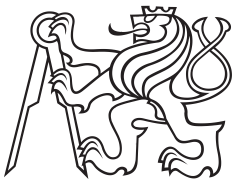
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

CAN Bus Latency Test Automation for Continuous Testing and Evaluation

Matěj Vasilevski

Supervisor: Ing. Pavel Píša, Ph.D.
May 2022

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Vasilevski Matěj** Personal ID number: **474605**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

CAN Bus Latency Test Automation for Continuous Testing and Evaluation

Master’s thesis title in Czech:

Automatizace průběžného testování latencí na sběrnici CAN

Guidelines:

The goal of this thesis is to update system for latency and throughput measurements to support CAN FD devices connected to multiple CAN buses. CTU original CAN FD IP core will be used on the tester side for frames generation and time-stamping. The project is continuation of CTU previous Linux CAN gateway assessment done for Volkswagen Research.
1) Familiarize with previous work on Xilinx Zynq based systems and with CTU CAN FD IP core
2) Update original design based on OpenCores SJA1000 IP cores and integrated XCAN controllers to include additional CTU CAN FD channels
3) Study and update CAN latency tester to support CAN FD and prepare integration and web interface for continuous testing
4) Implement time-stamping SocketCAN extension for XCAN and CTU CAN FD controllers
5) Document designed system and setup to be prepared for continuous evaluation and testing in-house. Offer the project to OSADL to be integrated into their QA real-time farm

Bibliography / sources:

[1] Jeřábek, M.: FPGA Based CAN Bus Channels Mutual Latency Tester and Evaluation; Bahelor's thesis, CTU 2016; https://gitlab.fel.cvut.cz/canbus/zynq/zynq-can-sja1000-top/wikis/uploads/56b4d27d8f81ae390fc98bdce803398f/F3-BP-2016-Jerabek-Martin-Jerabek-thesis-2016.pdf,
[2] Jeřábek, M.: Open-source and Open-hardware CAN FD Protocol Support; Masters's thesis, CTU 2018; https://dspace.cvut.cz/bitstream/handle/10467/80366/F3-DP-2019-Jerabek-Martin-Jerabek-thesis-2019-canfd.pdf
[3] Ille, O.: CTU CAN FD IP Core; CTU, Project site https://gitlab.fel.cvut.cz/canbus/ctucanfd_ip_core

Name and workplace of master’s thesis supervisor:

Ing. Pavel Píša, Ph.D. Department of Control Engineering FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **19.01.2022** Deadline for master's thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Pavel Píša, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to thank my supervisor Ing. Pavel Píša, Ph.D., for his invaluable guidance, hints and expertise. Also thanks to my family and friends for their support.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 20.5.2022

Abstract

This thesis documents the setup of an CAN bus latency testing system. This existing system has been revived, updated to work with CAN FD protocol and better documented for possible replication. Linux driver for CTU CAN FD IP cores has been extended to support timestamp reporting, and will hopefully be accepted into the mainline Linux driver. Existing patch for Xilinx CAN driver to report timestamps has been updated and will be submitted to the mainline too.

Also, latencies of the NuttX CAN driver for ESP32C3 TWAI peripheral, developed by my colleague Jan Charvát in his thesis, have been measured. NuttX proved to be oustandingly fast and capable of handling very high traffic loads.

Keywords: latency, testing, measurement, FPGA, CTU CAN FD, CTU CAN FD IP core, Xilinx, CAN, CAN bus, Linux, U-Boot, NFS, timestamping

Supervisor: Ing. Pavel Píša, Ph.D.

Abstrakt

Tato práce dokumentuje systém na měření latencí na sběrnici CAN. Již existující systém byl oživen, aktualizován aby fungoval i na protokolu CAN FD, a lépe zdokumentován pro možnost nasazení i jinde než zde na fakultě. Linuxový ovladač pro CTU CAN FD IP jádra byl rozšířen o podporu timestampů z IP jádra, a snad bude v blízké době přijat k zamergování do mainlinového Linux jádra. Existující patch Xilinx CAN ovladače pro podporu timestampů byl aktualizován a také bude v blízké době předložen návrh k jeho zamergování.

V této práci byla také měřena latence NuttX CAN ovladače pro ESP32C3 TWAI periferii, který byl vyvinut mým spolužákem Janem Charvátem v rámci jeho diplomové práce. NuttX se osvědčil jakožto mimořádně rychlý systém, schopný zpracovat velmi velký síťový provoz.

Klíčová slova: latence, testování, měření, FPGA, CTU CAN FD, CTU CAN FD IP core, Xilinx, CAN, sběrnice CAN, Linux, U-Boot, NFS, časová razítka

Překlad názvu: Automatizace průběžného testování latencí na sběrnici CAN

Contents

Figures

Tables

Chapter 1

Introduction

This thesis deals with latency testing of CAN devices on a CAN bus. CAN bus is a Controller Area Network bus, used mainly in vehicles to facilitate communication between microcontrollers. And in this work, the main focus will be on the latencies of a gateway device. Gateway device receives CAN frames on one CAN bus, optionally processes them, and transmits them on another CAN bus. It is desirable to know latencies of such gateway, when the gateway is used in a car as a middleman between two subsystems on different CAN buses.

This work aims to revive the old project for continuous latency testing of Linux based CAN gateway[19]. One goal of this thesis is to update the system to support CAN FD.

To measure the latency, timestamp is stored when a CAN frame is received. At first, software timestamps created in the kernel were used, but this approach is not as accurate. Then M. Jerabek designed an expansion board for MicroZed Zynq board [10], which enabled him to use hardware timestamps provided by the Xilinx CAN controllers. However, those controllers support only classical CAN, therefore CTU CAN FD IP cores will be used instead of the Xilinx controllers. The CTU CAN FD IP core has also better support for timestamping. Another goal of this thesis is to add support for timestaping in the Xilinx and CTU CAN FD Linux drivers.

Finally, the system needs to be better documented, to make it possible to replicate and restore the system more easily. This system could also be integrated into the OSADL QA farm[1], which monitors the performance of mainline Linux with `PREEMPT_RT` patch.

As an extra work, this thesis measures the latencies of NuttX CAN driver for ESP32-C3 TWAI peripheral. My colleague Jan Charvát developed this driver in his thesis and needed to verify that the driver works correctly.

[1] `https://www.osadl.org/OSADL-QA-Farm-Real-time.linux-real-time.0.html`

Chapter 2

Project Overview

Projects revolving around timing analysis of CAN drivers date back to 2006, when the LinCAN CAN driver[1] was in development at FEE CTU. There was a project OCERA - Open Components for Embedded Realtime Applications, which created the need for latency testing. The original program is called `canping`[2].

Then in 2011 came a contract for testing a Linux-based CAN gateway, done by the Department of Control Engineering CTU FEE in collaboration with the Volkswagen Group Research [18]. The experiments measured latency of the gateway, that is, the time spent between receiving a CAN message on one bus, processing it, and transmitting the message on a second bus. Such gateway can be used for example in a car, where it separates different networking subsystems. One subsystem can contain Electronic Control Units (ECUs) controlling engine and brakes, the other subsystem consists of e.g. infotainment unit and other non-critical systems. With the gateway, we can restrict the communication between those two subsystems, apply custom rules for network traffic filtering, or do some message processing to make the two CAN buses compatible (change message IDs, perform data manipulation, calculate new CRC checksums).

This latency testing then continued in 2014, comparing the performance of multiple interfaces used to access a CAN bus [19]. Namely, they compared gateway implementations with `read` / `write` system calls, non-blocking `read` / `write`, memory-mapping `PF_PACKET` socket receive/transmission ring buffers (to avoid copying frames from user space to kernel space), and finally using `recvmmsg` / `sendmmsg` interface to receive/send multiple messages on a socket. Measured latencies range from 50 µs in kernel gateway to 200 µs for simple blocking implementations. Some form of busy waiting was required to achieve about 100 µs latency. To see the overhead of the Linux kernel, an RTEMS based gateway has been used as a reference, achieving only 15 µs latency on the same PowerPC MPC5200 based mid range hardware. .

There has been a continuous testing set up to regularly measure latencies of new Linux kernel releases. The test runner would periodically fetch latest Linux versions from git, compile it, reboot the gateway to load the new kernel,

[1] `http://ortcan.sourceforge.net/lincan/`

[2] `https://sourceforge.net/p/ortcan/canping/ci/master/tree/`

and measure and collect new data. Results from this testing are published on website[3].

2.1 Measurement Setup

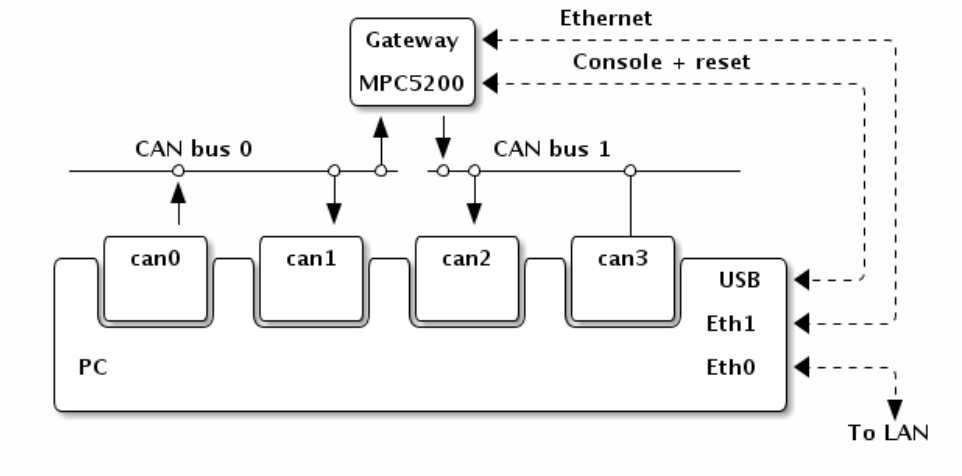


Figure 2.1: Original testbed setup [19].

Figure 2.1 shows the original measurement setup. The PC runs the aforementioned continuous testing. There were also other tests, measuring latency profiles, comparing syscalls performance, showing latency profiles depending on the frame rate (rate at which frames are sent from the PC). But here we will deal only with continous testing collecting average latency.

Figure 2.2 shows the definition of packet latency. It is also possible to calculate the CAN frame exact length in bits (including bit stuffing etc.) and use it to obtain the time spent transmitting the message. If we subtract it from the measured time between the two frames, we can get the gateway latency.

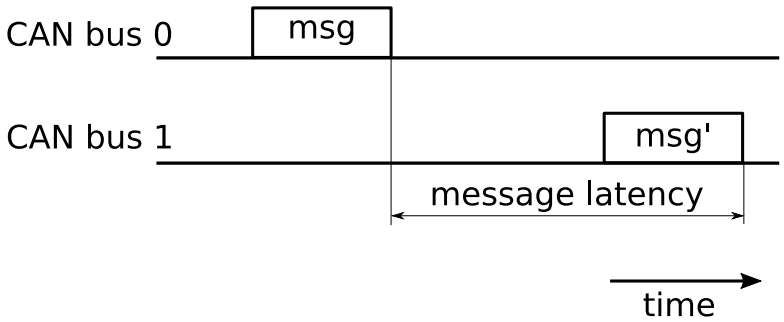


Figure 2.2: Definition of packet latency, image source is [18].

[3] `https://rtime.felk.cvut.cz/can/perf/`

Martin Jerabek then upgraded the original setup in his bachelors thesis [10]. He designed a CAN-BENCH carrier board for the Xilinx MicroZed board based on Zynq 7000 SoC [1]. This enabled him to use hardware timestamps from the CAN controllers, and thus achieve higher precision of measured latencies.

2.1.1 MicroZed APO Board

Because only two copies of the CAN-BENCH carrier board were manufactured and both are unavailable (one board died, the other isn't available), the MicroZed APO (MZAPO) boards[4] are used as a flexible alternative. The MPC5200 gateway is also quite obsolete, so it was replaced by MZAPO board too. The current setup is depicted on figure 2.4. One MZAPO board sends the CAN messages and measures the latencies, the other board serves as a gateway. The PC on the right only coordinates the measurement.

MicroZed APO (MZAPO) boards are MicroZed boards based on Xilinx Zynq 7000 SoC [1], with a custom-designed expansion board and additional hardware. The additional hardware includes LCD display, 3 rotary inputs with button functionality, PMOD connectors for connecting (slow) peripherals to FPGA. In this project we will use only the D-sub 9-pin connector, which exposes two CAN bus channels to the FPGA, and a serial console output. The expansion board is open source, the design of the board and the acrylic casing is available at[5].

2.2 Latency Testing

This section describes the latency testing procedure, see figure 2.4.

The gateway board has one simple purpose - receive messages from CAN bus 0 and instantly send them to CAN bus 1. In Linux, multiple gateway implementations are available - in kernel space and in userspace. Additionally, the userspace gateway can use multiple different programming interfaces: simple `read()` / `write()` system calls, using `PF_SOCKET` sockets and memory mapping the ring buffer memory etc. All those options are described in detail in [19]. The RTEMS based gateway will not be used here, because RTEMS doesn't have a driver for the CTU CAN FD IP core (otherwise Xilinx Zynq 7000 is well supported).

The sender board runs program called `latester`, which keeps sending CAN messages from the interface can0. The messages can be sent in three different modes: periodically, one at a time (wait until the previous message is received from gateway), and flood mode (as fast as possible). Here the one at a time mode is used. The length of outgoing messages, together with the total message count, can be configured from the command line. After a message is sent from can0, it is also immediately received and timestamped

[4] `https://cw.fel.cvut.cz/b212/_media/courses/b35apo/en/semestral/mz_apo-datasheet-en.pdf`

[5] `https://gitlab.com/pikron/projects/mz_apo/microzed_apo`

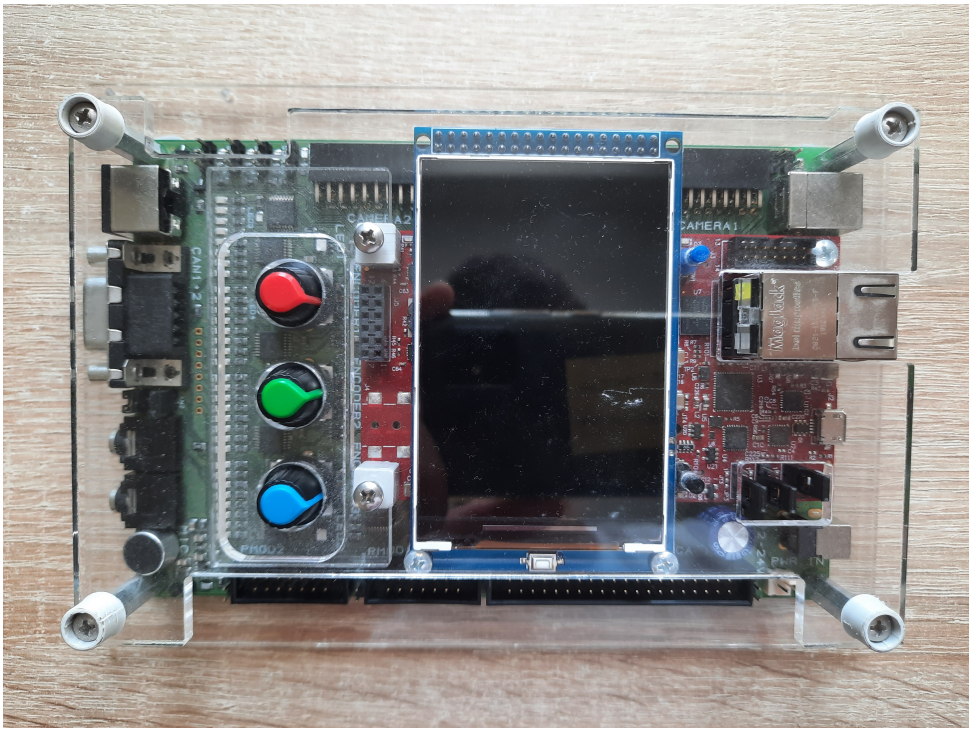


Figure 2.3: MZAPO board.

on can1 interface. The gateway then processes the message and sends it to CAN bus 1, where the message is received and timestamped on interface can2. The two timestamps are then subtracted to calculate the delay. And because the timestamps from can1 and can2 devices are not synchronized, the constant delay (measured at latester startup) also has to be subtracted.

It should be also noted that the CAN controllers are connected using can-crossbar [11]. The crossbar configuration used here is `0x03040150`, see the [11] for more details about the crossbar configuration register bits. This config value means "connect interfaces can0, can4, can5 to Line 1, interfaces can1, can2, can3 to Line 2, assing Lines 1 and 2 to external outputs 1 and 2 respectively, and enable them". The interface's names "can1, can3" used here are the reported names in the running Linux system. Interfaces can0, can1 are Xilinx CAN controllers, can2 to can5 are CTU CAN FD IP cores. However, for the crossbar configuration, the actual hardware description (FPGA design) has to be taken into account. The CAN controllers CAN4 to CAN1 (in the crossbar documentation) are CTU CAN FD IP cores, and controllers CAN5, CAN6 are Xilinx CAN controllers.

The coordinating PC boots the gateway into a Linux kernel of selected version and starts the gateway software itself. Then it starts the latency testing on the sender board, collects the logs from the measurement and processes them. Finally it publishes the results on a website.

The sender board can run regular Linux (for this setup with two CAN buses it seems ok), but RT patched Linux would be preferable. For measurements on

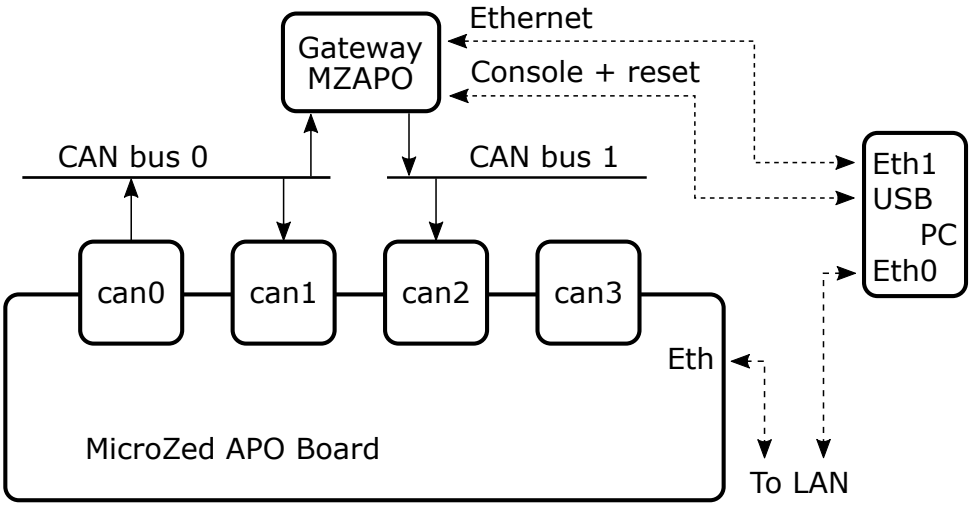


Figure 2.4: Testbed setup, [10] used as a template.

one CAN bus, it is neccessary to run RT Linux, see chapter 6 for more details about the system performance. And for the MZAPO boards specifically, the cpu frequency scaling governor has to be set to `performance` (instead of the default `ondemand`), to force maximal CPU frequency and avoid performance issues (again, critical for one CAN bus measurement).

2.2.1 Latester Documentation

Latester is a part of the `can-benchmark` project[6] (new repository, previously the project has been hosted at[7]). This project is part of the larger CTU FEE CAN bus related projects porfolio listed at[8]. The `can-benchmark` repository contains:

- the latester itself,
- `ugw`: userspace gateway, sends CAN frames from one interface to another, using various (configurable) Linux networking interfaces,
- numerous shell scrips for latency testing (with cpu load, ethflood, etc.),
- `continous` folder with setup scripts and tests for everyday automatic testing,
- and much more software, however it is not relevant for this actual work.

The latester runs in two threads: main thread and measure thread, and they communicate with each other using pipe. Measurement messagess are internally represented using struct `msg_info`, which contains:

- the CAN identifier (`canid_t`),

[6] `https://gitlab.fel.cvut.cz/canbus/can-benchmark/`
[7] `http://rtime.felk.cvut.cz/gitweb/can-benchmark.git`
[8] `https://canbus.pages.fel.cvut.cz/`

- message length,
- timestaps:
  - when the CAN frame was sent: `ts_sent` userspace timestamp, `ts_sent_kern` kernel timestamp - not used,
  - when the CAN frame was received on the first interface ("onwire" a.k.a. `can1` in figure 2.4): `ts_rx_onwire` is a userspace timestamp (obtained by calling `clock_gettime()` when the CAN frame is received), `ts_rx_onwire_kern` which is the hardware timestamp from the CAN controller,
  - when the CAN frame was received on the second interface ("final" a.k.a. `can2` on the second bus): analogous timestamps `ts_rx_final`, `ts_rx_final_kern`.

The measure thread is spawned from the main thread, and it starts with creating and binding the CAN sockets. Then is determined the timestamp difference between the two receiving CAN interfaces to synchronize them, since both Xilinx CAN driver and CTU CAN FD driver report timestamps with slightly shifted time bases for each CAN device (see chapter 5 for more details about both drivers). This synchronization is done by connecting all of the CAN interfaces in CAN Crossbar, sending a CAN message with payload `SYNC` from the transmitting interface (`can0`), and receiving the message on both receiving interfaces (`can1`, `can2`). Because the interfaces are connected in the programmable logic, we know that the delay is exactly 0, and by subtracting the hardware timestamps we can get the constant difference ($diff = t_{can2} - t_{can1}$).

The measure thread then requests `SCHED_FIFO` scheduling policy (real-time scheduling policy - first in, first out; process runs until blocked by I/O or preemted by higher priority process), and higher priority (40). Then it finally gets to the main loop: polling on the CAN interfaces for data to read and reading it, and sending CAN frames. When a CAN message is read, its timestamp is stored into the corresponding `msg_info` structure. If the message was received on the second bus, the `msg_info` structure containing all the timestamps and metadata is sent to the main thread through a pipe.

In the main thread is processing of the `msg_info` structures received through the pipe. The difference between timestamps from both receiving interfaces is calculated, as well as the transmission time of the CAN frame (to get exact gateway latency), histograms are updated and information about the message is printed to a file. After the measurement is done, the main thread prints all the statistics, histograms, and exits.

Latester produces the following output writen into `.txt` files (assuming "meas" is the measurement name specified on the command line, see paragraph below for more details):

- raw gateway latencies (time difference between CAN frames on `can1` and `can2`) in the form of cumulative histogram with inverted vertical axis, time in milliseconds - file `meas-hist-raw.txt`

- gateway latencies reduced by the amount it takes to send the CAN frame (latester can calculate the exact frame length in bits -> calculate the time it takes to transmit the frame) - file `meas-hist.txt`
- statistics from the measurement: total duration, number of sent and lost frames, average latency, and latency percentiles by 5 percents - file `meas-stat.txt`
- complete log of all testing messages: for every message the following is logged: its number, frame info (id and data), all the message timestamps (userspace/hardware timestamps for all two/three CAN interfaces) - file `meas-msgs.txt`

The following command line options are available when running latester:

- `-d / --device` - specify CAN interface to use. Must be given two or three times - transmission interface, receiving interface #1, optionally receiving interface #2
- `-c / --count` - number of testing messages
- `-i / --id` - id of the testing frames, default is 10
- `-p / --period` - period for sending messages (in microseconds), or 0 to send as fast as possible
- `-t / --timeout` - socket polling timeout when period is set to 0
- `-o / --oneattime` - sends next message only after the previous message was finally received
- `-n / --name` - string with the measurement name, used as a prefix for the output file names
- `-l / --length` - length of the generated CAN frames in bytes, default 2
- `-u / --userhist` - generate histogram from userspace timestamps
- `-q / --quiet` - do not print progress and statistics
- `-f / --fd` - send CAN FD frames
- `-b / --bitrate` - specify CAN bus bitrate, default 1000000
- `--dbitrate` - specify data bitrate when using CAN FD, default 2000000

As you can see from the first `--device` option, latester can run with either 2 or 3 CAN interfaces. When you use 3 interfaces, the measurement is done as described in section 2.2. When you use only two interfaces, latester uses the userspace timestamp from the sent frame to compute latency. Hardware timestamp for the sent CAN frame is not supported (neither in hardware nor in the latester itself). Example latester usage is in listing 2.1.

```
./latester -d can2 -d can3 -d can4 -n meas -c 3200 -l 8 -b
    ↪ 500000 -o
```

Listing 2.1: Example latester usage - using 3 CAN interfaces, sending 3200 messages of length 8 B, bus bitrate is 500000, and messages are sent one at a time.

To compile latester, simply run `make` in the `latester/` subdirectory in the can-latester repository. You will need two dependencies - `popt` and `talloc`. On Ubuntu, those can be installed from the packages `libpopt-dev` and `libtalloc-dev` (or `libpopt:armhf` when crosscompiling). If you are targeting different architecture (e.g. crosscompiling for ARM - MZAPO with Xilinx Zynq), use
`export CC=arm-linux-gnueabihf-gcc` before running the Makefile.

2.2.2 Future Work on Latester

In the future, latester should be rewritten to not pass the `struct msg_info` messages through the pipe to the main thread. Instead, a large enough circular buffer should be used and the main thread should be notified using a semaphor or conditional variable to read the buffer.

Also, the `can-benchmark` repository will be split into multiple repositories (`can-latester` etc.) and those will be included as submodules.

Chapter 3

Running Linux on MZAPO board

This section describes the setup used to run Linux on the MZAPO boards. U-Boot bootloader is installed on the board, where it loads the boot instructions from TFTP server, and boots a Linux kernel with NFS mounted root partition.

3.1 Das U-Boot

U-Boot is a bootloader used mainly in embedded applications to boot the operating system (oftentimes Linux) [4]. Bootlader's other tasks include initializing the CPU, memory controllers and other hardware. U-Boot's operating system booting routine is composed of multiple steps. High level overview of the routine is: it loads all the required components (kernel, device tree, initial ramdisk, FPGA bitstream) at explicit memory addresses, sets up kernel boot arguments, and finally boots the kernel.

On MZAPO boards, the U-Boot is installed on SD card inserted into the MicroZed Zynq. The Boot ROM contains a first-stage bootloader, which is configured using jumpers on the board to boot from SD card (or from internal Flash memory). This first-stage bootloader then loads another piece of code from the SD card from predefined memory location. This piece of code then loads the U-Boot SPL (secondary program loader) in a `boot.bin` file, which in turn loads the full U-Boot bootloader (second-stage bootloader) from `u-boot.img`. In the SD card filesystem is also an `uEnv.txt` configuration file, which contains U-Boot environment variables.

Variables in the `uEnv.txt` are used to configure the boot proces. Listing 3.1 shows the contents of this file. Variable `set_ethaddr` contains command to configure MAC address using the variable `ethaddr` (which is automatically picked up by U-Boot) [5]. Setting `autoload=no` then disables automatic image loading when running `dhcp` (and other) commands. The `dhcp` will then only perform DHCP configuration without automatic image loading (from whatever the configured image source is). `tftpserverip` is a helper variable containing IP address of the TFTP server, in this case it is the IP address of my desktop (running the TFTP server). This is followed by another two variables `bootscript_addr` and `bootscript_path`, which contain the memory location (0x01000000) where the bootscript image will be loaded and the bootscript image file address on the TFTP server

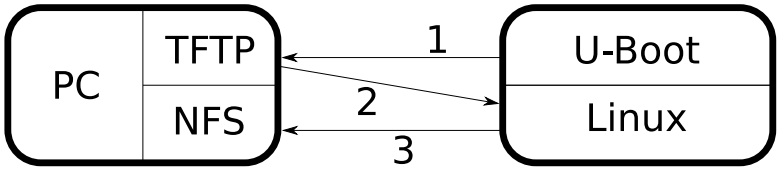


Figure 3.1: Diagram of the booting process. First the U-Boot loads from the TFTP server a single uImage with everything needed to boot, this uImage is extracted and booted, and the booted Linux kernel mounts its root partition from the NFS server.

(`/tftp/zynq/autoscr.scr`).

Then there is a collection of variables and `use_uart0` to configure serial line to be on the USB Type B connector on expansion board in MZAPO (which is more robust and student-proof connector than microUSB). Variable `sboot` then configures the actual boot procedure used in our setup. U-boot first obtains IP address with the `dhcp` command, then sets `serverip` variable (needed for `tftp` command which isn't used here, so maybe it could be deleted). Then the `tftpboot` command is executed with parameters `bootscript_addr` and `tftpserverip:bootscript_path`, to load the boot script into memory at specified address. And finally the `source` command runs the boot script from given memory address.

At the end is `default_bootcmd` variable with the default boot command, which however isn't used, because we also set the `uenvcmd`. This `uenvcmd` is picked up by U-Boot, and in the variable we tell the U-Boot to `run` the commands in variables `set_ethaddr` and `sboot`, so U-Boot sets the MAC address and then runs the `sboot` series of commands described above.

```
set_ethaddr=setenv ethaddr 00:0a:35:00:22:02
autoload=no
tftpserverip=192.168.0.110

bootscript_addr=0x01000000
bootscript_path=/tftp/zynq/autoscr.scr

SLCR_UNLOCK=0xF8000008
SLCR_LOCK=0xF8000004
APER_CLK_CTRL=0xF800012C
UART_CLK_CTRL=0xF8000154
MIO_PIN_10=0xF8000728
MIO_PIN_11=0xF800072C

use_uart0=mw.l ${SLCR_UNLOCK} 0xDF0D ; mw.l ${APER_CLK_CTRL}
    ↪ 0x01ff044d ; mw.l ${UART_CLK_CTRL} 0x00001403 ; mw.l ${
    ↪ MIO_PIN_10} 0x16E1 ; mw.l ${MIO_PIN_11} 0x16E0 ; mw.l $
    ↪ {SLCR_LOCK} 0x767B ; setenv stdout ttyPS0 ; setenv
    ↪ stderr ttyPS0 ; setenv stdin ttyPS0
```

```
sboot=dhcp && setenv serverip ${tftpserverip} && tftpboot ${
    ↪ bootscript_addr} ${tftpserverip}:${bootscript_path} &&
    ↪ source ${bootscript_addr}

default_bootcmd=run use_uart0 set_ethaddr

uenvcmd=run set_ethaddr sboot
```

Listing 3.1: uEnv.txt file used in MZAPO board.

The autoscr bootscript on TFTP server is a Hush shell (Bourne-like shell used in U-Boot) script. The script packed into U-boot image format (on the host PC running TFTP server) from text file into script using `mkimage` command

```
mkimage -A arm -O linux -T script -C none -a 0 -e 0   -n "
    ↪ autoscr example script"   -d autoscr.txt autoscr.scr
```

The `mkimage` tool is from U-Boot tools (package `u-boot-tools` on Ubuntu). Arguments have the following interpretation:

- `-A` sets the target architecture to ARM,
- `-O` sets the target operating system we will be booting,
- `-T` sets the compilation image target to `script` (it can also be a `kernel` for kernel image, `fpga` for FPGA image, `flat_dt` for flat device tree, `firmware` for firmware etc.),
- `-C` sets the compression type,
- `-a` sets the load address (in a hexadecimal number),
- `-e` sets the entry point (in a hexadecimal number),
- `-n` sets the image name,
- `-d` specifies the file with image data (here we are compiling shell script in text file to a "script" type image)

Contents of such autoscr script used in this setup are in listing 3.2. Command `echo` just prints some text, same as in regular shells. The `nfsserverip` then sets the NFS server IP address (same as the TFTP server, since both are running on the same desktop), and `nfspath` sets the path to the directory used as a root filesystem. Variable `image_img` sets the path to the uImage (a multi file image in the form of flattened device tree) with kernel and everything that is booted packed into a single file. Then there is a `test` whether `image_override` is defined, and if it is, the image path is overriden. Variable `image_tftp` then defines a series of commands that load the boot image from the TFTP server into memory at address `netstart`. Then `bitstream_unpack` defines another series of commands to prepare the

FPGA bitstream and load it into memory. And `boot_now` variable uses `bootm` command to boot image from memory.

Finally the kernel arguments are set in the `bootargs` variable. Here is the NFS root definition in `nfsroot=IP:path` format. Also the `root=/dev/nfs` to tell the kernel that root partition is mounted over NFS. Documentation about the NFS root setup for Linux is here [13]. And at last is executed our boot procedure: load the image from TFTP server, unpack bitstream and load it, and boot the loaded image from memory.

```
echo Running autoscr from ftp

setenv tftp_path /tftp/zynq

setenv nfsserverip ${tftpserverip}
setenv nfspath /srv/nfs/debian-armhf

setenv netstart 0x01000000

setenv image_img ${tftp_path}/image.ub
test -n "$image_override" && setenv image_img "
    ↪ $image_override"

setenv image_tftp 'echo === Loading boot image; tftpboot ${
    ↪ netstart} ${tftpserverip}:${image_img}; fdt addr ${
    ↪ netstart}'
setenv bitstream_unpack 'fdt get size filesize /images/fpga@1
    ↪  data; imxtract ${netstart} fpga@1 ${
    ↪ bitstream_load_address}'
setenv boot_now 'bootm ${netstart}'

setenv bitstream_load_address 0x04000000
setenv bitstream_load 'fpga loadb 0 ${bitstream_load_address}
    ↪  ${filesize}'

setenv bootargs ${bootargs} console=ttyPS0,115200
setenv bootargs ${bootargs} clocksource=ttc_clocksource
setenv bootargs ${bootargs} ip=${ipaddr} root=/dev/nfs ro
    ↪ nfsroot=${nfsserverip}:${nfspath}
setenv bootargs ${bootargs} mzapo_lcdip=yes

run image_tftp bitstream_unpack bitstream_load boot_now
```

Listing 3.2: Autoscr script on TFTP server.

3.1.1 FIT File

The U-Boot uImage loaded from TFTP server is created from U-Boot FIT (Flattened uImage Tree) file, shown in appendix A (it is too long to fit in here). It's structure is similar to Device Tree. There is a node for every image needed to boot - the Linux kernel, flattened device tree blob, FPGA bitstream, ramdisk. Each node has a lot of properties (data, type, compression, architecture and os, what kind of hashes should be calculated, etc.), but here we can focus only on the data property. This property is path to the data file, which will be used to build the single uImage. After putting together all the necessary files (compressed kernel, FPGA bitstream, initramfs), the U-Boot FIT image is compiled using

```
mkimage -f uboot-image.its image.ub
```

Although this setup is very convoluted and complex, it allows for extreme flexibility. For example, the same board can be used during Real Time Systems course seminars, to boot VxWorks system by simply changing the `bootscript_path` variable in `uEnv.txt` on the SD card.

3.2 Linux Kernel

The kernel used is here is versioned in Zynq RT utils and builds repository[1]. It contains linux kernel config suitable for the MZAPO board, and a Makefile that does all the things neccessary to build the kernel (clones kernel git, builds the kernel, installs modules). Simply calling `make` should suffice to build the kernel. Optionally, call `make KERNEL_GIT_REF=$YOUR_PATH_TO_KERNEL_GIT` The repository also contains three patches to device tree scripts (setting UART0 as a console output, device tree for system with 4x SJA1000 in programmable logic). However, those patches are already applied if you use the default kernel[2] from the makefile.

Output of this build is the `zImage` file, which is a compressed version of the Linux kernel image. This is used in the U-Boot FIT image file. Also, modules from the build in `/lib/modules/` path are copied to the NFS root (section 3.4.1).

3.2.1 Loading Device Tree Blob Overlays at Runtime

Because the Linux kernel doesn't offer an interface to load Device Tree Overlays at runtime, an out-of-tree external module has to be used. Here, the `dtbocfg` [12] module is used. It enables user to load/unload DT overlays from userspace.

To compile dtbocgf, run the commands listed in listing 3.3. The variable `KERNEL_SRC_DIR` is generally a path to the build directory of your chosen Linux kernel (or directly the kernel source directory, in case you don't use

[1] `https://github.com/ppisa/zynq-rt-utils-and-builds`
[2] `https://github.com/ppisa/linux-kernel/tree/linux-5.10.y-rt-pi`

out-of-tree builds), and `ARCH` sets the target architecture to arm. After this, copy the kernel modules again to the NFS root.

```
make KERNEL_SRC_DIR=${ZYNQ-RT-UTILS-REPO}/projects/linux/
    ↪ build/arm/zynq/ ARCH=arm
make KERNEL_SRC_DIR=${ZYNQ-RT-UTILS-REPO}/projects/linux/
    ↪ build/arm/zynq/ INSTALL_MOD_PATH=${ZYNQ-RT-UTILS-REPO}/
    ↪ projects/linux/build/arm/zynq-modules/ ARCH=arm
    ↪ modules_install
```

Listing 3.3: Commands to compile dtbocfg.

The Device Tree overlay used in this project are in repository[3], branch `mz_apo-2x-xcan-4x-ctu`. The overlay can be compiled using `mkdtb` script in `scripts/` directory. If you need to (maybe when compiling for a new kernel version/vastly different than the host machine), you can add new include search path using `-i` argument to the `dtc` compiler in `mkdtb` script, pointing to the new kernel's device tree scripts directory.

The same repository contains the `upbit` script to upload the bitstream into the FPGA and load Device Tree overlay.

3.3 TFTP Server

Another ingredient in the mix is the TFTP server to serve U-Boot FIT images. On Ubuntu, a package `tftpd` has been used to install a TFTP server. To configure the TFTP server, edit the `/etc/xinetd.d/tftp` configuration file - set the `server_args` variable to point to your desired TFTP served directory.

```
server_args = /path/to/your/tftp/directory
```

Also, restart the TFTP server to pick up the changes.

```
systemctl reload xinetd.service
systemctl restart xinetd.service
```

3.4 NFS Root and Debian System Setup

3.4.1 NFS Root

The board is booted into an NFS (Network FileSystem) mounted root filesystem. This allows us to prolong the SD card lifespan by not even mounting it at runtime (it is used only for booting). We also avoid memory corruption in case of kernel crashes (can happen since we will be developing/debugging kernel drivers). Also this setup is overall more flexible (has "unlimited" space unlike the SD card), and allows for e.g. upgrading the Linux system without

[3] `https://gitlab.fel.cvut.cz/canbus/zynq/zynq-can-sja1000-top`

having to physically remove the SD card, plug it into the host PC and do the upgrade.

On Ubuntu, a NFS server can be installed using the package `nfs-kernel-server`. This server is configured using the `etc/exports` file. Add the following line to the config file:

```
/srv/nfs/debian-armhf    192.168.0.0/24(rw,sync,
    ↪ no_subtree_check,no_root_squash,insecure)
```

This exposes the `/srv/nfs/debian-armhf` path to all IP addresses on "192.168.0.0/24" subnet with the following options:

- `rw` - allow both read and write requests to the NFS server
- `sync` - NFS server responds to requests only after the requested changes have been committed - prevents data loss/corruption
- `no_subtree_check` - disables security checking whether the accessed subdirectory is in the exported tree
- `no_root_squash` - gives the root user on client authority to access files on the NFS server as a root too. Manual says this is good for diskless systems, which is our case.
- `insecure` - allow requests from all ports (not only <1024)

And finally export the directory with

```
exportfs -r
```

3.4.2 Debian Setup

Finally comes the Debian system setup for the MZAPO board. We will use Debian because I already have experience with managing Debian/Debian-based distributions, and I can ask my supervisor for help (because he also has experience with Debian). It is of course possible to use other distributions/solutions (maybe even Yocto/Buildroot), but Debian is very convenient. The following packages will be needed:

- `qemu-user-static` - static version of QEMU user emulation binaries
- `qemu-utils` - QEMU utilities
- `debootstrap` - tool for bootstrapping a Debian system
- `debian-archive-keyring` - GnuPG archive keys of the Debian archive

The debootstrap basically downloads all the necessary .deb packages, unpacks them into the target directory, chroots to the target directory, and runs the installation and configuration scripts from each package [3]. Because we are targeting arm architecture, we will need QEMU for the last step to execute the arm binaries. This can be also called CrossDebootstrap.

Execute the debootstrap command with the following arguments:

```
/usr/sbin/debootstrap \
    --no-merged-usr \
    --keyring=/usr/share/keyrings/debian-archive-keyring.gpg
    ↪ \
    --arch=armhf \
    --include=debian-keyring,mc,libc6-dev,libstdc++6,busybox,
    ↪ aptitude,etckeeper \
    buster /srv/nfs/debian-armhf http://ftp.cz.debian.org/
    ↪ debian/
```

This creates a Debian system (`buster` release) in the `/srv/nfs/debian-armhf` directory, for target architecture armhf (hf - hard float - for processors with hardware floating point support). Additionally the packages `debian-keyring` , `mc` , `libc6-dev` ... will be installed on the target system.

After the installation finishes, one more configuration has to be done. Some packages may contain system services, which are started post installation under normal circumstances. Such services are also restarted on package upgrades. But if we are chrooted inside the newly created system and install such a package, it will try to start the service, which is undesirable. Solution to this is to create a `/srv/nfs/debian-armhf/usr/sbin/policy-rc.d` script inside the newly created system, listing 3.4.2 shows the script used in this installation. It checks the hostname if it is equal to the chroot host computer hostname, and returns code 101 in such case. Error code means "action is denied", so when you install something from chroot on the host computer, it won't try to manage the services. If you install something from the system running on the MZAPO board, the services will be managed as usual. Also change the hostname in `/etc/hostname` in the new system, otherwise the script won't work.

Finally, don't forget to copy the kernel modules, as mentioned in section 3.2.

```
#!/bin/sh
[ "$(hostname)" = "$USE_HERE_THE_HOSTNAME_OF_THE_CHROOT_HOST"
    ↪  ] && exit 101
exit 0
```

Chapter 4

Upgrading to CAN FD

There are two things to upgrade for CAN FD capability: hardware and software. Hardware part is easy, CTU CAN FD IP cores[1] will be used instead of OpenCores SJA1000 IP cores and Xilinx CAN IP cores. Software wise, the latency tester and userspace gateway have to be updated to work with CAN FD.

4.1 Updating the Software

Software upgrade to CAN FD has been done according to the SocketCAN Linux kernel documentation [6]. That is, using the struct `canfd_frame` everywhere instead of struct `can_frame`, and checking how many bytes were received (16 or 72 - `CAN_MTU` vs `CANFD_MTU`, since the structures share the same structure, except the CAN FD structure can hold up to 64 bytes of data.)

Both latester and userspace gateway (ugw) have been upgraded to support CAN FD. Results of my work are in the repository[2]. Also, various small bugs fixes have been done in the latester.

4.2 Updating the FPGA Design

Previously, the design used in the MZ_APO board included 2 Xilinx CAN controllers, 2 OpenCores SJA1000 IP cores (modified to be CAN FD tolerant), and two CTU CAN FD IP cores. Now the 2 SJA1000 cores have been replaced with another 2 CTU CAN FD cores, to enable latency testing with CAN FD protocol.

The update has been done in Xilix Vivado 2018.2 using graphical interface. You can download the Xilinx Vivado from Xilinx, but you will have to obtain the board file for Zynq 7000 SoC somewhere. The blocks representing SJA1000 cores have been replaced with additional CTU CAN FD cores, and details in git-tracked files have been polished in a text editor. The updated FPGA

[1] `https://gitlab.fel.cvut.cz/canbus/ctucanfd_ip_core/`

[2] `https://gitlab.fel.cvut.cz/canbus/can-benchmark`

design is in repository[3], branch `mz_apo-2x-xcan-4x-ctu`.

4.3 Exact Frame Length Calculation

The project has an algorithm to compute the exact CAN frame length in bits, to calculate the transmission time for the frame. This time is then subtracted from the "end-to-end" latency to obtain the latency in the gateway. However, exact frame length algorithm for CAN FD frames has not been developed (due to time constraints), so for CAN FD measurements, the end-to-end latency has to be used. This does not hamper the functionality of latester, the latency profile is still usable, it is just offseted by the time it takes to send the CAN FD frame. This frame transmission time is of course variable (bit stuffing etc.), but the tail latencies are visible. An exact CAN FD frame length calculation algorithm should be developed in the future.

[3] `https://gitlab.fel.cvut.cz/canbus/zynq/zynq-can-sja1000-top`

Chapter 5

Timestamping in Linux Drivers

Hardware timestamps are time marks stored inside the CAN controller whenever a message is sent or received (depends on what the controller supports). The controller typically has an internal counter, which is sampled when an event happens (send/receive). This counter value (a.k.a. the timestamp) is then stored in some buffer along the frame data.

In order to achieve better precision of measured latencies, it is necessary to use hardware timestamps provided by the CAN bus controller. The alternative - using software timestamps from the kernel, stored when the frame gets to the operating system networking stack - is worse for multiple reasons. And software timestamping in the userspace is consequently even worse. The hardware timestamp is more precise, because it is stored exactly when the CAN frame is sent/received on the bus. Software timestamp stored in the kernel might or might not be close to real value, due to the processing as the frame bubbles up the kernel software stack. In fact, it can happen that even though a CAN message A was received before another CAN message B, the software timestamp t_A is greater (later in time) than timestamp t_B. Because the interrupt handlers (defined in a device driver) are preemtible, and the software timestamps are generated only after the packet is handed from the device driver to the kernel stack [17], two frames received immediately one by one can have reverse timestamps (the second interrupt preemts the first, finishes, and then the kernel might continue the first interrupt/continue with the second message and generate a timestamp).

5.1 Timestamps from Xilinx CAN

In the Xilinx CAN controller included in the MicroZed Zynq board [1], the timestamping counter is only 16 bits wide. This means that the counter will rollover very often. Quick calculation: assuming the CAN controller runs on 100 MHz clock, the counter will roll over every

$$\frac{2^{16}}{100 \cdot 10^6} = 0.6 \text{ ms},$$

which is very often.

To further complicate matters, the timestamping counter is not readable. If it was readable, one could set up a kernel worker to periodically read the counter and keep track of its state and rollovers. However, spawning a task with period $T < 0.6$ ms doesn't sound like a great idea either, and might not be possible on some constrained systems.

To convert the 16-bit timestamp to real time value and make use of the hardware timestamping, Martin Jerabek came up with an conversion algorithm is his bachelor thesis [10]. The algorithm is show in listing 5.1.

```
def convert(unsigned int timestamp, reference_time_point ref):
    ↪
    // ktime is abbreviation for kernel time
    ktime_now = ktime_get_real_ns()
    if (this CAN frame is the first received):
        set the reference point to now
        ref->ktime = ktime_now
        ref->tstamp = timestamp
        return ktime_now
    else:
        // calculate the number of rollovers since the
    ↪ reference
        rollovers = (ktime_now - ref->ktime) /
    ↪ counter_rollover_time_ns
        // C is a constant to convert from counter time to
    ↪ nanoseconds
        // C = NSEC_PER_SEC / counter_frequency
        timestamp_delta_ns = (timestamp - ref->ts) * C
        return ref->ktime
            + rollovers * counter_time_rollover_ns
            + timestamp_delta_ns
```

Listing 5.1: First iteration of M. Jerabek's algorithm for converting Xilinx CAN 16-bit timestamps to real time [10].

The algorithm is invoked from the function handling the received CAN frame (which itself is invoked from the Interrupt Service Routine (ISR)), and works as follows. If the received CAN frame is the first frame received, the reference time point is set to current kernel real time, together with the timestamping counter value associated with it, and the current kernel time is returned. If the received frame isn't the first: the number of timestamping counter rollovers since the reference time point is calculated. Then is calculated the difference between reference and current timestamp, and converted to nanoseconds. Finally we can return the reference kernel time, plus the number of rollovers times the time one rollover takes, plus the timestamps difference in seconds. Listing 5.2 shows simplified ISR handling function from the Linux Xilinx driver for context, source taken from[1].

[1] https://elixir.bootlin.com/linux/v5.18-rc4/source/drivers/net/can/xilinx_can.c#L741

```
static int xcan_rx(struct net_device *ndev, int frame_base) {
    allocate socket buffer
    read from the RX FIFO
        data length code (DLC) register
        extract timestamp from the DLC register
        get pointer to the buffer timestamp storing struct
        convert timestamp and store it
    some processing to match SocketCAN format
    pass the socket buffer up in the kernel networking stack
}
```

Listing 5.2: Simplified function handling received frames called from the ISR, with timestamp extraction from DLC register.

The reference time point can't be dynamically updated. That would require some synchronization primitive (spinlock/mutex) to avoid bugs when tow interrupts want to update the reference (remember, interrupts can be preemted), and such synchronization is not worth it (introducing synchronization and thus slowing down processing only to guarantee correct timestamps). Even the current implementation deserves some synchronization when setting the reference, but again, it is rather not worth it. We can live with the first/first few timestamp being off, or the reference kernel time point being inaccurate by a small amount (sub-millisecond error).

5.1.1 Final Timestamping Version

However, the algorithm listed in 5.1 "suffered slight problems at hardware counter overflows", as the author reports in [10]. Therefore a new version has been developed.

```
ktime_t get_frame_timestamp(u16 frame_cantime, ktime_t
    ↪ frame_ktime) {
    if first frame
        ref_ktime = frame_ktime
        ref_cantime = frame_cantime
        exact_frame_ktime = frame_ktime;
    else
        frame_cantime_full = ktime_to_cantime(frame_ktime) -
    ↪ ref_cantime
        replace the lower 16 bits of frame_cantime_full by
    ↪ frame_cantime
        frame_cantime_full += ref_cantime
        exact_frame_ktime = cantime_to_ktime(
    ↪ frame_cantime_full)
    return exact_frame_ktime
}
```

Listing 5.3: Fixed algorithm for converting 16-bit timestamps from Xilinx CAN controller [10].

The new algorithm is in listing 5.3. If the received frame is the first frame, it returns the current kernel real time, same as in the first algorithm version. Else, the current kernel time (`frame_ktime`) is converted to "frame_cantime_full" (as if the counter could count to infinity and didn't wrap around at 2^{16}). The `ktime_to_cantime` conversion is multiplying by $counter\_frequency/NSEC\_PER\_SEC$ constant, and `cantime_to_ktime` is its inverse. Then the 16 least significant bits of this `frame_cantime_full` are replaced by the hardware timestamp of current frame. Finally the reference timestamp is added and the full frame cantime is converted back to real time.

Both of the algorithms were implemented by Martin Jerabek in the Xilinx CAN Linux driver. My work has been to rebase the fixed algorithm on latest Linux kernel (since the patch was over 5 years old), polish it and prepare it for submission to Linux. The patch is in repository[2], and hopefully will be accepted by the maintainers. I've also tried to implement my conversion algorithm based on a consultation with my supervisor Pavel Pisa, but it turned out to be virtually the same as the first version from M. Jerabek [10], so I have abandoned it.

5.2 Timestamps in CTU CAN-FD IP core

With CTU CAN FD IP cores, the timestamping situation is much more interesting (and feasible). The CTU CAN FD has support for timestamps on received frames (configurably both at start of frame or end of frame) and supports even time trigerred transmission. By default RX timestamps are sampled at the end of frame [9]. The IP core is designed in such a way that timestamps are provided by the system integrator of the IP core - the integrator provides an up-counting counter (up to 64-bits wide) and connects it to the `timestamp` input of the IP core. This design allows all the IP cores in a single design to share a single time base [8].

There has been preliminary research on timestamps implementation done by Martin Jerabek in[3]. There is a question what to do when the provided timestamps counter is not 64-bits wide. He came to the conclusion that the counter signal should be right-aligned, since we gain nothing by left-aligning and it is even counter intuitive. Which is a good conclusion and we will continue with it. The timestamping frequency will be configurable in two ways: you can pass a second clock phandle in the `clocks` property in Device Tree, or you can pass directly a number as a `ts_frequency` property.

5.2.1 Quick Introduction to Device Tree

This subsection contains a brief introduction to Device Tree, what it is and how it helps Linux users/developers.

[2] `https://gitlab.fel.cvut.cz/vasilmat/linux-can-test`
[3] `https://gitlab.fel.cvut.cz/canbus/ctucanfd_ip_core/-/issues/158`

Device Tree is a system (in the form of a tree data structure) for describing non-discoverable hardware [24]. Because not every bus supports device enumeration (e.g. PCI/PCIe support enumeration, APB and Avalon don't), the kernel source code had to contain full description of the hardware. That includes even register addresses, how many cores given CPU has, how much memory is available in the system, and so on. Now with Device Tree you can describe your hardware in the form of Device Tree Script (dts), compile it to Device Tree Blob (dtb) using the Device Tree Compiler, and then the Linux kernel can load this blob during boot. Alternatively, you can load an Device Tree Blob Overlay during run time, using the `dtbocfg` kernel module [12]. Listing 5.4 shows an example Device Tree script.

As you can see, there are two nodes `node1` and `node2`, both with a bunch of properties in the form of key-value pairs. Device Tree supports a few data types for the propeties:

- String - `A string`
- Cell - a list of 32-bit unsigned integers inside angle brackets - ⟨123 0x1A &clock⟩
- Comma separated list of strings - "first string", "second string"

There is also the `phandle` type used inside cells (e.g. `clocks = <&clkc 15>;`), which is a reference to another node in the device tree. Phandle value itself is a 32-bit unsigned integer, a number referring to another node. In the example above, `&clkc` refers to the with label `clkc` (which actually defines clocks and clock names for MicroZed Zynq 7000 SoC based boards), and the device tree compiler would resolve the `&clkc` phandle to the numerical value of the `clkc` label. There is much more going on in the Device Tree, but we will not go into details here, since we don't need to. For more information about Device Tree, you can refer to the specification [20] or some of the wiki pages on elinux.org [24],[25].

```
/dts-v1/;

/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second
    ↪ string";
        // hex is implied in byte arrays. no '0x' prefix is
    ↪ required
        a-byte-data-property = [01 23 34 56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
```

```
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is
    ↪  a uint32 */
        child-node1 {
        };
    };
};
```

Listing 5.4: Example Device Tree Script, taken from [25].

5.2.2 Common Clock Framework in Linux

Before we get to the timestamping patch itself, there should be few words about the Common Clock Framework in the Linux kernel, since we will make use of it when setting up timestamping clock. The common clock framework was introduced in 2012 to unify the API across a variety of platforms [23]. It provides means to control the clock nodes (getting/setting the clock rate, phase), implement custom clock drivers and much more. But again, we will use only tiny subset of the API - specifically obtaining the clock rate and letting the framework setup the timestaping clock for us using a definition in Device Tree.

One of the main benefits of the framework for us is that it enables clocks to be declared from the Device Tree. The framework parses the device tree and sets up the clocks. Figure 5.1 shows the clock framework architecture - we are in the "Kernel space" part, specifically the user driver. For the clock consumer (us) it is possible to only declare the used clocks in `clocks` property in Device Tree for our hardware. Then in the driver, we obtain a clock reference using `clk_get` / `devm_clk_get` (devm version returns a managed reference to the clock producer - it will be automatically freed when the device is removed [14]). Then we prepare the clock using `clk_prepare` and enable it with `clk_enable`. Listing 5.5 shows some basic example of common clock framework usage (without error handling etc.).

However, it is not necessary to call those functions manually, because the current version of CTU CAN FD Linux driver uses[4] Runtime Power Management framework for I/O devices [16]. This framework takes care of the clock management (among other things). You just implement suspend and resume callback functions, and register those callbacks in `dev_pm_ops` structure (using some macro, e.g. `SIMPLE_DEV_PM_OPS`).

```
struct clk *clk;
u32 frequency;
clk = devm_clk_get(dev, "timestamping_clock");
frequency = clk_get_rate(clk);
```

[4] https://git.kernel.org/pub/scm/linux/kernel/git/mkl/linux-can-next.git/tree/drivers/net/can/ctucanfd/ctucanfd_base.c#n1431

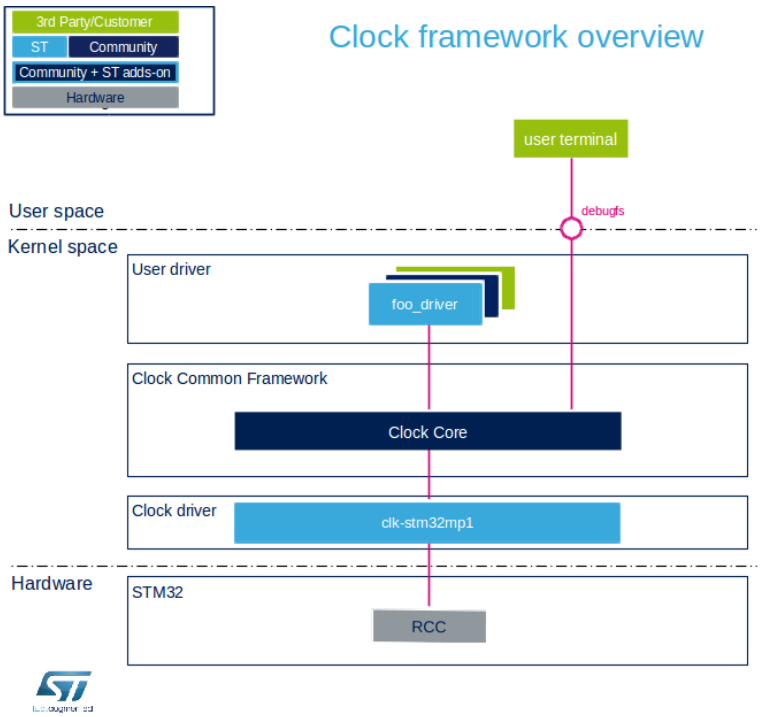


Figure 5.1: Graphical overview of the clock framework. Figure taken from [21].

```
clk_prepare(clk);
clk_enable(clk);
```

Listing 5.5: Example usage of the Common Clock Framework.

5.2.3 Timecounters and cyclecounters

Because the timestamps implementation in CTU CAN FD IP core isn't exactly defined and depends on what the system integrator provides, the Linux driver must be able to deal with smaller counters, which will wrap around. To put things in perspective: assuming 100 MHz timestamping counter frequency, 64-bit wide counter would wrap in almost 6000 years (so we wouldn't have to deal with rollovers, because they won't happen in a lifetime), while 32-bit counter wraps around every 42 seconds. As per the advice from Marc Kleine-Budde[5] (the maintainer of the Linux CAN networking subsystem), the timestamping patch has been implemented using `struct timecounter` and `struct cyclecounter` helper structures.

Cyclecounter is the lower-level structure used for abstracting the real free running counter [15]. It has no state, the struct provides only read access to the underlying counter, and helper constants/variables to convert read timestamp to nanoseconds. Listing 5.6 shows structure definition taken from

[5] `https://marc.info/?l=linux-can&m=163472962426850`

the Linux kernel source. Property `mask` is a bitmask for two's complement subtraction (for non-64bit wide counters). Using the equation below, `mult` and `shift` properties serve for the actual conversion to nanoseconds, see function `cyclecounter_cyc2ns`. Cycles is the number of counter ticks we want to convert to nanoseconds.

$$nanoseconds = (cycles * mult) >> shift$$

The calculation is in fixed point arithmetics, because the FPU's (Floating Point Unit) registers aren't saved when context is switched from userspace to kernel space. So if we were to use floating point operations, we would destroy userspace data. It is possible to save the floating point registers and do FP calculations, but it is not neccessary here, as the calculation is rather simple. Shift is used to avoid precision loss when calculating `mult` constant, e.g. when the frequency is about the same order as 10^9 (nanoseconds per second). E.g. with frequency 133 MHz you would get $\frac{10^9}{133*10^6} = 7.5 \doteq 7$, causing quite the error when converting counter cycles to nanoseconds. Function `clocksource_hz2mult` is used for calculating the multiplication constant from given shift and frequency.

$$mult = \frac{10^9 << shift}{frequency}$$

```
struct cyclecounter {
    u64 (*read)(const struct cyclecounter *cc);
    u64 mask;
    u32 mult;
    u32 shift;
};
```

Listing 5.6: Struct cyclecounter, source [15].

Timecounter is the higher-level structure above cyclecounter. This structure keeps count of the elapsed nanoseconds since it has been initialized. Listing 5.7 shows the structure definition taken from the Linux kernel. Property `nsec` is where the elapsed nanoseconds are kept. Property `frac` is used for accumulating fractional nanoseconds (since the `mult` constant in cyclecounter can be left shifted for better accuracy, the lower masked part of the result is then added to the fractions before shifting back to the right). And finall the property `mask` is a bit mask to maintain fractions as mentioned above. Regarding timecounter usage, you will mostly want to use function `timecounter_cyc2ns`, which converts given counter value to nanoseconds. Also, users of the `timecounter` structure are responsible for reading the counter more often than it wraps around.

```
struct timecounter {
    const struct cyclecounter *cc;
    u64 cycle_last;
    u64 nsec;
```

```
    u64 mask;
    u64 frac;
};
```

Listing 5.7: Struct timecounter, source [15].

The timestamps from different CTU CAN FD controllers are not synchronized, i.e. the timestamps from two controllers for some CAN frame will be slightly shifted, even though the CAN frame has been received at the very same time on both controllers.

5.2.4 Device Tree Bindings

The developed driver expects two properties in the Device Tree bindings:

- second `clocks` phandle or directly `ts-frequency` timestamping frequency
- `ts-used-bits` bit width of the timestamping counter

The work delay for the `struct timecounter` periodic reads is calculated using the algorithm in listing 5.8. Formula for the work delay is

$$work\_delay\_jiffies = \frac{2^{ts\_used\_bits-1}}{ts\_frequency} * HZ.$$

HZ is the number of system ticks per second, and the result is in jiffies (jiffy is the kernel unit of time, 1 jiffy = 1 system tick). Instead of using full $2^{ts\_used\_bits}$, the bit width is reduced by one to read the counter roughly twice per its rollover time. The algorithm works in fixed point arithmetics, so the `fls(HZ)` function is used to get the last set bit in the binary representation of the HZ number. This allows us to shift left as much as possible to get the best achievable precision.

```
u32 jiffies_order = fls(HZ);
u32 max_shift_left = 63 - jiffies_order;
s32 final_shift = (priv->timestamp_bit_size - 1) -
    ↪ max_shift_left;
u64 tmp = div_u64((u64)HZ << max_shift_left, priv->
    ↪ timestamp_freq);

if (final_shift > 0)
    priv->work_delay_jiffies = tmp << final_shift;
else
    priv->work_delay_jiffies = tmp >> -final_shift;

if (priv->work_delay_jiffies == 0)
    return -EINVAL;

priv->work_delay_jiffies = min(priv->work_delay_jiffies,
    ↪ CTUCANFD_MAX_WORK_DELAY_SEC * HZ);
```

```
return 0;
```

Listing 5.8: Algorithm for work delay calculation from timestamps frequency and counter bit width

5.3 Results and Future Work on Timestamping Patches

The produced patches are in the linux-can-next mirror repository[6], in branches `xilinx-timestamping` and `ctucanfd-timestamping`.

Initial RFC version of the patch for CTU CAN FD driver has been sent for review to the linux-can community[7]. Main takeaways from this RFC are:

- `ts-used-bits` will be (most likely) deleted from the DT bindigs. Instead, the counter width will be read from a register in the IP core.
- `ts-frequency` will be deleted, only the `clocks` phandle will be used.
- `Kconfig` option enabling the timestamps by default will be deleted, instead the `SIOCSHWTSTAMP` ioctl should be used.

This DT bindings based solution works only for platform devices. For CTU CAN FD cores used on the PCI bus, a table vendor with vendor name and timestamping frequency will be (probably) maintained in the driver source code.

The CTU CAN FD IP core also allows to set timestamping at the start/end of frame, but there isn't an kernel interface to expose this setting to userspace. One option is to maintain an out-of-tree driver with e.g. various module parameters, which probably wouldn't get merged into the mainline kernel. PEAK-system do something similar with their driver[8].

[6] `https://gitlab.fel.cvut.cz/vasilmat/linux-can-test`

[7] `https://lore.kernel.org/linux-can/20220512232706.24575-1-matej.vasilevski@seznam.cz/t/#u`

[8] `https://www.peak-system.com/fileadmin/media/linux/implementation-details.html`

Chapter 6

Measuring Latency of NuttX OS Running on ESP32

This chapter deals with measuring latency of NuttX OS CAN driver for ESP32C3 TWAI peripheral. My colleague Jan Charvát developed a NuttX driver for CAN controller on a ESP32-C3 board [2], and needed to verify it works correctly. This measurement confirmed that not only it works correctly, but it is also quite fast and can handle heavy load testing.

6.1 Measurement Using One CAN Bus

During the spring semester, an opportunity arose to measure latencies on yet another CAN device. My classmate Jan Charvát was developing a CAN driver for NuttX running on an ESP32 board. And our supervisor came with the idea to test our projects against each other.

There is, however, one catch. Only one CAN device is available on the ESP32 board, so the latency testing has to be slightly modified. Figure 6.1 shows modified setup. The gateway running on ESP32 retransmitts the CAN message with ID decremented by one to increase the message's priority, and ensure that the gateway will be able transmit its response, even when latester is flooding the CAN bus with its messages. Crossbar configuration here is `0x01000000` - tie all the CAN controllers together in one line and enable the line.

6.2 NuttX

NuttX is a real time operating system with focus on being scalable (from 8-bit to 64-bit microcontrollers), compliant with standards (POSIX), highly configurable and extensible to new processor/SoC/board architectures [7]. It has small footprint (the smallest NuttX build can fit into 32 kB of memory). NuttX aims to be ANSI and POSIX compliant, so programming for NuttX is easier when you already know development for Linux. Of course NuttX doesn't implement everything that Linux provides, only a small subset, to keep its small footprint. It could be called "Linux lite". And it also provides some features of VxWorks (task management, watchdog timers), another

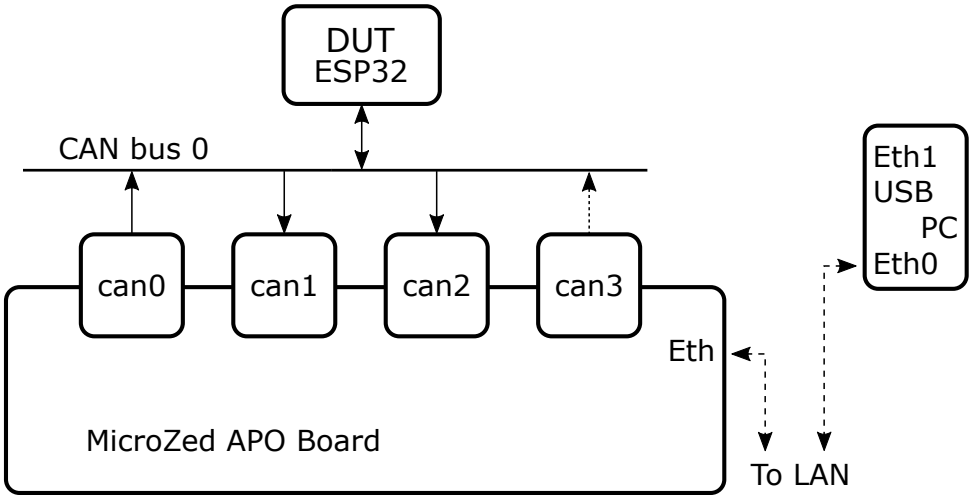


Figure 6.1: Modified setup with ESP32 board running NuttX as a Device Under Test (DUT). Interface can3 can be used for flooding the CAN bus with low priority traffic.

real time OS developed by Wind River Systems. The NuttX system is fully pre-emptible, supports FIFO and round-robin scheduling, sporadic events, tickless operation, and has support for priority inheritance.

NuttX manages to keep its small footprint by using a special source code tree structure. It consists of a lot of small files, and most files contain only one function. Those source files are then compiled into objects and saved as static libraries. This allows the linked to include only the actually used parts from the archives into the final binary. Also the configuration file allows you to only compile features you plan to use (similar to Linux config). Besides that, it makes use of GNU toolchain's support for *weak* symbols to further help keep the footprint tiny.

From device drivers, NuttX supports character and block devices, has network drivers, USB drivers, serial, I2C, CAN bus, and much more [7]. My colleague Jan Charvát worked on the lower-half of a CAN driver for Espressif ESP32-C3 microcontroller.

6.3 Espressif ESP32-C3

The NuttX OS runs on an Espressif ESP32-C3-Mini-1, a module based on the ESP32-C3 series SoC, running on a 32-bit single core RISC-V architecture microprocessor [22]. It has WiFi and Bluetooth connectivity, accomodates 15 GPIOs (General Purpose Input/Output pins), and has built-in TWAI controller. TWAI is practically the CAN protocol, but renamed to avoid licensing costs. It supports only the classical CAN, not the CAN FD.

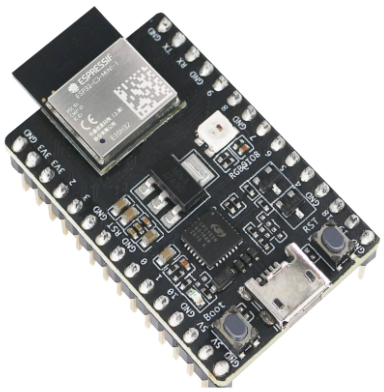


Figure 6.2: ESP32-C3 DevKit.

6.4 Latester Modifications

Latester (the application used for measuring latencies) had to be modified for this new use case. At first, only slight modification in the measurement thread in frames processing to differentiate the original frame and the responded frame with ID smaller by 1. Then we ran the measurements, only to find out there were huge frame losses, ranging roughly from 30 % up to 60 % of all frames sent. A loss means that frame was successfully sent, but latester haven't received back the same frame with modified ID.

We verified that NuttX responded to all CAN frames correctly, see the note on frame loss in section 6.6. So I went looking for bugs in latester and found a bug with timeouting at the end of measurement, but this didn't help with the lost frames. We then identified that the problem is that it the system runs is not fast enough to process all the CAN frames, as there have been multiple `rx fifo overflow` errors from the CAN controllers. After all, the system now has to handle double the amount of interrupts from CAN frames reception, compared to the regular measurement with two CAN buses.

So acceptance filters have been used to filter frames on both reception interfaces. Interface `can1` is set up to ignore the modified frame ID (regular-1). Likewise, interface `can2` ignores the regular measurement ID. Those filters helped to reduce processing load, because it avoids context switches between the kernel and userspace (latester no longer has to `recvmmsg()` a CAN frame only to find out it can be ignored).

Acceptance filters can be set using SocketCAN [6]. Listing 6.1 shows an example snippet with two CAN filters. Those filters are inverse, so only frame with ID different than the filtered passes. However, the filter passed to `CAN_RAW_FILTER` option are in logical `OR`. Which makes the two inverted filters useless. To use the filters in logical `AND`, a `CAN_RAW_JOIN_FILTERS` socket option is needed. Listing 6.2 shows a snippet of how the join option is used.

```
struct can_filter rfilter[2];

rfilter[0].can_id   = 0x00A | CAN_INV_FILTER;
rfilter[0].can_mask = CAN_SFF_MASK;
rfilter[1].can_id   = 0x00B | CAN_INV_FILTER;
rfilter[1].can_mask = CAN_SFF_MASK;

setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(
    ↪ rfilter));
```

Listing 6.1: Example filter setup on CAN socket, the first filter blocks messages with ID `0xA`, the second blocks IDs `0xB`.

```
int join_filter = 1;
setsockopt(s, SOL_CAN_RAW, CAN_RAW_JOIN_FILTERS, &join_filter,
    ↪  sizeof(join_filter))
```

Listing 6.2: Using `CAN_RAW_JOIN_FILTERS` to join the filters on socket `s` into logical `AND`.

Those filters, together with making sure we are running RT patched Linux, helped to ensure that all CAN frames are and will be handled.

6.5 Measurements

We have tried to automate the measurements as much as possible, so I've developed a Python script (included in the `can-benchmark` repository) to run parametrized tests. The altered variables are:

- CAN bus speed: we used 3 different bitrates: 125k, 500k, 1M,
- CAN frame length: 2, 4 or 8 bytes,
- number of messages sent: 100, 1000 or 10000,
- boolean switch indicating whether the messages are sent one by one or in "flood" mode,
- boolean switch indicating whether NuttX had artificial load.

Artifical load on NuttX is done by running another thread which runs a busy loop and prints into the console. This thread has lower priority than the main thread which sends CAN frames.

From the list of variables, CAN bus speed and NuttX artificial load were not automated. So we set those things manually. It could be done, but we didn't devote resources to it. To automate the CAN bus speed setting, one would have to recompile the NuttX OS with new configuration, and then load the binary onto the ESP32 board. This could be hacked together with Python's `os` standard library. To automate starting the busy loop on NuttX,

one could use some Python library to connect NuttX's NSH shell over serial line and send a command to start the load.

Results from those automated tests were processes using another Python script. This script gathered all available files with histograms, grouped them by the flood switch, bus speed, and number of messages, and generates another script (this time shell) to run `gnuplot` commands and create .pdf files with the plots. The generated plots are in appendix C. However, those measurements were done in the beginning of April, and by then we still didn't have everything figured out about both the testing and the tested parts, so those plots should be taken with a grain of salt.

We have also done a test measurement to verify NuttX scheduling and demonstrate the effect of incorrectly set process priorities. The busy-loop process's priority has been changed to be higher than that of the CAN responder.

6.6 Results

Table 6.1 shows measured NuttX latencies when sending messages one at a time. For bus speed 125 kb/s, the measured latency is zero, which means that NuttX managed to process the message during the time for 3 IFS (Inter Frame Space) bits. At the 125 kb/s bitrate, one bit takes about 8 µs, so the inter frame space is 24 µs. Therefore we can conclude that the NuttX processing latency is smaller than 24 µs, and thus is unmeasurable. Keep in mind that "zero NuttX latency" means "zero NuttX latency when measuring on a single CAN bus". If we had another CAN bus available, we would measure some nonzero latency.

And judging from the results for 1 Mb/s bitrate, we can estimate the NuttX latency to be around 15 µs. At 1 Mb/s, the inter frame space is only 3 µs (1 bit takes 1 µs). Also the effect of frame length on processing latency can be seen here, for frame length 2 the latency is on average 10 µs, for frame length 8 it goes up to 12 µs. For bus speed 500 kb/s, the average latencies are 8 µs (10 µs). As the inter frame space is 6 µs, more of the processing latency hides in the IFS time.

	2 data bytes		4 data bytes		8 data bytes	
bitrate	avg [us]	worst [us]	avg [us]	worst [us]	avg [us]	worst [us]
125 Kbps	0	0	0	0	0	0
500 Kbps	8	10	8	10	10	12
1 Mbps	10	12	10.25	12	12	14

Table 6.1: Measured NuttX latencies, messages sent one at a time. 3200 messages were sent.

Table 6.2 shows the measured latencies when sending messages in flood mode (as fast as possible). For bitrate 125 kb/s, the latencies are zero. NuttX is fast enough to process the CAN message during inter frame space, and then sends back the response without problem, because the response has higher

priority (ID `0x9` vs original ID `0xA`). For bitrates 500 kb/s and 1 Mb/s, we can see nonzero latencies. Those average latencies rougly correspond to the time it takes to send one CAN frame. E.g. for bus speed 500 kb/s, one frame of length 8 is 16 bytes long (for more accurate terminology, "data length" should be used instead of "frame length"), 16 bytes is 128 bites, and 1 bite takes 2 µs. Thus the average frame takes about 256 µs (not including bit stuffing, CRC field etc.). Because at those bitrates NuttX won't process the message during IFS, another CAN frame is sent by the latester, and NuttX has to wait until it another CAN frame is completely send before it can send its response back. The data back this up, as the average latency is proportionally lower for shorter frame lengths/faster bitrate.

We have also tested how NuttX handles heavy traffic load by running `cangen`, and `candump`-ing all CAN frames into a text file (to a ramdisk to minimize network traffic). This log file confirmed that NuttX correctly responded to all the messages, and endured traffic of about 10k frames per second. The command used to generate frames is `cangen can2 -I 0xA -g 0 -i -x` (full load test ignoring "No buffers available" error).

	2 data bytes		4 data bytes		8 data bytes	
freq	avg [us]	worst [us]	avg [us]	worst [us]	avg [us]	worst [us]
125 Kbps	0	0	0	0	0	0
500 Kbps	132.5	286	170.3	178	246.2	250
1 Mbps	66	141	85.2	89	123.2	127

Table 6.2: Measured NuttX latencies, messages sent a flood mode. 3200 messages were sent.

Finally graph 6.3 shows the effect of incorrectly set process priorities. When the priorities are correct (busy loop has lower priority than the useful task), the latency is mostly zero, and only for few frames (count in lower tens) gets above zero. When the priorities are inverted, the latency profile degraded and more than 200 messages have rather large latency - roughly around 250 µs and more. There are also two big steps at the end (for both profiles), which means that few frames had latency in the order of milliseconds, which is quite bad. We were not able to pinpoint why it is happening, doesn't seems like a bug in the latester (we have exact timestamps from CTU CAN FD cores, which prove that the frame appeared on bus much later). And during regular measurement without CPU load on NuttX, such latencies were not observer.

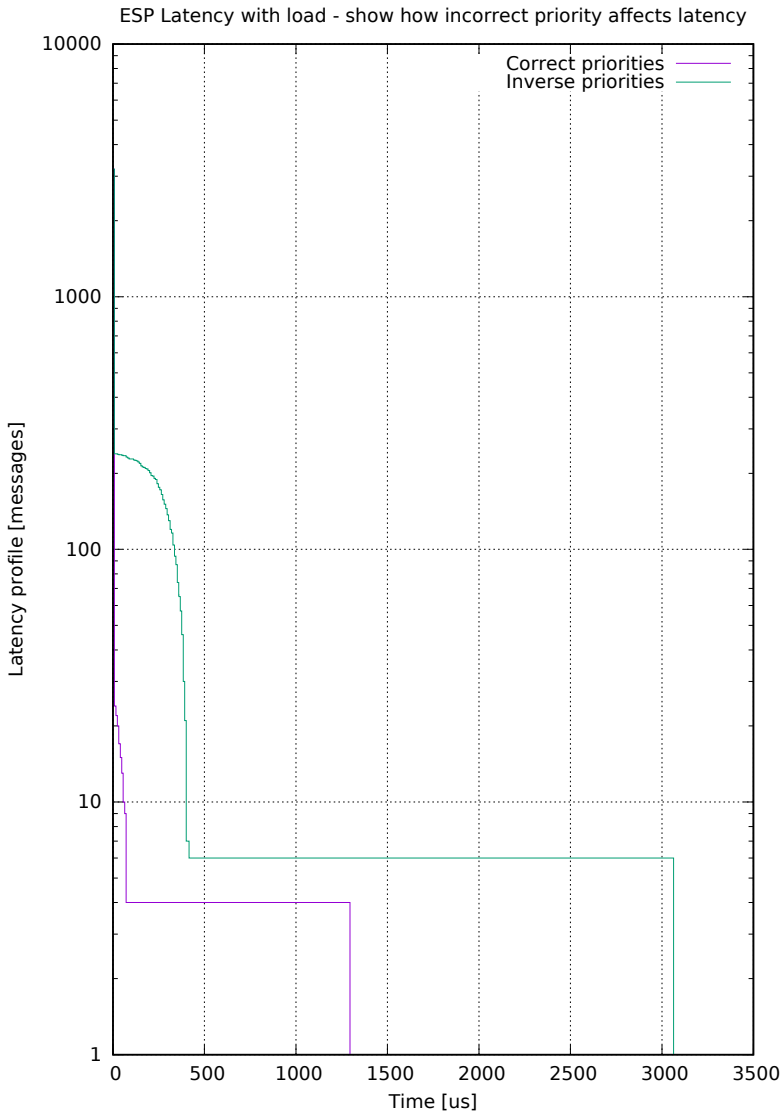


Figure 6.3: NuttX latencies, system is under load by busy loop, showing the effect of incorrectly set priorities. Bus speed is set to 125k, frame length 2 is used, 3200 messages sent, sending messages one at a time.

Chapter 7
Conclusion

The goal of this thesis was to update the system for latency testing to support CAN FD devices. This has been accomplished.

The software for latency testing (latester, userspace gateway ugw) have been updated to support CAN FD. The FPGA design has been extended with two additional CTU CAN FD IP cores, to include total of 4 IP cores and facilitate testing on CAN FD bus (as the Xilinx CAN doesn't support CAN FD, nor would it be desirable to use them because of the small timestamping counter).

Linux driver for CTU CAN FD has been extended to report frame timestamps. Feedback on the version of this extension has been gathered from the Linux community, and I hope to get a new fixed patch version merged in the near future. The timestamping patch for Xilinx CAN driver from M. Jerabek has been updated to latest kernel version and will be submitted soon, too.

The latency testing system has been thoroughly documented (mainly chapters 2 and 3). However, it hasn't been offered to OSADL to be integrated into their QA realtime farm.

As an extra work, we have together with Jan Charvát measured latencies of his NuttX CAN driver for ESP32-C3 TWAI peripheral. The latester has been extended to support also this measurement type (using only one CAN bus). This work helped to verify the correctness and robustness of his implementation. We have also showed that NuttX is very fast and is capable of handling very high traffic loads on CAN bus.

However, this extra testing has been at the cost of not setting up a web server with continous latency testing results. The web server setup remains to be finished after the deadline for this thesis.

Appendix A

U-Boot FIT file

```
/dts-v1/;

/ {
        description = "Kernel for MicroZed";
        #address-cells = <1>;

        images {
                kernel@1 {
                        description = "Linux kernel";
                        data = /incbin/("./images/linux/
    ↪ zImage");
                        type = "kernel";
                        arch = "arm";
                        os = "linux";
                        compression = "none";
                        load = <0x00008000>;
                        entry = <0x00008000>;
                        hash@1 {
                                algo = "crc32";
                        };
                        hash@2 {
                                algo = "sha1";
                        };
                };
                fdt@1 {
                        description = "Flattened Device Tree
    ↪ blob";
                        data = /incbin/("./images/linux/
    ↪ system.dtb");
                        type = "flat_dt";
                        arch = "arm";
                        compression = "none";
                        hash@1 {
                                algo = "crc32";
                        };
```

```
                        hash@2 {
                                algo = "sha1";
                        };
                };
                fpga@1 {
                        description = "FPGA bitstream";
                        data = /incbin/("./images/linux/
    ↪ system.bit.gz");
                        type = "filesystem";
                        compression = "gzip";
                        hash@1 {
                                algo = "crc32";
                        };
                        hash@2 {
                                algo = "sha1";
                        };
                };
                ramdisk@1 {
                        description = "Initial RAM Filesystem
    ↪ ";
                        data = /incbin/("./images/linux/
    ↪ initramfs.cpio.gz");
                        type = "ramdisk";
                        arch = "arm";
                        os = "linux";
                        compression = "none";
                        hash@1 {
                                algo = "crc32";
                        };
                        hash@2 {
                                algo = "sha1";
                        };
                };
        };

        configurations {
                default = "conf@1";
                conf@1 {
                        description = "Boot Linux kernel with
    ↪  FDT blob";
                        kernel = "kernel@1";
                        fdt = "fdt@1";
                        ramdisk = "ramdisk@1";
                };
        };
};
```

Listing A.1: uboot-image.its file - U-Boot FIT (Flattened uImage Tree).

Appendix B

Attachment Contents

The following files are in the attachment/attached CD.

- `ctucanfd_patch` directory with the patch files for CTU CAN FD driver with timestamping. Linux kernel git is too big to be included. It is also available at `https://gitlab.fel.cvut.cz/vasilmat/linux-can-test`, branch `ctucanfd-timestamping`.
- `xilinx_patch` with the timestamping patch for Xilinx CAN driver. Also available at `https://gitlab.fel.cvut.cz/vasilmat/linux-can-test`, branch `xilinx-timestaping`.
- `can-benchmark` repository with my modifications to the latester and ugw. Also available at `https://gitlab.fel.cvut.cz/canbus/can-benchmark`.
- `zynq-can-sja1000-top` repository with my tiny change in HDL to add the two CTU CAN FD IP cores. Also available at `https://gitlab.fel.cvut.cz/canbus/zynq/zynq-can-sja1000-top`, branch `mz_apo-2x-xcan-4x-ctu`.

Appendix C

Plots From Latency Measurement on NuttX

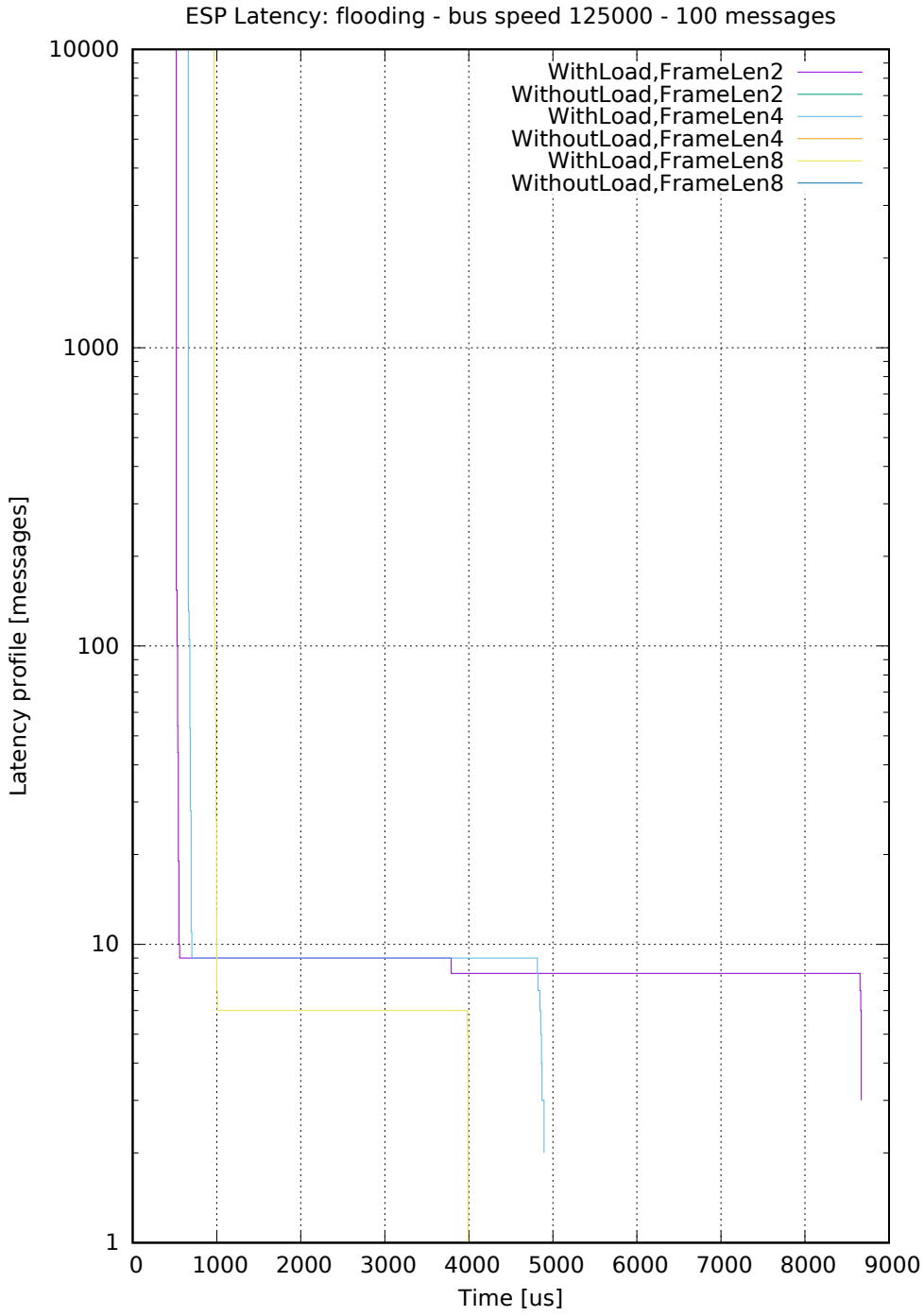


Figure C.1: ESP latency profile: flooding - bus speed 125000 - 100 messages

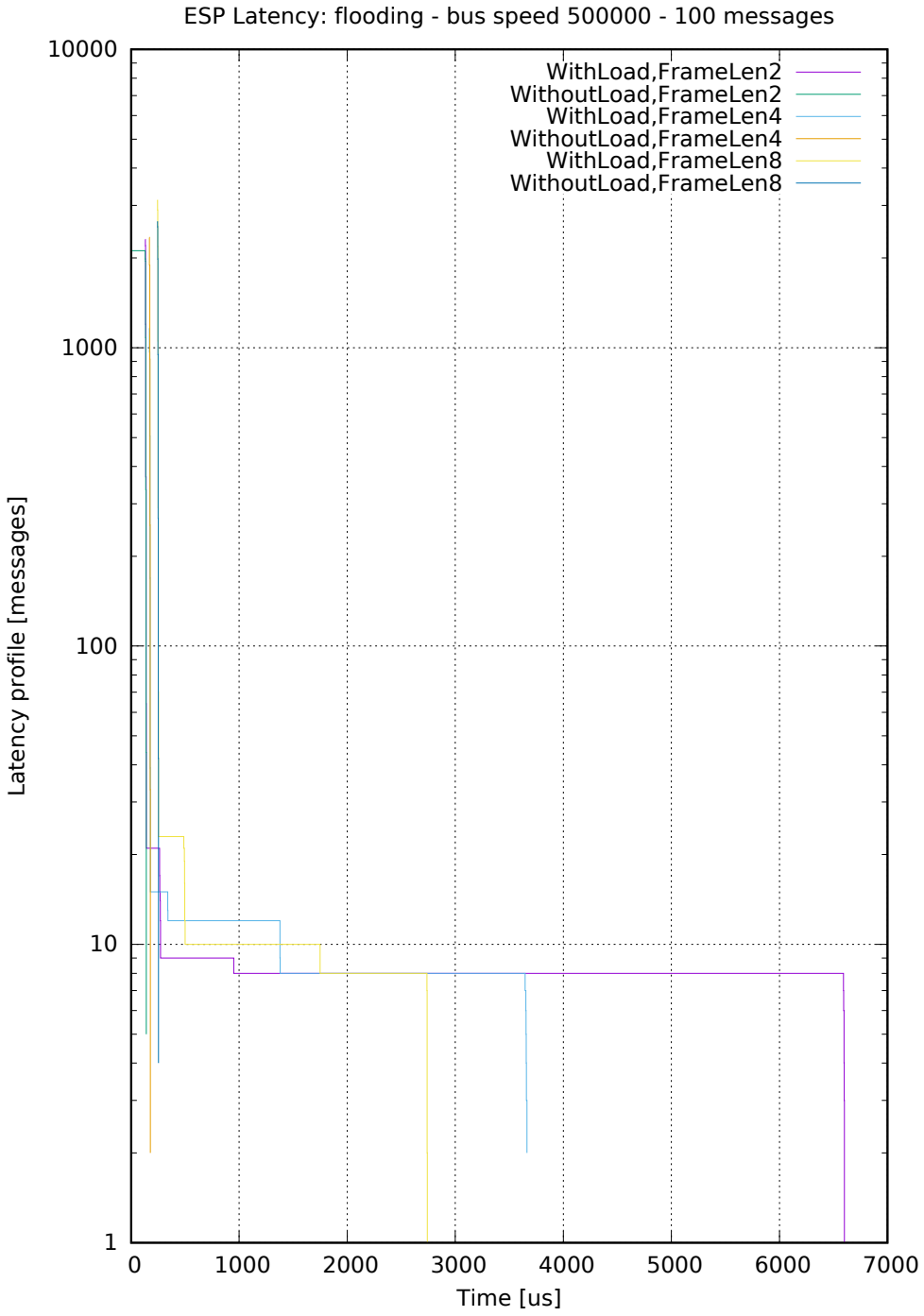


Figure C.2: ESP latency profile: flooding - bus speed 500000 - 100 messages

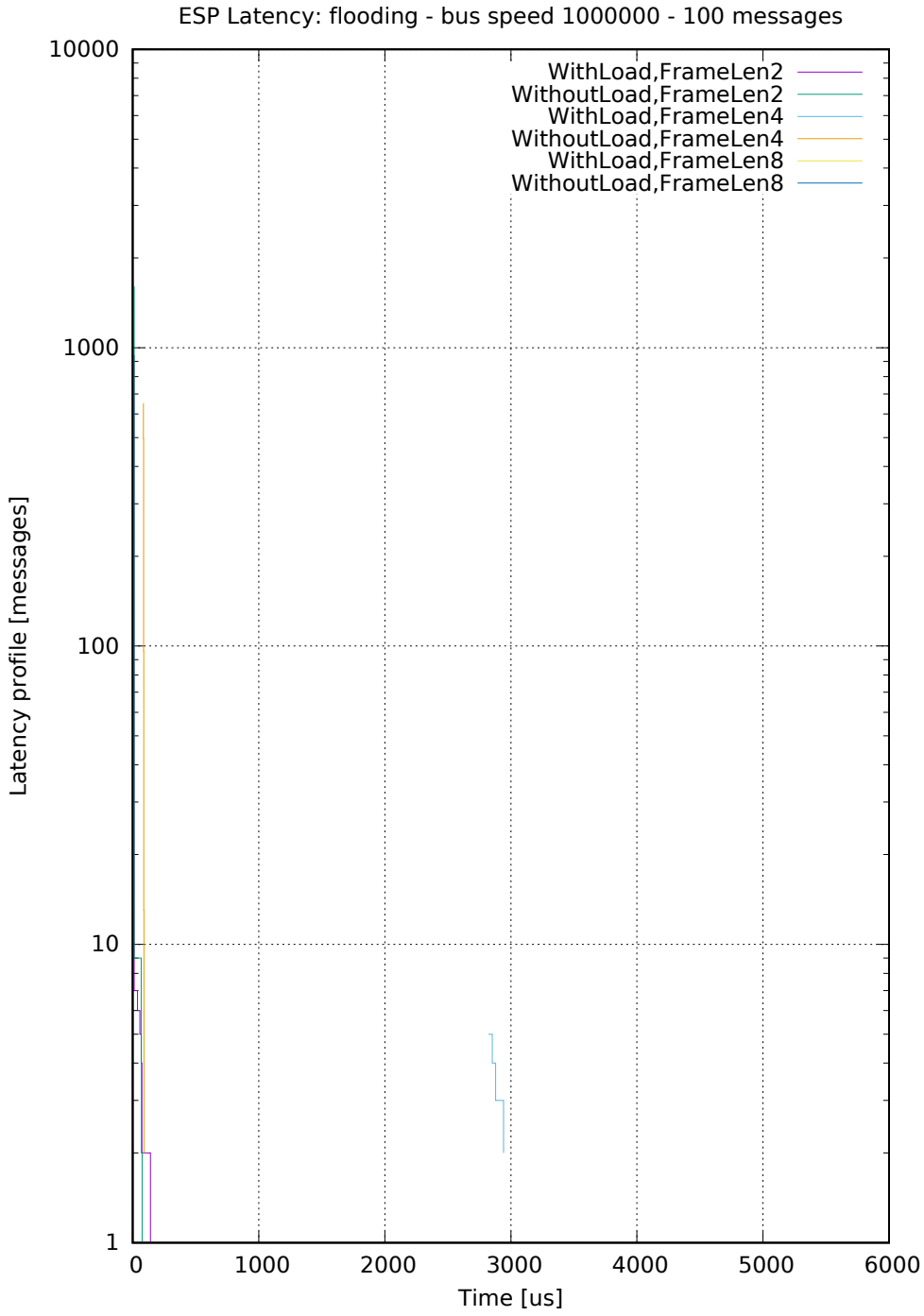


Figure C.3: ESP latency profile: flooding - bus speed 1000000 - 100 messages

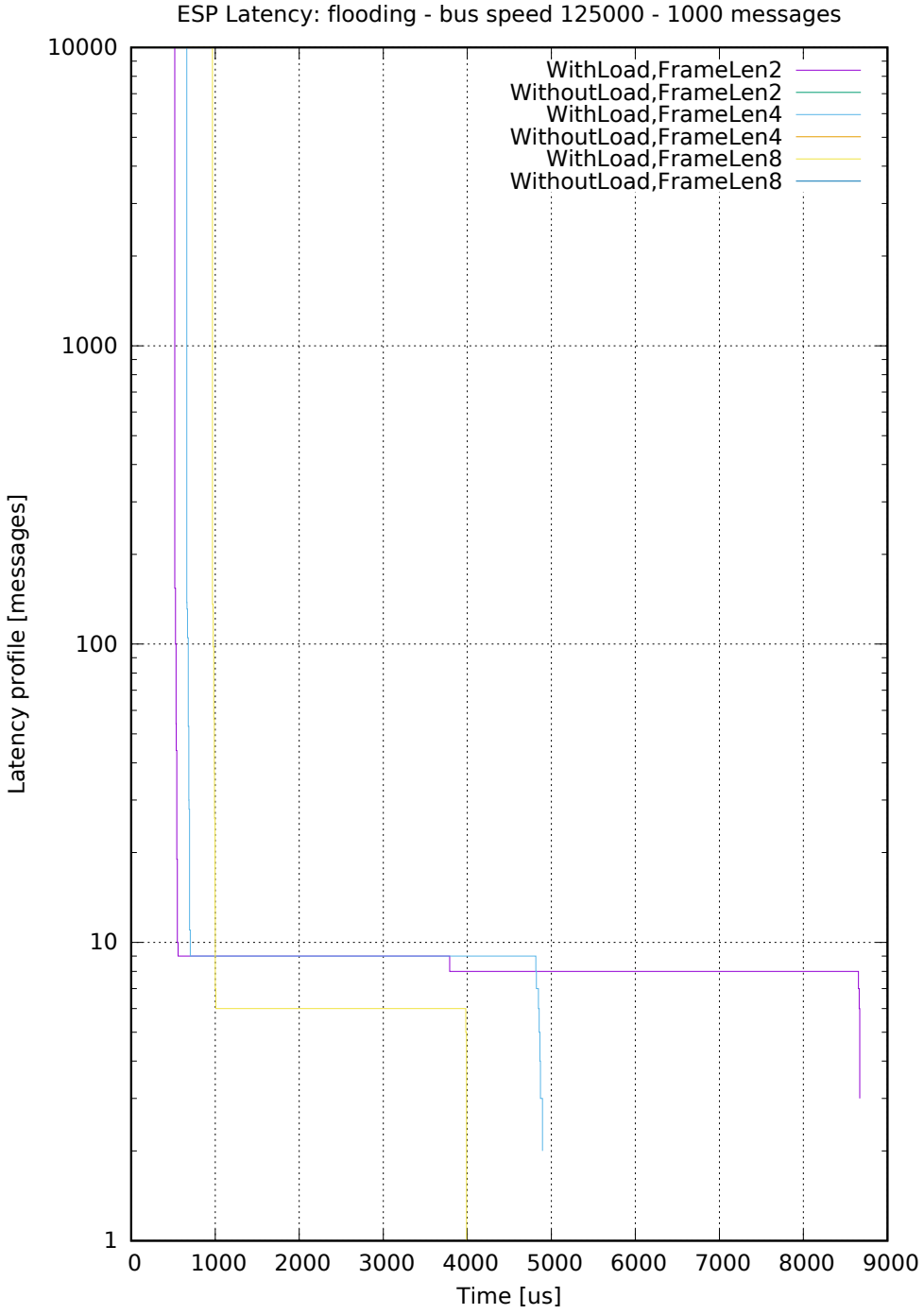


Figure C.4: ESP latency profile: flooding - bus speed 125000 - 1000 messages

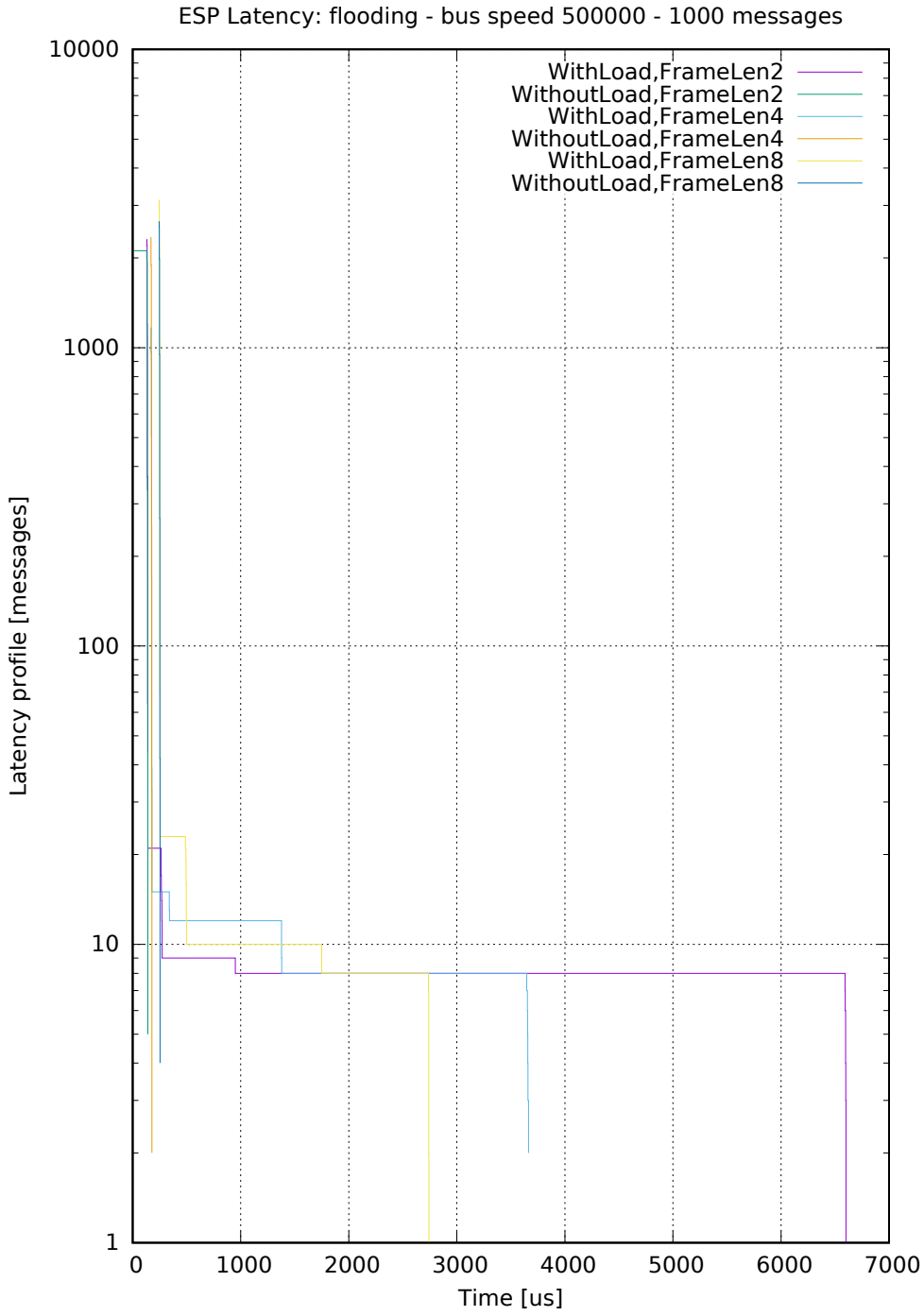


Figure C.5: ESP latency profile: flooding - bus speed 500000 - 1000 messages

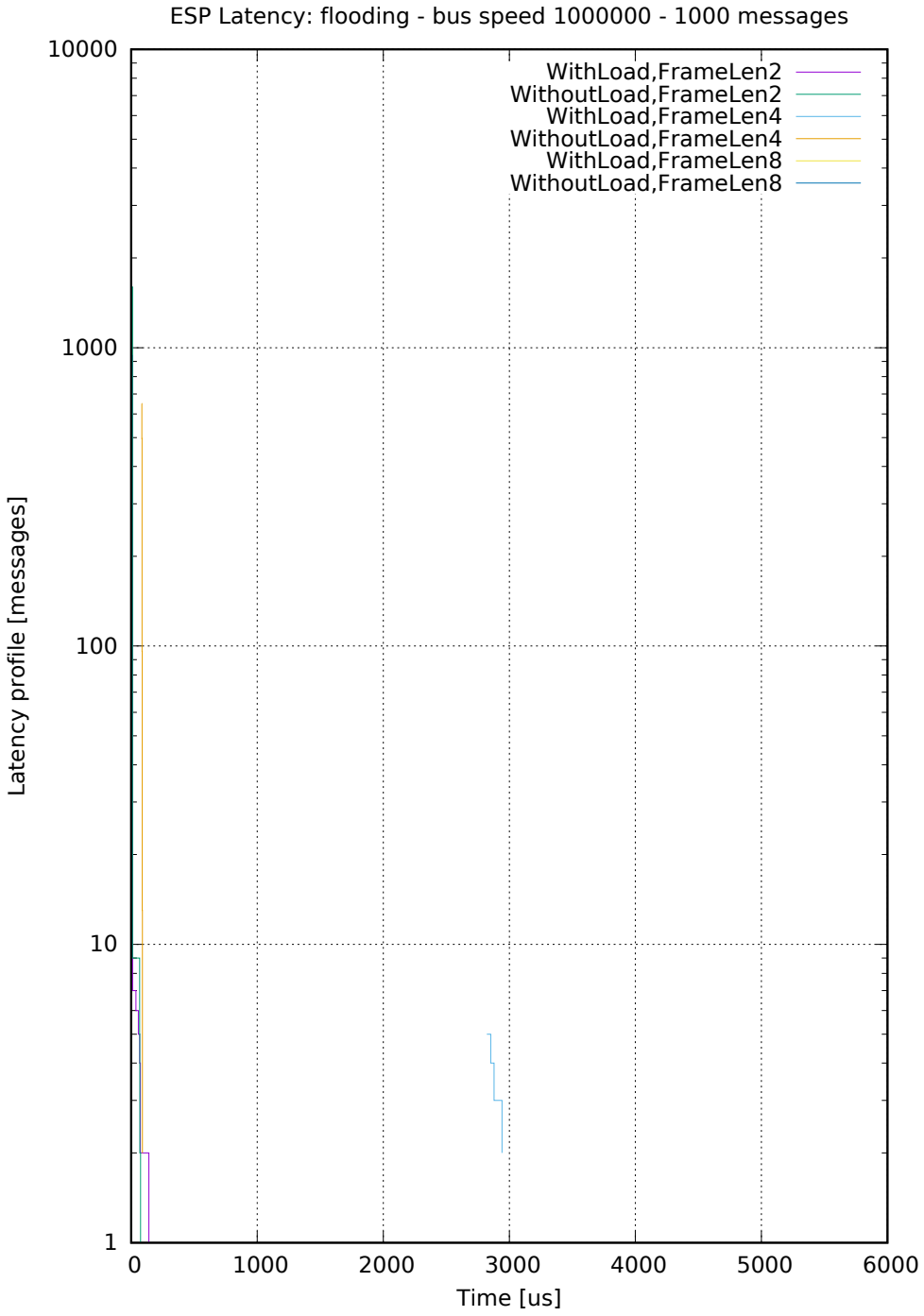


Figure C.6: ESP latency profile: flooding - bus speed 1000000 - 1000 messages

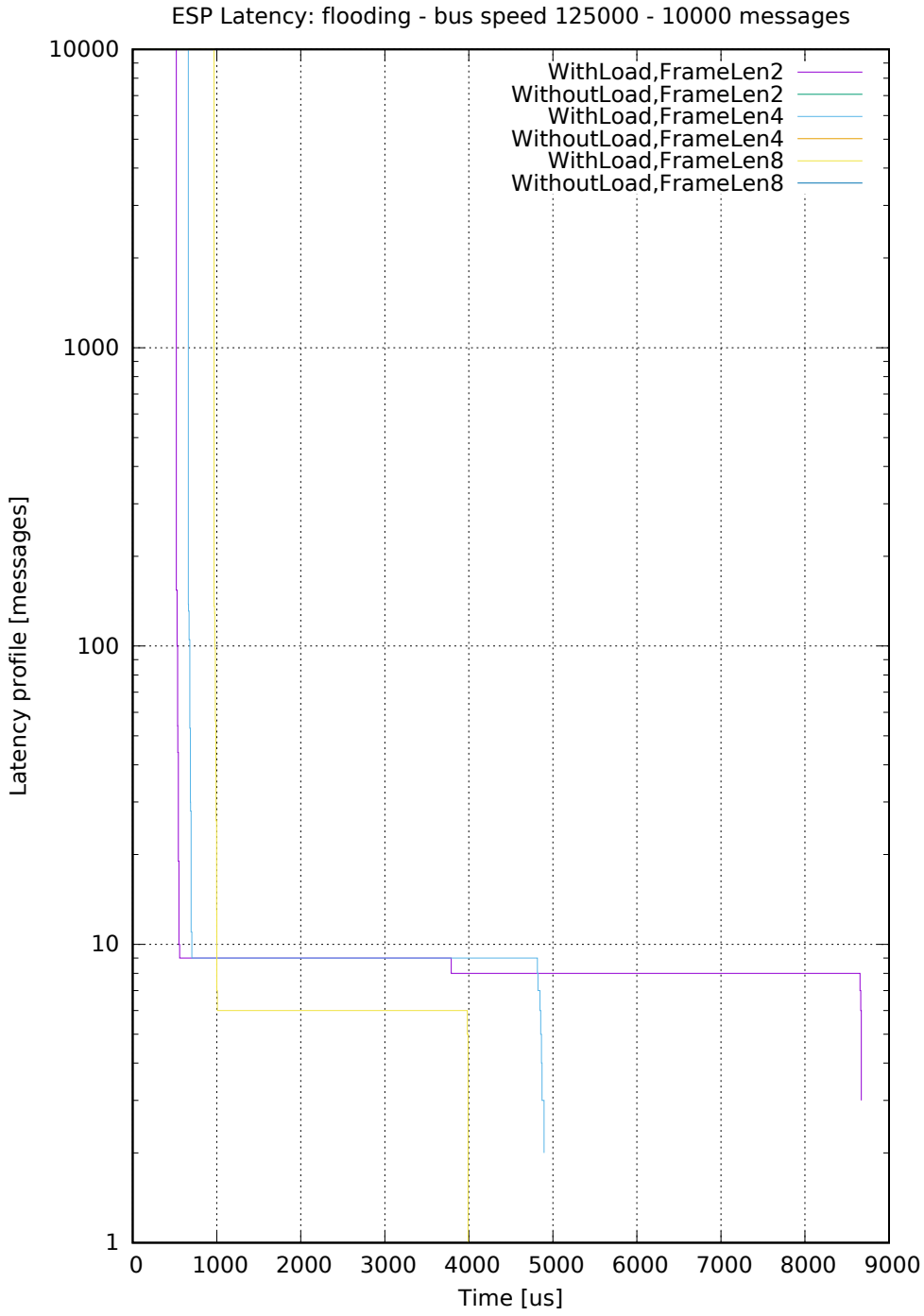


Figure C.7: ESP latency profile: flooding - bus speed 125000 - 10000 messages

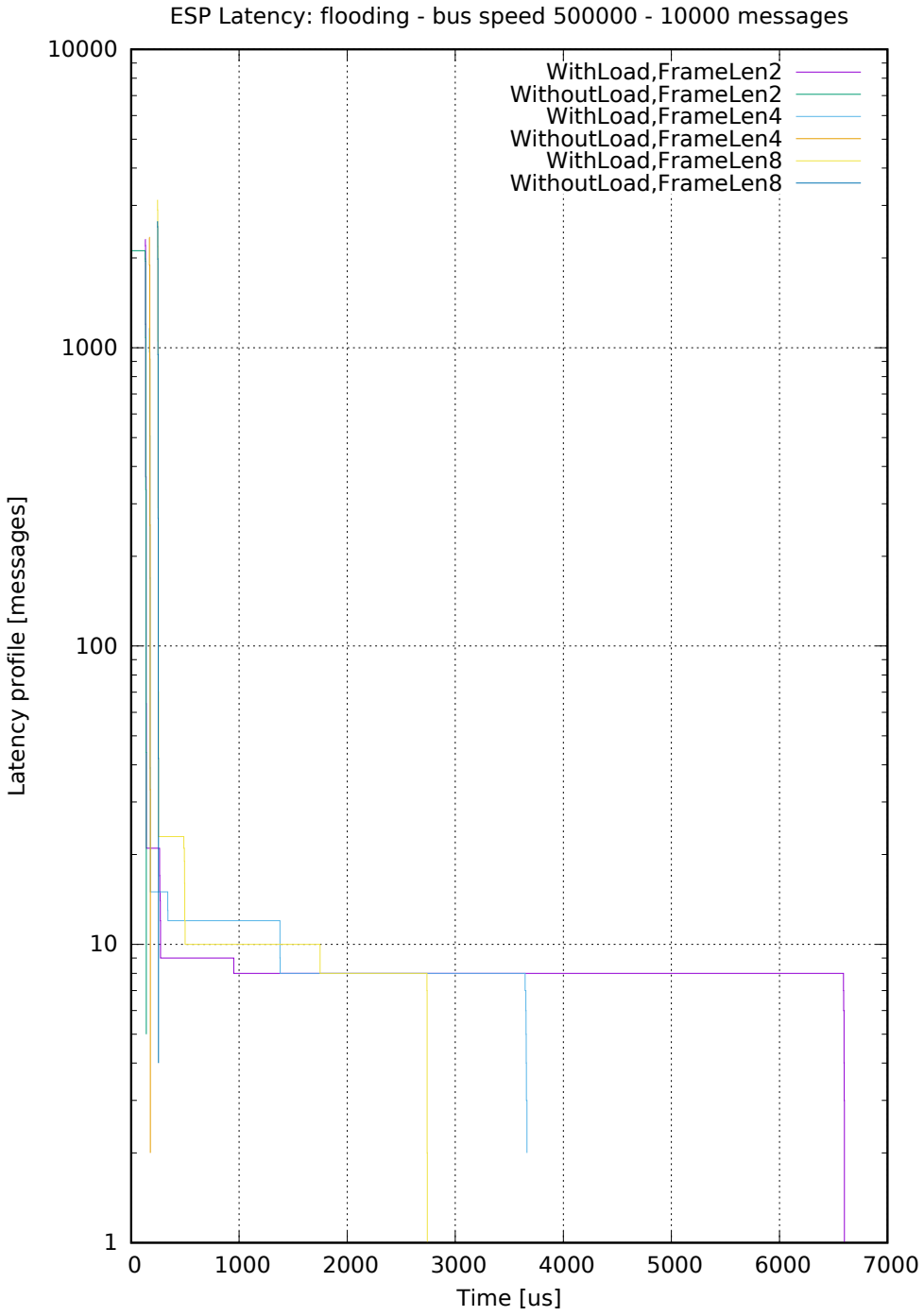


Figure C.8: ESP latency profile: flooding - bus speed 500000 - 10000 messages

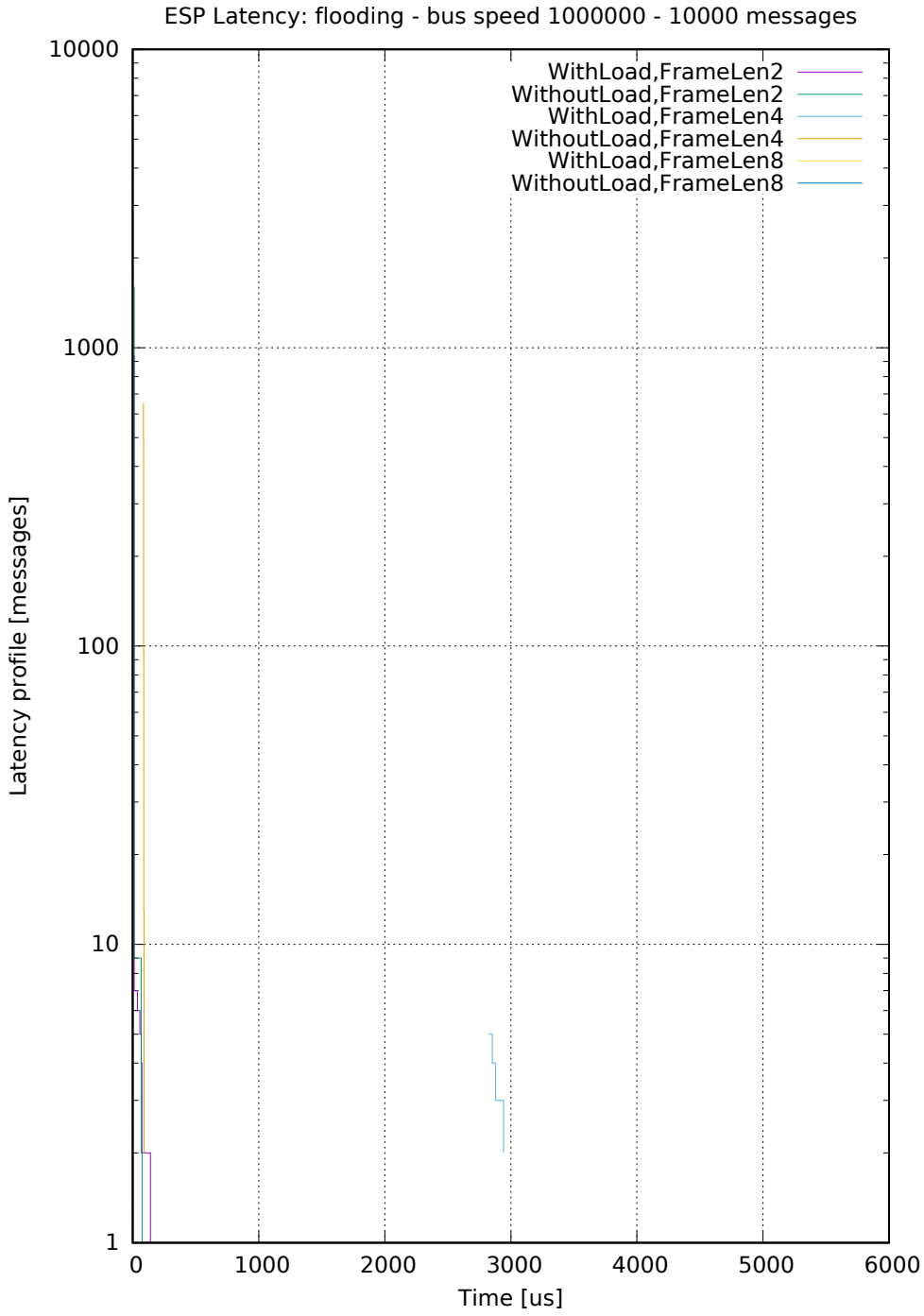


Figure C.9: ESP latency profile: flooding - bus speed 1000000 - 10000 messages

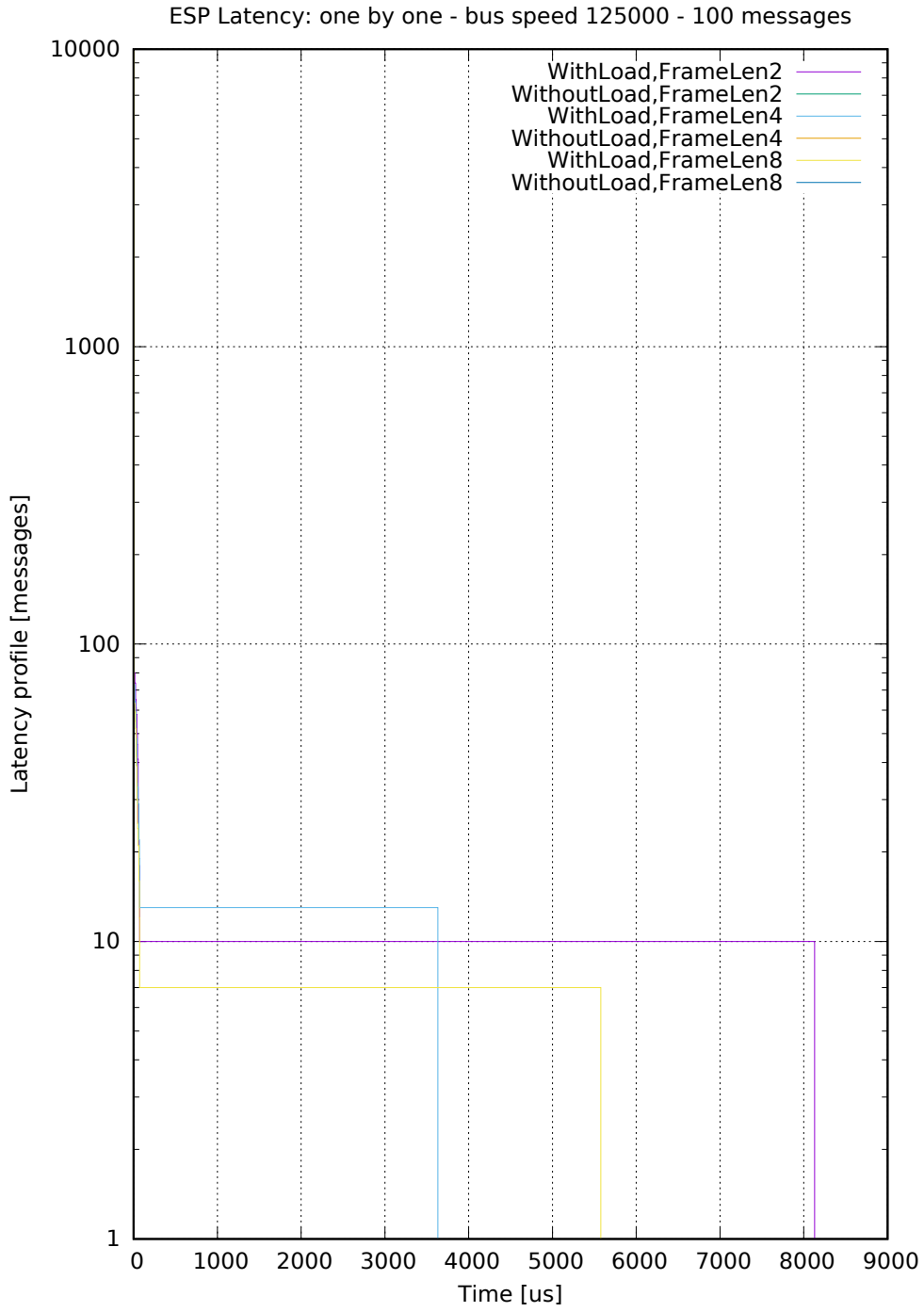


Figure C.10: ESP latency profile: one by one - bus speed 125000 - 100 messages

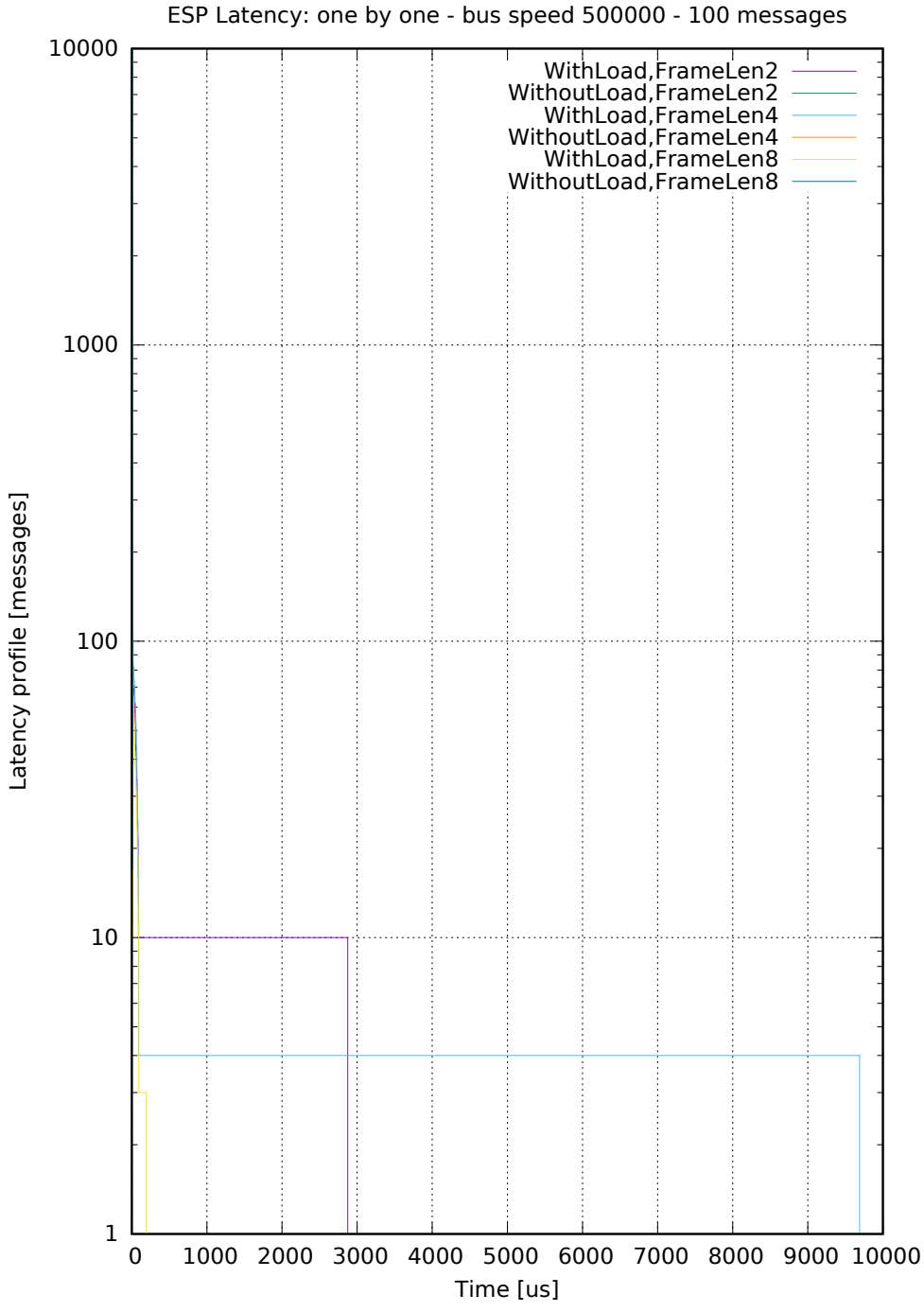


Figure C.11: ESP latency profile: one by one - bus speed 500000 - 100 messages

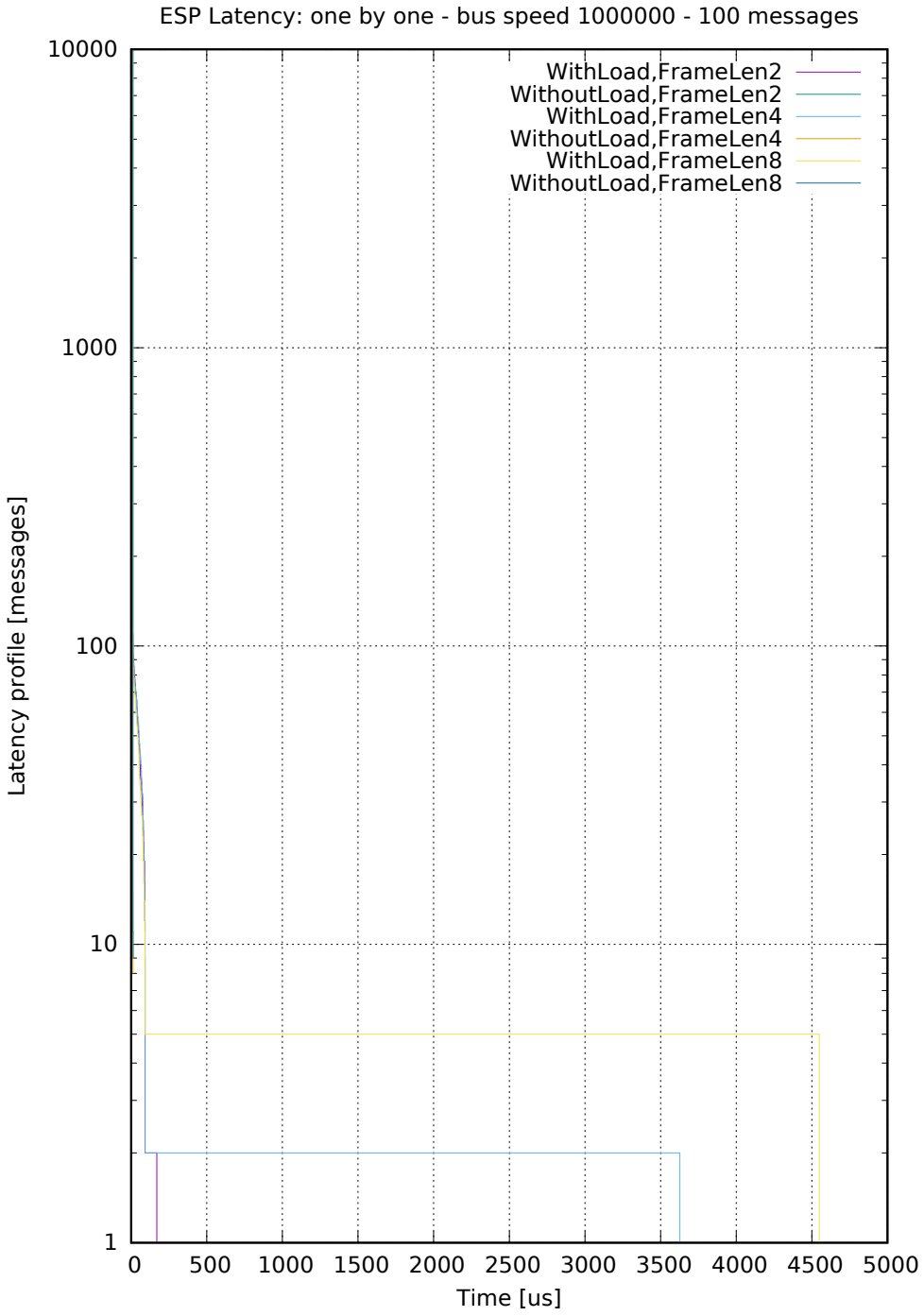


Figure C.12: ESP latency profile: one by one - bus speed 1000000 - 100 messages

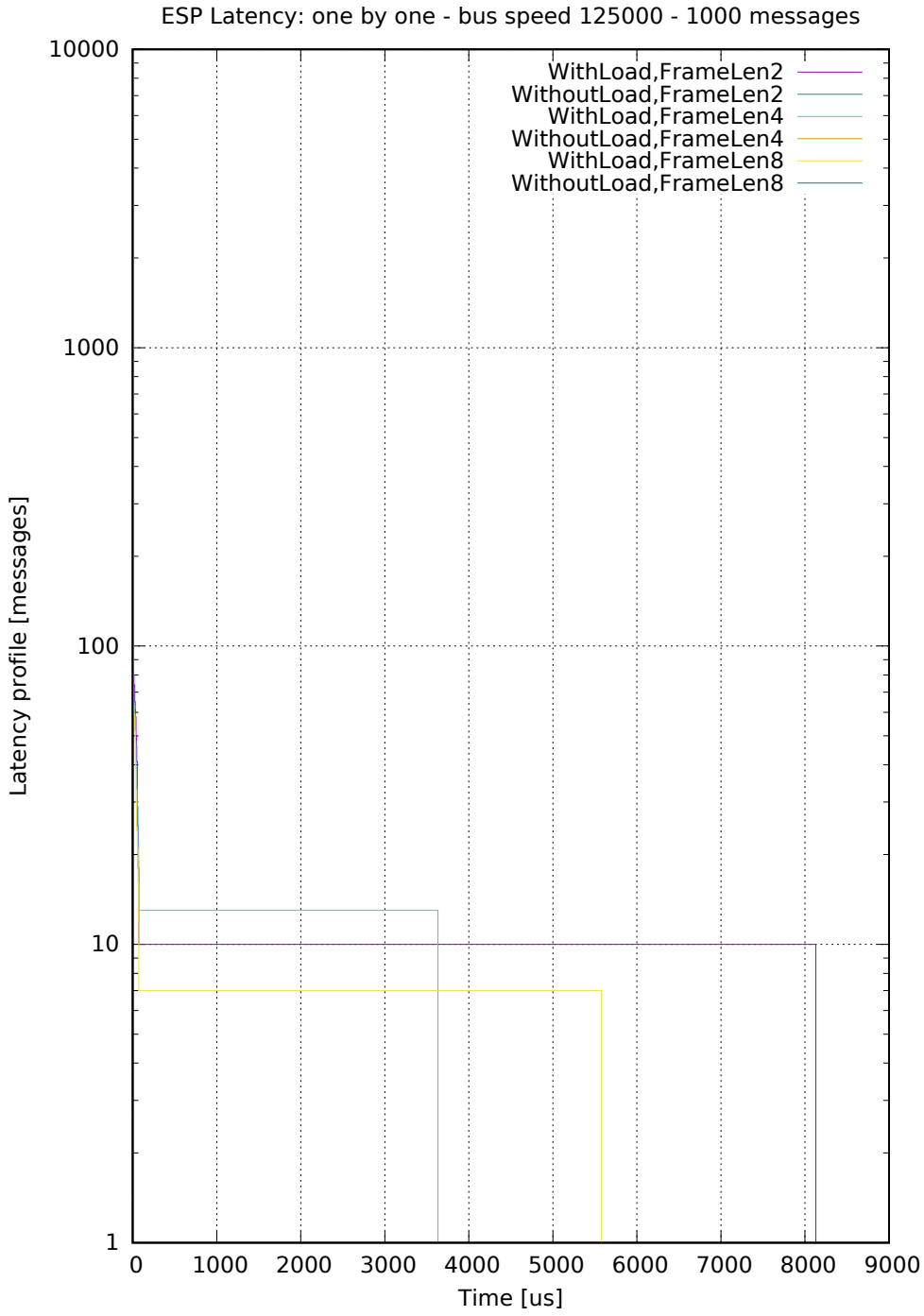


Figure C.13: ESP latency profile: one by one - bus speed 125000 - 1000 messages

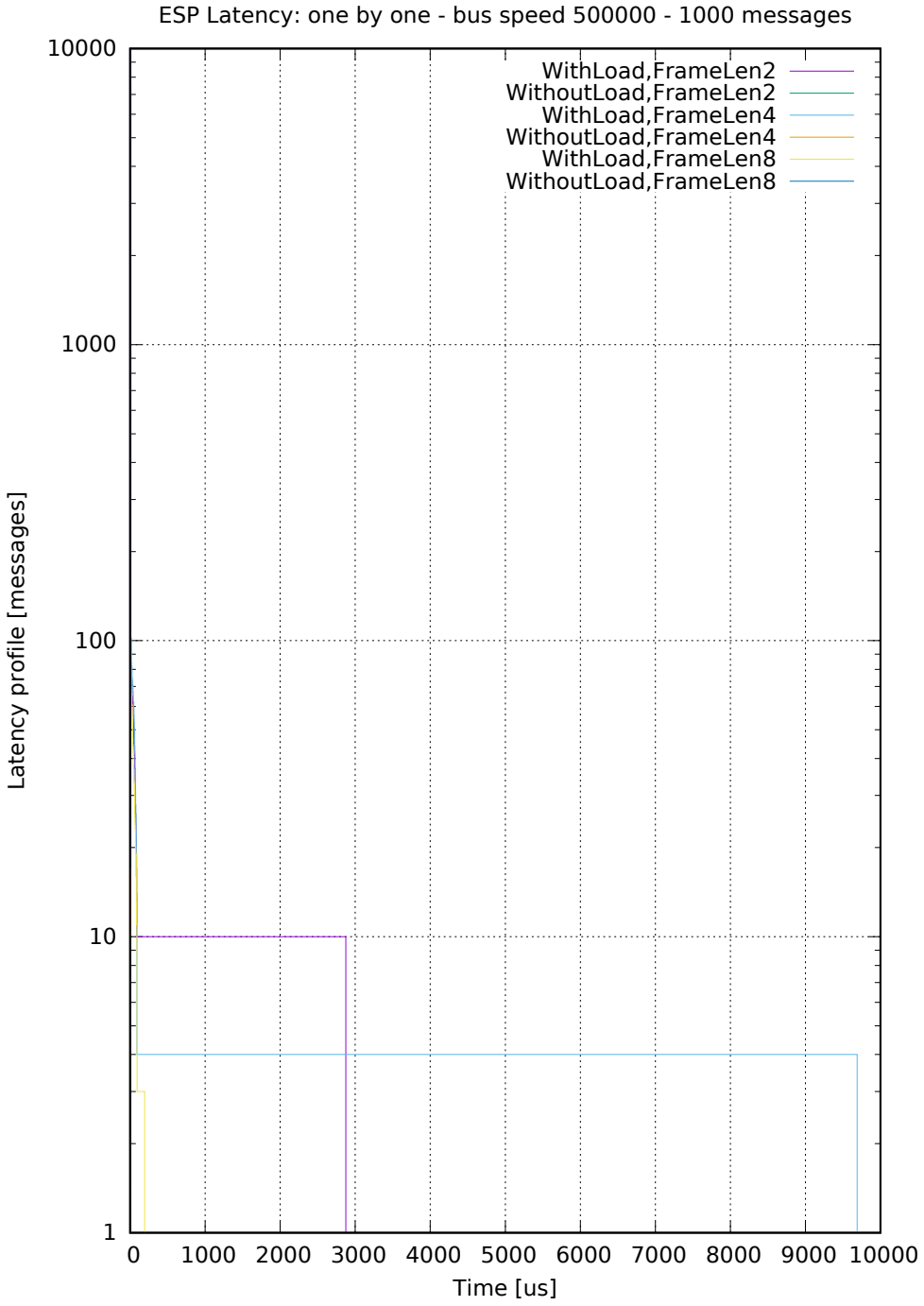


Figure C.14: ESP latency profile: one by one - bus speed 500000 - 1000 messages

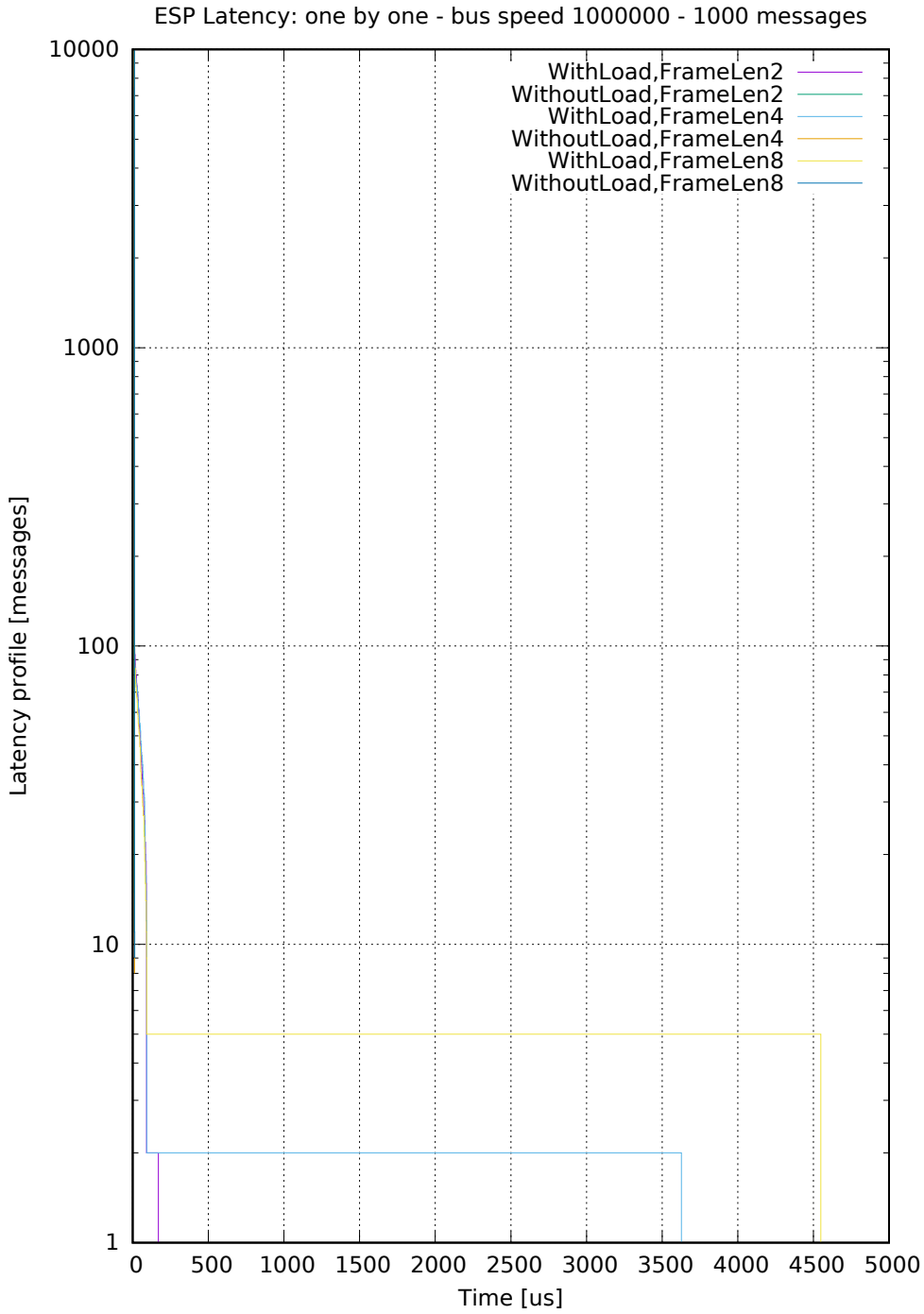


Figure C.15: ESP latency profile: one by one - bus speed 1000000 - 1000 messages

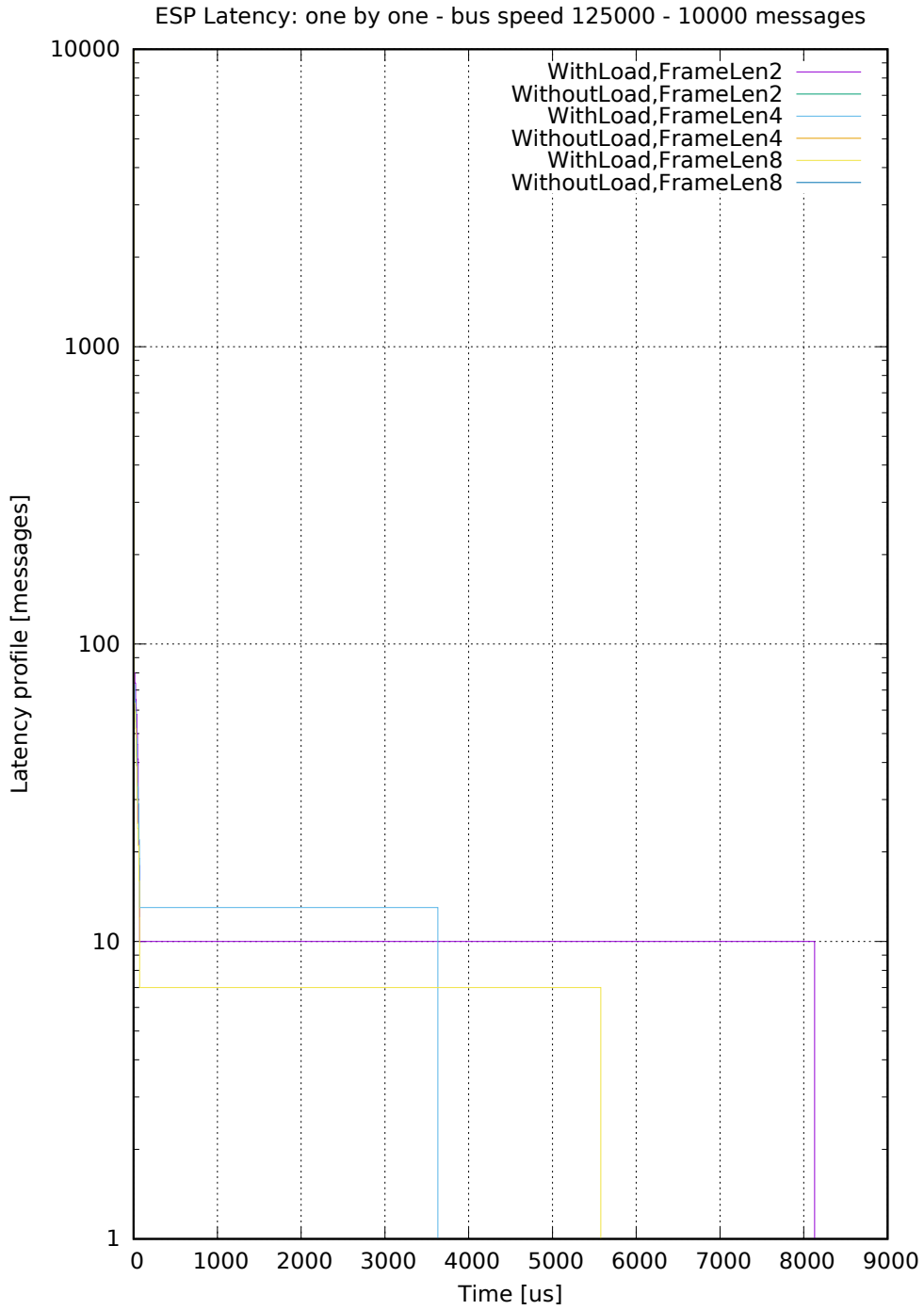


Figure C.16: ESP latency profile: one by one - bus speed 125000 - 10000 messages

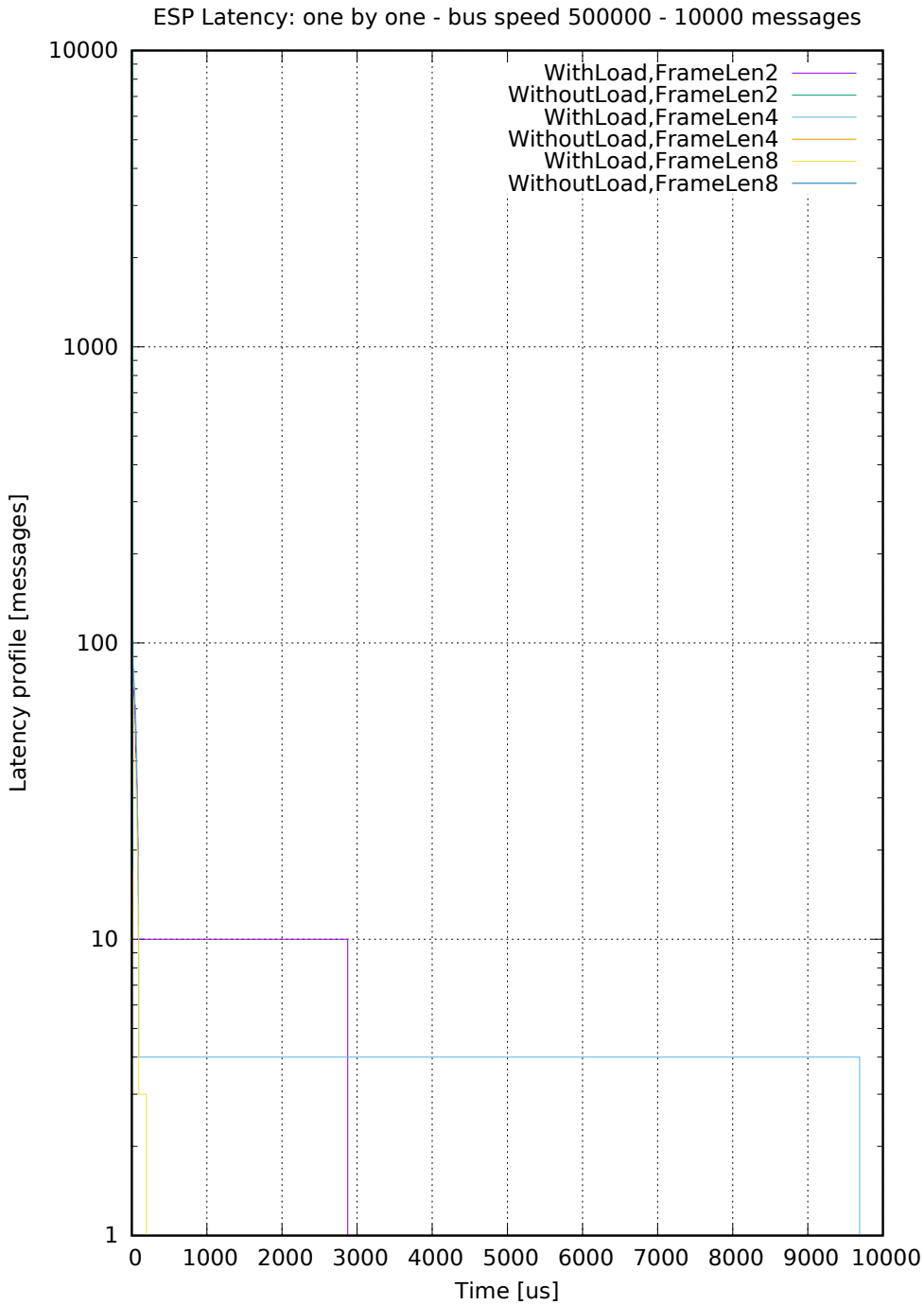


Figure C.17: ESP latency profile: one by one - bus speed 500000 - 10000 messages

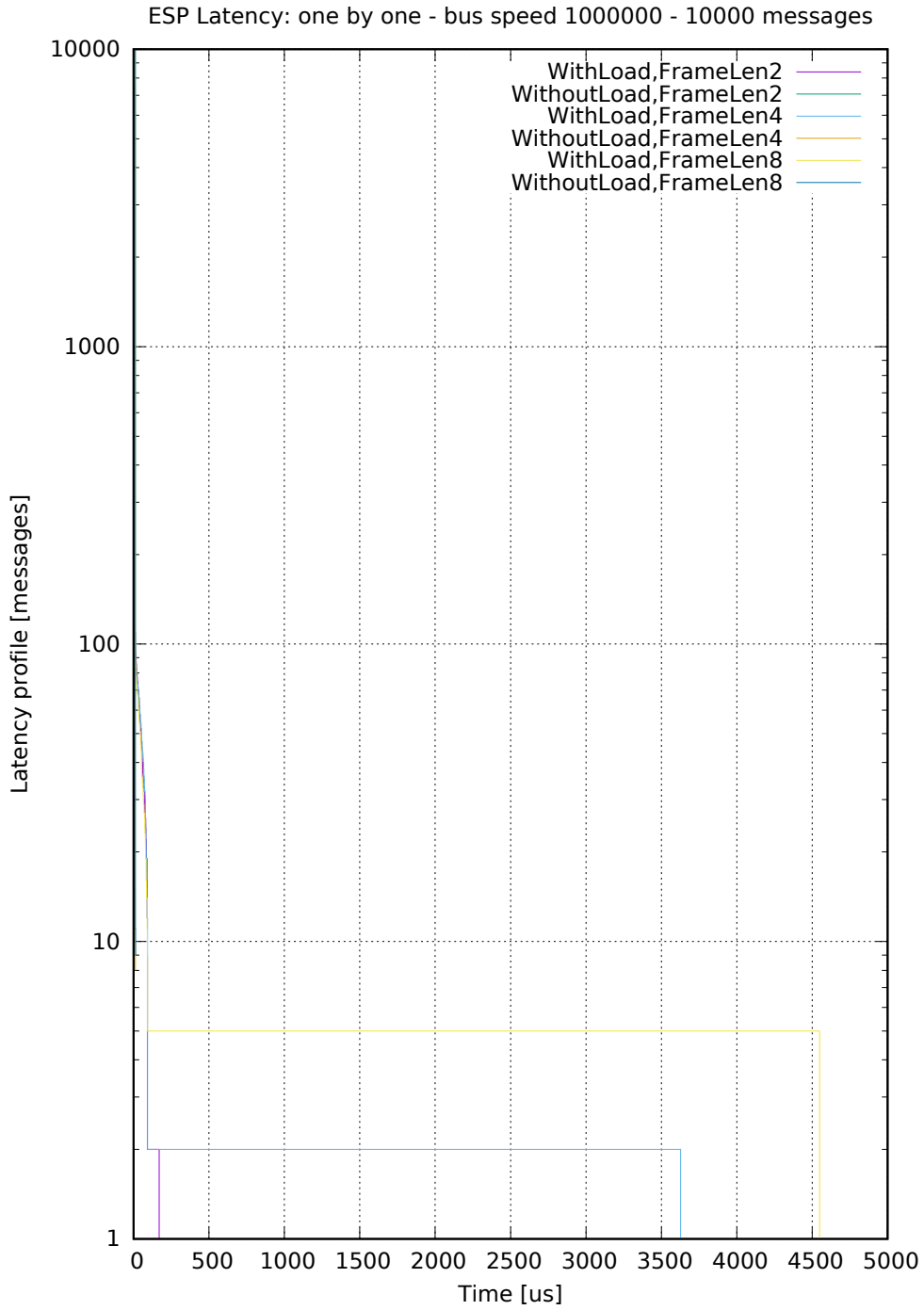


Figure C.18: ESP latency profile: one by one - bus speed 1000000 - 10000 messages

Appendix D

Bibliography

[1] Avnet. Microzed - development board based on the zynq-7000 soc, 2022. URL: https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/microzed/.

[2] Jan Charvat. Nuttx rtos can bus driver for espressif esp32c3. Master's thesis, CTU FEE, Prague, May 2022.

[3] Debian. Emdebian - crossdebootstrap guide, 2022. URL: https://wiki.debian.org/EmDebian/CrossDebootstrap.

[4] Wolfgang Denk. U-boot readme, 2022. URL: https://source.denx.de/u-boot/u-boot/raw/HEAD/README.

[5] DENX Software Engineering. U-boot manual - environment variables, 2022. URL: https://www.denx.de/wiki/DULG/UBootEnvVariables.

[6] Oliver Hartkopp et al. Socketcan linux kernel documentation, 2022. URL: https://www.kernel.org/doc/Documentation/networking/can.txt.

[7] Apache Foundation. About apache nuttx, 2022. URL: https://nuttx.apache.org/docs/latest/introduction/about.html.

[8] Ondřej Ille. Ctu can fd ip core - system architecture document, 2022. URL: https://canbus.pages.fel.cvut.cz/ctucanfd_ip_core/doc/System_Architecture.pdf.

[9] Ondřej Ille. Ctu can fd ip core datasheet, 2022. URL: https://canbus.pages.fel.cvut.cz/ctucanfd_ip_core/doc/Datasheet.pdf.

[10] Martin Jeřábek. Fpga based can bus channels mutual latency tester and evaluation, May 2016. URL: https://rtime.felk.cvut.cz/can/F3-BP-2016-Jerabek-Martin-Jerabek-thesis-2016.pdf.

[11] Martin Jeřábek. Open-source and open-hardware can fd protocol support. Master's thesis, CTU FEE, Prague, January 2019. URL: https://dspace.cvut.cz/bitstream/handle/10467/80366/F3-DP-2019-Jerabek-Martin-Jerabek-thesis-2019-canfd.pdf.

[12] Ichiro Kawazome. Device tree blob overlay configuration file system, 2022. URL: https://github.com/ikwzm/dtbocfg.

[13] Gero Kuhlmann, Martin Mares, Nico Schottelius, Horms, and Chris Novakovic. Linux kernel documentation - mounting the root filesystem via nfs (nfsroot), 2022. URL: https://www.kernel.org/doc/Documentation/filesystems/nfs/nfsroot.txt.

[14] The Linux Kernel Organization. Linux documentation - devm-clk-get function, 2022. URL: https://www.kernel.org/doc/htmldocs/kernel-api/API-devm-clk-get.html.

[15] The Linux Kernel Organization. Linux kernel - timecounter and cyclecounter, 2022. URL: https://elixir.bootlin.com/linux/v5.17.5/source/include/linux/timecounter.h.

[16] The Linux Kernel Organization. Linux kernel documentation - runtime power management framework, 2022. URL: https://www.kernel.org/doc/html/latest/power/runtime_pm.html.

[17] The Linux Kernel Organization. Linux networking documentation - timestamping, 2022. URL: https://www.kernel.org/doc/Documentation/networking/timestamping.txt.

[18] Michal Sojka, Pavel Pisa, Ondrej Spinka, and Zdenek Hanzalek. Measurement automation and result processing in timing analysis of a linux-based can-to-can gateway. In *Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems*, pages 963–968. IEEE, 2011. URL: http://ieeexplore.ieee.org/document/6072917/, doi:10.1109/IDAACS.2011.6072917.

[19] Michal Sojka, Pavel Píša, and Zdeněk Hanzálek. Performance evaluation of linux can-related system calls. In *2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014)*, pages 1–8, 2014. doi:10.1109/WFCS.2014.6837608.

[20] Devicetree Specification. Device tree specification v0.3, 2022. URL: https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.3.

[21] STMicroelectronics. Clock overview, 2022. URL: https://wiki.st.com/stm32mpu/wiki/Clock_overview.

[22] Espressif Systems. Esp32-c3-mini-1 datasheet, 2022. URL: https://www.espressif.com/sites/default/files/documentation/esp32-c3-mini-1_datasheet_en.pdf.

[23] Mike Turquette. Linux documentation - common clock framework, 2022. URL: https://www.kernel.org/doc/Documentation/clk.txt.

[24] Embedded Linux Wiki. Device tree reference, 2022. URL: https://elinux.org/Device_Tree_Reference.

[25] Embedded Linux Wiki. Device tree usage, 2022. URL: https://elinux.org/Device_Tree_Usage.