modifications to the kernel configuration were necessary. I had to enable a driver for the i.MX6 PCIe host controller and and the card drivers as well. Since my WiFi card was made by Intel (see subsection 2.4.1), I enabled the iwlwifi driver. The modifications added the following lines to the kernel config file:

```
CONFIG_PCI_MSI=y
CONFIG_PCI_IMX6=y
CONFIG_IWLWIFI=m
CONFIG_IWLDVM=m
CONFIG_IWLMVM=m
```

I also had to add the card's firmware. That can be done in Buildroot's `menuconfig` as follows:

```
Target packages → Hardware handling → Firmware
[*] linux-firmware
      WiFi firmware  --->
          [*] Intel iwlwifi 7260
```

When the `lspci` command showed some devices and both `lsmod` and `dmesg` mentioned iwlwifi, I knew all drivers were successfully installed, so I moved on.

Most of today's WiFi networks use some security protocols; therefore I installed wpa_supplicant, which is able to connect to these protected networks. It comes with a lot of optional features and I enabled most of them. See an excerpt from the configuration below.

```
Target packages → Networking applications
[*] wpa_supplicant
[*]   Enable nl80211 support
[*]   Enable autoscan
[*]   Enable EAP
[*]   Enable HS20
[*]   Install wpa_passphrase binary
```

With wpa_supplicant in the system, one can run the following sequence of commands to connect to a WiFi network:

```
ip link set wlp1s0 up                               #bring the card up
wpa_passphrase MYSSID passphrase > /tmp/wpa_s.conf  #create a configuration
wpa_supplicant -B -i wlp1s0 -c /tmp/wpa_s.conf      #connect to the network
udhcpc -i wlp1s0                                    #acquire an IP address
```

The configuration file `/tmp/wpa_s.conf` looks as follows:

```
network={
    ssid="MYSSID"
    #psk="passphrase"
    psk=59e0d07fa4c7741797a4e394f38a5c321e3bed51d54ad5fcbd3f84bc7415d73d
}
```

To make the configuration permanent, I saved the file as `etc/wpa_supplicant.conf` to the board's `rootfs_overlay` directory in the Buildroot tree. The same directory also contains an overlay of `/etc/network/interfaces`. This file is used to set up networks automatically on system start-up. I extended it by the following lines:

```
auto wlp1s0
iface wlp1s0 inet dhcp
  wireless-essid "MYSSID"
  wireless-key "passphrase"
  pre-up wpa_supplicant -i wlp1s0 -c /etc/wpa_supplicant.conf -B
  post-down killall -q wpa_supplicant
```

3.4 Graphics

Finally, it was time to make use of all the graphics accelerators I listed in subsection 2.1.1 and deploy a hardware-accelerated graphical web browser. All required kernel drivers had already been enabled by the SBC's configuration file. I checked that in under the following `linux-menuconfig` entries:

```
Device Drivers → MXC support drivers
Device Drivers → Graphics support → Frame buffer Devices
```

The Linux kernel provides an abstraction for graphics hardware in the form of the frame buffer devices. They allow 'application software to access the graphics hardware through a well-defined interface', says [22], 'so that the software is not required to know anything about the low-level hardware registers'. On the i.MX 6Quad platform, each IPU display interface (see subsection 2.1.1) can be associated with a `/dev/fb*` file via Freescale's MXC driver. Table 3.1 shows how the frame buffer devices are set on RSB-4411; this is done in the SBC's Device Tree file. The kernel also provides a newer abstraction layer called DRM, which solves the problem of multiple applications using a single frame buffer device simultaneously. However, Advantech didn't patch the DRM drivers and I wasn't even able to compile them.

Display interface	Frame buffer device	
HDMI	`/dev/fb0`	
VGA	`/dev/fb1`	
LVDS0	`/dev/fb2`	`/dev/fb2` (split mode)
LVDS1	`/dev/fb3`	

Table 3.1: Frame buffer devices on RSB-4411

At first, I took advantage of the display control board, discussed in subsection 2.2.3, and connected the display via HDMI. Freescale's HDMI driver lacks WUXGA support, but the control board was able to adapt to the resolution of 1920×1080 as well. In this section, I will describe all steps I had to take to run a touch-enabled web browser.

3.4.1 Hardwre acceleration

As I said in subsection 2.1.1, i.MX 6Quad features a GPU able to accelerate graphics rendering through the OpenGL ES API. The unit is a Vivante model. 'The GPU driver is divided into two layers', says [22]. One layer runs in kernel mode. This layer of the driver is included into the kernel tree and compiled as a part of the kernel (see the `linux-menuconfig` entries I mentioned earlier). The other layer is a library that runs in user mode provides all the APIs. The library is distributed only as a binary. There is also an alternative reverse-engineered GPU driver called Etnaviv.

The VPU driver is also included in the kernel tree. In addition, this unit, which provides hardware accelerated video decoding, requires firmware. Freescale provides a user-space helper library for VPU, too. This library is used e.g. by the GStreamer plugins from [23] to facilitate hardware-accelerated video playback.

I added the libraries to the Buildroot system as follows:

```
Target packages → Hardware handling → Freescale i.MX libraries
i.MX platform (imx6q/imx6dl)  --->
[*]   imx-codec
-*-   imx-vpu
-*-   firmware-imx
[*]   imx-gpu-g2d
-*-   imx-gpu-viv
        Output option (Framebuffer)  --->
```

When I first installed the libraries, I was still using Buildroot 2017.11.x. Back then, no OpenGL application would start and I was unable to fix it. Luckily, around that time Buildroot ‘released another dragon’, meaning version 2018.02 came out. Before trying anything else, I updated the system, and suddenly everything worked like a charm. The failing was probably caused by the old GPU library version not being compatible with the kernel driver.

I installed the examples that come with the GPU library and used them for some testing. I also installed GStreamer (`gstreamer 1.x`) along with the plugins from [23] I mentioned (`gst1-imx`) and was able to smoothly play a 1080p video. See some example commands below.

```
/usr/share/examples/viv_samples/vdk/tutorial1       #display some animations

gst-launch-1.0 \                                    #play a video
  filesrc location=/mnt/video.h264 \
  ! h264parse ! imxvpudec ! imxg2dvideosink sync=false async=false
```

3.4.2 Candidates

When I started the development, I had no experience of embedded Linux. My initial idea thus was to deploy software I would install on desktop: the Chromium web browser, which uses the GTK+ graphical library, running on top of the X Window System (X). However, I quickly realized this wouldn’t do. Embedded Linux systems have little in common with desktop-targeted Linux distributions. First of all, using a touchscreen instead of a keyboard and a mouse asks for a completely different user interface.

Another aspect I had to consider was the complexity of the system. After all, we only wanted to display a single web page. I realized installing a full-featured browser would be an overkill; and probably mistake. The less users are allowed to do, the less likely they break something. Besides, neither Chromium nor any other ‘high’ browser is available as a Buildroot package.

In addition to a web browser, I had to install a usable on-screen keyboard. This is something quite common in smartphone OSs; however, on Linux, quality options are scarce.

Thanks to Advantech’s Yocto image, I was able to try the Matchbox window manager running on top of X. When I tried the GUI on the touchscreen display, it felt clumsy and I spent most of the time trying to hit the close button. The window manager features an on-screen keyboard, but that wasn’t satisfactory either. It was clear X wasn’t the most suitable option and I had to look for a more touch-friendly interface.

3.4.3 Qt

Qt is a cross-platform framework for creating graphical applications. When viewing Qt’s website at qt.io, I realized this was the way to go. These are the main reasons I chose Qt:

- Qt targets embedded systems.
- Qt’s components cover all areas of graphical software development.
- Qt is fast and simple.
- Qt is included in Buildroot.
- Qt is widely used in industrial applications.

I found an excellent guide at [24]. The page introduces Qt’s component called EGLFS: ‘a platform plugin for running Qt5 applications on top of EGL and OpenGL ES 2.0 without an actual windowing system’. EGLFS ‘is the recommended plugin for modern Embedded Linux devices that include a GPU’, continues [24]. On the i.MX 6Qaud platform, EGLFS is able to utilize the Vivante drivers.

The best thing about Qt and EGLFS is that all configuration is done by Buildroot automatically. I only enabled Qt5 as follows:

```
Target packages → Graphic libraries and applications (graphic/text) → Qt5
      Qt5 version (Latest (5.9))  --->
-*-   qt5base
[*]     Compile and install examples (with code)
        SQLite 3 support (System SQLite)  --->
-*-     gui module
-*-       widgets module
-*-       OpenGL support
            OpenGL API (OpenGL ES 2.0+)  --->
-*-       eglfs support
[*]       GIF support
[*]       JPEG support
[*]       PNG support
[*]   qt5virtualkeyboard
(en_GB) language layouts
[ ]     handwriting
[*]   qt5webengine
[*]     proprietary codecs
[*]     alsa
```

Two of the Qt components interested me the most: Qt WebEngine and Qt Virtual Keyboard. Qt WebEngine is a module that integrates the Chromium engine into Qt. That means it provides fast hardware-accelerated web-page rendering and supports latest web technologies. The only downside of Qt WebEngine is that hardware-accelerated video decoding does not work. However, that might change in future, as [25] suggests. There is also the Qt WebKit module, but that one is obsolete. Qt Virtual Keyboard is a nice on-screen keyboard that can be easily integrated into any Qt application. As of Qt 5.10, it supports Czech keyboard as well.

Finally, I had a pretty clear idea of what I was going to deploy: A Qt WebEngine-based browser with Qt Virtual Keyboard support running on top of EGLFS.

3.4.4 Touchscreen

Thanks to its USB interface, there was no problem with connecting the touchscreen and Qt instantly recognized it as an input device. However, multi-touch gestures didn't work at first, because the touchscreen showed up as a generic mouse. I searched through the kernel configuration and enabled the following options:

```
CONFIG_INPUT_TOUCHSCREEN=y
CONFIG_TOUCHSCREEN_USB_COMPOSITE=y
CONFIG_HID_MULTITOUCH=y
```

After that, multi-touch started working. I tested it using the `fingerpaint` Qt demo (see the command below) and I was able to paint with all 10 fingers simultaneously. I am not sure which of the options were really necessary though.

```
/usr/lib/qt/examples/touch/fingerpaint/fingerpaint      #paint with fingers!
```

Qt can handle input events on its own (via the evdev interface). In case this approach doesn't work, the manual at [24] mentions alternative libraries like `tslib` or `libinput`.

3.4.5 Web browser

While searching the Internet, I discovered Qt WebBrowser at [26] which exactly matched my expectations. This simple web browser serves as a demonstration of Qt's capabilities, including Qt WebEngine and Qt Virtual Keyboard. Qt WebBrowser was not included in Buildroot, so I copy-pasted some other Qt package files and added it manually.

However, the application disappointed me. I fixed some really annoying bugs, but it was still quite slow and laggy. In the end, I decided to choose a different approach. However, Qt WebBrowser is still available in the system: in can be enabled among Merica packages and launched as follows:

```
qtwebbrowser                                      #launch Qt's demo web browser
```

Qt WebEngine comes with a few examples. I took the simplest one (basically just a full-screen browser window), added virtual keyboard according to the Deployment Guide and designed a simple toolbar using QML. My browser, of course, doesn't support any advanced features, but it launches quickly and scrolls web pages smoothly. Moreover, it is so simple anyone could improve it. I set `http://localhost/` as a starting webpage. See the screenshot in Figure 3.9a.

Except for the video decoding I mentioned in subsection 3.4.3, the browser is fully hardware accelerated. The status is shown Figure 3.9b. I tested the acceleration also at helloracer.com/webgl and the Ferrari was being rendered really beautifully and quickly.

The browser can be installed via the `mtbrowser` package and launched using the following script:

```
mtbrowser.sh
```

I wrote this wrapper script to set some Qt parameters and enable CORS (see section 4.1). Just like the program I described in subsection 3.2.4, the browser is launched automatically on system start-up.

In addition to the browsers, I installed some fonts and some certificates for HTTP Secure:

```
Target packages → Fonts, cursors, icons, sounds and themes
[*] DejaVu fonts
[*]   mono fonts
[*]   sans fonts
[*]   serif fonts
[*]   sans condensed fonts
[*]   serif condensed fonts

Target packages → Libraries → Crypto
[*] CA Certificates
```

3.4.6 LVDS configuration

Finally, I had to set up the LVDS connection. This required several changes throughout the board's Device Tree file. The guide at [27] was a big help here. This was the most difficult task to set up right, because I simply didn't know whether I configured something wrong, made a bad cable or even bricked the display.

I began by copying the timing values listed in [8]. It didn't work. Because I thought there might be a problem with the EDID data not being sent (see 2.9), I took the EDID numbers from [8] and put them into a binary file. While checking the file using the `parse-edid` command (part of the `read-edid` package), I noticed the timing values were quite different. I tried the values from the EDID file and this time the display successfully woke up

My Device Tree configuration follows:

```
mxcfb2: fb@1 {
    status = "disabled"; /* hdmi + lvds = no vga */
};

mxcfb3: fb@2 {
   compatible = "fsl,mxc_sdc_fb";
   disp_dev = "ldb";
   interface_pix_fmt = "RGB24";
   default_bpp = <32>;
   int_clk = <0>;
   late_init = <0>;
   status = "okay";
};

&ldb {
    status = "okay";
    split-mode; /* dual-channel setting */

    lvds-channel@0 {
        fsl,data-mapping = "spwg";
        fsl,data-width = <24>;
        crtc = "ipu2-di0";
        primary;
        status = "okay";

        display-timings {
            native-mode = <&timingauo>;

            timingauo: b101uan021 {
                clock-frequency = <152720000>;
                hactive = <1920>;
                vactive = <1200>;
                hback-porch = <60>;
                hfront-porch = <60>;
                vback-porch = <30>;
                vfront-porch = <10>;
                hsync-len = <60>;
                vsync-len = <10>;
            };

        };
    };
};
```

First I tested the display configuration using the commands below.

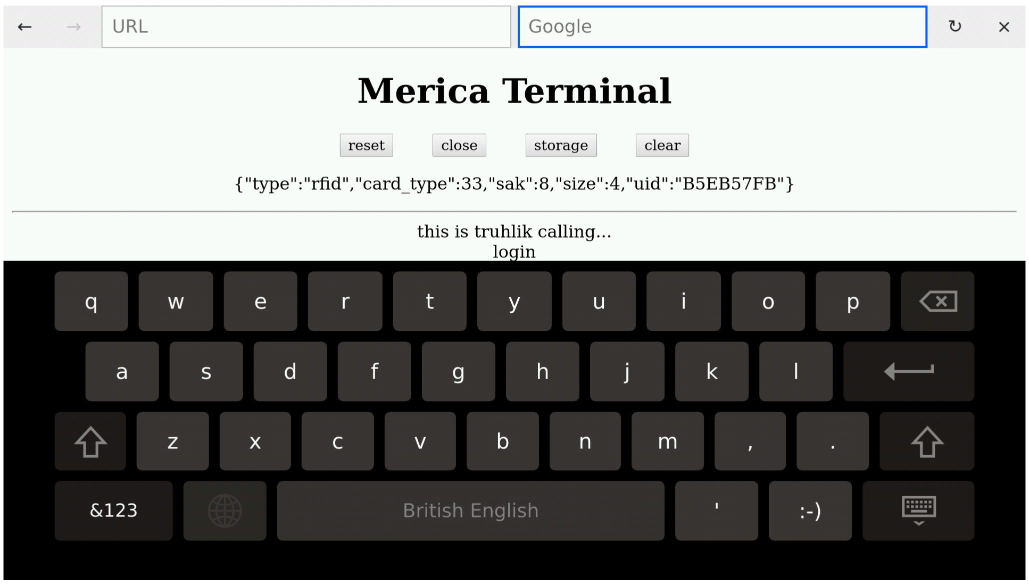
```
fbset -fb /dev/fb2                           #check display configuration
cat /dev/urandom > /dev/fb2                  #check that the display lives
```

The fbset produced the following output:

```
mode "1920x1200-60"
        # D: 157.604 MHz, H: 75.050 kHz, V: 60.040 Hz
        geometry 1920 1200 1920 2400 32
        timings 6345 60 60 30 10 60 10
        accel false
        rgba 8/16,8/8,8/0,8/24
endmode
```

From there, it was a small step towards launching the browser. I only had to set some Qt variables, as mentioned at [24], and everything started working like a charm.

```
export QT_QPA_EGLFS_FB=/dev/fb2
export QT_QPA_EGLFS_PHYSICAL_WIDTH=216
export QT_QPA_EGLFS_PHYSICAL_HEIGHT=135
```

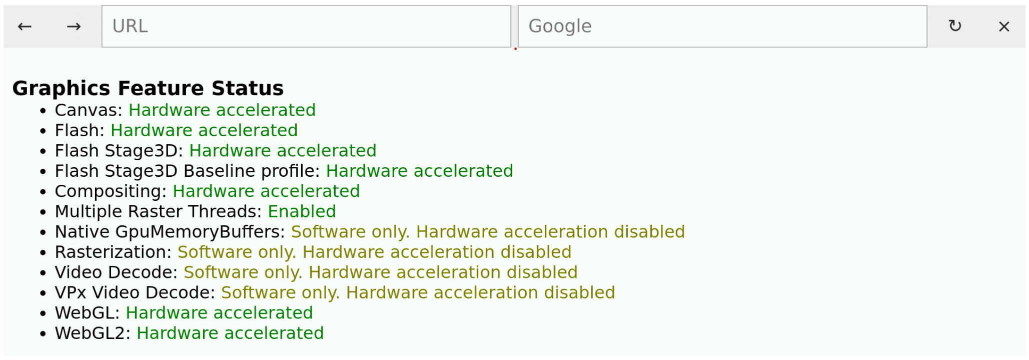


Merica Terminal

{"type":"rfid","card_type":33,"sak":8,"size":4,"uid":"B5EB57FB"}

this is truhlik calling...
login

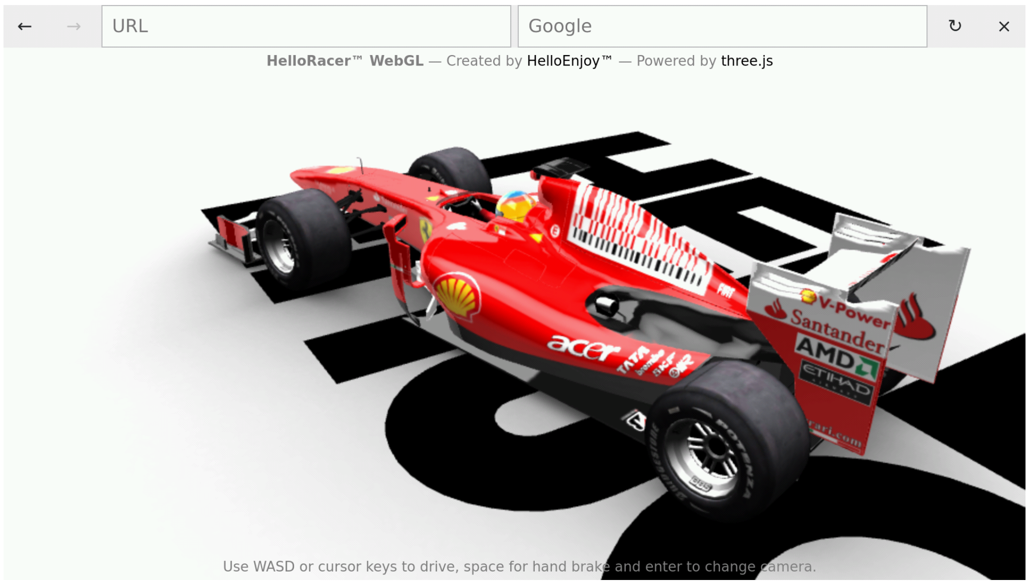
(a) On-screen keyboard



Graphics Feature Status

- Canvas: Hardware accelerated
- Flash: Hardware accelerated
- Flash Stage3D: Hardware accelerated
- Flash Stage3D Baseline profile: Hardware accelerated
- Compositing: Hardware accelerated
- Multiple Raster Threads: Enabled
- Native GpuMemoryBuffers: Software only. Hardware acceleration disabled
- Rasterization: Software only. Hardware acceleration disabled
- Video Decode: Software only. Hardware acceleration disabled
- VPx Video Decode: Software only. Hardware acceleration disabled
- WebGL: Hardware accelerated
- WebGL2: Hardware accelerated

(b) Hardware acceleration status at chrome:gpu


←
→
URL
Google
↻
×
HelloRacer™ WebGL — Created by HelloEnjoy™ — Powered by three.js
Use WASD or cursor keys to drive, space for hand brake and enter to change camera.

(c) WebGL demo

Figure 3.9: Screenshots of the web browser

Chapter 4

Testing

We tested the terminal with the Philips Saeco Incanto coffee machine as we had planned. We opened the machine and my supervisor was able to solder wire connections between the machine's control board and a flat cable connector. See the result in Figure 4.1. Then we connected the machine's buttons to the SBC's GPIO pins through the level shifter (see section 2.4 for details).

The last task for me was to develop a server application for demonstration. The application would collect data from the coffee machine and the terminal, maintain a database of coffee-drinking users and show some statistics.

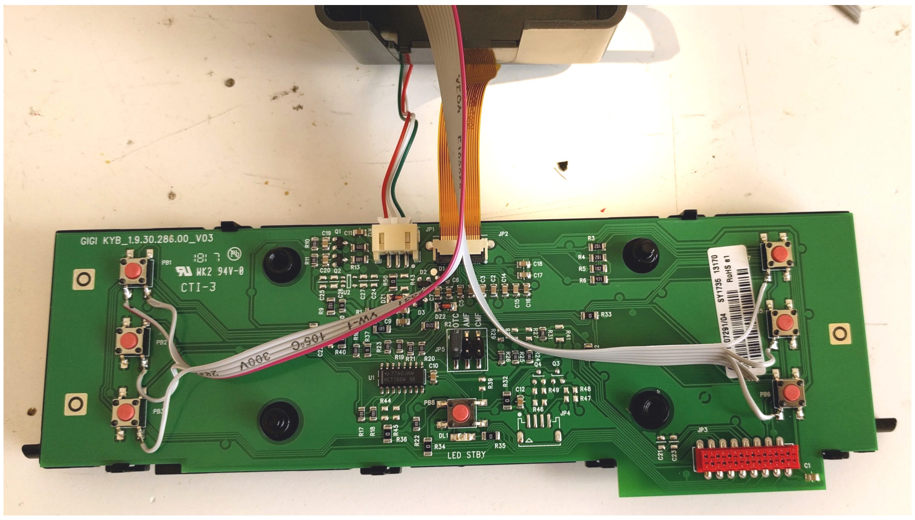


Figure 4.1: Control board of the coffee machine

4.1 Remote server

My supervisor recommended me to build the remote server with the Flask web framework. Flask is written in Python and is really easy to set up. I'd had some experience of plotting graphs with the Matplotlib library, which is also in Python, so I gave it a try. Figure 4.2 depicts a scheme of the server including technologies I used.

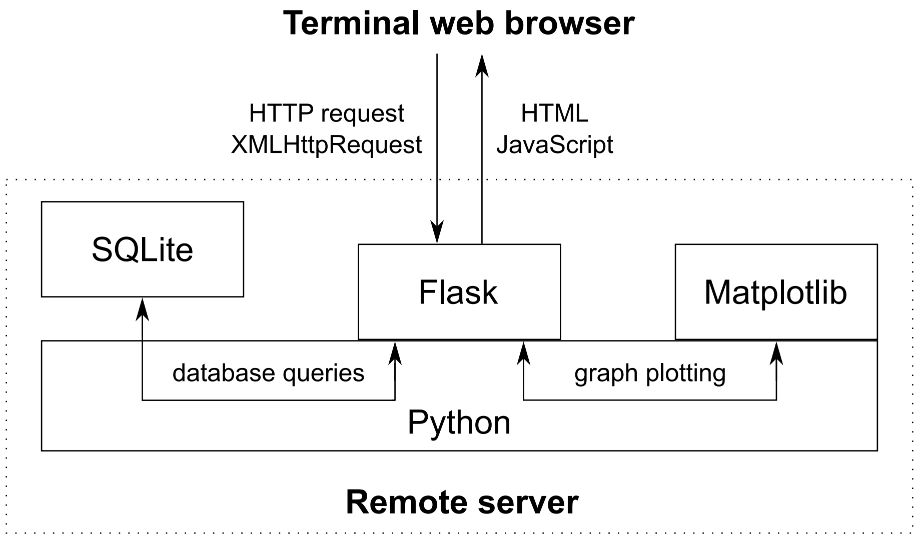


Figure 4.2: Scheme of the remote server

4.1.1 Database

I defined the database using SQL. Specifically, I used the SQLite engine which is, according to its web page, ‘the most used database engine in the world’. SQLite is natively supported by Python, too. The SQL code follows. I believe it can be easily ported to another SQL engine, should SQLite not be sufficient.

```
pragma foreign_keys = ON;

create table if not exists users (
    id varchar(24) primary key not null,
    name varchar(255) default "human"
);

create table if not exists flavors (
    name varchar(255) primary key not null
);

insert or ignore into flavors values
    ("espresso"),
    ("espresso lungo"),
    ("cappuccino"),
    ("latte macchiato");

create table if not exists coffees (
    num integer primary key,
    id varchar(24) references users(id),
    flavor varchar(255) not null references flavors(name),
    time datetime default current_timestamp
);
```

In order to access the database from Flask, I wrote a Python wrapper module. It imports `sqlite3` and defines several helper functions. Below, I include a wrapper function for one of the more advanced queries as an example.

```
def count_coffees_of_each_flavor(user):
    conn = sqlite3.connect("coffee.db")
    c = conn.cursor()
    res = list(c.execute("""
        select f.name, count(c.flavor) from flavors f
        left join (select * from coffees where id = ?) c
        on f.name=c.flavor group by f.name
        """, (user,)))
    conn.commit()
    conn.close()
    return res
```

4.1.2 Graphs

Statistical graphs are indeed generated by Matplotlib. I told the library to `use('Agg')` and not to open any windows. A new `Figure` is created for each graph to enable concurrent plotting. Graphs are saved into `BytesIO` streams and sent immediately. The following code shows how Matplotlib is integrated with Flask:

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

@app.route("/coffee/graph_flavors")
def coffee_graph_flavors():
    if "uid" not in session:
        return "nope"
    uid = session["uid"]
    flavors, counts = zip(*db.coffee_flavors(uid))
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_aspect(1)
    ax.pie(counts)
    b = BytesIO()
    fig.savefig(b, format="svg", bbox_inches="tight")
    b.seek(0)
    return send_file(b, mimetype="image/svg+xml")
```

4.1.3 Flask

Using Flask, I defined routes to manipulate the database and templates to provide content. I used Flask's `session` module to maintain user sessions. In addition to Flask, I had to install Flask-CORS: an extension that enables Cross-Origin Resource Sharing. Enabling CORS was necessary to let the remote server control contents of the local web page I described in subsection 3.2.3. A Flask application with CORS enabled can be set up as follows:

```
app = Flask(__name__)
CORS(app, supports_credentials=True)
```

Finally, I created the controlling remote script that would be downloaded and executed by the terminal web browser. I wrote a set of JavaScript functions to:

- transfer data from and to the remote server using XMLHttpRequests

- handle JSON strings received from the local server via Websocket
- manage contents of the web page and localStorage

4.2 Final test

When everything was ready, we quickly assembled the terminal and connected it to the coffee machine. The test scenario is depicted in Figure 4.3.

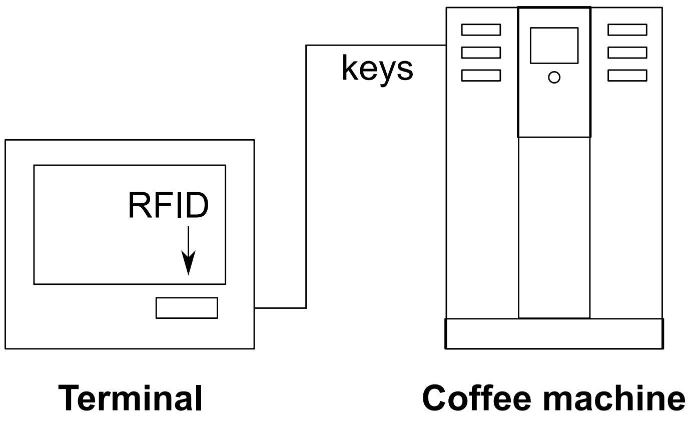


Figure 4.3: Test scenario

The system passed. It successfully connected to a WiFi network and accessed the remote server. Known as well as new users were able to log in via RFID and add coffees to the database by pressing keys on the coffee machine. Their statistics were displaying on the screen.

See the photo in Figure 4.4.

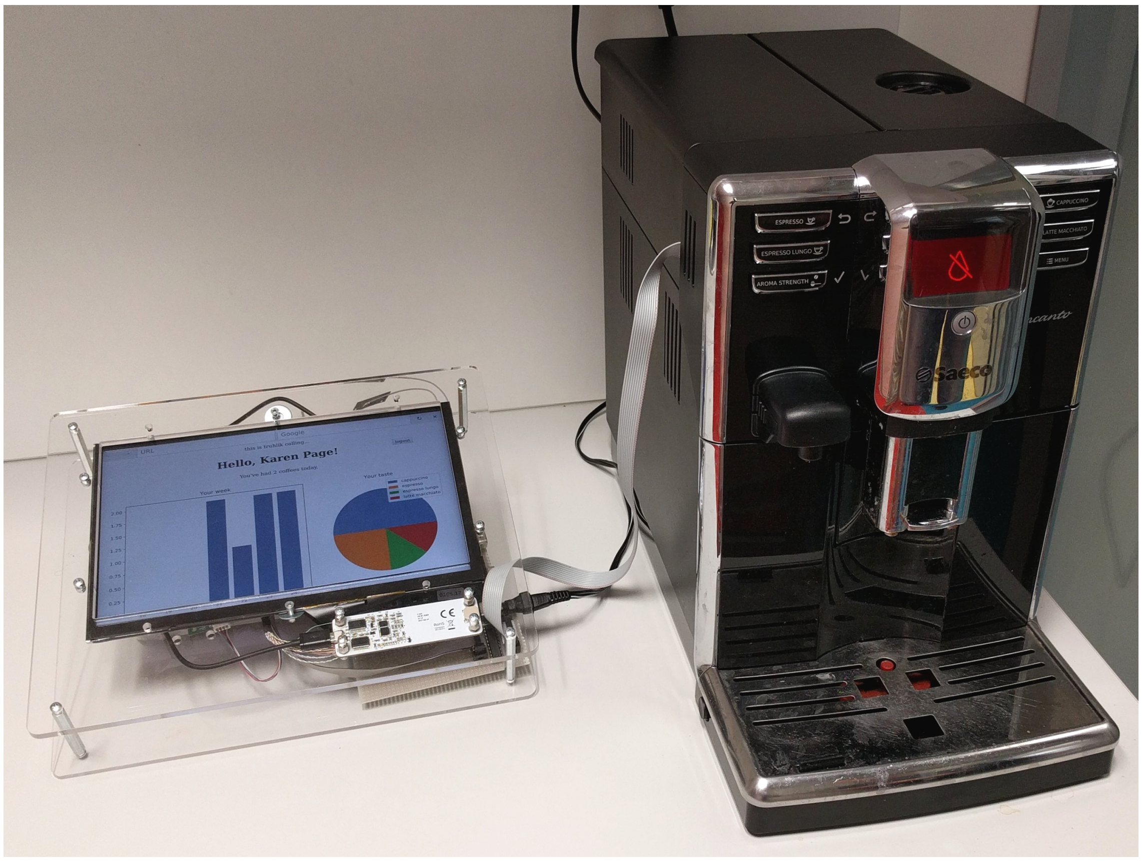


Figure 4.4: Final test

Chapter 5

Results

The goal of my work was to develop a prototype of an industrial terminal that would run a Linux operating system and feature, among other things, a touchscreen display and an RFID reader.

First, I chose and purchased suitable hardware components. I have chosen: the Advantech RSB-4411 single-board computer, which best matched our requirements and was affordable; a WUXGA touchscreen display from China; and a unique and feature-rich RFID reader, Digital Logic μFR Nano. The display model may not be suitable for production, but possible replacement will require only one modification to the system configuration. I have obtained necessary accessories, made cables and designed the terminal's body.

Then, I built a Linux system for the terminal. I have created an easy-to-update Buildroot configuration and successfully deployed a GPU-accelerated touchscreen-friendly web browser. Using the libev and libwebsockets libraries, I have developed an application to handle peripheral events. In addition to being a HTTP server, the application sends data from the RFID reader and GPIO pins to the browser via WebSocket. It also provides the browser with a link to a remote server.

Finally, I tested the terminal with a coffee machine. I have written a Flask server to maintain an SQLite user database and provide content to the terminal. I have successfully deployed the whole system, as Figure 4.4 shows.

I believe my work has been successful. To quote one of the coffee machine users: 'This is the best thing I've ever seen.'

References

1. ADVANTECH CO., LTD. *2018-2019 Embedded IoT Master Catalog* [online]. 2018 [visited on 2018-04-18]. Available from: `http://advcloudfiles.advantech.com/ecatalog/2018/02231128.pdf`.
2. NXP SEMICONDUCTORS. *i.MX 6Dual/6Quad Applications Processors for Consumer Products* [online]. 2017 [visited on 2018-04-21]. Available from: `https://www.nxp.com/docs/en/data-sheet/IMX6DQCEC.pdf`.
3. NXP SEMICONDUCTORS. *i.MX 6Dual/6Quad Applications Processor Reference Manual* [online]. 2017 [visited on 2018-04-22]. Available from: `https://www.nxp.com/docs/en/reference-manual/IMX6DQRM.pdf`.
4. ADVANTECH CO., LTD. *RSB-4411* [online]. 2017 [visited on 2018-04-29]. Available from: `http://www.advantech.eu/products/single_board_computer/rsb-4411/mod_d3901250-b0a0-4a5f-9762-b26fa0c36858`.
5. ADVANTECH CO., LTD. *User manual for RSB-4411* [online]. 2018 [visited on 2018-04-21]. Available from: `http://advdownload.advantech.com/productfile/Downloadfile1/1-1JRB5MK/RSB-4411_User_Manual_Ed.1.pdf`.
6. GALUS, David. *Migration to New Display Technologies on Intel Embedded Platforms* [online]. 2013 [visited on 2018-05-01]. Available from: `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/new-display-technologies-on-intel-embedded-platforms-paper.pdf`. Technical report. Intel Corporation.
7. *10.1" 1920x1200 B101UAN02.1 LCD Capacitive Touch Screen TFT Monitor With HDMI + DVI + VGA + Audio Input LCD Controller Board* [online] [visited on 2018-05-01]. Available from: `https://www.amazon.co.uk/1920x1200-B101UAN02-1-Capacitive-Monitor-Controller/dp/B06Y29YHS5/`.
8. AU OPTRONICS CORPORATION. *Product Specification* [online]. 2012 [visited on 2018-05-22]. Available from: `https://rabbit-note.com/wp-content/uploads/2014/08/B101UAN021.pdf`.
9. VS DISPALY TECHNOLOGY(HONGKONG) LTD. *MONITOR CONTROL BOARD SPECIFICATION* [online]. 2011 [visited on 2018-05-22]. Available from: `http://www.vslcd.com/Specification/M.NT68676.2A.pdf`.
10. WIKIPEDIA CONTRIBUTORS. *Radio-frequency identification — Wikipedia, The Free Encyclopedia* [online]. 2018 [visited on 2018-04-30]. Available from: `https://en.wikipedia.org/wiki/Radio-frequency_identification`.
11. DIGITAL LOGIC LTD. *NFC RFID Reader – μFR Nano* [online] [visited on 2018-04-30]. Available from: `https://www.d-logic.net/nfc-rfid-reader-sdk/products/nano-nfc-rfid-reader/`.

12. DIGITAL LOGIC LTD. *NFC RFID Reader Writer Comparison Table* [online] [visited on 2018-05-01]. Available from: `https://webshop.d-logic.net/nfc-rfid-device-comparison`.

13. *Bracket convert mini PCIExpress Half to Full size.* [online] [visited on 2018-05-20]. Available from: `https://www.thingiverse.com/thing:232143`.

14. *ADVANTECH-Corp · GitHub* [online] [visited on 2018-05-05]. Available from: `https://github.com/ADVANTECH-Corp`.

15. ADVANTECH CO., LTD. *Yocto Linux BSP Ver.8 User Guide for iMX6 series* [online] [visited on 2018-05-05]. Available from: `http://ess-wiki.advantech.com.tw/view/IoTGateway/BSP/Linux/iMX6/Yocto_LBV8_User_Guide`.

16. *The Buildroot user manual* [online] [visited on 2018-05-06]. Available from: `https://buildroot.org/downloads/manual/manual.html`.

17. ROBIN, Philippe. *Experiment with Linux and ARM Thumb-2 ISA* [Embedded Linux Conference]. 2007 [visited on 2018-05-07]. Available from: `https://elinux.org/images/8/8a/Experiment_with_Linux_and_ARM_Thumb-2_ISA.pdf`. ARM Ltd.

18. DIGITAL LOGIC LTD. *nfc-rfid-reader-sdk · GitLab* [online] [visited on 2018-05-12]. Available from: `https://www.d-logic.net/code/nfc-rfid-reader-sdk`.

19. GOLASZEWSKI, Bartosz. *New GPIO interface for linux user space* [FOSDEM '18]. 2018 [visited on 2018-05-13]. Available from: `https://fosdem.org/2018/schedule/event/new_gpio_interface_for_linux/`. BayLibre.

20. FARIAS, Facundo. *Awesome WebSockets* [online] [visited on 2018-05-13]. Available from: `https://github.com/facundofarias/awesome-websockets`.

21. *libwebsockets.org lightweight and flexible C networking library* [online] [visited on 2018-05-13]. Available from: `https://libwebsockets.org/`.

22. FREESCALE SEMICONDUCTOR, INC. *i.MX 6 Linux Reference Manual*. 2017.

23. *GStreamer 1.0 plugins for i.MX platforms* [online] [visited on 2018-05-19]. Available from: `https://github.com/Freescale/gstreamer-imx`.

24. THE QT COMPANY LTD. *Qt for Embedded Linux* [online] [visited on 2018-05-19]. Available from: `http://doc.qt.io/qt-5/embedded-linux.html`.

25. *QtWebEngine/VideoAcceleration - Qt Wiki* [online] [visited on 2018-05-21]. Available from: `https://wiki.qt.io/QtWebEngine/VideoAcceleration`.

26. *Qt WebBrowser* [online] [visited on 2018-05-21]. Available from: `http://doc.qt.io/QtWebBrowser/`.

27. DIGI INTERNATIONAL INC. *Adding a custom display* [online]. 2017 [visited on 2018-05-21]. Available from: `https://www.digi.com/resources/documentation/digidocs/90001945-13/reference/yocto/r_an_adding_custom_display.htm`.

Acronyms

ADC analog-to-digital converter

API application programming interface

bpp bits per pixel

BSP board support package

CAN Controller Area Network

CORS Cross-Origin Resource Sharing

CPU central processing unit

CRC cyclic redundancy check

DC direct current

DHCP Dynamic Host Configuration Protocol

DRM Direct Rendering Manager

EDID Extended Display Identification Data

eDP Embedded DisplayPort

eMMC embedded MultiMediaCard

GPIO general-purpose input/output

GPU graphics processing unit

GPU2D 2D Graphics Processing Unit

GPU3D 3D Graphics Processing Unit

GPUVG Vector Graphics Processing Unit

GUI graphical user interface

HDMI High-Definition Multimedia Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

I/O input/output

IPU Image Processing Unit

ISIC International Student Identity Card

I^{2}C Inter-Integrated Circuit

JSON JavaScript Object Notation

LDB LVDS Display Bridge

LVDS low-voltage differential signaling

MIPI Mobile Industry Processor Interface

NFC near-field communication

OS operating system

PCIe Peripheral Component Interconnect Express

PWM pulse-width modulation

QML Qt Modeling Language

RFID radio-frequency identification

SBC single-board computer

SD Secure Digital

SPI Serial Peripheral Interface bus

SQL Structured Query Language

TCP Transmission Control Protocol

TLS Transport Layer Security

TTL transistor–transistor logic

UART universal asynchronous receiver-transmitter

UID unique identifier

USB Universal Serial Bus

VFP Vector Floating Point

VGA Video Graphics Array

VPU Video Processing Unit

WUXGA Widescreen Ultra Extended Graphics Array; 1920×1200

X X Window System

Appendix A

CD contents

```
.
|-- buildroot        #terminal system
|-- coffee-flask     #remote server
|-- dip              #this thesis in LaTeX
|-- dip-matejj28.pdf #this thesis
|-- mt-apps          #peripheral handlers
|-- mtbrowser        #web browser
|-- qtwebbrowser     #another web browser
`-- README.md
```