Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Embedded Debug Interface for Robots

vorgelegt von

Robin Mirow

MIN-Fakultät

Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

Studiengang: Informatik

Matrikelnummer: 6946287

Erstgutachter: Dr. Andreas Mäder

Zweitgutachter: M. Sc. Marc Bestmann

Abstract

Modern robot platforms consist of a multitude of sensors, servos, and other interconnected devices. Particularly in the case of robots that are designed to operate independently, it is often required to diagnose problems with these devices without connecting the robot to a dedicated computer.
This bachelor thesis describes the hardware layout and software implementation for a microcontroller board with an LCD touchscreen that can be attached directly to a robot. It continuously listens on an *RS-485* bus using the *ROBOTIS Dynamixel Protocol 2.0* [ROBe] and displays detailed information about supported devices.
Other robot platforms using the same protocol would also work as long as they use a bus compatible with a UART (Universal Asynchronous Receiver Transmitter) interface.

Zusammenfassung

Moderne Roboterplattformen bestehen aus einer Vielzahl von Sensoren, Servomotoren und anderen miteinander verbundenen Geräten. Insbesondere im Fall von Robotern, die darauf ausgelegt sind unabhängig zu agieren, ist es oft notwendig Probleme mit diesen Geräten zu diagnostizieren ohne den Roboter an einen dedizierten Computer anzuschließen.
Diese Bachelorarbeit beschreibt den Hardware-Aufbau und die Software-Implementation für eine Mikrocontrollerplatine mit einem LCD Touchscreen, die direkt an einen Roboter montiert werden kann. Diese hört kontinuierlich einen *RS-485* Bus ab, der das *ROBOTIS Dynamixel Protocol 2.0* [ROBe] verwendet, und zeigt detailierte Informationen über unterstützte Geräte an.
Andere Roboterplattformen, die das gleiche Protokoll verwenden, würden auch funktionieren, solange sie einen Bus benutzen, der mit einer UART (Universal Asynchronous Receiver Transmitter) Schnittstelle kompatibel ist.

Contents

List of Figures

1 Introduction

As robots become cheaper and more popular for common tasks, they are frequently used as independent platforms without significant supporting infrastructure or personnel. Because of this, it is important that robots are equipped with easy to use interfaces to control them and diagnose problems. Touchscreen displays are ideal for this use case, as they can be attached to any even surface and do not require any specialized input devices.
This thesis implements an example of such an interface as a standalone microcontroller board that can be directly attached to a robot built on the *Wolfgang* robot platform (see section 1.1 for more details). It can be used to debug common issues with the devices used by the robot, such as connectivity or misconfiguration.
This introduction gives a brief motivation for this bachelor thesis in section 1.1. Since the *Wolfgang* robot platform is used for the *RoboCup* competition, it is explained in section 1.2. Section 1.3 defines the goals of the thesis.
Chapter 2 presents related work in robot interface design, focusing on robots designed for frequent interaction with humans. After that, chapter 3 explains some basics as well as the bus and protocol used by the implementation. Next, chapter 4 describes the actual implementation and rationale behind important design decisions. Chapter 5 then provides a short overview of the usability of the finished work and relevant benchmarks while chapter 6 discusses these results. Finally, Chapter 7 summarizes the findings and lists possible improvements to the current implementation.

1.1 Motivation

The work done in this thesis is primarily intended for use with the *Wolfgang* robot platform used by the *RoboCup* team *Hamburg Bit-Bots*. It consists of various devices [bit19] that communicate using the *ROBOTIS Dynamixel Protocol 2.0* [ROBe] over an *RS-485* or TTL bus.
Without a computer connected to the robot, it is not possible to monitor the status of the connected devices. Devices may be unreachable for different reasons:

- the device is physically disconnected
- the device is powered off or otherwise malfunctioning

- the device's packets are lost, either due to interference or a misbehaving bus participant
- the device never sends any packet because it is waiting for another device that is not sending for one of the reasons above

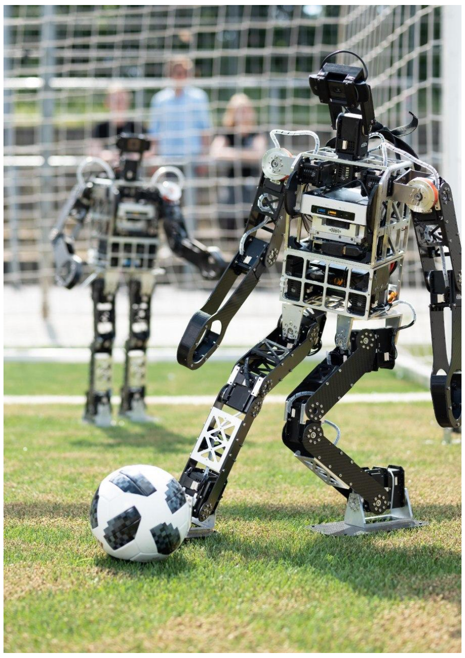


Figure 1.1: The *Wolfgang* robot platform

While it is always possible to connect a computer in case of an obvious malfunction, this is a significant amount of overhead. It does not allow for quick detection of disconnected devices or anomalous readings of a single device. In a competition like *RoboCup*, it is important to quickly detect problems in order to fix them in the field. Noncritical errors may not disable the robot but they can still have an impact on its performance in a game.
Due to the extensible nature of the protocol, other robot platforms using the same protocol and a bus compatible with a UART (Universal Asynchronous Receiver Transmitter) interface would also work, with code changes only required for adding support for new device models.

1.2 RoboCup

RoboCup (`https://www.robocup.org/`) is a competition designed to promote robotics and AI research. It intends to set challenges that are both technically difficult and socially impactful. The long term goal is to create fully autonomous robots that can play soccer and win against the current winners of the FIFA World Cup championship [robb].

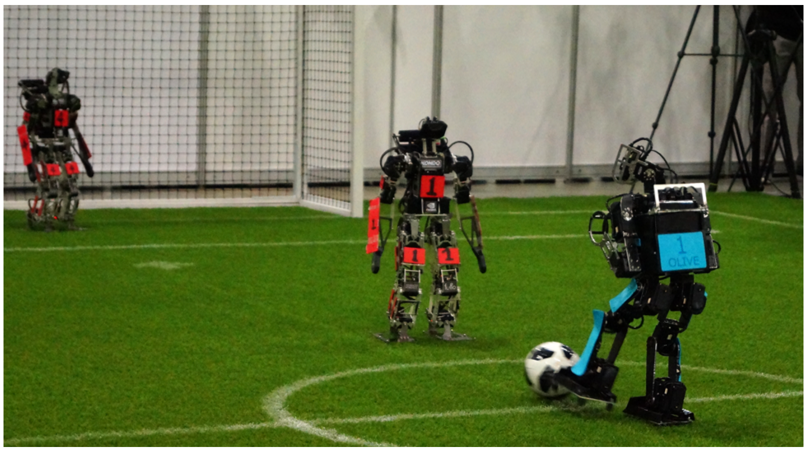


Figure 1.2: A *RoboCup* soccer game

While soccer was the original idea behind *RoboCup* [roba], there are now many different categories that focus on specific challenges:

- *RoboCupSoccer*
- *RoboCupRescue*
- *RoboCup@Home*
- *RoboCupIndustrial*
- *RoboCupJunior*

The leagues in each category are based on the physical layout of the robot or the specific task. For example, the *RoboCupSoccer* category [robd] includes the following leagues:

- *Humanoid*
- *Standard Platform*
- *Middle Size*
- *Small Size*
- *Simulation*

The *Hamburg Bit-Bots* team competes in the *Humanoid* league (*KidSize* and *TeenSize*) with robots built on the *Wolfgang* robot platform [robc].
By defining clear rules and goals, the various *RoboCup* leagues make it possible to compare and evaluate different approaches in both robotics and AI research. They also make research in these areas more visible to the general public.

1.3 Thesis Goal

The goal of this thesis is to determine a suitable microcontroller board and develop the required firmware for

- collecting status information of the connected devices by passively listening on the bus ...
- displaying this information on an integrated touchscreen display and ...
- navigating between detailed views for each device/model using the touchscreen.

In particular, it should be possible to identify unreachable or disconnected devices at a glance. Adding support for new device models should be easy and effortless. Both display and microcontroller should be compact enough to be able to attach and detach them from a robot quickly.

2 Related Work

None of the qualified teams for the *RoboCup Humanoid League 2019* had a dedicated display installed on their robots [robc]. Outside of research contests like *RoboCup* and industrial applications, user interactions with robots are still rare.

Relatively common robots for domestic use are automatic vacuum cleaners and lawnmowers. These robots usually feature only simple displays. Controls are limited to buttons. Complex configuration and maintenance for most models can be performed using a separate application that has to be installed on the user's phone. Examples include the *Roomba* and *Braava* product lines by iRobot [iro]. The *Pepper* robot by SoftBank Robotics is a humanoid robot that focuses on social interactions in human-centered environments like stores, schools or homes. It is one of the few social robots that has seen limited success outside of research. In addition to speakers and LEDs, it also features a large touchscreen on the chest. The touchscreen is intended for interaction but can also be used for debugging during development [PG18].

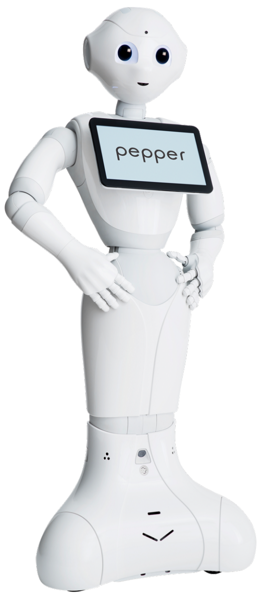


Figure 2.1: The *Pepper* robot[1]

[1] Source: `https://www.softbankrobotics.com/emea/themes/custom/softbank/images/full-pepper.png` (Accessed: 2020-03-30)

Since the *Pepper* robot is highly customizable through software, it can be adapted for a wide variety of scenarios. However, the lack of powerful or accurate arms limits it to mostly informational roles. It is used as the platform for the *RoboCup@Home Social Standard Platform League* [PG18].
Similarly, the *Human Support Robot* by Toyota is used for the *RoboCup@Home Domestic Standard Platform League*. As the name suggests, these robots are supposed to assist humans in domestic contexts and are not optimized for complex social interactions. They feature a simple display for showing additional information [YTO+19].
Many teams in the *RoboCup@Home Open Platform League 2019* also used various types of displays for interfacing with their robots:

- The *CATIE Robotics* team uses modified PAL Robotics *TIAGo* robots. The *TIAGo* is equipped with a laptop tray that was customized to make it adjustable. An Android tablet was mounted to the back of the head to serve as a touch-capable input device [FAD+19].
- The *homer@UniKoblenz* team also uses a *TIAGo* as well as a custom robot-based on the *CU-2WD-Center* robot platform. It has a small head-mounted display that displays a face [MMW+19].
- The *RoboFEI@Home* team uses a custom robot with an enclosure for an Apple iPad 2" tablet. The tablet is primarily used to display a face [PMM+19].
- The *RT Lions* team use a screen and a mini-beamer that projects images on a plastic dome to display the faces for their two robots [RWT+19].

3 Basics

This chapter explains some of the technologies used by the implementation. Section 3.1 and section 3.2 describe the bus and protocol used by the *Wolfgang* robot platform. Section 3.3 gives a brief overview of embedded operating systems. The implementation uses an embedded OS to meet its latency requirements.

3.1 RS-485

RS-485 is a commonly used name for the ANSI TIA/EIA-485 standard. It is a specification for a half-duplex serial bus. Data is transmitted over two wires, usually labeled A and B, with a positive voltage on A and a negative voltage on B. The voltage difference between these two wires determines the actual logic level. A voltage difference less than or equal to $-200\,\mathrm{mV}$ is a logical 0, while a difference greater than or equal to $200\,\mathrm{mV}$ is a logical 1 [SZC02].
This two-wire setup makes the bus more resilient towards noise, allowing for operation in noisy environments or over long distances.
For the purpose of this thesis, the only interesting fact is the differential transmission. It requires an additional transceiver that converts the two signals into a single signal at the microcontroller's logic level. Such a signal can then easily be consumed by a UART.

3.2 ROBOTIS Dynamixel Protocol 2.0

The ROBOTIS Dynamixel Protocol is a packet-based master/slave protocol used by all of ROBOTIS's products. This section describes only the improved version 2.0. Official documentation can be found at ROBOTIS's website [ROBe].
Each device is assigned a unique 8-bit ID that is used to determine the sender or receiver of packets. IDs must be configured before connecting a device to the bus. The user must make sure that IDs do not overlap. The IDs `0xff` and `0xfd` are not allowed and `0xfe` is reserved for broadcasts.
Once connected, the master (usually a powerful computer controlling the various devices) can send an instruction packet. Instruction packets can target one or more devices (for details see 3.2.1). Only devices targeted by an instruction are allowed to write a status packet to the bus. If an instruction targets more than one device, the responses are ordered by their ID. Due to this, no external synchronization for

the bus is required. Since instruction or status packets may be lost, the master has to resend instruction packets if no packets have been received after a certain amount of time.
Devices are seen as linear byte-addressed memory with 16-bit addresses (referred to as *control tables* by the official documentation). Usually, these are memory-mapped registers that can be used to read or change the state of a device. However, due to this simple view of a device as some amount of memory, the protocol can be easily extended or reused by custom devices.

Name	Type	Content
Header	uint8_t	0xff
	uint8_t	0xff
	uint8_t	0xfd
Reserved	uint8_t	0x00
Device Id	uint8_t	Device Id
Length	uint16_t	Length
Instruction	uint8_t	Instruction
Payload	uint8_t	Byte 0
	...	...
	uint8_t	Byte n
CRC	uint16_t	CRC

(a) Generic layout of an instruction packet

Name	Type	Content
Header	uint8_t	0xff
	uint8_t	0xff
	uint8_t	0xfd
Reserved	uint8_t	0x00
Device Id	uint8_t	Device Id
Length	uint16_t	Length
Instruction	uint8_t	0x55
Error	uint8_t	Error
Response	uint8_t	Byte 0
	...	...
	uint8_t	Byte n
CRC	uint16_t	CRC

(b) Generic layout of a status packet

Figure 3.1: Generic packet layout

Each packet has some metadata preceding the actual payload. It starts with a fixed byte sequence that allows detecting a packet start without knowing where the last packet ended. It is followed by the device ID. For instruction packets, this is the device that is targeted, for status packets it is the device that sent the packet. The *Length* field determines the remaining length of the packet. This allows for a payload of variable size, depending on the instruction used or the data returned by the status packet. The next field is the instruction; status packets use a reserved instruction. While the exact content of the payload varies, status packets always store an additional error field as the first byte. This field can be used to indicate any errors that occurred during the processing of an instruction. Since the payload can contain arbitrary values, it can also contain the byte sequence used to mark the start of a new packet. To prevent a receiver from misinterpreting this byte sequence, *byte stuffing* is used. Whenever a payload contains the byte sequence `0xff 0xff 0xfd`, it is replaced by `0xff 0xff 0xfd 0xfd`. This makes it impossible to send a payload that could be interpreted as the start of a new packet. The receiver simply removes the extra byte that was added. Note that the *Length* field specifies the number of bytes with byte stuffing already applied. The last two bytes of every packet contain the *CRC-16/BUYPASS* [Coo] checksum of every byte in the packet, including the starting sequence (and excluding the CRC itself). This checksum can be used to detect transmission errors, in case the underlying bus does not have error detection built-in (*RS-485* does not).

3.2.1 Instructions

Ping (`0x01`) Tests whether a device is connected. If the device ID is broadcast (`0xfe`), all devices respond. The status packet contains the device's model number and firmware version.

Ping Instruction (0x01)

instruction packet payload

Name	Type	Content
<no payload>		

status packet payload

Name	Type	Content
Model Number	uint16_t	Model Number
Firmware Version	uint8_t	Firmware Version

Figure 3.2: *Ping* instruction payloads

Read (`0x02`) Reads bytes from the *control table* of a device. The status packet contains the requested bytes.

Read Instruction (0x02)

instruction packet payload

Name	Type	Content
Address	uint16_t	Address
Length	uint16_t	Length

status packet payload

Name	Type	Content
Data	uint8_t	Byte 0
	...	...
	uint8_t	Byte (Length - 1)

Figure 3.3: *Read* instruction payloads

Write (`0x03`) Writes bytes to the *control table* of a device. The status packet indicates whether the write was executed successfully and contains no additional payload. Many devices can be configured to not send any status packets on writes.

Write Instruction (0x03)

instruction packet payload			status packet payload		
Name	Type	Content	Name	Type	Content
Address	uint16_t	Address	<no payload>		
Data	uint8_t	Byte 0			
	...	...			
	uint8_t	Byte n			

Figure 3.4: *Write* instruction payloads

Reg Write (`0x04`) Identical to *Write*, except that the write is delayed until an *Action* instruction is received.

Reg Write Instruction (0x04)

instruction packet payload			status packet payload		
Name	Type	Content	Name	Type	Content
Address	uint16_t	Address	<no payload>		
Data	uint8_t	Byte 0			
	...	...			
	uint8_t	Byte n			

Figure 3.5: *Reg Write* instruction payloads

Action (`0x05`) Executes the write registered by the previous *Reg Write* instruction. The status packet indicates whether the write was executed successfully and contains no additional payload. Many devices can be configured to not send any status packets on writes.

Action Instruction (0x05)

instruction packet payload			status packet payload		
Name	Type	Content	Name	Type	Content
<no payload>			<no payload>		

Figure 3.6: *Action* instruction payloads

Factory Reset (`0x06`) Resets the device's *control table* fields to their default values. *Reset Mode* determines the values that are reset:

- `0xff`: resets all values
- `0x01`: resets all values except for the ID
- `0x02`: resets all values except for the ID and the baud rate

The status packet indicates whether the reset was executed successfully and contains no additional payload.

`Factory Reset Instruction (0x06)`

instruction packet payload			status packet payload		
Name	Type	Content	Name	Type	Content
Reset Mode	uint8_t	Reset Mode	<no payload>		

Figure 3.7: *Factory Reset* instruction payloads

Reboot (`0x08`) Reboots the device. The status packet indicates whether the reboot was successful and contains no additional payload.

`Reboot Instruction (0x08)`

instruction packet payload			status packet payload		
Name	Type	Content	Name	Type	Content
<no payload>			<no payload>		

Figure 3.8: *Reboot* instruction payloads

Clear (`0x10`) Resets the multi-turn information of the device. This instruction is very closely tied to servos and generally not useful for other kinds of devices.

`Clear Instruction (0x10)`

instruction packet payload			status packet payload		
Name	Type	Content	Name	Type	Content
Fixed	uint8_t	0x01	<no payload>		
	uint8_t	0x44			
	uint8_t	0x58			
	uint8_t	0x4c			
	uint8_t	0x22			

Figure 3.9: *Clear* instruction payloads

Sync Read (`0x82`) Reads bytes from the *control tables* of multiple devices at once. The device ID of the instruction packet must be broadcast (`0xfe`). The status packet of each device contains the requested bytes. Devices respond in the same order as their IDs in the instruction packet payload.

Sync Read Instruction (0x82)

instruction packet payload

Name	Type	Content
Address	uint16_t	Address
Length	uint16_t	Length
Device IDs	uint8_t	Device ID 1
	...	...
	uint8_t	Device ID n

status packet payload

Name	Type	Content
Data	uint8_t	Byte 0
	...	...
	uint8_t	Byte (Length - 1)

Figure 3.10: *Sync Read* instruction payloads

Sync Write (`0x83`) Writes bytes to the *control tables* of multiple devices at once. The data for each device can be different. The device ID of the instruction packet must be broadcast (`0xfe`). The status packet of each device indicates whether the write was executed successfully and contains no additional payload. Devices respond in the same order as their IDs in the instruction packet payload. Many devices can be configured to not send any status packets on writes.

Sync Write Instruction (0x83)

instruction packet payload

Name	Type	Content
Address	uint16_t	Address
Length	uint16_t	Length
Device ID 1	uint8_t	Device ID 1
Data 1	uint8_t	Byte 0
	...	...
	uint8_t	Byte n
...	...	...
Device ID n	uint8_t	Device ID n
Data n	uint8_t	Byte 0
	...	...
	uint8_t	Byte n

status packet payload

Name	Type	Content
<no payload>		

Figure 3.11: *Sync Write* instruction payloads

Bulk Read (`0x92`) Reads bytes from the *control tables* of multiple devices at once. Unlike *Sync Read*, this instruction allows for different addresses and lengths for each device. The device ID of the instruction packet must be broadcast (`0xfe`). The status packet of each device contains the requested bytes. Devices respond in the same order as their IDs in the instruction packet payload.

Bulk Read Instruction (0x92)

instruction packet payload

Name	Type	Content
Device ID 1	uint8_t	Device ID 1
Address 1	uint16_t	Address 1
Length 1	uint16_t	Length 1
...	...	...
Device ID n	uint8_t	Device ID n
Address n	uint16_t	Address n
Length n	uint16_t	Length n

status packet payload

Name	Type	Content
Data n	uint8_t	Byte 0
	...	...
	uint8_t	Byte (Length n - 1)

Figure 3.12: *Bulk Read* instruction payloads

Bulk Write (`0x93`) Writes bytes to the *control tables* of multiple devices at once. Unlike *Sync Write*, this instruction allows not only for different data but also for different addresses and lengths for each device. The device ID of the instruction packet must be broadcast (`0xfe`). The status packet of each device indicates whether the write was executed successfully and contains no additional payload. Devices respond in the same order as their IDs in the instruction packet payload. Many devices can be configured to not send any status packets on writes.

`Bulk Write Instruction (0x93)`

instruction packet payload

Name	Type	Content
Device Id 1	uint8_t	Device Id 1
Address 1	uint16_t	Address 1
Length 1	uint16_t	Length 1
Data 1	uint8_t	Byte 0
	...	...
	uint8_t	Byte n
...	...	...
Device Id n	uint8_t	Device Id n
Address n	uint16_t	Address n
Length n	uint16_t	Length n
Data n	uint8_t	Byte 0
	...	...
	uint8_t	Byte n

status packet payload

Name	Type	Content
<no payload>		

Figure 3.13: *Bulk Write* instruction payloads

3.2.2 Byte-Order

All multi-byte values (packet fields, values in packet payloads and *control table* fields) are *little-endian*. That is, for multi-byte values, the least significant byte comes first, then the second least significant byte, etc.

3.3 Embedded Operating Systems

Embedded operating systems are libraries linked directly with the user's code. They are small both in runtime overhead and in program size, making them ideal for microcontrollers that cannot run full operating systems like Linux or Windows. Embedded OSs only provide a small subset of the many features offered by traditional operating systems. Common features are:

- threads (sometimes also called tasks)
- concurrency primitives (mutexes, semaphores, queues, etc)
- memory allocators
- filesystems

The most significant difference to traditional operating systems is the lack of security. Since the operating system is part of the program itself, all code can be trusted and process boundaries are not needed [TB14].
The core of an embedded OS are threads and context switching. Being part of the user program, the application must configure an interrupt that calls the scheduler, which in turn runs application code in one of the threads. Threads can run on separate CPU cores but they are also useful for single-core systems. Threads are preemptive: one thread can be paused and another one started instead. This makes it possible to guarantee that some threads always get the CPU time they need, for example to process incoming data [freb].
The concurrent nature of threads makes it necessary to synchronize access to shared data. Since synchronization is deeply intertwined with the scheduler, concurrency primitives must also be provided by the embedded OS.

4 Implementation

This chapter describes the hardware the implementation uses (section 4.1) and the developed firmware (section 4.2). The source code for the firmware can be found at `https://github.com/Laegluin/embedded_debug_interface_for_robots/tree/thesis-ref`. The version referenced by this thesis is tagged as `thesis-ref`. The description of the firmware is intended to give a high-level overview and discusses tradeoffs as well as points of interest.
Since the implementation targets the *Wolfgang* robot platform, it must be compatible with its *RS-485* bus running at a speed of 2 MBd.

4.1 Hardware

The following hardware is used:

- STM32F7508-DK development board[1]
- MAX485 RS-485/RS-422 transceiver[2]
- FT232R USB to UART converter (for testing)[3]

The MAX485 transceiver is used to connect the *RS-485* bus to the STM32F7508-DK board. It outputs a 5 V signal that can be interpreted by one of the board's UARTs. STM32F7508-DK is built around an STM32F750N8H6 Arm Cortex-M7 based microcontroller. The controller's *UART6* is used, as it is easily accessible through the board's Arduino Uno compatible connectors at the back. All UARTs have a theoretical maximum speed of 27 MBd [STM18c] (depending on the clock configuration), making the MAX485's maximum speed of 2.5 MBd the maximum speed of this setup [Max14][STM18d].
The STM32F7508-DK board has a built-in 4.3" 480x272 LCD-TFT capacitive touchscreen. It is used to display and interact with the UI. Because it is built-in, no further assembly is required [STM18d].
The STM32F750N8H6 microcontroller is based on an Arm Cortex-M7 CPU. It supports clock speeds of up to 216 MHz and has both data and instruction caches.

[1]`https://www.st.com/resource/en/user_manual/dm00537062-discovery-kit-for-stm32f7-series-with-stm32f750n8-mcu-stmicroelectronics.pdf`
[2]`https://datasheets.maximintegrated.com/en/ds/MAX1487-MAX491.pdf`
[3]`https://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf`

It is equipped with a single-precision FPU (floating-point unit) and a DMA controller [STM18a]. While this is quite a lot of processing power for a microcontroller, it is needed to drive an interactive UI while at the same time processing incoming data quickly enough. The DMA controller allows transferring data from the UART to memory at high data rates without using up CPU time.

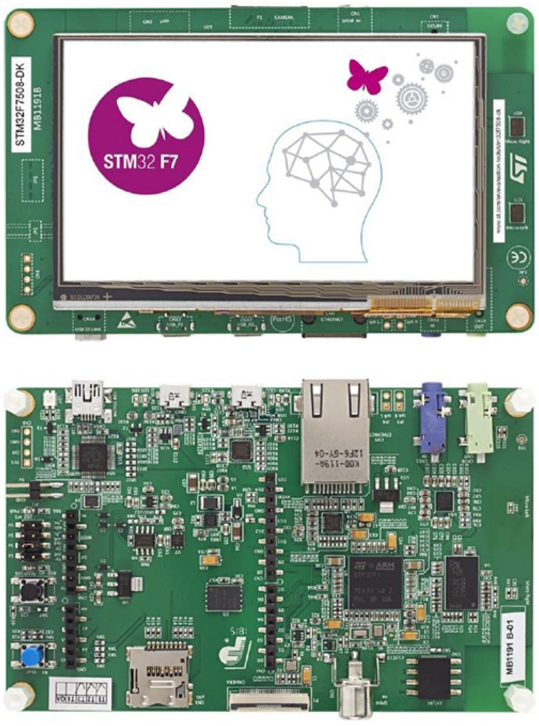


Figure 4.1: The STM32F7508-DK development board[4]

The controller itself comes with only 64 KiB of embedded flash memory. However, it also has a Quad-SPI memory interface that can be used with external flash memory [STM18a]. The STM32F7508-DK board has 16 MiB of pre-installed flash memory [STM18d].
Similarly, 320 KiB of RAM are embedded in the controller [STM18a]. This is more than enough for most applications but not enough for the video RAM (VRAM) used by the display. When using double buffering and 8 bits per pixel, at least $480 \cdot 272 \cdot 3 \cdot 2 = 765$ KiB are needed for VRAM alone. VRAM can be located in external RAM which can be controlled by the microcontroller's FMC (flexible memory controller). The board has 8 MiB of pre-installed external RAM [STM18a][STM18d].

[4] Source: `https://www.st.com/bin/ecommerce/api/image.PF267270.en.feature-description-include-personalized-no-cpn-large.jpg` (Accessed: 2020-03-30)

The STM32F7508-DK board is equipped with an ST-LINK/V2-1 in-circuit debugger. It can be used for debugging and for programming the internal flash memory [STM18d]. External flash cannot be programmed using the ST-LINK; instead, a standalone bootloader must be used (see subsection 4.2.1).
The FT232R USB to UART converter is only used for testing and benchmarking. It can easily be connected to *UART6* instead of the MAX485. Data can then be sent to the USB serial port emulated by the FT232R from the computer connected to it [Fut]. This makes testing extremely easy since the device can be treated as a serial terminal.

4.2 Software

This section describes the firmware for the STM32F7508-DK board. The bootloader is written in C (C99), the actual firmware itself is written in C++ (C++14). All code is compiled with the *GNU arm-none-eabi* toolchain. *Newlib* is used as the C standard library implementation.

4.2.1 Bootloader

When the microcontroller is reset, it loads the *vector table* from address `0x00000000`. The *vector table* contains the addresses of all possible interrupt handlers. In addition, the first item is the initial value of the stack pointer and the second is the address of the entry point. After loading the *vector table*, the stack pointer is set to the configured value and the processor jumps to the entry point [STM18c]. By default, addresses in the range `[0x00000000, 0x08000000)` are mapped to `0x08000000`, which in turn is mapped to internal flash memory [STM18c].
This poses a problem for applications that do not fit into internal flash memory: there is no way to boot from the much larger Quad-SPI flash memory or to program it. An additional bootloader has to be used instead. A bootloader is a minimal application that initializes a peripheral device for I/O, accepts commands as well as data from said device and allows programming and booting from otherwise unsupported memory.
In this case, the bootloader must allow programming and booting from Quad-SPI flash memory. It uses one of the STM32F7508-DK board's micro-USB ports for I/O [STM18d]. USB is a well-suited interface because data integrity checks are built-in [usb00]. STMicroelectronics provides a USB 2.0 compliant library for applicable microcontrollers that the bootloader uses for USB support [STM19a].
When the bootloader starts, it initializes the USB controller and the Quad-SPI memory interface. It also blinks the board's LED at regular intervals to indicate that it is running. It then listens on the USB interface as a communication class device.

Name	Type	Content
Start	uint8_t	not 0xff
	uint8_t	0xff
	uint8_t	not 0xff
Command	uint8_t	0x00
Image Length	uint32_t	Image Length
Image	uint8_t	Byte 0
	...	...
	uint8_t	Byte (Length - 1)

(a) *flash* packet layout

Name	Type	Content
Start	uint8_t	not 0xff
	uint8_t	0xff
	uint8_t	not 0xff
Command	uint8_t	0x01

(b) *start* packet layout

Figure 4.2: Bootloader packet layout

The bytes received are interpreted according to a custom, packet-based protocol. The packet layout can be seen in figure 4.2. Packets start with a start sequence that makes it possible to identify the beginning of a packet even if it is preceded by unrelated bytes. To prevent unwanted start sequences in the packet itself, all bytes following the start sequence must be escaped by adding *byte stuffing*: whenever a byte that is not `0xff` is followed by a `0xff` byte, another `0xff` byte must be added. For example, the bytes `0x00 0xff 0xff` must be sent as `0x00 0xff 0xff 0xff`. The bootloader simply removes the added bytes before processing the rest. Values of more than one byte are encoded as *little-endian* and lengths are always given as the length before any *byte stuffing* was applied.

A *flash* packet instructs the bootloader to erase and then program the Quad-SPI flash memory with all the bytes in the *Image* field, beginning at the start address of the Quad-SPI flash memory.

The *start* packet instructs the bootloader to start an application previously flashed. It disables all (pending) interrupts, exits interrupt handler mode if necessary and enables the memory-mapped mode for the Quad-SPI memory interface. In addition, it also enables the board's LED permanently, indicating that the application is now running. Finally, it starts the application by loading the stack pointer and jumping to the entry point in the same way the microcontroller does when it boots from internal flash memory. The bootloader assumes that the application's *vector table* starts at the beginning of the Quad-SPI flash memory space (`0x90000000` [STM18c]). The application can also be started by pressing the board's programmable button (next to the reset button). The procedure is identical to the one described above.

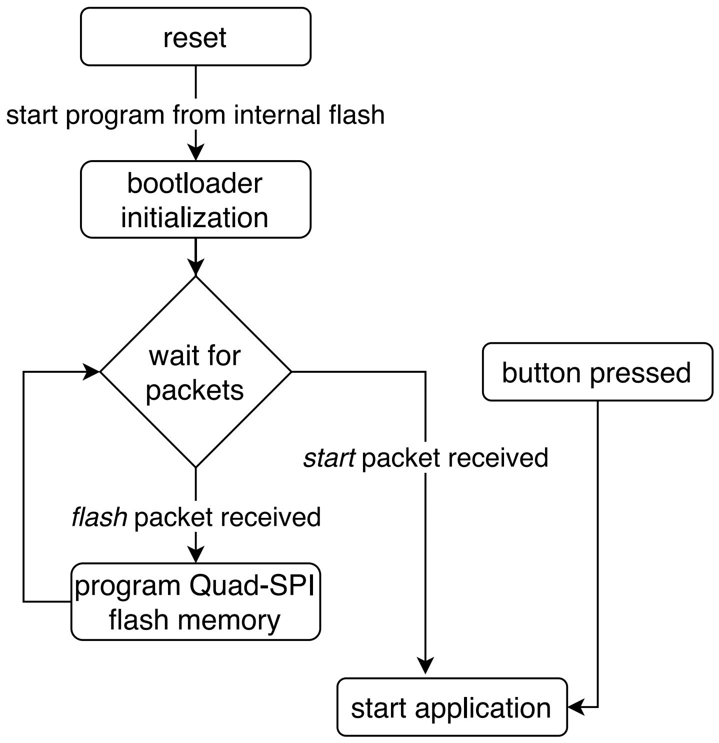


Figure 4.3: After reset, the microcontroller starts the program in internal flash memory (the bootloader). It then waits for packets to arrive over USB. If a *start* packet has been received or the button was pressed, it starts the application in Quad-SPI flash memory.

The bootloader only has to be flashed to internal flash memory once. Afterward, it can be used to program the Quad-SPI flash memory and start an application from it using the board's micro-USB port. In this case, the application is the actual firmware that controls the UI and processes packets sent over the *Wolfgang* robot platform's *RS-485* bus.

4.2.2 Used Libraries

The following software libraries were used to aid the development of the firmware:

- STM32F7 HAL and Low-layer drivers[5] (part of STM32CubeF7 1.15.0)
- STM32F7508-Discovery board support package[6] (part of STM32CubeF7

[5]https://www.st.com/resource/en/user_manual/dm00189702-description-of-stm32f7-hal-and-lowlayer-drivers-stmicroelectronics.pdf
[6]https://github.com/STMicroelectronics/STM32CubeF7/tree/master/Drivers/BSP/STM32F7508-Discovery

1.15.0)

- STemWin version 5.44 (part of STM32CubeF7 1.15.0)
- FreeRTOS[7] version 10.0.1 (part of STM32CubeF7 1.15.0)
- Catch2[8] version 2.10.2

The STM32F7 HAL and Low-layer drivers are various independent drivers for peripherals found on hardware produced by STMicroelectronics. Despite the name (HAL - hardware abstraction layer) they still require the user to write hardware-specific code. They do offer simplified interfaces that do not require manipulation of memory-mapped peripheral device registers. For more complicated peripherals like the LCD controller this can be helpful [STM17].
Similarly, the STM32F7508-Discovery board support package offers drivers based on the HAL and Low-layer drivers that are customized for the STM32F7508-DK board. They encapsulate all of the hardware and have easier to use interfaces than their HAL counterparts [STM19b].
STemWin is a GUI library based on emWin by SEGGER Microcontroller GmbH & Co. KG. It is effectively a precompiled version of emWin with driver support for STMicroelectronics' hardware. While emWin is commercially licensed, STemWin is licensed as free of charge for use with STMicroelectronics' products [STM19b][STM18b][SEG17]. Since it is otherwise indistinguishable from emWin, it will be referred to as emWin in the following. emWin was chosen instead of TouchGFX (also part of STM32CubeF7) because it does not require code generation and has extensive documentation.
FreeRTOS is a popular embedded operating system. It is permissively licensed and widely used by major companies. It is specifically designed to run with minimal overhead (both code size and execution speed) [fred]. STMicroelectronics provides a distribution of FreeRTOS that is already configured for their microcontrollers as part of STM32CubeF7 [STM19b].
Catch2 is a unit testing library for C++11 and above. Tests are written as simple functions with assertions and do not require complex setup. It is a header-only library, meaning that no additional build system configuration is necessary [cat].

4.2.3 Testing

Automatic testing of the hardware-specific code and the UI is difficult. Tests concerning the bus were done using the FT232R USB to UART converter to send

[7] `https://www.freertos.org/`
[8] `https://github.com/catchorg/Catch2`

test data as well as traces of real bus traffic directly to the microcontroller's UART. UI tests were also performed manually, usually as part of the aforementioned tests. Most of the core logic deals with data received over the bus. The actual origin of the data is not relevant for testing. This code is carefully structured to avoid any dependencies on hardware-specific code. It is tested using Catch2 unit tests running on the host platform (Linux on AMD64). While this is not a perfect solution as there are still differences between the platforms like pointer sizes, alignment requirements or hardware protection mechanisms provided by the operating system, it creates high confidence in the correctness of the code nevertheless.
In addition, manual tests on a real robot were also performed whenever possible. Due to the high setup time these tests were intended as a final quality measure and not as a general tool for finding bugs.

4.2.4 Overview

On startup, the firmware first configures the required peripheral devices. This includes the UART (specifically *UART6*), the LCD controller and the external RAM used as a frame buffer for emWin. Care must be taken when using the external RAM in combination with another device (the LCD controller in this case) as external RAM is cached by the CPU. Since the LCD controller does not know about any CPU caches, writes to the frame buffers will not be visible to it until the writes are flushed to memory, causing visible artifacts on the screen [STM18c]. This can be prevented by using the microcontroller's MPU (memory protection unit). The MPU allows disabling write caching for certain memory regions [STM18c]. Reads can still be cached since only the CPU writes to the frame buffer. Additionally, the MPU is also used to disable access to a 4 KiB region starting at `0x00000000`. This helps catch dereferences of *null*-pointers—a common programming error—early. Otherwise, *null*-pointer dereferences would be perfectly valid, since the address `0x00000000` is mapped to the start of the internal flash memory (see subsection 4.2.1).
Finally, the FreeRTOS scheduler has to be started. It takes over the execution and starts scheduling tasks (FreeRTOS uses this term instead of thread). The *SysTick*, *SVCall*, and *PendSV* interrupts must use the handlers provided by FreeRTOS in order for FreeRTOS to function correctly. The *SysTick* interrupt must also run at the lowest possible interrupt priority [frec]. This conflicts with STMicroelectronics' HAL library, which expects to be called from the *SysTick* interrupt running at the highest priority [STM17]. As a workaround, the HAL library is called from a separate timer interrupt that is running at the highest priority and identical frequency to the *SysTick* interrupt.
Since tasks scheduled by FreeRTOS run concurrently, it is necessary to secure any shared state against concurrent accesses. This also applies to *Newlib*, the C

standard library implementation that is used. Its `malloc` implementation is used for all heap allocations in the standard library, including C++ containers like `std::vector`, but is not safe to use concurrently [new]. There are ways to secure it by providing a global lock; however, FreeRTOS already includes an optimized concurrency safe memory allocator [frea]. The linker's `--wrap` option [Fre16] is used to replace the malloc function with a custom implementation that simply calls the FreeRTOS allocator. This way all code automatically uses the correct allocator.

There are two tasks that have to run concurrently: one is processing incoming data and the other is updating the UI. Both of these tasks share two data structures that are each protected by a mutex. The `Log` object stores recent errors and profiling information whereas the `ControlTableMap` stores all currently known information about devices connected to the *RS-485* bus (for more detail see 4.2.6).

The packet processing task runs at a higher priority than the UI task. This means it will run until it deliberately yields control to lower priority tasks for a short amount of time. This way the UI task only gets a limited amount of processing time, leaving the rest for packet processing. If the UI task is currently holding the lock on a mutex and the packet processing task is trying to lock the same mutex, the FreeRTOS scheduler temporarily assigns the priority of the packet processing task to the UI task (priority inheritance). The UI task can now run until it releases the lock, at which point it reverts to its previous priority and is preempted by the packet processing task [free]. Again, the UI task only runs as long as it has to, freeing the rest of the time for packet processing.

The following two subsections describe the design of the two tasks in more detail. Subsection 4.2.5 describes the UI task, subsection 4.2.6 the packet processing task.

4.2.5 UI Task

The UI task is responsible for updating and drawing the UI as well as processing touch input. The task simply consists of an infinite loop calling the emWin function `GUI_Exec`, which handles user input and timers. A separate hardware timer interrupt polls the touch controller at 30 Hz and stores updates to its state in emWin's input queue.

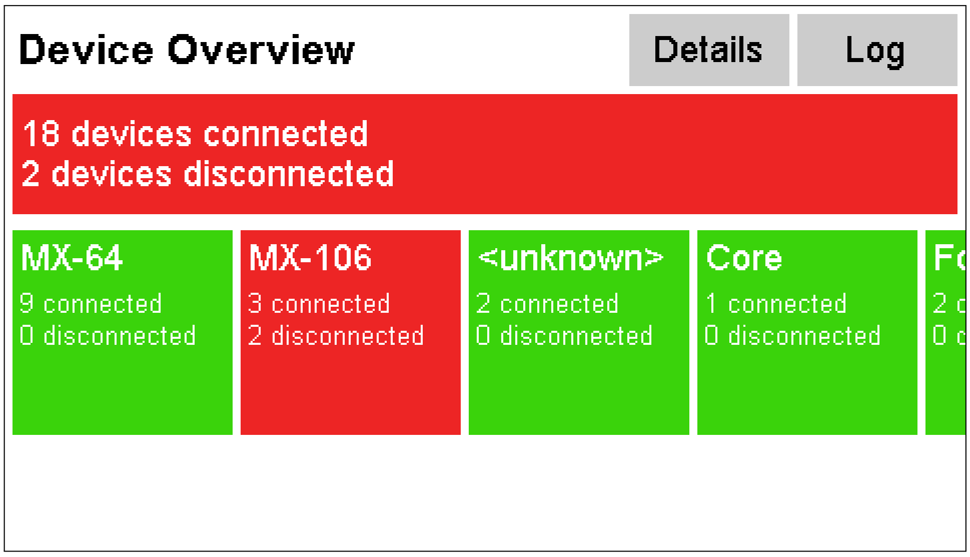
Listing 4.1: Main loop of the UI task

```
while (true) {
    GUI_Exec();
}
```

In emWin, every part of the user interface is a window: windows themselves, buttons, scrollbars, lists, etc. Every window is identified by a *handle*. A *handle* is an integer that emWin associates with a window. All functions operating on a

window take its handle as an argument. Some of these functions only apply to some types of windows, like for example functions for manipulating buttons [SEG17].
Every window also has a *callback* function associated with it. This function is called whenever the window receives a message. Messages can be user input, elapsed timers, notifications from other windows or even a request to draw itself. *Callbacks* can be overridden by supplying a new function to use instead. Often, this function only handles some messages and delegates the remaining messages to the original *callback* function. Most importantly, *callbacks* are a way to react to user input on a specific window [SEG17].
The goal of the UI is to make it easy to identify the status of devices at a glance. It is split into three separate views, each displaying information at a different level of detail: the device overview, the model overview, and the device details view. A fourth view displays profiling information and log messages. Every view is a distinct window that covers the entire screen. Color coding is used to differentiate between connected and disconnected devices. Every list item or button representing a connected device is colored green, those representing disconnected devices are colored red. The following four paragraphs give a brief overview of each view.

Device overview Displays the total number of devices and the number of devices for each model. It is the view shown when the application is started and is meant to give a brief summary of all devices. Each summary for a certain model can be clicked to open the model overview of that model. It also features buttons to reach the device details view and the log view. While this view does not provide much detail, disconnected devices are immediately visible. It is updated every 500 ms.



18 devices connected
2 devices disconnected

MX-64
9 connected
0 disconnected

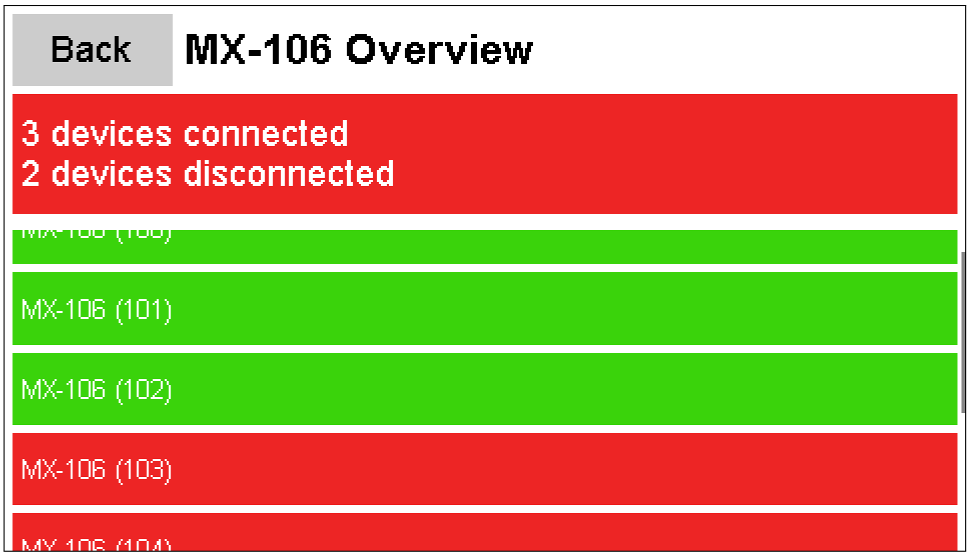
MX-106
3 connected
2 disconnected

<unknown>
2 connected
0 disconnected

Core
1 connected
0 disconnected

Figure 4.4: Screenshot of the device overview. The top bar is red because at least one device (two in this case) is disconnected. The squares below the bar show the status of the devices of a specific model. They also turn red if at least one device is disconnected. The list containing them is horizontally scrollable. Each square can be clicked and will open the model overview for the corresponding model.

Model overview Displays the status and ID of each device of a certain model. Each device in the list can be clicked to open the device details view and select that device. This view does not provide as much detail as the device details view but it only shows devices of one particular model, making it easy to find the exact device that is disconnected. It is updated every 500 ms.



Back

MX-106 Overview

3 devices connected
2 devices disconnected

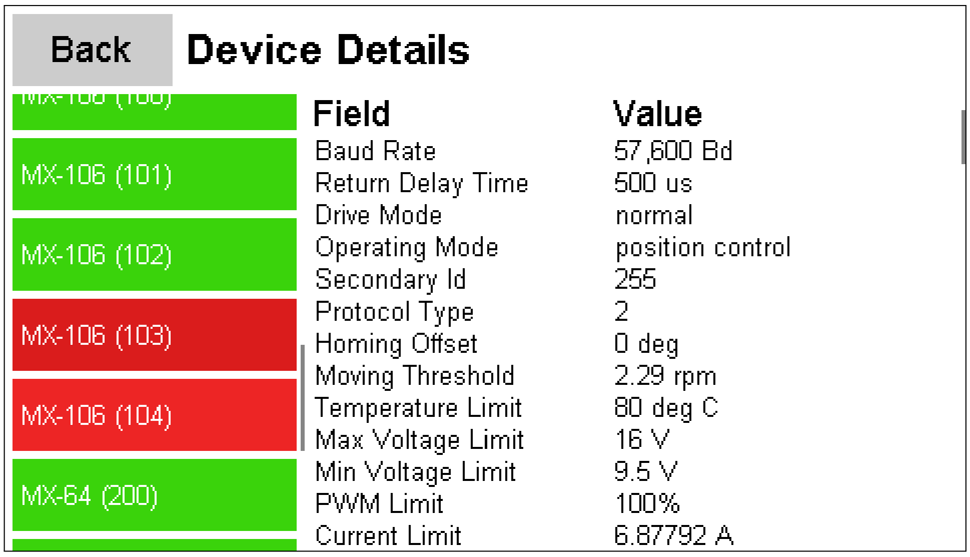
MX-106 (101)

MX-106 (102)

MX-106 (103)

Figure 4.5: Screenshot of the model overview. The top bar is red because at least one device for the selected model (MX-106) is disconnected. Below the bar is a scrollable list with an item for each device of the selected model. The device's ID is shown in parentheses and the item is colored red when it is disconnected. Clicking an item will navigate to the device details view and select the clicked device.

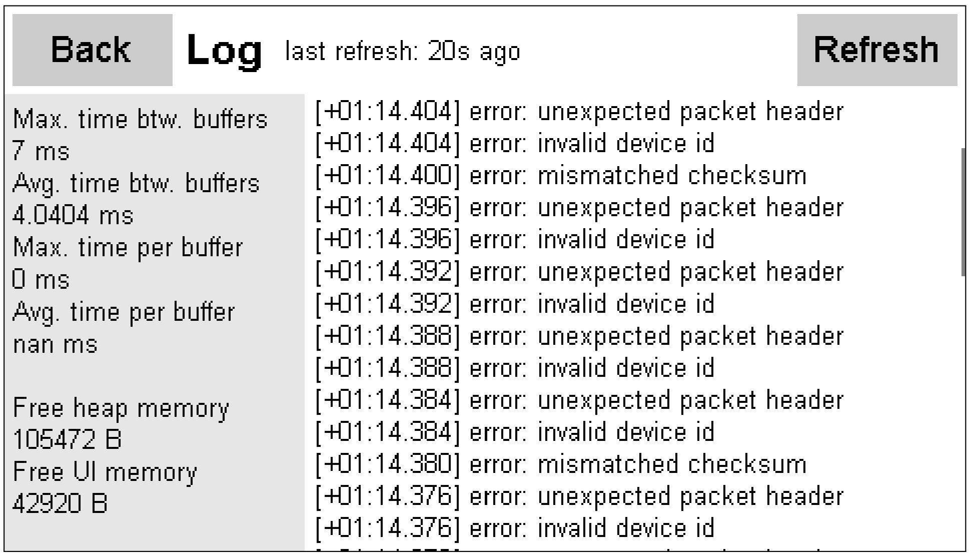
Device details view Displays all known values of a single device's *control table*. Different devices can be selected by using the list on the left-hand side. This view provides the maximum amount of detail per device but makes it harder to determine the status of all connected devices at once. It is updated every 500 ms.



Field	Value
Baud Rate	57,600 Bd
Return Delay Time	500 us
Drive Mode	normal
Operating Mode	position control
Secondary Id	255
Protocol Type	2
Homing Offset	0 deg
Moving Threshold	2.29 rpm
Temperature Limit	80 deg C
Max Voltage Limit	16 V
Min Voltage Limit	9.5 V
PWM Limit	100%
Current Limit	6.87792 A

Figure 4.6: Screenshot of the device details view. The list on the left-hand side contains an item for each connected device that is colored green when the device is connected and red otherwise. Clicking an item will select that device and show all values of its *control table* in the table on the right-hand side.

Log view Displays profiling information such as free memory and maximum and average processing times, as well as the last 50 errors that occurred during packet processing. This view is mostly intended for debugging. It has to be refreshed manually since a lot of errors may occur in short bursts, making it impossible to actually read an individual error message.



Max. time btw. buffers
7 ms
Avg. time btw. buffers
4.0404 ms
Max. time per buffer
0 ms
Avg. time per buffer
nan ms

Free heap memory
105472 B
Free UI memory
42920 B

```
[+01:14.404] error: unexpected packet header
[+01:14.404] error: invalid device id
[+01:14.400] error: mismatched checksum
[+01:14.396] error: unexpected packet header
[+01:14.396] error: invalid device id
[+01:14.392] error: unexpected packet header
[+01:14.392] error: invalid device id
[+01:14.388] error: unexpected packet header
[+01:14.388] error: invalid device id
[+01:14.384] error: unexpected packet header
[+01:14.384] error: invalid device id
[+01:14.380] error: mismatched checksum
[+01:14.376] error: unexpected packet header
[+01:14.376] error: invalid device id
```

Figure 4.7: Screenshot of the log view. The left-hand side contains profiling information like processing times and free memory, the right-hand side shows the 50 most recent error messages. The timestamps in the error log are relative to the start of the application. The view is only refreshed when the *Refresh* button is clicked.

All of the views have to regularly access data shared with the packet processing task. While they are holding the lock for this data, the packet processing task may be blocked on this lock. This makes it extremely important to minimize the time spent holding any locks. All of the views do this by copying the needed data before processing it. Thus, they only ever hold a lock to make copies, which is relatively quick compared to updating the entire UI at the same time. This does decrease the theoretical performance of the UI; however, the UI only needs to be fast enough to not appear slow to the user. In comparison, the packet processing task has strict deadlines. If it is blocked for too long, it will miss incoming data.

4.2.6 Packet Processing Task

The packet processing task is responsible for processing the data the UART connected to the *RS-485* bus receives. Since the bus has a data rate of 2 MBd,

performance is crucial. Data must be processed faster than it arrives in order to not lose any of it. At the same time, there needs to be enough processing time left to also update to UI.
The code is designed to work with more than one bus at the same time. Each bus is associated with a `Connection` object. It holds the state of the packet parser, some profiling information and a pointer to a buffer that stores received data. At the moment, only a single bus connected to *UART6* is supported.

Listing 4.2: Main loop of the packet processing task

```
while (true) {
    for (auto& connection : connections) {
        if (connection.last_processing_start == 0) {
            connection.last_processing_start = HAL_GetTick();
        }

        process_buffer(log, connection, control_table_map);
    }

    vTaskDelay(4 / portTICK_PERIOD_MS);
}
```

The buffer is automatically filled by the DMA controller. It transfers the bytes from the UART's receive register and raises interrupts when half of or the entire buffer has been filled. Once it has been filled, the DMA controller starts at the beginning of the buffer again. The interrupt handler sets a flag that determines which part of the buffer is now ready. As long as the data is processed faster than it is received, no data will be lost. Even when the data is not processed quickly enough, there will be no serious malfunctions. Some data may be corrupted but this will usually be detected by the CRC checksums that are part of the ROBOTIS Dynamixel protocol (see section 3.2).
The size of the buffer has a significant impact on the maximum time allowed before data is lost. A larger buffer increases this time but also increases the latency. The latency can largely be ignored because it is still low (for human time scales) given the possible buffer sizes. Latency would increase drastically if there were no or only very little traffic on the bus, as data is only ever processed once one half of the buffer is ready. Other than by memory constraints, the size of the buffer is limited to 65 535 B by the size of the DMA controller's NDT register [STM18c].
The buffer's size is 8192 B. This means that even assuming the absolute worst case, the maximum amount of time for processing one half of the buffer is $\frac{4096\,\mathrm{B}}{2\,\mathrm{MBd}/8} \approx 16\,\mathrm{ms}$. Realistically, the maximum amount of time will be higher for multiple reasons:

- The baud rate of the bus is not equivalent to the bit rate of the incoming data, since this equation ignores overhead like stop bits.

- In practice, it is impossible to have 100 % load on the bus.
- The equation assumes that processing starts right after an interrupt signals that one half of the buffer is ready, and does not actually process a single byte. Normally, data is then being processed, which continuously moves the point at which a collision with the DMA controller can occur.

For these reasons, if the maximum amount of time between processing one half of the buffer never exceeds 16 ms, there will be no data loss or corruption.

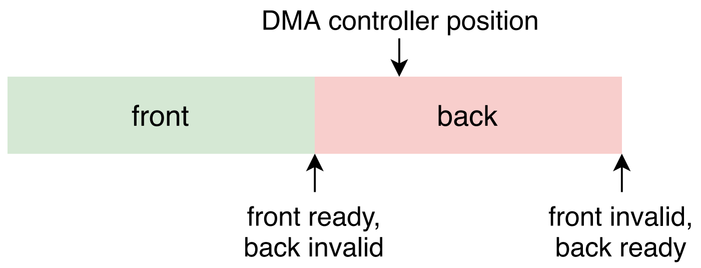


Figure 4.8: After more than half of the DMA buffer has been filled, the front of it is ready and can be read. At the same time, the back of it is now invalid because the DMA controller is writing to it. The situation is reversed when the DMA controller has finished writing to the back and starts at the front again.

Since the buffer is written to by the DMA controller and then read by the CPU, the CPU must not cache reads. This is accomplished by placing the buffer in the DTCM RAM section of the microcontroller's memory. DTCM RAM is never cached [STM18c].
After processing one half of each buffer per bus, control is yielded to the UI task for 4 ms. The UI can only be updated during this time. Increasing this time also increases the responsiveness of the UI while increasing the time spent between the processing of data.
The `process_buffer` function then parses packets until all the data in the ready buffer has been consumed. Each successfully parsed packet is passed to `ControlTableMap::receive`, which processes the contents of the packets. Errors and profiling information are passed to the `Log` object afterward.
Due to this, the mutexes for the `ControlTableMap` and the `Log` object are never locked at the same time. The UI task also never locks more than one of these mutexes at a time. Holding only one lock at the same time prevents deadlocks.
The lock for the `ControlTableMap` object is intentionally held for almost the whole duration of the call. Unlike with the UI task where holding the lock for a long time would block the packet processing task, there is no task to block since the only

other task is the UI task which can only run when the packet processing explicitly yields control. Maximizing the time holding a mutex removes the overhead of locking it multiple times.

Parser The `Parser` is responsible for parsing packets from the ready part of a buffer. It consumes bytes from a `Cursor`. A `Cursor` is essentially a pointer to a part of the buffer that also tracks how many bytes have already been read. This enables a simple API for the `Parser` itself. Each call to the `parse` method either parses one packet successfully, encounters an error or parses a packet partially. In all cases, the `Cursor` tracks the current position and indicates when all bytes have been consumed, while the `Parser` holds state that must be retained across parses. Most importantly, it stores the previous three bytes. When inspecting the next byte there are three possible scenarios:

- the byte and the previous three bytes are the header of a new packet
- the byte is stuffing (it can be ignored)
- the byte is a regular byte

In case the header of a packet is detected while already parsing a packet, an error is returned. Returning early when encountering an error or only partially parsing a packet is possible because the `Parser` stores the part of the packet currently being parsed and the incremental CRC checksum.

Listing 4.3: Definition of the `Packet` struct

```
struct Packet {
    DeviceId device_id;
    Instruction instruction;
    Error error;
    std::vector<uint8_t> data;
};
```

The speed of the `Parser` is critical to performance. To avoid allocating memory while parsing, the `parse` method takes a pointer to a `Packet` struct as argument. Since packets are processed sequentially, the same `Packet` struct can be reused. Packets themselves can have different lengths but when defining a maximum allowed packet length, the memory for a `Packet` of that length can be preallocated. Defining a maximum packet length also makes sense in order to avoid running out of memory due to packets with incorrect *Length* fields.
The `Packet` struct represents both instruction and status packets since the only difference between them is the addition of an *Error* field for status packets (see section 3.2). For instruction packets the `error` member can be ignored because it will never contain an error.

ControlTableMap The `ControlTableMap` object then processes these `Packet` structs further. Status packets are left as is but instruction packets are parsed into `InstructionPacket` structs. An `InstructionPacket` is a discriminated union of structs for each instruction. Because the instruction specific fields are part of the payload of a packet, parsing them is simple. Unlike the packet parser, the instruction packet parser is just a function. It also reuses previous `InstructionPackets` but does allocate memory. Further optimization was not required to achieve the desired performance.

Listing 4.4: Definition of the `InstructionPacket` struct

```
struct InstructionPacket {
    // constructors, destructor and member functions omitted

    Instruction instruction;
    union {
        PingArgs ping;
        ReadArgs read;
        WriteArgs write;
        RegWriteArgs reg_write;
        ActionArgs action;
        FactoryResetArgs factory_reset;
        RebootArgs reboot;
        ClearArgs clear;
        SyncReadArgs sync_read;
        SyncWriteArgs sync_write;
        BulkReadArgs bulk_read;
        BulkWriteArgs bulk_write;
    };
};
```

For every received instruction packet, the `ControlTableMap` records the devices that are expected to respond. When a new instruction packet is received, a counter for each device from which a status packet was not received is incremented. These counters are reset when a status packet from the corresponding device has been received. Devices that have not responded to more than four instruction packets are considered disconnected.
Ping instructions are handled differently: the status packet responding to a *Ping* instruction contains the model number of the device. The model number is used to register a new device. Each device is mapped to a `ControlTable` object that stores the current state of the device's *control table*. If the device is not registered, has a different or an unknown model number, a new `ControlTable` matching the model number is allocated. A device's model can be unknown if status packets from that device have been received without previously receiving the response to a *Ping* instruction.
Whenever data is written to or read from a device, the written or read data is

also updated for the `ControlTable` object belonging to that device. This is the data that is displayed by the UI. Only the following instructions are handled properly:

- *Ping*
- *Read*
- *Write*
- *Sync Read*
- *Sync Write*
- *Bulk Read*
- *Bulk Write*

The remaining instructions are mostly ignored but still used for detecting disconnected devices. Adding support for them would be a significant effort especially considering that these instructions do not appear in most traffic. The current implementation also assumes that writes do not require a status packet response, as that is how the *Wolfgang* robot platform is configured.
The `ControlTable` objects are identified by the ID of the device they belong to. Initially, a `std::unordered_map` was used but the performance proved to be insufficient. Instead, the objects are stored in a custom data structure. It takes advantage of the fact that device IDs are only one byte and that `ControlTable` objects must be accessed through a pointer (four bytes on a Cortex M7 processor [STM18c]). The objects are simply stored in an array that is indexed by the device ID. An additional boolean flag indicates the presence or absence of the object. The entire array only requires $256 \cdot (4+1+3) = 2048\,\mathrm{B}$ of memory (this includes three bytes of padding). This solution trades memory and iteration speed for fast access times. Since objects must be accessed for almost every received packet, this provides a serious speedup compared to `std::unordered_map`.

ControlTable The *control table* of each device is represented by a `ControlTable` object. For each device model, there is a different subclass of `ControlTable`. Most received packets update the `ControlTable` object of at least one device, performance is thus an important design consideration. Adding or changing the `ControlTable` of a device has to be easy.
The interface definition of the `ControlTable` base class can be seen in listing 4.5. `is_unknown_model` determines if the model number is known and `model_number` returns its value if present. `set_firmware_version` is called when the firmware version is reported by the response to a *Ping* instruction. `device_name` simply returns a

human-readable name of the model represented by the `ControlTable`. It is used for display purposes only.

Listing 4.5: Definition of the `ControlTable` class

```cpp
class ControlTable {
  public:
    virtual ~ControlTable() = default;

    virtual std::unique_ptr<ControlTable> clone() const = 0;

    virtual bool is_unknown_model() const;

    virtual uint16_t model_number() const = 0;

    virtual void set_firmware_version(uint8_t version) = 0;

    virtual const char* device_name() const = 0;

    virtual ControlTableMemory& memory() = 0;

    virtual const ControlTableMemory& memory() const = 0;

    virtual const std::vector<ControlTableField>& fields() const =
        0;

    bool write(uint16_t start_addr, const uint8_t* buf, uint16_t
       len);

    std::vector<std::pair<const char*, std::string>> fmt_fields()
       const;
};
```

The core functionality of `ControlTable` class uses the `memory` and `fields` methods. The `ControlTableMemory` object returned by a call to `memory` is responsible for storing the actual data. It is made up of multiple `Segments` that each describe a region of memory:

- data segments are some memory starting at a certain address
- indirect address segments allow redirecting addresses (required to support ROBOTIS *MX-64* and *MX-106* servos)
- unknown segments allow no reads and ignore writes (these are used when the model number is not known)

The descriptions are used by the `ControlTableMemory` object to map data contained in packets to a single, flat byte array. Calls to the `write` method of a `ControlTable`

object are forwarded to the `ControlTableMemory`. By using an accessor, only one virtual call has to be performed, even when writing more than one byte.
The `fields` method returns a description of all fields in a device's *control table*. Each `ControlTableField` stores the address, type, name, and default value of a single field. It also contains a function for formatting the value of the field as a human readable string.

Listing 4.6: Definition of the control table fields of a Rhoban DXL Board

```
const std::vector<ControlTableField> CoreBoardControlTable::FIELDS
    {
  ControlTableField::new_uint16(0, "Model Number",
      CoreBoardControlTable::MODEL_NUMBER, fmt_number),
  ControlTableField::new_uint8(2, "Firmware Version", 0,
      fmt_number),

  ControlTableField::new_uint16(10, "LED", 0, fmt_bool_on_off),
  ControlTableField::new_uint16(12, "Power", 0, fmt_number),
  ControlTableField::new_uint32(14, "RGB LED 1", 0, fmt_core_rgb),
  ControlTableField::new_uint32(18, "RGB LED 2", 0, fmt_core_rgb),
  ControlTableField::new_uint32(22, "RGB LED 3", 0, fmt_core_rgb),
  ControlTableField::new_uint16(26, "VBAT", 0, fmt_core_voltage),
  ControlTableField::new_uint16(28, "VEXT", 0, fmt_core_voltage),
  ControlTableField::new_uint16(30, "VCC", 0, fmt_core_voltage),
  ControlTableField::new_uint16(32, "VDXL", 0, fmt_core_voltage),
  ControlTableField::new_uint16(34, "Current", 0, fmt_core_current
      ),
  ControlTableField::new_uint16(36, "Power On", 0,
      fmt_core_power_on),
};
```

These descriptions are used by the `fmt_fields` method. It reads the current value of a field from the `ControlTableMemory` and formats it using the formatting function. This function is used to display the contents of a device's *control table*. Formatting all fields is quite slow, which is why the UI task first copies the entire `ControlTable` object using the `clone` method. It effectively acts as a virtual copy constructor.
To add support for a new device model, one only has to create a new subclass of `ControlTable` and construct an instance of it when a response to a *Ping* instruction is received. Currently, the following device models are supported:

- ROBOTIS MX-64 servos (`http://emanual.robotis.com/docs/en/dxl/mx/mx-64-2/`)
- ROBOTIS MX-106 servos (`http://emanual.robotis.com/docs/en/dxl/mx/mx-106-2/`)
- Rhoban DXL Boards (`https://github.com/Rhoban/DXLBoard`)

- BitFoot foot pressure sensors (`https://github.com/bit-bots/bit_foot`)
- *Hamburg Bit-Bots* IMU modules (`https://github.com/bit-bots/imu_module`)

5 Evaluation

This chapter evaluates the performance of the implementation described in chapter 4. Two measurements were taken to assess the performance:

- The time required to process the ready half of the receive buffer (see subsection 4.2.6). This measurement will be called **time per buffer** in the following.

- The time that passes between two calls of the `process_buffer` function (again, see subsection 4.2.6). This measurement will be called **time between buffers** in the following.

The *time per buffer* measures the performance of the part of the system that actually deals with incoming data. It does not measure idle time and synchronization overhead. The *time between buffers* on the other hand measures the performance of the entire system. It indicates how much time passes between two opportunities to process incoming data. This means that if this time does not exceed the maximum amount of time of 16 ms calculated in subsection 4.2.6, no incoming data will be lost.
In addition, the perceived responsiveness of the UI is also reported. These observations are entirely subjective and only meant to give an impression of the responsiveness to be expected.
The source code of the firmware and scripts used for measurements as well as the results can be found at `https://github.com/Laegluin/embedded_debug_interface_for_robots/tree/thesis-benchmark-ref`. The version used for measurements is tagged as `thesis-benchmark-ref`.
All code was compiled with GCC 6.3.1 with maximum optimizations, link-time optimization and no exception support (`-O3 -flto -fno-exceptions`).
Section 5.1 describes the method used to obtain the measurements and section 5.2 presents the results.

5.1 Measurement Method

The FT232R USB to UART converter listed in section 4.1 was used to simulate an actual *RS-485* bus. Three different kinds of simulated traffic were used:

- A trace of real traffic from the bus of a robot of the *Hamburg Bit-Bots*.

- Synthetic traffic of 20 devices consisting of *Ping* instructions followed by *Read*, *Write*, *Sync Read*, *Sync Write*, *Bulk Read* and *Bulk Write* instructions for each device, repeated 300 times.
- Synthetic traffic of 20 devices that are constantly responding to *Ping* instructions. Devices change their model number after each *Ping* instruction.

The simulated traffic was sent to the USB to UART converter in two ways:

- the entire file of simulated traffic at once
- one packet at a time, with 100 µs delay per packet

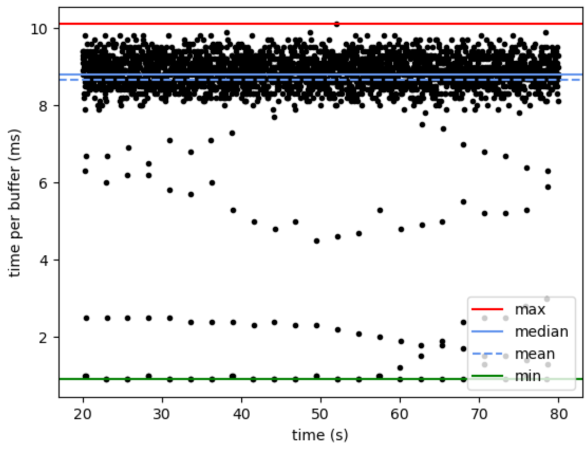
A python script was used to send the simulated traffic. Output buffering was disabled. To minimize the system call overhead when not adding any delay, the files containing the simulated traffic all had a size of at least 400 KiB. When the file was smaller, it was made bigger by simply repeating the contents.
All combinations of the aforementioned parameters (for a total of six different combinations) were tested as separate experiments. Before starting the experiments, the microcontroller was reset and every view of the UI was opened at least once to ensure that every view would be updated. Measurements were taken for at least 80 s after reset.
The firmware had to be modified slightly to allow for the collection of the measurements. An additional timer was used to generate an interrupt every 0.1 ms. The interrupt handler incremented a counter every time it was called. This counter was used in calls to `process_buffer` to determine the current time. Thus, every measurement has an accuracy of 0.1 ms.
Measurements were stored on external RAM. After enough measurements had been taken, the microcontroller was halted using the debugger. The debugger was then used to dump the memory that stored the measurements. The raw data was converted to a JSON file and then plotted using two python scripts.
The interrupts generated by the additional timer and the extra code to store the measurements on comparatively slow external RAM imposed overhead that is not present during normal execution. As a consequence, all times measured are an upper bound on the expected values. The differences should be minimal, however.

5.2 Results

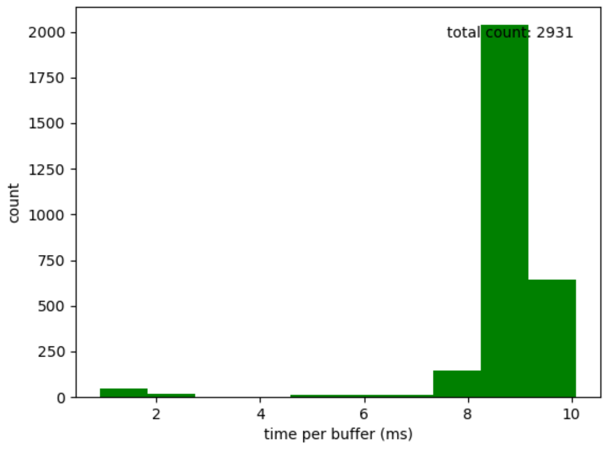
This section presents the results of the experiments for each of the six combinations mentioned above. Every experiment is presented in a separate subsection. There are four different plots for each experiment. The scatterplots show the *time per buffer* and *time between buffers* over time. Since each data point represents a

duration, points in the plots are placed at the geometric mean of the start- and endpoint of the duration. The histograms show the distribution of durations for each scatterplot.
Subsection 5.2.7 contains tables for maximum, median, mean and minimum *time per buffer* and *time between buffers* for each experiment. It also shows the data rates that can be derived from the results in an additional table.
The first 20 s of measurements were discarded to allow for the manual setup, the next 60 s were used to generate the following plots and tables.

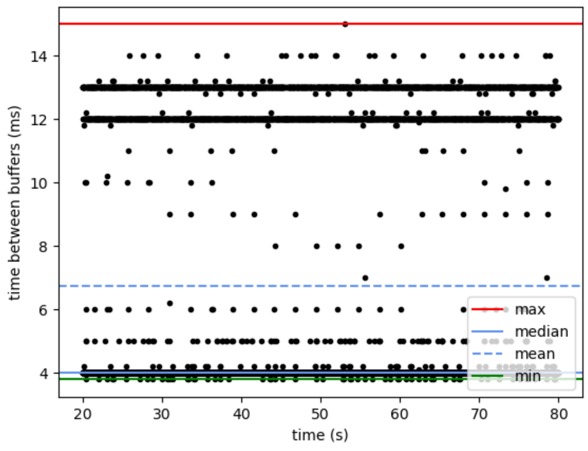
5.2.1 Trace, no delay



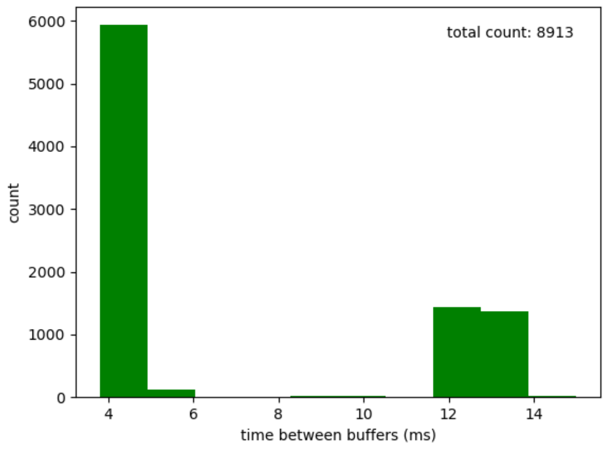
(a) Scatterplot of *time per buffer*



(b) Histogram of *time per buffer*



(c) Scatterplot of *time between buffers*



(d) Histogram of *time between buffers*

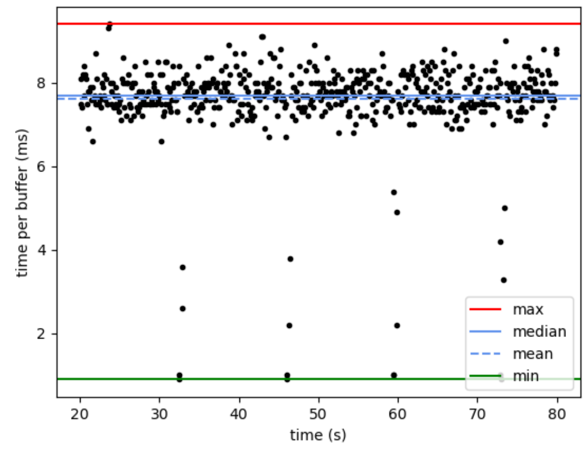
Figure 5.1: *Time per buffer* and *time between buffers* for trace traffic with no delay

UI responsiveness The UI was responsive, there was regular but short stuttering.

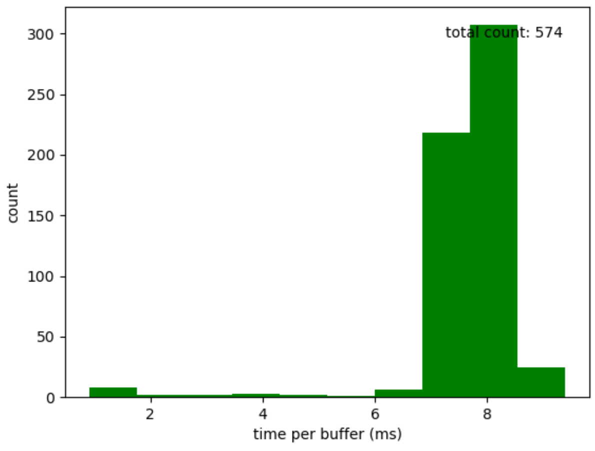
Most of the measurements of *time per buffer* are clustered around 8–10 ms. There are noticeable exceptions that occur at regular intervals. These are most likely packets that were not transmitted correctly at the time the trace was recorded. Because they were malformed, they were ignored after parsing, leading to less time

spent processing the buffer that contained these errors. Transmission errors that occurred during the measurements are unlikely to have had a significant impact due to the extremely short cable length. In addition, these errors would not appear at regular intervals, while recorded errors would be repeated every time the trace is repeated, which happened frequently since the size of the trace was only 516 KiB.
There are two distinct spikes in the *time between buffers*. The higher spike is caused by those measurements that were taken while a buffer was being processed. Most of the measurements are about 4 ms. These are the measurements taken when there was no data to process. They are the majority, indicating that higher data rates could be handled without a problem.
It is unclear why the *time between buffers* is so tightly clustered around certain times. It is possible that these clusters were caused by synchronization with one of the mutexes. Since the time the UI task was holding a lock was always the same and since FreeRTOS could only preempt a task every millisecond, there was only a very limited set of possible wait times for the packet processing task. These wait times may have been what caused the tight clustering.

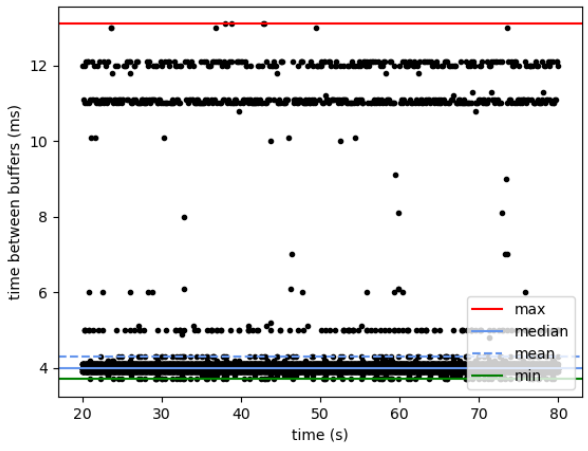
5.2.2 Trace, 100 µs delay



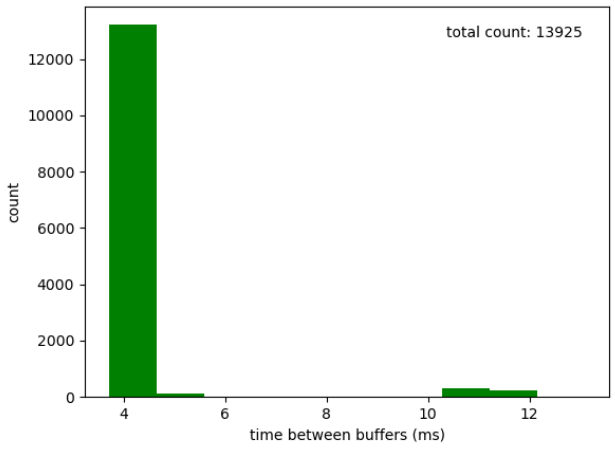
(a) Scatterplot of *time per buffer*



(b) Histogram of *time per buffer*



(c) Scatterplot of *time between buffers*



(d) Histogram of *time between buffers*

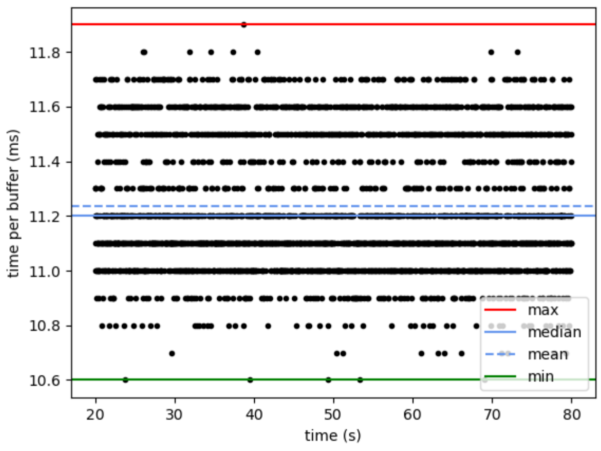
Figure 5.2: *Time per buffer* and *time between buffers* for trace traffic with 100 µs delay

UI responsiveness The UI was responsive, there was only occasional minimal stuttering.

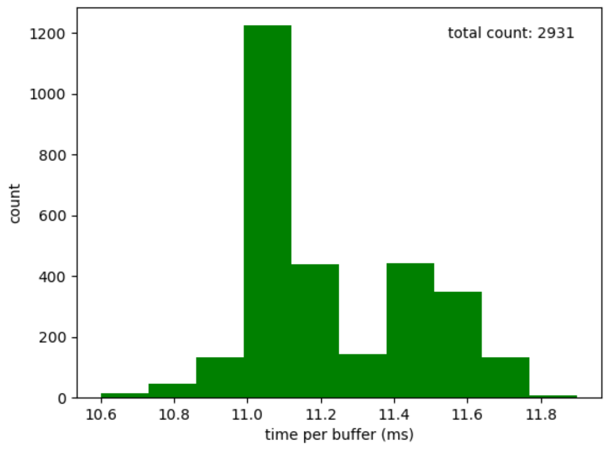
All measurements look extremely similar to those in the experiment with no delay. Since the additional delay reduced the effective data rate (see table 5.3), there were

fewer packets processed in the same time frame and the influence of malformed packets was not as severe.
The *time between buffers* is absolutely dominated by measurements taken when there was no data to process. This is not surprising, as even less time was spent processing data when there was also a delay per packet.

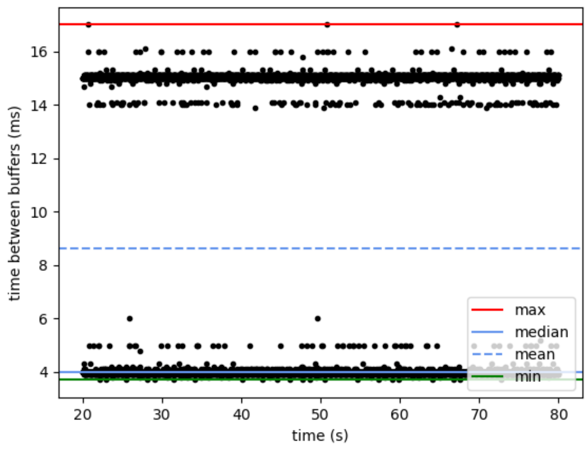
5.2.3 Synthetic Read/Write instructions, no delay



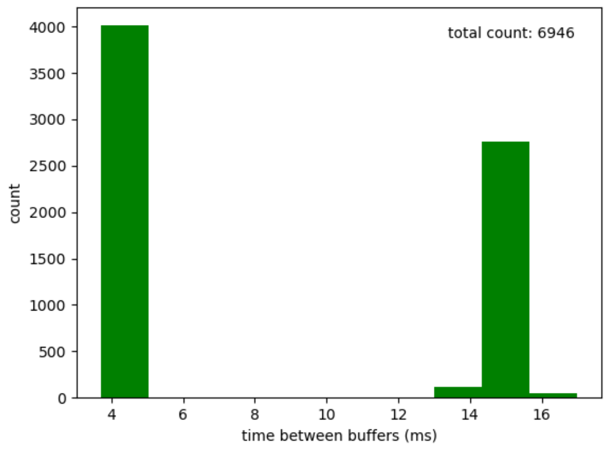
(a) Scatterplot of *time per buffer*



(b) Histogram of *time per buffer*



(c) Scatterplot of *time between buffers*



(d) Histogram of *time between buffers*

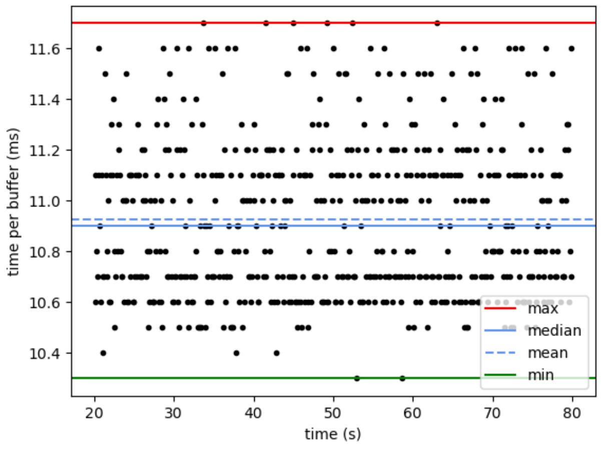
Figure 5.3: *Time per buffer* and *time between buffers* for synthetic read/write traffic with no delay

UI responsiveness The UI was still usable but there was heavy stuttering. Using the UI was not pleasant.

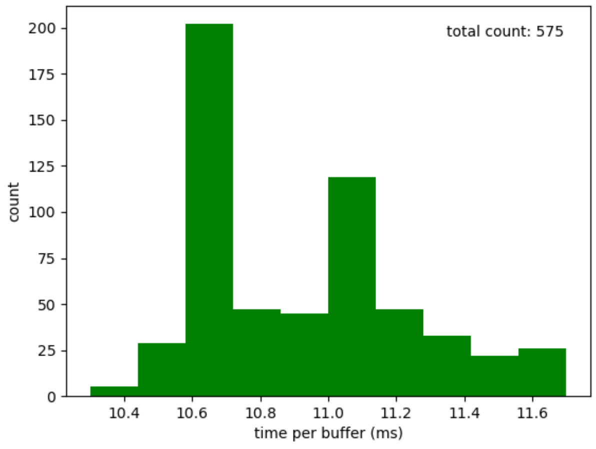
Compared to the experiments using the recorded trace, there are almost no outliers

in the measurements. The synthetic nature of the data is clearly visible. Measurements are extremely clustered and all clusters are very close to each other.
These clusters were most likely caused by the different types of packets that are part of the data. The same types of packets were always sent as a sequence. Every type had a slightly different length and took a slightly different path through the program. This did not make a significant difference to the overall time but it was enough to consistently change the time it took to process a buffer, depending on the types of packets it contained.

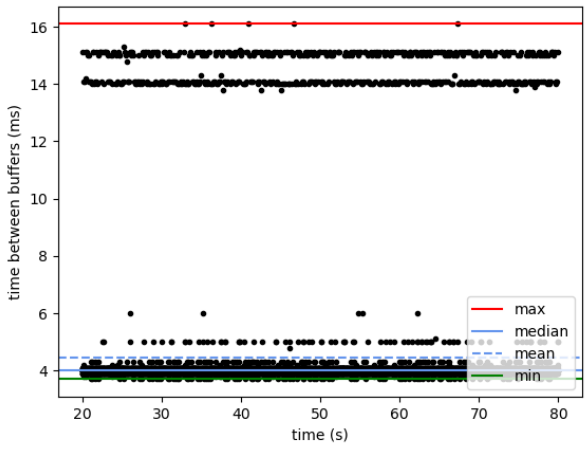
5.2.4 Synthetic Read/Write instructions, 100 µs delay



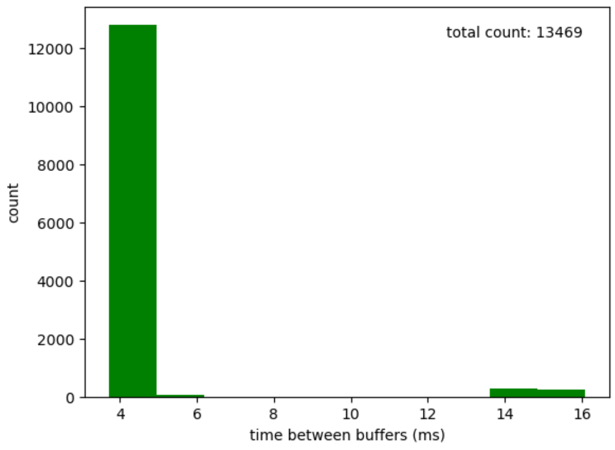
(a) Scatterplot of *time per buffer*



(b) Histogram of *time per buffer*



(c) Scatterplot of *time between buffers*



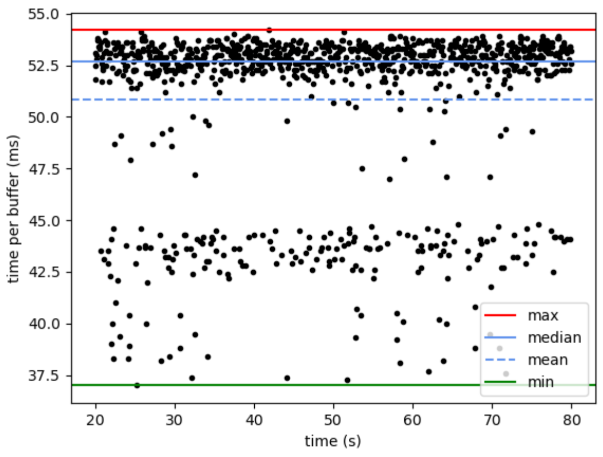
(d) Histogram of *time between buffers*

Figure 5.4: *Time per buffer* and *time between buffers* for synthetic read/write traffic with 100 µs delay

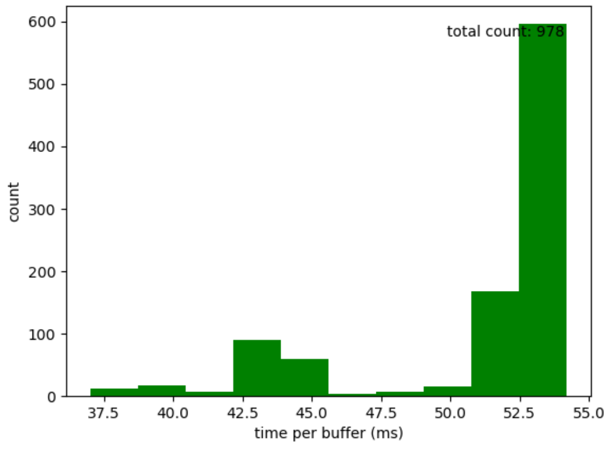
UI responsiveness The UI was responsive, there was only occasional short stuttering.

Like the experiment using the recorded trace, results when adding a delay per packet are very similar to those without delay. Again, clusters in the *time per buffer* plots are less pronounced as there are less measurements and the measurements for the *time between buffers* are dominated by constant overhead.

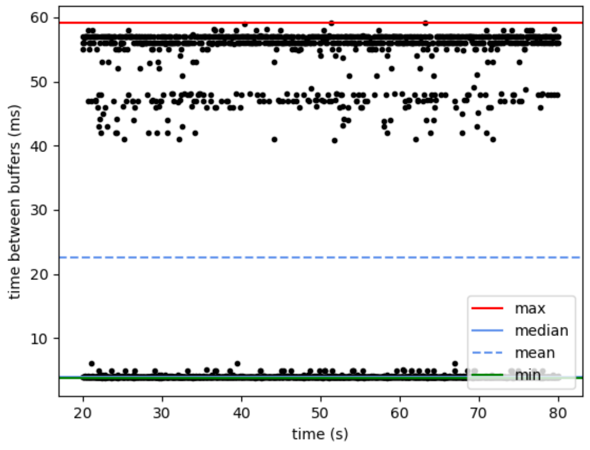
5.2.5 Synthetic Ping instructions, no delay



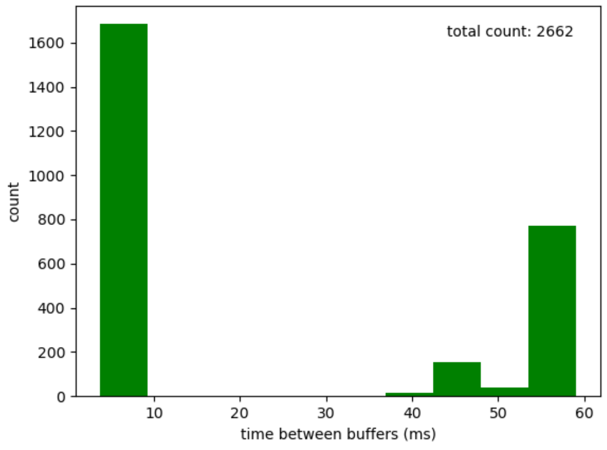
(a) Scatterplot of *time per buffer*



(b) Histogram of *time per buffer*



(c) Scatterplot of *time between buffers*



(d) Histogram of *time between buffers*

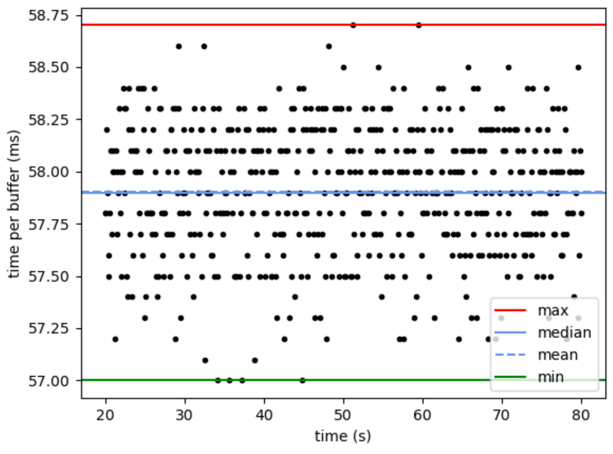
Figure 5.5: *Time per buffer* and *time between buffers* for synthetic ping traffic with no delay

UI responsiveness The UI was not responsive at all. While it was still possible to use, scrolling did not work properly and any action caused significant and noticeable delays.

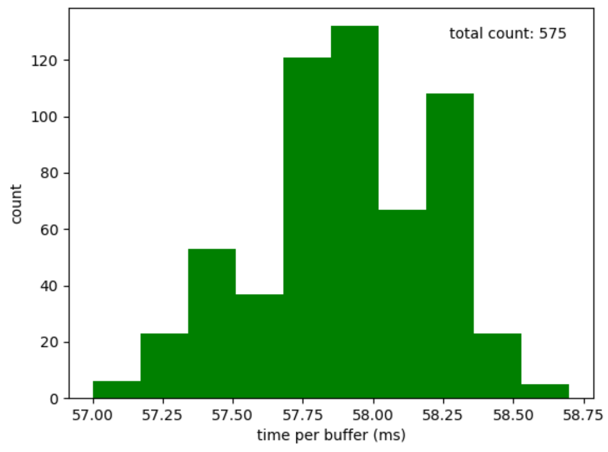
The results for the *time per buffer* are significantly worse than in the previous

experiments. The maximum *time per buffer* exceeds 50 ms and there are many outliers. These outliers were most likely caused by errors during parsing: since the buffers were not processed quickly enough, the DMA controller was overwriting buffers as they were being read.
The *time between buffers* reflects the measurements for *time per buffer*. However, it is notable that there are many measurements of only about 4 ms, even though the processing was too slow. This is because after processing one buffer, there was a high likelihood that the next buffer was not currently ready. Data was not lost during these waits but during the processing itself, meaning that the number of waits only decreased because more time was spent processing data.
Because so much time was spent processing data, the amount of time dedicated to the UI decreased and the latency increased. The high latencies caused problems with input handling since input could not be processed in time. Less time for the UI task also meant the real-time required for rendering the UI increased.

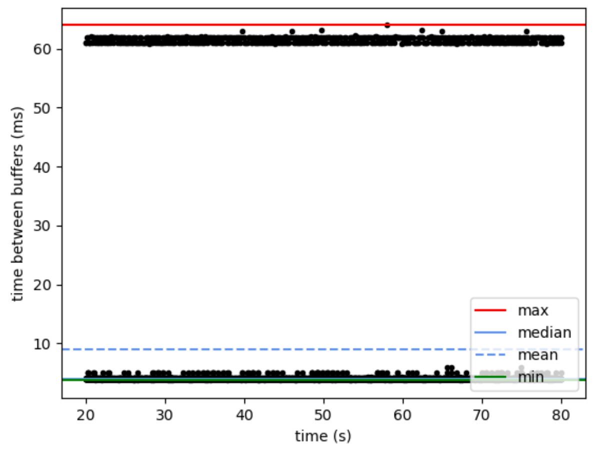
5.2.6 Synthetic Ping instructions, 100 µs delay



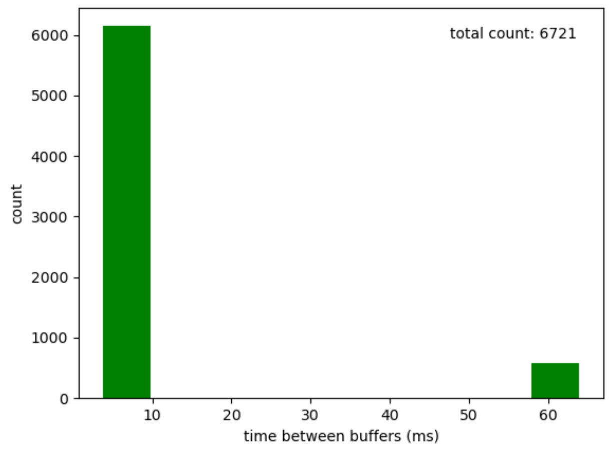
(a) Scatterplot of *time per buffer*



(b) Histogram of *time per buffer*



(c) Scatterplot of *time between buffers*



(d) Histogram of *time between buffers*

Figure 5.6: *Time per buffer* and *time between buffers* for synthetic ping traffic with 100 µs delay

UI responsiveness The UI was mostly usable but there was significant stuttering. Scrolling mostly worked but was not pleasant to use. There were noticeable but not significant delays when interacting with buttons.

Results with delay per packet look remarkably similar to those for the experiments

with synthetic *Read* and *Write* instructions. The only difference is that all times increased drastically to about 57–59 ms. The reason is that while the average *time per buffer* was still about four times higher than the allowed maximum of 16 ms, the delay per packet slowed down the effective data rate enough to make the actual allowed maximum greater than the measured maximum *time per buffer*.
This can clearly be seen when comparing the effective data rates in table 5.3: the effective data rates with added delay are all comparable, whereas the effective data rate for synthetic *Ping* instructions with no delay is significantly lower than the others.
It is also the reason the UI was so much more responsive than without any delay. The problems caused by high latencies were still present but the UI task did have enough time to render the UI at a sufficient speed.

5.2.7 Results by Experiment

Experiment	**Maximum**	**Median**	**Mean**	**Minimum**
Trace, no delay	10.1 ms	8.8 ms	8.6 ms	0.9 ms
Trace, 100 µs delay	9.4 ms	7.7 ms	7.6 ms	0.9 ms
Synthetic Read/Write instructions, no delay	11.9 ms	11.2 ms	11.2 ms	10.6 ms
Synthetic Read/Write instructions, 100 µs delay	11.7 ms	10.9 ms	10.9 ms	10.3 ms
Synthetic Ping instructions, no delay	54.2 ms	52.7 ms	50.8 ms	37.0 ms
Synthetic Ping instructions, 100 µs delay	58.7 ms	57.9 ms	57.9 ms	57.0 ms

Table 5.1: Maximum, median, mean and minimum *time per buffer* for each experiment

Experiment	**Maximum**	**Median**	**Mean**	**Minimum**
Trace, no delay	15.0 ms	4.0 ms	6.7 ms	3.8 ms
Trace, 100 µs delay	13.1 ms	4.0 ms	4.3 ms	3.7 ms
Synthetic Read/Write instructions, no delay	17.0 ms	4.0 ms	8.6 ms	3.7 ms
Synthetic Read/Write instructions, 100 µs delay	16.1 ms	4.0 ms	4.5 ms	3.7 ms
Synthetic Ping instructions, no delay	59.1 ms	4.0 ms	22.6 ms	3.7 ms
Synthetic Ping instructions, 100 µs delay	64.0 ms	4.0 ms	8.9 ms	3.7 ms

Table 5.2: Maximum, median, mean and minimum *time between buffers* for each experiment

Experiment	**Effective data rate**	**Processing data rate**
Trace, no delay	1 600 717 bit/s	3 788 589 bit/s
Trace, 100 µs delay	313 481 bit/s	4 306 544 bit/s
Synthetic Read/Write instructions, no delay	1 600 717 bit/s	2 916 174 bit/s
Synthetic Read/Write instructions, 100 µs delay	314 027 bit/s	2 999 109 bit/s
Synthetic Ping instructions, no delay	534 118 bit/s	644 432 bit/s
Synthetic Ping instructions, 100 µs delay	314 027 bit/s	565 894 bit/s

Table 5.3: The effective and processing data rates for each experiment. The effective data rate is the number of bits that were received divided by the duration of the experiment (60 s). The processing data rate is the rate at which the actual packet processing task could in theory process packets. It is calculated by dividing the number of bits that were received by the time spent processing them (the sum of all *time per buffer* measurements).

6 Discussion

This chapter discusses the results gathered from the experiments that were presented and evaluated in chapter 5.

All of the experiments except for 5.2.5 and 5.2.6 showed a maximum *time between buffers* of below or slightly above 16 ms. This means that in practice, the implementation is performant enough to not lose any data, especially considering that a bus load of 100 % is not possible in reality.
It is clear that there is a constant overhead of about 4 ms in the *time between buffers*. This overhead can be seen as the median *time between buffers* in all experiments. This overhead is not small; compared to the approximate maximum of 16 ms, it is 25 % of the entire time spent.
From both experiments that used synthetic *Ping* instructions as data it is apparent that *Ping* instructions are the absolute worst case when it comes to processing time. This is not surprising considering that *Ping* instructions may require allocating a new `ControlTable` object (see subsection 4.2.6). However, these allocations are only necessary when the model number changes constantly as was the case in these two experiments. This is simply not a realistic scenario; in practice, *Ping* instructions are incredibly rare, usually only sent when the master connects to the bus for the first time.
The number of *Ping* instructions in the other experiments was already higher than to be expected because the data they used contained at least one *Ping* instruction and response for every device. This means that *Ping* instructions were sent every time the data was repeated. This unusually high frequency did not impact performance in any meaningful way, however. All four experiments processed the incoming data in time.
Generally, the amount of load on the bus was noticeable when interacting with the UI. It manifested as periodic increases in latency and low framerate. These became especially severe when the load was too high to be processed in time.
This is a direct consequence of how the UI and the data processing task are configured. Since the UI task is only assigned a fixed amount of time after all incoming data in one half of the buffer has been processed, higher processing times decrease the overall amount of time spent updating the UI (see subsection 4.2.4 for details). While this only seriously affected the UI once incoming data was already getting lost, the user experience was still not acceptable. There is certainly room for improvement, either by optimizing the firmware, redesigning the way the priorities

of the two tasks are handled or by switching to hardware with greater processing power.

7 Conclusion and Future Work

This chapter gives a conclusion on the work done as well as the results presented in section 7.1. Section 7.2 discusses possible future work.

7.1 Conclusion

In this thesis firmware for the STM32F7508-DK board has been developed that allows monitoring and debugging the status of devices connected to a ROBOTIS Dynamixel Protocol 2.0 based bus. The targeted *Wolfgang* robot platform uses *RS-485* but any bus compatible with a UART interface can be used. Like with *RS-485*, a transceiver or converter has to be used to interface the bus with the board itself. It is also possible to directly connect a bus using TTL logic levels (3.3–5 V). This is effectively what was done during testing and evaluation.
The STM32F7508-DK board is well suited for the development of graphical applications. Because the touchscreen display is already part of the board, no further assembly was required. The speed of the STM32F750N8H6 microcontroller is sufficient but any less powerful hardware would most likely have caused serious issues. A significant amount of processing power is needed just to render and update the UI.
The UI has been designed to clearly highlight any disconnected devices. Consistent color coding (red for disconnected, green for connected) is used across all views. The different views allow users to select varying levels of detail. While the UI performs well enough to be usable, there is noticeable lag, especially under high load.
The firmware is performant enough to handle the data rate of 2 MBd used with the *Wolfgang* robot platform. Future improvements would allow for even higher data rates or multiple bus connections (see section 7.2).
The source code of the firmware makes it easy to add support for new device models or update the definition of already supported ones by constraining necessary changes to one well-defined interface. The code must be recompiled and flashed to the board to deploy changes.
Most of the firmware code is either platform-independent or uses the emWin library. Porting the firmware to different (Arm Cortex-M based) hardware should be possible with reasonable effort: both bootloader and hardware initialization, as well as the emWin and FreeRTOS configuration, would have to modified.

7.2 Future Work

There are various ways in which the current implementation can be improved:

- Currently, the device details view shows all fields in the device's *control table*. Only a small number of values for these fields are ever observed. There is no way to differentiate a default value from a value that was actually observed. Default values could be highlighted in a different color or left out altogether.
- It should be possible to accept data from more than one *RS-485* bus at the same time without difficulties. Further investigation would be required on how to physically connect additional buses. The increased data rate most likely also requires performance improvements.
- Performance may be improved by increasing the size of the receive buffers. This may also allow improving UI performance by increasing the yield time of the data processing task.
- Performance may be improved by further optimizing the code and data structures used. This is unlikely to result in large improvements, however.
- Performance could also be improved by using more powerful hardware.
- Some display flickering may be removed by enabling and using V-Sync with the LCD controller.
- UI performance may be improved by using the dedicated 2D hardware accelerator included in the STM32F750N8H6 microcontroller [STM18c]. This is unlikely to have a noticeable effect, as the UI task appears to spend most of the time updating the state of the widgets.
- UI performance may be improved significantly by using a dual-core CPU. A dedicated CPU core could be used for rendering the UI, which would also decrease latency and simplify the code since the data processing task would not have to yield to allow for UI updates.

Bibliography

[bit19] Robots of the Hamburg Bit-Bots. `https://submission.robocuphumanoid.org/uploads/Hamburg_Bit_Bots-specs-5de3cecc85cbe.pdf`, 2019. Accessed: 2020-03-02.

[cat] Why do we need yet another C++ test framework? `https://github.com/catchorg/Catch2/blob/87950d9cfa87eb41ff60b7e5f7e11ad21749a2a1/docs/why-catch.md`. Accessed: 2020-03-11.

[Coo] Greg Cook. Catalogue of parametrised CRC algorithms with 16 bits. `http://reveng.sourceforge.net/crc-catalogue/16.htm`. Accessed: 2020-03-04.

[FAD$^+$19] Rémi Fabre, Boris Albar, Clément Dussieux, Ludwig Joffroy, Zhe Li, Clément Pinet, Jennifer Simeon, and Sébastien Loty. CATIE Robotics @Home 2019 Team Description Paper. Technical report, Centre Aquitain des Technologies de l'Information et Electroniques, 1 Avenue du Dr Albert Schweitzer, 33400 Talence, France, 2019.

[frea] Memory Management. `https://www.freertos.org/a00111.html`. Accessed: 2020-03-11.

[freb] RTOS Fundamentals. `https://www.freertos.org/implementation/a00002.html`. Accessed: 2020-03-06.

[frec] Running the RTOS on a ARM Cortex-M Core. `https://www.freertos.org/RTOS-Cortex-M3-M4.html`. Accessed: 2020-03-11.

[fred] The FreeRTOS^{TM} Kernel. `https://www.freertos.org/RTOS.html`. Accessed: 2020-03-11.

[free] xsemaphorecreatemutex. `https://www.freertos.org/CreateMutex.html`. Accessed: 2020-03-11.

[Fre16] Free Software Foundation. *LD(1) Linux User's Manual*, October 2016. Part of binutils-arm-none-eabi 2.27.

[Fut] Future Technology Devices International. *Future Technology Devices International Ltd. FT232R USB UART IC Datasheet.* Version 2.15.

[iro] irobot: Saug-, Wisch- und Mähroboter. `https://www.irobot.de/`. Accessed: 2020-03-17.

[Max14] Maxim Integrated. *MAX481/MAX483/MAX485/MAX487–MAX491/MAX1487 Low-Power, Slew-Rate-Limited RS-485/RS-422 Transceivers*, September 2014. Revision 10.

[MMW+19] Raphael Memmesheimer, Ivanna Mykhalchyshyna, Niklas Yann Wettengel, Tobias Evers, Lukas Buchhold, Patrik Schmidt, Niko Schmidt, Ida Germann, Mark Mints, Greta Rettler, Christian Korbach, Robin Bartsch, Isabelle Kuhlmann, Thomas Weiland, and Dietrich Paulus. RoboCup 2019 - homer@UniKoblenz (Germany). Technical report, University of Koblenz-Landau, Universitätsstr. 1, 56070 Koblenz, Germany, 2019.

[new] The Red Hat newlib C Library. `https://sourceware.org/newlib/libc.html`. Accessed: 2020-03-11.

[PG18] A. K. Pandey and R. Gelin. A Mass-Produced Sociable Humanoid Robot: Pepper: The First Machine of Its Kind. *IEEE Robotics Automation Magazine*, 25(3):40–48, 2018.

[PMM+19] Bruno F. V. Perez, Douglas R. Meneghetti, Enrico Matiuci, Leonardo C. Neves, Fagner Pimentel, Gabriel S. Melo, João Victor M. Santos, Lucas I. Gazignato, Marina Y. Gonbata, Mateus G. Carvalho, Matheus V. Domingos, Rodrigo C. Techi, Thiago S. B. Meyer, William Y. Yaguiu, Flavio Tonidandel, Reinaldo Bianchi, and Plinio T. Aquino Junior. RoboFEI@Home Team Description Paper for RoboCup@Home 2019. Technical report, FEI University Center, Sao Paulo, Brazil, 2019.

[roba] A Brief History of RoboCup. `https://www.robocup.org/a_brief_history_of_robocup`. Accessed: 2020-03-24.

[robb] Objective. `https://www.robocup.org/objective`. Accessed: 2020-03-24.

[robc] Qualified teams for RoboCup 2019. `https://humanoid.robocup.org/hl-2019/teams/`. Accessed: 2020-03-17.

[robd] RoboCupSoccer. `https://www.robocup.org/domains/1`. Accessed: 2020-03-24.

[ROBe] ROBOTIS. Protocol 2.0. `http://emanual.robotis.com/docs/en/dxl/protocol2/`. Accessed: 2020-03-01.

[RWT$^+$19] Prof. Dr. Matthias Rätsch, Thomas Weber, Sergey Triputen, Marvin Ott, Peter Stengl, Pengfei Huyan, Bo Zhang, He Lin, Steffen Eißler, Michael Litz, Moritz Mähr, Lennart Kraft, Le Ping Peng, Päivi Kärnä, Patrik Huber, and Michael Danner. RT Lions Team Description Paper. Technical report, Reutlingen University, Alteburgstraße 150, 72762 Reutlingen, Germany, 2019.

[SEG17] SEGGER Microcontroller GmbH & Co. KG. *emWin - Graphic Library with Graphical User Interface - User & Reference Guide*, November 2017. UM03001 Revision 1, part of STM32CubeF7 1.15.0.

[STM17] STMicroelectronics. *UM1905 User Manual - Description of STM32F7 HAL and Low-layer drivers*, February 2017. Revision 3.

[STM18a] STMicroelectronics. *DS12535 datasheet - STM32F750x8*, June 2018. Revision 1.

[STM18b] STMicroelectronics. *Release Notes for STemWin Library*, March 2018. Part of STM32CubeF7 1.15.0.

[STM18c] STMicroelectronics. *RM0385 Reference manual - STM32F75xxx and STM32F74xxx advanced Arm®-based 32-bit MCUs*, June 2018. Revision 8.

[STM18d] STMicroelectronics. *UM2470 User manual - Discovery kit for STM32F7 Series with STM32F750N8 MCU*, October 2018. Revision 1.

[STM19a] STMicroelectronics. *UM1734 User manual - STM32Cube^{TM} USB device library*, February 2019. Revision 4.

[STM19b] STMicroelectronics. *UM1891 User manual - Getting started with STM32CubeF7 MCU Package for STM32F7 Series*, January 2019. Revision 9.

[SZC02] Manny Soltero, Jing Zhang, and Chris Cockril. RS-422 and RS-485 Standards Overview and System Configurations. Technical report, Texas Instruments, June 2002. Revised May 2010.

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*, pages 37–38. Pearson, fourth edition, 2014.

[usb00] *Universal Serial Bus Specification Revision 2.0*, April 2000.

[YTO$^+$19] Takashi Yamamoto, Yutaro Takagi, Akiyoshi Ochiai, Kunihiro Iwamoto, Yuta Itozawa, Yoshiaki Asahara, Yasukata Yokochi, and Koichi Ikeda. Human Support Robot as Research Platform of Domestic Mobile Manipulator. Technical report, Toyota Frontier Research Center and Toyota Research Institute, 2019.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

[illegible]

Hamburg, den 29.04.2020 Robin Mirow

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

[illegible]

Hamburg, den 29.04.2020 Robin Mirow