Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

# CAN Bus Latency Test Automation for Continuous Testing and Evaluation

Pavel Hronek

Supervisor: Ing. Pavel Píša, Ph.D.
Field of study: Cybernetics and Robotics
May 2023

# BACHELOR‘S THESIS ASSIGNMENT

## I. Personal and study details

| | | | |
|---|---|---|---|
| Student's name: | **Hronek Pavel** | Personal ID number: | **499283** |
| Faculty / Institute: | **Faculty of Electrical Engineering** | | |
| Department / Institute: | **Department of Control Engineering** | | |
| Study program: | **Cybernetics and Robotics** | | |

## II. Bachelor’s thesis details

Bachelor’s thesis title in English:

**CAN Bus Latency Test Automation for Continuous Testing and Evaluation**

Bachelor’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) Study and continue work on CAN latency tester and solve its integration with web interface for continuous testing
3) Setup server side and a cabinet with tested systems for unattended testing of actual Linux kernel version CAN performance
4) Document designed system and setup
5) Cooperate with OSADL to integrate solution 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
[2] Jeřábek, M.: Open-source and Open-hardware CAN FD Protocol Support; Masters's thesis, CTU 2018
[3] Ille, O.: CTU CAN FD IP Core; CTU, Project site
https://gitlab.fel.cvut.cz/canbus/ctucanfd_ip_core
[4] Ille, O.; Novák, J.; Píša, P.; Vasilevski, M.: CAN FD open-source IP core, In: CAN Newsletter 3/2022, PDF, CAN in Automation, 2022

|  |
|---|
| Name and workplace of bachelor’s thesis supervisor: |
| **Ing. Pavel Píša, Ph.D.   Department of Control Engineering  FEE** |
| Name and workplace of second bachelor’s thesis supervisor or consultant: |
|  |

Date of bachelor’s thesis assignment: **18.01.2023**   Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

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

## III. Assignment receipt

The student acknowledges that the bachelor’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 bachelor’s thesis, the author must state the names of consultants and include a list of references.

|  |  |
|---|---|
| . <br>Date of assignment receipt | Student’s signature |

## Acknowledgements

I would like to thank my supervisor, Ing. Pavel Píša, Ph.D., for his guidance, thorough advice and feedback throughout the work, and Ing. Matěj Vasilevski for his assistance in the beginning as I took over the project from him. I would also like to thank my parents for their support and care during my studies and all the people who encouraged me with kind words as I was working on this thesis.

## 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, on 26. May 2023

## Abstract

The goal of this thesis is to automate continuous latency testing on the CAN bus for the purpose of ensuring reliability of real-time applications. It is based on the existing implementation of *can-latester* software, which measures the time required to forward a CAN frame by a gateway. This thesis focuses on designing and building a system to autonomously perform this measurement on a gateway running on a Linux system with and without real-time patches and record the results.

The system is assembled in a 19" data rack cabinet and runs daily tests on the latest version of the Linux kernel. The results of individual measurements and long time trends are visualized in a web interface. Everything is designed with the intent to enable integration into a real-time system QA farm.

**Keywords:** CAN bus, latency testing, CTU CAN FD, Linux, automation, continuous monitoring, real-time processing

**Supervisor:** Ing. Pavel Píša, Ph.D.

## Abstrakt

Cílem této práce je automatizovat průběžné sledování latencí na sběrnici CAN pro účely zajištění spolehlivosti v aplikacích reálného času. Vychází se z existující implementace softwaru *can-latester*, který měří dobu potřebnou pro zpracování a přeposlání rámce CAN bránou. Obsahem této práce je navrhnout a sestavit systém, který bude samostatně provádět tato měření na bráně běžící na Linuxovém systému s real-time úpravami i bez nich a zaznamenávat výsledky.

Výsledný systém je sestaven ve skříňce 19palcového datového rozvaděče, kde každý den provádí testy na aktuální verzi Linuxového jádra. Výsledky z jednotlivých měření jsou spolu s dlouhodobými trendy vizualizovány prostřednictvím webového rozhraní. Vše je navrženo tak, aby systém bylo možné integrovat do QA farmy systémů reálného času.

**Klíčová slova:** sběrnice CAN, testování latencí, CTU CAN FD, Linux, automatizace, průběžné sledování, zpracování v reálném čase

**Překlad názvu:** Automatizace průběžného testování latencí na sběrnici CAN

# Contents

# Figures

# Tables

# Chapter 1

# Introduction

This thesis deals with software and setup for latency testing automation on the CAN bus. The purpose is to provide quality assurance for real-time applications on operating systems that go through continuous development. This assurance is vital in applications that depend on consistent timing to function correctly, and any regression in latency could lead to unpredictable behavior. The system designed as part of this thesis will help spot such regressions early and track latency performance over time. In this thesis, the subject of testing is the Linux kernel.

The CAN bus is a dominant technology in the field of industrial networks, automotive and embedded applications. In many of them, it is beneficial to use systems based on the Linux kernel, which is very versatile and has excellent support thanks to being widely used. To ensure reliability of using CAN on Linux in time-sensitive applications, testing and monitoring of CAN frame processing latencies is necessary throughout the development. An ideal application to conduct this testing on is a CAN gateway, which simply forwards frames between two buses. For more elaborate reasoning, see section 1.1.

The first part of the work focuses on designing a system that will facilitate this testing. The software that is used for the measurement of a gateway's latency (*can-latester*) was the subject of previous works [1][2]. In this thesis, these works are followed up on by setting up a server that is used to schedule and automate continuous testing, as well as assembling and configuring a physical testbed with the hardware necessary for the testing. The hardware and principle of measurement are described in chapter 2.

The rest of the work is concerned with the software and setup for testing automation. The configuration of the testbed devices, how they boot, and how they are managed is all documented in the first two sections of chapter 3. The later sections explain the testing process, the software written to conduct it, and how it functions. In short, the final system runs daily tests on the latest available version of the mainline Linux kernel as well as a version with the newest batch of RT patches applied. Then it processes the results for visualization.

The final chapter 4 describes how the data acquired during the testing are presented via an interactive website.

## 1.1 CAN, Linux, and Latency Testing Motivation

The Control Area Network (CAN) bus is popular mainly in the automotive industry, where it is used to facilitate communication between various subsystems of a vehicle, and in many industrial and embedded applications. The specification covers the physical and link layer. The physical bus can be as simple as a single twisted-pair cable terminated at both ends. Data are transmitted as a differential signal, providing increased noise resistance. Bus arbitration is based on the priority of frames being sent and collisions are resolved in a non-destructive manner: the higher priority frame goes through and the sender of the lower priority frame waits until the medium becomes available again. The standard also covers error detection and recovery. All of this makes CAN both highly flexible and reliable. An update to the CAN standards introduces an extension, CAN FD, that allows a larger payload per frame (64 bytes, up from 8 bytes) and a higher bitrate during data transfer phase. Further information about CAN can be found at [3].

Linux is a general-purpose operating system kernel that is used in a variety of applications spanning from high-performance servers, desktops, phones (Android) to embedded applications in various industries, including automotive. As such, it often needs to access the CAN bus to collect data or even control other devices in real time. For this purpose, a CAN bus support was implemented for the kernel. The LinCAN project was one of the first solutions providing CAN bus interface drivers for the Linux kernel and allowing them to be interacted with via a character device. It had been used in industrial control applications for many years [4]. Later the SocketCAN subsystem was developed in collaboration with Volkswagen and is now part of the kernel. It uses the socket API similarly to how TCP/IP operating system services are accessible to user-space applications (`socket`, `send`, `recv`, `close`) [5]. An interface analogous to SocketCAN has even been adopted[1] into the open-source real-time operating system NuttX.

An effort is being made to make Linux capable of handling real-time tasks, originating over 20 years ago. A set of patches is being maintained (some of which have now been merged into the mainline) that introduce preemptible locking mechanisms to the kernel and strive to reduce the amount of non-preemptible code as much as possible. These changes are often referred to as PREEMPT_RT and are meant to make the maximum response time of tasks with real-time priority more deterministic [6].

As stated at the beginning, latency and strict timing properties are crucial in many applications. To ensure reliability of such applications, these properties need to be monitored throughout the development process of all components. One prominent example of this monitoring is the OSADL QA farm. They run stress tests on the PREEMPT_RT-patched Linux kernel in an effort to spot unsatisfying system latencies, crashes, and similar misbehavior early in the development of the next kernel version.[7] The system developed as part

[1]see `https://github.com/apache/nuttx/pull/1238`

of this thesis aims to provide similar testing and QA for applications utilizing CAN on Linux.

A CAN gateway is a ubiquitous device in CAN networks. It simply forwards chosen frames from one bus to another, providing isolation of critical systems and adding resilience in case of failure on one of them. It runs no complicated control algorithms that would cause a variance in latencies; at most, it performs some simple filtering. A gateway implementation is even a part of the Linux CAN subsystem. As such, gateways are very suitable for having their latencies tested.

# Chapter 2
# Measurement Setup

This chapter describes the physical setup needed in order to measure latency of a CAN gateway, as well as the measurement process itself.

A gateway is a simple device whose function is to forward CAN frames from one bus to another. On Linux, there is a gateway implementation that is part of the CAN subsystem in the kernel and can be controlled via the *cangw* utility. To test what gateway performance can be achieved in user space, a program called *ugw* can be used, which is part of the *can-latester* project[1].

To measure the performance of a gateway device, one has to use another CAN-capable device connected to the same two buses as the gateway. This device will then send a long sequence of frames to the gateway and measure the time between when each frame is sent and when it appears on the second bus to determine the typical performance of the gateway with some reasonable certainty. One program that can be used to perform this measurement from Linux is called the CAN Latency Tester (*can-latester*[1]). The principle of its functioning is described in section 2.3.

## 2.1 Used Components

This proposed setup requires two devices equipped with at least two CAN interfaces. The specified goal is testing of Linux system based gateway, so it is natural to use the same system for frame generation and timestamp recording as well. The devices chosen for this project are two Xilinx Zynq MicroZed SBC-based educational kits called MicroZed APO (MZAPO for short), which were available at the department. They come with programmable logic (PL/FPGA) that can realize logic designs of choice, CAN FD controllers in the described case. Two CAN buses are exposed via a DE-9 connector on each board, which allows the boards to be connected with a regular serial cable. [8]

The MZAPO boards are a combination of a MicroZed development board and a custom IO expansion board. Their Xilinx Zynq-7000 SoCs are equipped with two ARM Cortex A9 cores and Xilinx's programmable logic [8]. The

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

boards can run mainline Linux with just a few patches that add an appropriate device tree[2].

For the CAN controller, the CTU CAN FD was chosen. It is an open-source soft core developed at CTU FEE and has driver support in the Linux kernel. It also allows precise hardware timestamping, which is essential for this use case [9]. The specific design[3] used in the FPGA contains 4 CTU CAN FD controllers and connects the 2 XCAN controllers available directly on the SoC silicon into programmable logic. It also includes a crossbar switch, which dynamically connects the controllers to internal lines and the lines to external buses from within the operating system.

Only the CTU CAN FD controllers are used for the measurement. On the measuring side, two controllers (can0 and can1) are connected to internal line 1 and another one (can2) to line 2. Line 1 is connected to bus A, line 2 is connected to bus B. The need for two controllers on bus A comes from the principle of the measurement (section 2.3). On the device under test, one controller is connected to line 1 and another to line 2. The lines are in turn connected to physical buses A and B, which connect both devices through a cable. The connection of the boards is schematically depicted in figure 2.1.

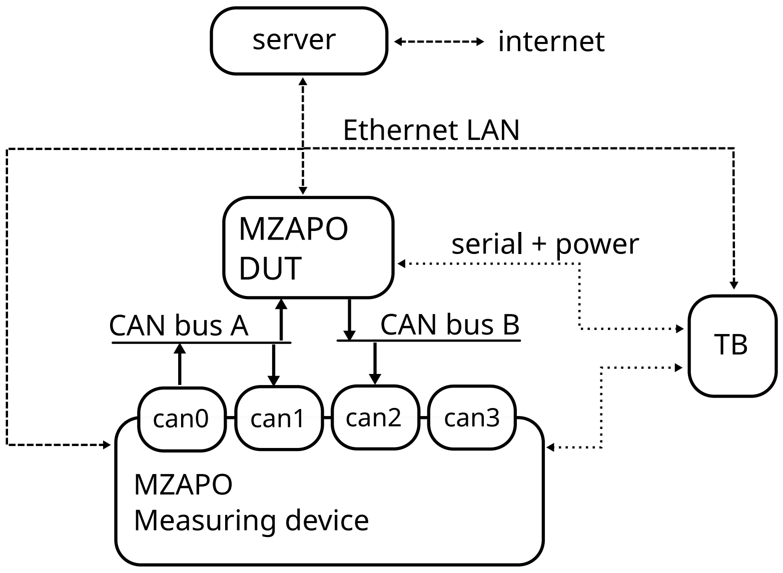


**Figure 2.1:** Diagram of the setup, adapted from [2]

The operation of both devices needs to be coordinated during measurement and the results need to be stored and processed. For this purpose, a server was added to the setup and is connected to the devices via Ethernet. The advantage of using an additional device rather than doing these tasks from the measuring device is that a server has way more CPU power and larger storage, which is useful when building new versions of the Linux kernel. It

[2]`https://github.com/ppisa/zynq-rt-utils-and-builds/tree/master/projects/linux/patches`

[3]`https://gitlab.fel.cvut.cz/canbus/zynq/zynq-can-sja1000-top/-/tree/mz_apo-2x-xcan-4x-ctu`

can also directly host the root filesystem, which the MZAPO boards mount over NFS, and provide them a TFTP server with kernel images, which are loaded during boot up. This makes it easier to manage them – e.g., there is no more need to take out the SD card to revert a new kernel build that no longer boots.

The measuring device and the device under test (DUT) are stressed by the measurement, which can lead to crashes and freezes when some problem is uncovered. In order to recover from such a situation automatically, it is helpful to have a serial (over USB) connection to the boards, as the MZAPO has a feature that allows it to be reset by a 2 second long break signal on the serial console port. The server, however, does not necessarily have to be in the same location as the boards, making direct connection impractical. This was solved by adding another testbed control device (TB) – a combination of the ARM64 OrangePi Zero Plus board and an expansion board designed for hardware testing by Ing. Petr Porazil of PiKRON. Beside a USB hub for connecting to the MZAPO boards, it has power headers with the ability to control power to them and turn the other two boards on or off remotely. [10] It also means that all three devices can be powered from a single power supply. The TB is connected to the same Ethernet LAN as the other devices.

## 2.2 Final assembly

The final setup was assembled into a 19" data rack cabinet, as seen in picture 2.2. The cabinet also hosts a network switch, which connects all devices into one Ethernet LAN. One additional Ethernet cable connects to the server running on central server virtualization technology elsewhere in the building. A single power supply enters the cabinet and powers the Orange Pi testbed control device, which in turn powers the two MZAPO boards. The boards are connected via serial over USB to the TB and also via CAN over serial cable to each other.

## 2.3 Principle of Measurement

The measurement itself is carried out by the *can-latester* software. The device under test can be used in either regular gateway mode (retransmit on a different bus) or single interface mode. The single interface mode is useful to test CAN frame processing latencies of systems equipped with only one CAN interface, like the ESP32. In this case, the DUT retransmits each frame onto the same bus, just with a higher priority.

In gateway mode it works by sending a CAN frame on bus A from its first CAN interface (can0). The frame is then received by can1 on the same device, as well as by the second device, which is set up to operate as a gateway. The gateway forwards the frame onto bus B, where it is received by the can2 interface of the first device. The time it takes to forward the frame is influenced by the operating system's task scheduling and is the main

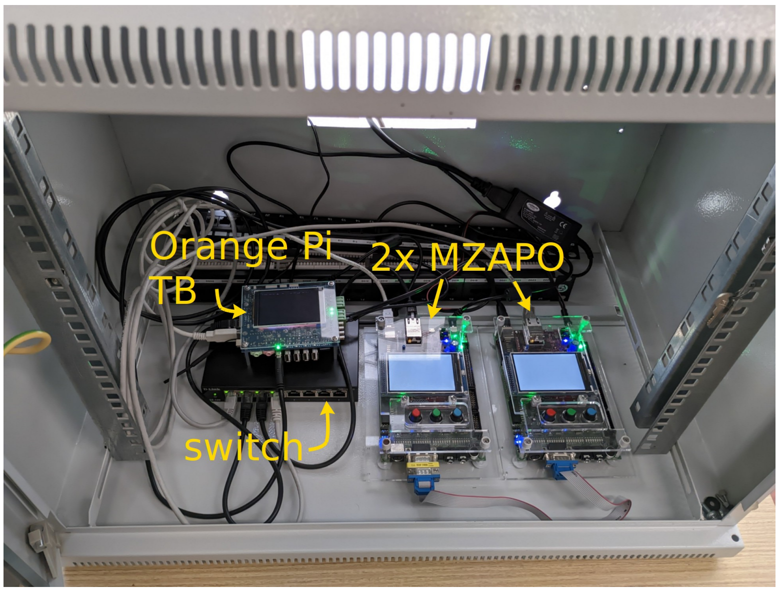


**Figure 2.2:** The final assembly in a cabinet

source of variance in latencies that are being measured. The measuring device reads the timestamps of when the frame was received by can1 and can2 from the hardware and computes the latency. Using the timestamp from can1 ensures we know the precise time instant of when the frame actually appears on DUT's incoming bus. The computed latency is stored in a histogram structure – an array of bins (for every 0.001 ms up to 5000 ms) containing the number of frames whose latency falls into each bin. After a set number of frames sent and received (typically 10,000), the histogram of measured latencies is saved in a text file for later processing. [2]

It is possible to send the frames in multiple modes. Periodically (with a fixed interval), one at a time (wait for the previous to return), or as fast as possible (flood mode). The automated tests can be performed in any mode.

Single interface DUT is not suited to be used in flood mode since it has to wait for the medium to become available (end of the current frame), which affects the latency slightly. On the other hand, the measuring device cannot send a frame while the DUT is transmitting, meaning that the DUT cannot be fully saturated and the results are not representative of what the performance would be under full load.

The timestamps returned by the two receiving interfaces (can1 and can2) on the measuring device might not be consistent with one another because the two controllers don't have to start their time counters at the same time. Therefore synchronization of timestamps must be performed before the measurement. To do so, all interfaces are temporarily connected by the CAN crossbar switch when a single frame is sent. The frame is received by can1 and can2 at exactly the same time, so the difference in timestamps can be used to correct future measurements. More detailed documentation can be found in [2].

# Chapter 3
# Automation of Testing

This chapter focuses on the device setup and software necessary to automate latency testing of a Linux-based CAN gateway. As stated in previous chapters, two MZAPO boards (measuring device and device under test) and a server is used. The server needs to build the latest version of Linux, which will be subject to the test. It currently builds two variants; one version is the latest state of the master branch, the second one is the current development version including PREEMPT_RT real-time modifications. The DUT then boots each kernel build and runs a CAN gateway. The measuring device runs tests on the gateway by generating CAN traffic and measuring the latency of the gateway under various conditions. The server collects these results and processes them for being displayed by the web interface described in chapter 4. The scripts that handle the automation and the website source code have been published in a git repository[1].

## 3.1 MZAPO Boards Setup

To simplify management of the boards, they have been set up to boot from the network and have their filesystem hosted on the server. The boards use the U-Boot bootloader, which can be controlled by the `uEnv.txt` file on the boards' boot partition of the SD card. The boot command (`uenvcmd`) variable and other options can be configured to control the boot process. [11] The following command can be used to acquire an IP address for the board as well as the address of the server via DHCP, download a boot script (`autoscr.scr`) from the server via TFTP (located by `${bootscript_path}` variable) and execute the boot script:

```
dhcp && tftpboot ${bootscript_addr} ${serverip}:${
    ↪ bootscript_path} && source ${bootscript_addr}
```

The downloaded boot script then actually downloads the boot image and loads the kernel. The startup commands sequence, kernel image used, and root filesystem location can therefore be altered without touching the SD card.

[1] `https://gitlab.fel.cvut.cz/canbus/can-benchmark/can-latester-automation`

The measuring device needs to boot a different kernel than the DUT. This is so that the kernel of the DUT is the main variable that influences the results. The measuring device's kernel also includes patches[2] adding precise timestamping support to the CTU CAN FD driver, while DUT's kernel should be unmodified. The measuring device therefore needs to download a different boot image containing a version of the kernel that will not change every day. This can be achieved by putting some logic into the boot script. Currently it checks the MAC address of the device and based on that chooses which boot image to download. IP address could also be used for this decision.

After the kernel is loaded, bootup continues by running the init process as specified by kernel command line argument `init=`, in this case `/sbin/init-overlay`[3]. This script mounts a read-only NFS share (which is the same for both devices) and constructs a combined root filesystem with a writable *tmpfs* overlay on top of it. After that it hands over to `/sbin/init` and the boot process continues regularly.

The kernel command line arguments are passed to the kernel by U-Boot, which reads them either from `uEnv.txt` or they are set by the the `autoscr.scr` boot script as `bootargs` environment variable. The complete list of parameters used is the following:

```
console=ttyPS0,115200 clocksource=ttc_clocksource ip
   ↪ =192.168.0.10 root=/dev/nfs ro nfsroot=192.168.0.1:/srv
   ↪ /nfs/debian-armhf init=/sbin/init-overlay
```

The `console` argument redirects serial output to a USB type B port on the expansion board rather than using the default USB micro-B on the MicroZed board. This port also supports resetting the system via break signal on the serial line. [8] The `clocksource` argument is passed because the default clocksource on older versions of Linux (like 4.19) does not support frequency scaling. The remaining arguments are required to boot from an NFS filesystem.

Some further initialization needs to happen before testing can begin. The FPGA needs to be programmed with the correct design, the CAN crossbar switch needs to be configured on each device, and the CAN interfaces need to be brought up and have their `txqueuelen` increased (to prevent `ENOBUF` errors which *latester* and *ugw* don't handle). Additionally, the irq threads of the measuring device's receiving CAN interfaces need to have a higher priority than the transmitting interface, otherwise the system sometimes was not reading out frames quickly enough, which led to buffer overflows. This all was automated in a script called `init-device.sh`, which is included in the repository[1]. The script is registered as a *systemd* unit and runs as part of the boot process once the `multi-user.target` is reached. The systemd unit is located at `/etc/systemd/system/init-device.service` and looks something like this:

[2]`https://gitlab.fel.cvut.cz/vasilmat/linux-can-test/-/commits/net-next/ctucanfd-timestamps-v5-with-config-pm-squashed/`

[3]`https://github.com/ppisa/rpi-utils`

```
[Unit]
Description=Device initializer
[Service]
ExecStart=/usr/src/can-automation/device-scripts/init-device.sh
[Install]
WantedBy=multi-user.target
```

It also needs to be enabled by `systemctl enable init-device.service`.

The process to update the FPGA is the following. We need a file containing the synthesized FPGA design (firmware) and a device tree script that describes the devices in the design to the operating system. The design file needs to be put under `/lib/firmware/`. The device tree script is compiled into a device tree overlay (dtbo) using the *dtc* tool (part of the Linux kernel repository). The dtbo is then loaded using *dtbocfg* kernel module. The module first needs to be loaded (`modprobe dtbocfg`), then we can create a directory under `/sys/kernel/config/device-tree/overlays`, for example `my-overlay`. Inside this new directory we copy the dtbo file named as `dtbo` and activate the overlay by writing `1` into `status` file in the same directory. The correct firmware is automatically loaded into the FPGA as specified in the device tree script (e.g. `firmware-name = "system.bit.bin";`). [12]

### 3.1.1 CAN Crossbar Driver

The CAN crossbar switch is a part of the design used in the FPGA that allows connecting the CAN controllers to external buses from within the operating system. Without it the connections could only be changed by reprogramming the FPGA. Up until now the switch had to be controlled by mapping and reading/writing its memory in user-space. A simple driver that exposes the register through the filesystem was written to avoid this. It registers itself as a platform driver, so it gets loaded by the kernel automatically whenever a design having the crossbar in its corresponding device tree overlay is programmed into the FPGA. The source code of the driver is available in a public repository[4].

When loaded, the driver registers a *misc* device to create `/dev/ctucancrossbar`, which allows reading and writing the entire 32-bit register as a hex number (e.g., `0x01000000`). The individual flags of the register are exposed as device attributes in *sysfs*. The attributes are `line1` to `line4` (value 0 or 1 whether to enable output from `lineX` to an external bus), `bus1` to `bus4` (values 0 to 3 to specify which of `line1` to `line4` is connected to `busX`) and `ctrl1` to `ctrl8` (values 0 to 3 to specify which line controller `ctrlX` should be connected to). The layout of the register is depicted in figure 3.1; see page 45 of [13] for more details on the values in the register.

[4] `https://gitlab.fel.cvut.cz/canbus/can-benchmark/can-crossbar-driver`

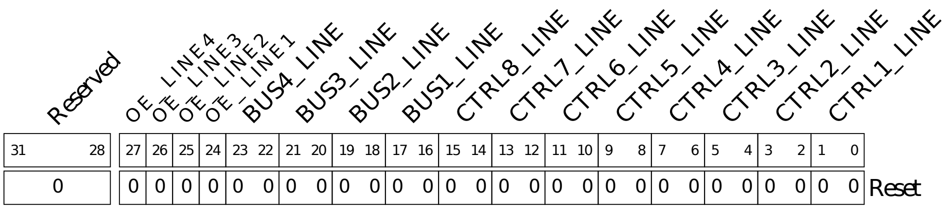


**Figure 3.1:** CAN crossbar switch register layout [13]

## 3.2 Server Setup

A server is used to coordinate the testing process. Currently it is an x86-64 virtual machine with 4 CPU cores running Debian as its operating system. It provides the boards with a TFTP server containing boot files as well as an NFS server that hosts their read-only root filesystem. This allows the server to have complete control over the boards' files as well as allowing the boards to be reset without risking filesystem corruption.

To provide the TFPT server, a package like `tftpd-hpa` needs to be installed. It serves the `/srv/tftp/` directory by default. Support for an NFS server is provided by the `nfs-kernel-server` package. It is configured to serve the `/srv/nfs/debian-armhf` directory, which contains the root filesystem. The boards' operating system was installed into the NFS share with *debootstrap*. This is described in more detail in [2].

Sometimes it might be needed to run programs from the target device's installation on the server as if they were run directly on the device – for example to install packages. This is possible by "changing root" (*chroot*) to the device's filesystem. As the boards use the ARM architecture, we need an x86-64 version of `qemu-user-static` in the chroot environment to be able to execute ARM binaries. Ideally, the package would be extracted into the root directory before installing the operating system.

To avoid having to hard-code the IP address of the TFTP sever on each board individually, this information can be provided by the DHCP server. In the configuration file for *dnsmasq* under `/etc/dnsmasq.d/`, the appropriate lines look like this:

```
dhcp-host=00:0a:35:00:22:01,set:mach_mzapo,192.168.0.10,mzapo0
dhcp-host=00:0a:35:00:22:04,set:mach_mzapo,192.168.0.11,mzapo1
dhcp-boot=tag:mach_mzapo,/zynq/autoscr.scr,srvname,192.168.0.1
```

The first two lines assign a static IP address to the boards and tag them as `mach_mzapo`. The third line says that the DHCP server should advertise to `mach_mzapo` devices that they can boot from file `/zynq/autoscr.scr` on `192.168.0.1`. The boards and server are the only members of their own VLAN, therefore the server can run a DHCP server configured as needed.

## 3.3 The Testing Process

A set of scripts was written to automate the testing process. They are currently hosted in the repository[1] mentioned at the beginning of the chapter. That repository is cloned on the server and the scripts are run from there. These scripts handle the tasks of compiling Linux, booting it on the DUT, executing a range of tests, doing the same for RT patched Linux and finally processing the results.

The main script is called `run_daily_tests.py`. It contains most of the logic regarding *which* tests to run. The script takes several command line arguments. `--skip-build` can be used to skip the compilation and rebooting in case the DUT is already running the intended version. `--branches` specifies which Linux kernel branches should be tested. Currently it accepts `master` and `rt`, but more can be added through a config file. All branches are tested when the argument is omitted. Finally `--conf` can instruct the script to use a different config file than default and `-q` suppresses output from *make* to decrease the size of the resulting log file.

Some tasks are delegated to utility scripts. `build_linux.py` is used to compile the version of Linux needed at the moment. A script called `run_test.py` executes a single test with selected configuration. It prepares the DUT as specified by the configuration, runs *latester* and fetches the results. The downloaded histogram file is converted to JSON using `hist_to_json.py`. `process_json_dir.py` finally takes all the JSON files of past individual tests, groups them by the test configuration and merges them into one large file per group. These large files are then served together with the website. Most of these utility scripts can also be run from the command line and used independently, although the main script simply imports them as modules and calls their functions. The option for manual running of scripts was useful during development when some phase of the testing failed.

An additional script called `build_web.py` is part of the repository. It only needs to be executed manually when some configuration changes. Its function is described in chapter 4.

All scripts use the `conf.json` config file, which by default is located at `/var/lib/latester/conf.json`. The file is used to parametrize some parts of the process. For example, the directories where results are stored, where the website root is located, IP addresses of the devices doing the testing, which commands to run in certain situations, and more. The fields contained in the file will be referenced further in the text, e.g., `SOME_FIELD` or `{SOME_FIELD}` when part of a formatted string. The fields are listed and explained in appendix A.

The testing begins every day at 4:00 AM when the server's *cron* executes `run_daily_tests`, a simple shell wrapper for `run_daily_tests.py` that redirects its output to a log file and stores its return code. The following subsections describe the process in detail.

### 3.3.1 Building Linux

The first step of the process is to build the latest version of the Linux kernel that will run on the DUT. The compilation and installation are delegated to a utility script called `build_linux.py`, which can also be run manually from the terminal to perform the build in the current working directory. The remaining parameters are read from `conf.json`.

The server maintains a clone of the mainline Linux repository, pulls the latest changes, checks out the required branch and builds it. This happens two times in our case: for the `master` branch and for the latest RT branch. Branches of Linux versions with RT patches are hosted at the rt-devel repository[5]. This repository also contains a branch called `for-kbuild-bot/current-stable` which conveniently tracks the latest changes and is the one used by the script.

The build outputs are placed in a separate directory (`BUILD_DIR` field in `conf.json`) to avoid cluttering the source tree (`SOURCE_DIR`). It is completely deleted before every build since we switch between two significantly different branches and incremental build brought no speed benefit in this case – it still took about 20 minutes. The solution for optimal speed would be to have two build directories, although at the cost of disk space. The build directory is then recreated to contain only two files: a customized *Makefile* override based on the one in MZAPO kernel config repository[6] (named *GNUmakefile*, which in *GNU make* takes precedence) and a *.config* appropriate for the branch being built (RT config has `PREEMPT_RT` enabled). These files are stored in a separate location known from the `conf.json` file and the script simply copies them to `BUILD_DIR`.

The purpose of the *Makefile* override is to simplify the call to *make* by injecting some constant parameters. It tells it where to put build outputs with `O=` and that it is supposed to cross-compile with a specific toolchain using `ARCH=arm` and `CROSS_COMPILE=arm-linux-gnueabihf-`. On top of that, we add `DTC_FLAGS=-@` to enable overlay support on all device tree blobs.

An alternative to having entirely different .config files for each branch could be having a set of .config fragments that are applied over a base .config file or even the default (*defconfig*) configuration. This is possible with the `merge_config.sh` tool, which is part of the Linux kernel repository. The current solution works well for the time being, but this should be considered in the future for the sake of robustness of the system.

On kernel versions prior to 5.19, it was necessary to patch the kernel to include the CTU CAN FD driver, but since it is now included in the mainline, this is no longer the case. It however still needs to be enabled in the `.config` by setting `CONFIG_CAN_CTUCANFD=m` and `CONFIG_CAN_CTUCANFD_PLATFORM=m`.

Before the build itself, the script calls `make olddefconfig` to update the kernel *.config* with default values in case new options were added. Then it executes `make` with the `-j 4` parameter to parallelize the build onto the 4 CPU

[5] `https://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-rt-devel.git/`

[6] `https://github.com/ppisa/zynq-rt-utils-and-builds/blob/master/projects/linux/build/arm/zynq/GNUmakefile`

cores available on the server as well as with a specified `LOCALVERSION`. When called from the main script, the `LOCALVERSION` is set as `-dut` to distinguish daily builds from ones that were run manually. This is useful in the next step, which is cleaning up old modules. The script lists the directories under `INSTALL_DIR/lib/modules` and deletes all `-dut` versions that don't match the version that was just built and are older than 2 days.

The newly built modules are installed to the target filesystem by running `make INSTALL_MOD_PATH={INSTALL_DIR} modules_install`. Files `zImage` and `zynq-microzed-uart0.dtb` that were produced by the build are then copied to `BUILD_OUT_DIR` to be archived (version suffix is appended to their file name). The automated builds use constant *autobuild* suffix to avoid filling up the disk. Subsequently, the kernel and device tree are packaged into an image for U-Boot which is served by the TFTP server. Additionally, the *dtbocfg* [12] module is built against the current kernel and installed. This module is used to program the FPGA and update the device tree from user space. Finally, the DUT is rebooted to load the new kernel by sending a 2.5 second long break signal to its serial port via the Orange Pi testbed control and monitoring device.

The `run_daily_tests.py` script waits for the DUT to boot up by trying to open a connection to port 22 (SSH). If a connection cannot be established after 120 tries, it gives up and exits the scripts.

### 3.3.2 Running Tests

A series of tests is run for the master and RT branches of the Linux kernel. The tests are run in multiple configurations to evaluate performance under different conditions, such as DUT being under CPU and IO load, frames being sent in flood mode, or using the *ugw* gateway as opposed to the kernel gateway. One test corresponds to one run of *latester*, which measures the gateway's latency on 10,000 frames. The results are recorded in a histogram.

Each test is performed by `run_test.py`, which can again be called both as a function and from the command line. The test configuration is determined by a set of options that can be turned on or off. When testing a kernel from the master branch, the options that make sense are `flood`, `kern`, `stress` and `fd`. When testing an RT kernel, there is an additional `rt` option that does nothing on master. These options are passed in a list as the `args` parameter or on the command line. The meaning of these options is explained later in this section. The last command line argument is the kernel version identifier which should be used in the names of output files.

Once the DUT boots up, `run_daily_tests` sets its CPU frequency scaling governor to `performance` so the CPU is not throttled during the test. Right after that, `run_test` is called for every combination of test options appropriate for the branch currently being tested.

`run_test` connects to the gateway and measuring device using Fabric[7],

[7]`https://www.fabfile.org/`

a higher level API to the Paramiko[8] library, which implements SSH in Python. Then it proceeds along these steps:

- If the `kern` argument is specified, it configures the kernel-space gateway on DUT via the *cangw* utility from *can-utils* by running `cangw -A -s can3 -d can2` to forward frames from one interface to the other. Otherwise it launches the user-space gateway (*ugw*), which is a part of the latester project: `ugw can3 can2`. If `fd` is specified, it adds the `-f` argument to *ugw* and `-X` argument to *cangw* to indicate the gateways should expect CAN FD frames. The commands can be set using options `UGW_CMD` and `CGW_CMD` in the conf file (or `UGW_FD_CMD` and `CGW_FD_CMD` respectively). These configuration options can be used to customize which interfaces are used, the path where to find the executables, and the tx and rx method *ugw* should use, as this can also affect resulting latencies. The performance of these methods (such as using `read`/`write` system calls or `send`/`recvmmsg`) was evaluated in [14]. The user-space gateway is always run with real-time priority: `chrt -r {UGW_PRIO} {UGW_CMD}`.

- If the `rt` argument is specified, it sets a higher real-time priority on interrupt handlers for CAN by running `chrt -f --pid {IRQ_PRIO} {PID}`, otherwise it sets it back to `DEFAULT_PRIO` (50).

- If the `stress` argument is specified it launches a `stress --cpu {STRESS_CPUS}` process to simulate load on the CPU of the gateway and starts scanning the network-attached filesystem with `find / -type f -exec cat '{}'';'`. This stresses IO as well as generates network traffic. Before doing that it also drops caches (`echo 3 >/proc/sys/vm/drop_caches`) so that reads do not just retrieve data from RAM.

- After a second of delay it runs latester on the measuring device with provided arguments (`OPTS` or `OPTS_FLOOD` if `flood` argument is specified). Latester automatically sets its own scheduling policy to `SCHED_FIFO` and priority to 40.

- When latester finishes, the script copies the results from the measuring device over SFTP to the server and stores them under `DEST_DIR/branch` (branch is either `master` or `rt`), where they will be picked up for further processing. The result files being copied are `auto-hist.txt`, `auto-stat.txt` and `auto-msgs.txt`. They are named to include the time, kernel version and used options (e.g., `run-230504-094750-hist+6.3.0-g35ea14bea980+oaat-kern.txt`). The data from `auto-hist.txt` and `auto-stat.txt` are combined and stored in a JSON file (described in 3.3.3). This is done by a function from `hist_to_json.py`.

[8]https://www.paramiko.org/

- Finally, it stops all previously started processes by sending them an interrupt (same as Ctrl+C) and closing the SSH connection. The kernel gateway is also deactivated if it was used (`cangw -F`).

After all tests are finished, the CPU frequency scaling governor is reverted back to `ondemand` to save power.

### 3.3.3 Processing Results

The *txt* histogram files from latester are formatted as two columns separated by a space. The first column contains the latency bin and the second one has the cumulative frame count with this latency (that is, the number of frames with a latency greater or equal to the bin). Bins with identical cumulative value as the previous one are omitted. The step between bins is 0.001 ms. As most ways we display the data are not cumulative, the values of individual bins are computed by taking the difference of two adjacent cumulative bins. Beside that a zero-value bin is added right next to every non-zero bin in places where it was previously omitted. This makes the data less sparse, but it is convenient when displaying it in a line plot so that the line makes a peak rather than interpolating to the next non-zero bin. The bin and value series are saved as `x` and `y` arrays in the resulting JSON. An additional `stats` object is added that contains some values from the `-stat.txt` file. These values include the number of lost frames, minimum, maximum, median, average, 5th and 95th percentile latency, and the number of frames with latency above or below the range of the histogram (below 0 and above 5,000 ms).

After all tests are finished, the resulting JSON files from individual tests are processed into a much smaller number of files that can be served together with the web interface. The `process_json_dir.py` script is called for both subdirectories under `DEST_DIR`, lists all contained files and groups them by the configuration options used (part of the file name). Each of these groups is merged into one large JSON by `hist_cat.py`. It merges the `x` bin arrays from source files into one and pads the `y` arrays with zeros to align the values with the new bins and have the same length. Then the `y` arrays can be put into a 2D "array of arrays" that will be rectangular in shape. The final JSON contains fields `x`, `y`, `stats` and `labels`, where `y` is the 2D array of histograms stacked one behind another, `x` is an array of bin labels corresponding to the second dimension of the 2D array, `stats` is an array of stats objects corresponding to each histogram and `labels` is an array of identifiers (just the file names of the individual results) containing information about kernel version, date and time of the test, and test configuration.

An additional `overview.json` file is created to reduce the amount of data that needs to be transferred just to show the main page of the web interface. It only contains data about a limited number of test configurations (currently 4, but can be changed by modifying `OVERVIEW_CONFIGS`). The data consists of a time series of daily maximum measured latencies of each configuration, as well as the latest histogram. The fields included in the file are: `xs` (an array of arrays of UNIX timestamps of the tests in each time series), `ys` (an array of arrays of maximum latencies), `labels` (the identifiers of each configuration) and `last_hists` (an array of objects containing `x` and `y` data for the histogram and a `label`).

## 3.4 Adding Test Configurations

The kernel branches that should be built are specified in the configuration file. All that is needed to add a new branch is to add an object to `KERNEL_VARIANTS` with the git branch and kernel .config file (see Appendix A for the format of the object). Custom branches will likely need adding of their remote to the local repository (`SOURCE_DIR`). `run_daily_tests` runs tests on all available variants by default.

Adding test configurations is slightly less straightforward. It is difficult to predict what kinds of actions need to be taken in order to achieve the desired test conditions and how they should interact with the other options, hence doing this through the conf file would be limiting.

First, the `run_test.py` must be modified to perform the desired actions on the DUT when the new option is passed in the `args` list argument. Ideally, this would be done by running commands through the `gw` SSH `Connection` object. An identifier of the option also needs to be appended to the test `name`, which should uniquely identify the set of options used. The option must also be added to the `opts` array of `run_tests` function in `run_daily_tests.py` so that test configurations that include this option are run automatically.

# Chapter 4
# Web Presentation

The web interface was designed to be a statically hosted site, that is, all pages are generated beforehand and can be served directly by a server like Apache or NGINX without any processing. Dynamic elements can be achieved with JavaScript on the client side.

The main benefit of the website being static is simplicity, greater loading speed, and potentially better security thanks to a smaller attack surface on the server. The downside is typically the need to move logic onto the client, although in our case very, little logic could be done on the server to begin with. [15] A benefit of a dynamic page specific to this use case would be the possibility of transferring just the right amount of data in a single request instead of downloading everything in one very large request or in small units in many requests. This can be partially imitated in a static site by choosing which sets of data are most commonly needed and precomputing them, although with the downside of taking up more storage on the server.

The plots are drawn using a JavaScript library called *plotly*. It is a feature-rich and easy to use graphing library with a comprehensive selection of plot types. Using it requires little code and the result is a nicely polished interactive graph, rendered as an SVG object inside HTML. 3D graphs are drawn using WebGL. The graph can be panned, zoomed in, or hovered over with mouse to display data point values in a tooltip. [16] It is a bit heavy at 3.4 MB of uncompressed minified JavaScript, although fortunately, there is an option to create a custom bundle[1] that only includes the required chart types, which shrinks the size to about 1.5 MB. That might still be a bit much, but the time saved by not having to replicate the same functionality and polish with less sophisticated tools seems to have been worth it. Further removal of the surface chart type could have reduced the size to just under 1 MB.

An alternative could have been generating static SVGs beforehand with *matplotlib*. The obvious downside would be the absence of interactive features (or the need for some complicated workarounds) and therefore some abnormal histograms spanning a broad range of latencies would be hardly readable without the ability to zoom in. There is a vast selection of alternative charting libraries; the closest of which to achieving the same result is probably *vis.js*, which has the benefit of being modularized by default, and *Apexcharts*. Many

[1]`https://github.com/plotly/plotly.js/blob/master/CUSTOM_BUNDLE.md`

other plotting libraries either lack some useful chart types (like heatmap) or are proprietary.

*Plotly* can be used in a website simply by including its script in the page's `head` tag. The page must also contain a `div` tag that should host the chart. The chart is then drawn by calling `Plotly.react(div_id, data, layout);` from JavaScript and passing it the id of the `div` and the `data` and `layout` objects, constructed according to the documentation to display the loaded data and configure desired functionality. Plotly also supports adding event listeners to allow implementation of custom interactive features. [16]

## 4.1 Serving the Website

In the end, NGINX was chosen as the web server. The web directory of the cloned repository is symlinked as `latester` under WWW Root (`/var/www/html`). The result files that need to be served are put into the `results` subdirectory, which is ignored by *git*. The server configuration described in the following paragraphs was used during development. In later stages of the work, it was decided to mirror the main presentation to a more permanent hosting solution[2], which is not managed by me. Therefore the configuration does not apply there. It is, however, still described here for future reference.

Web browsers cache the downloaded data, so sometimes the visitor could see outdated results. The caching means that the browser does not even begin a request for the file. It is, however, possible to instruct the browser to occasionally revalidate the file, for example, when it was last loaded more than 10 minutes ago. The revalidation means that the browser makes a request that specifies the last version of the file in the HTTP header, and if the file has not since changed, the server responds with status 304 (not modified) and no further data is transferred. This behavior is controlled by the `Cache-Control` response header. To enable this, the line `expires 10m;` is added to the NGINX site configuration under the `/latester/results` location directive, so that it only applies to the changing result files. [17][18]

The single results files typically weigh anywhere from 500 B to 5 KiB, average for the RT branch after running for two months being 1979 B. These are not served, and the merged files that contain the whole series have slightly lower overhead (JSON object keys). Even if they were merely concatenated, that would make about 705 KiB for each test series after a year of testing, 7 MiB in 10 years. With today's internet speeds, this amount of data is perfectly acceptable. This can be further reduced by utilizing gzip compression during transfer. In NGINX, it can be enabled with `gzip on;` and `gzip_types application/json application/javascript;`. Compressing the result files reduces the transfer size by 12 times on average (ranging from 3 to 24, measured on a subset of 16 randomly picked files). It is also possible to precompress the files on disk and then send those directly, decreasing storage use and computational cost of serving them. NGINX allows this using the

[2]`https://canbus.pages.fel.cvut.cz/can-latester/`

gzip_static module [17]. This option is not enabled at the moment as the storage use is very marginal.

The structure of the website is as follows:

```
WWW Root
\- latester
   |- results
   |  |- retcode (symlink)
   |  |- daily-log.txt (symlink)
   |  |- overview.json
   |  |- master-*.json
   |  \- rt-*.json
   |- index.html
   |- overview.js
   |- inspect.html
   |- inspect.js
   |- compare.html
   |- compare.js
   \- style.css
```

## 4.2 Overview Page

The main page (index.html) of the web interface is an overview page that displays the time series of worst latency measured in a subset of test configurations specified by OVERVIEW_CONFIGS. Below the main chart, there is an appropriate number of smaller boxes containing the latest histogram of each series. At the bottom, the page shows whether the last execution of daily tests was successful and when it happened. It also links to the log file, which can be viewed in the browser or downloaded.

Once the page loads, it downloads the overview.json file containing all the required data and instructs *plotly* to draw it. The main chart is a simple scatter plot containing multiple data series. The x-axis shows the date at which the test was run and the y-axis shows the worst latency measured in that day's run on a logarithmic scale. Further, there is a *range slider* showing an overview of the whole graph and the current location when zoomed in. Each data series is automatically assigned a color and can be identified in the legend in the top-right corner of the chart. When the mouse hovers over a data point, a tooltip appears, showing the date and time of the test, the value of the point, and the identifier of the test configuration.

The small histogram charts contain a bar plot that shows how many frames fell into each latency bin during the last measurement (data from last_hists array of the JSON file). There is a bar for every 0.001 ms and it can go up to 5000 ms (this is when the ability to zoom in proves very useful). The y-axis is logarithmic here as well. This chart also shows a tooltip with the nearest x and y data point when hovered over with mouse. Below the small charts is a button allowing toggling cumulative display of the histogram. When

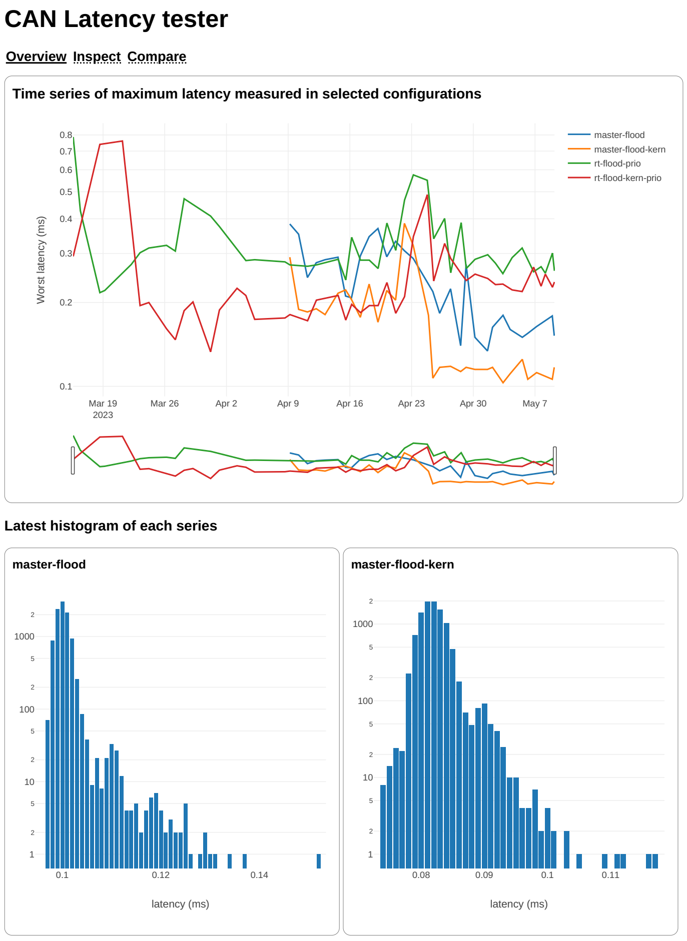


**Figure 4.1:** Overview page

toggled, all small charts are switched to show a line plot instead of the bars. The cumulative values are computed from the regular histogram before being drawn.

To simplify the process of dynamically creating variable number of the small histogram boxes, a script called `build_web.py` is used to generate the HTML code. It needs to be manually called whenever `OVERVIEW_CONFIGS` changes, so probably not very often. It works by formatting and concatenating a predefined HTML string for each box, then replacing with it `{hist-boxes}` in `index.html.in` and saving it as `index.html`. Manipulating DOM on an already loaded page with JavaScript is possible (and used in `compare.js`). However, it results in less readable code, and I prefer to have as little code running in the browser as possible.

Another file downloaded upon page load is `retcode` (symlinked from its actual location). It contains a single number, the return code of the last execution of `run_daily_tests.py` as captured by its wrapper. A successful message is shown if the return code is 0; otherwise, an error is indicated. The date and time of the last execution is determined from the "last modified" timestamp of the file, accessible from the `Last-Modified` HTTP header.

## 4.3 Inspect Page

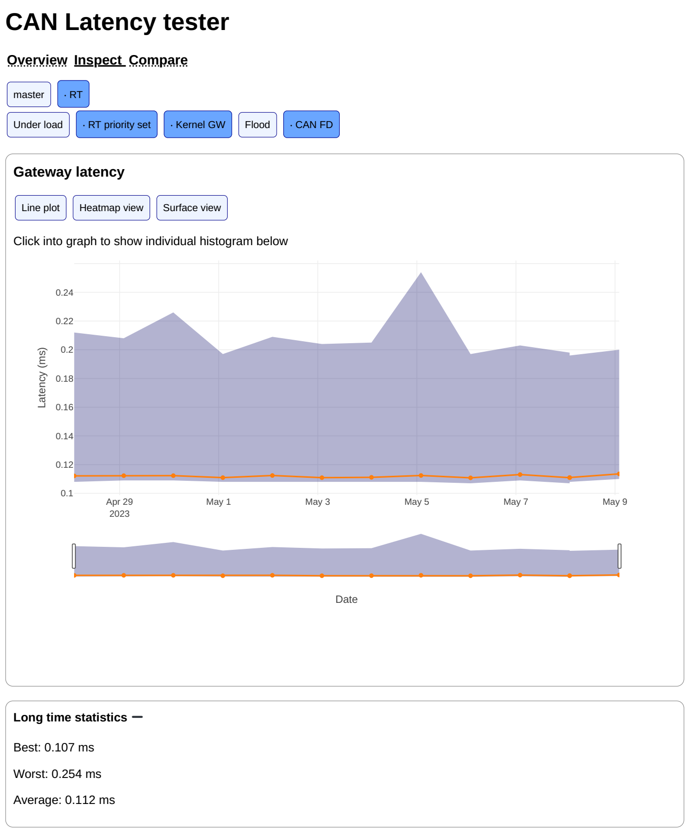
The inspect page allows viewing a single test series as a whole (in 3 different ways – *Line plot*, *Surface* and *Heatmap*) and exploring individual histograms. A test series is a set of tests run on the same kernel branch with the same test configuration. Measurements with different kernel variants or test options are not mixed because they cannot be used to observe a trend in latencies.

The main *Line plot* displays the trend in minimum, average and maximum latency of each test in the selected series over time. An alternative option is the *Surface view*, which shows each histogram stacked one behind another in a 3D scene. In this case, the z-axis (up) is in logarithmic scale. The *Heatmap view* is similar to the surface view – it shows histograms stacked one behind another, except that it is in a top-down view and height of each bar is signified by a color. The result is a grid of colored boxes, rows going upward with time the same way as a spectrogram. The color scale is near-logarithmic (looks better than precisely logarithmic). The darker the box, the more frames it represents. Switching between these views is done by three buttons above the chart. The line plot is depicted in Figure 4.2. The remaining modes can be seen in Appendix B.

The configuration options are specified in the query (search parameters) part of the URL and can also be toggled using buttons. These parameters are only used by client-side JavaScript and do not necessarily need to be passed to the server, but they are useful as the page can be refreshed without losing the selected options and also can be used to link to a particular test series.

The top row of buttons selects one of the kernel branches (master or rt), the second row allows selecting which options were used during the test (for example, which gateway was used or whether the board was under synthetic load). Buttons that are toggled on have a darker shade of blue and either a checked box or filled circle Unicode character in front of the label. Toggling of a button modifies the query parameters in the address bar by using the *history* API without reloading the whole page. Subsequently, the correct JSON file is downloaded based on these parameters. The path to the file is easily determined since the file names contain used options in a fixed order. The JSON file contains all the histograms from measurements matching the selected configuration, each including its identifier (information on kernel version and time of measurement) and stats object.

Hovering over the main chart shows the identifier of the specific test just below the chart and also displays a tooltip showing the value of the data point closest to the mouse pointer. In the line plot mode, it shows all three values (min, max and average). "Brushing" (clicking and dragging) over the chart in Line plot and Heatmap view zooms in on the selected range. The surface view can be zoomed in by scrolling. Double-click resets the view to



Click into graph to show individual histogram below

Best: 0.107 ms

Worst: 0.254 ms

Average: 0.112 ms

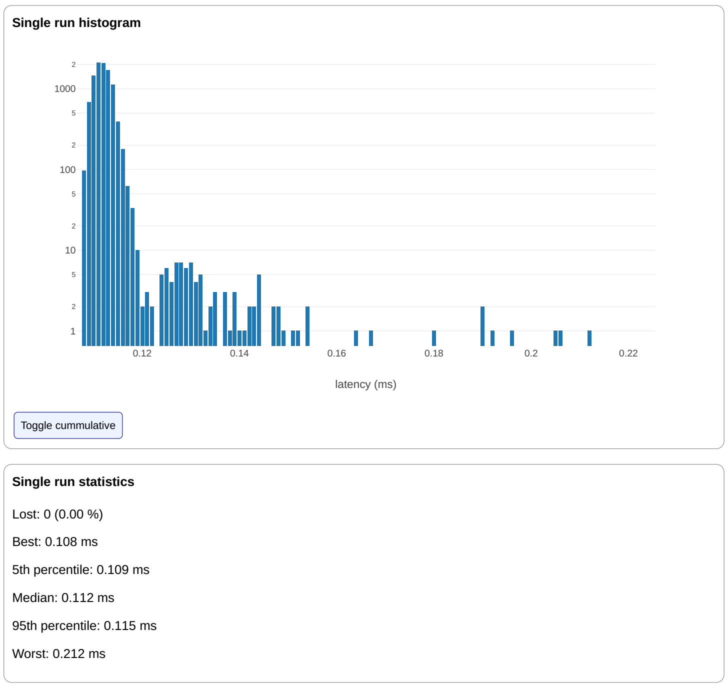
**Figure 4.2:** Inspect page – main chart and stats

default.

A small box below the main chart shows the minimum, maximum and average latency over the whole test series. The box is collapsible by clicking on its header so that both the main plot and individual histogram can be visible at the same time without scrolling.

Clicking into any of the main plots shows a second plot below, which displays the particular histogram for a more detailed view. The style is the same as with the small charts in Overview and can also be switched to cumulative view. The cumulative values are computed and saved when the source JSON is loaded, so switching the chart back and forth does not incur unnecessary computational cost.

The kernel branch buttons are generated automatically by `build_web.py` according to the configuration file – it processes `inspect.html.in` the same way as the Overview page. The test option buttons need to be added manually to the HTML with the same style as the existing ones. The click event should call `toggleparam('opt')` with the same parameter as the identifier used in the test result file names. The option identifier also must be added as a field to the `opts` object in `inspect.js`. The value of the field is a boolean



Single run statistics

Lost: 0 (0.00 %)

Best: 0.108 ms

5th percentile: 0.109 ms

Median: 0.112 ms

95th percentile: 0.115 ms

Worst: 0.212 ms

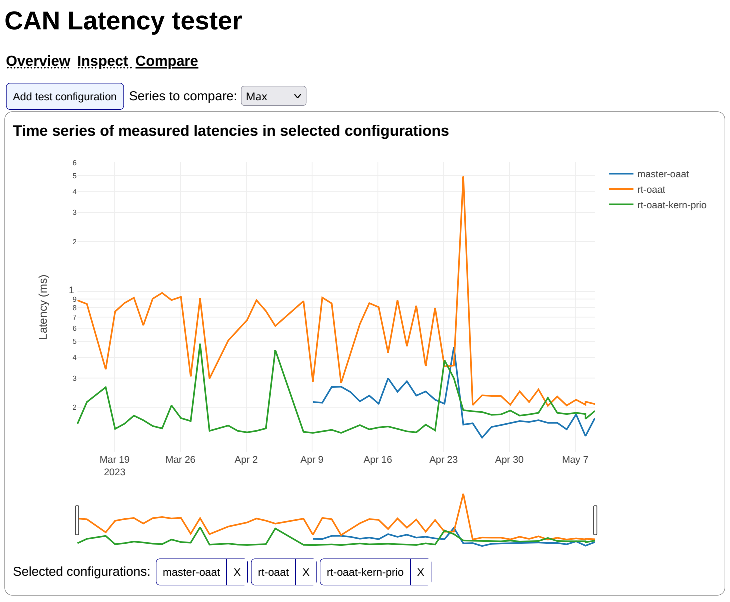
**Figure 4.3:** Inspect page – individual histogram

specifying whether the option should be on by default when the page is first loaded. The order of the fields must be the same as the order in which the identifiers appear in the result file names.

## 4.4 Compare Page

The *Compare* page is meant to view an arbitrary number of test series together for comparison. The main plot shows a line trace for each selected series. The data shown can be switched between min, max and average latency of the tests in the series. This is done by a simple dropdown above the main chart. The test series are also added from a dropdown, and the number of them that can be shown together is not limited. The currently selected series are visible in the legend to the right of the chart, where they can be selectively hidden. The series also appear as a list of buttons below the chart. The button has two functions - the left part of the button with the series identifier can be clicked to show it on the *Inspect* page. Clicking on the smaller right part (with an 'X') completely removes the series from the view and the legend.

Adding a series causes its appropriate JSON file to be downloaded. These are the same files as used by *Inspect* page. The parsed JavaScript object is then stored in a list, so when it is removed and re-added, it does not need to be downloaded again (or even revalidated if the browser respects the cache policy).


CAN Latency tester

Overview Inspect Compare

Add test configuration Series to compare: Max

Time series of measured latencies in selected configurations

Selected configurations: master-oaat X rt-oaat X rt-oaat-kern-prio X

**Figure 4.4:** Compare page

The list of available test series is embedded in HTML by `build_web.py` that also generates the previous two pages. It must be run whenever a new test configuration is added or some are removed. It simply takes the `compare.html.in` and replaces `{config-items}` with a generated HTML string containing the dropdown items.

# Chapter 5
# Conclusion

The goals of this thesis were to automate latency testing on the CAN bus, assemble the testbed setup, and build a website to present the results.

A testbed to perform the continuous testing was designed and assembled. It consists of two SBCs with FPGA-based CTU CAN FD controllers, which are used to perform the measurement, and a support testbed device that controls their power supply and allows them to be interacted with via serial connection. A server was set up to manage the devices and coordinate the testing. The devices were configured to boot from the network and have their filesystems hosted on the server, which simplifies their management and recovery in case of software-related failure. On top of that, a small driver for the crossbar switch was written, which could streamline interacting with it in the future.

The process of latency testing was scripted and automated. The resulting system measures latencies of a CAN gateway under various conditions and is easily configurable and extensible. All important aspects were documented to enable possible future inclusion in a QA farm. The testing has been running on the assembled testbed every day since late March, although with some changes in methodology making the old results inconsistent with those recorded in May and onward.

Integration of the system with OSADL is being negotiated, a meeting is likely to happen at the end of June. However, this time frame is past the deadline for the completion of this thesis.

The testing system has already contributed to the stability of PREEMPT_RT Linux. When tests that happen under synthetic load were added, the DUT started experiencing lockups during the test, and warnings from the kernel were printed to the console. The maintainers were notified and it was discovered that there was an error in the logic for disabling preemption. The error has since been fixed, in Linux RT versions 6.3.3-rt15 and 6.4-rc2-rt1 respectively.

Finally, a website was built to present the results from this system. The site allows viewing the trends in latencies under the different test configurations, and offers multiple display options, including examining individual histograms. It also reports whether the last test run was successful and when it happened. In case of problems, it is possible to view the log file.

# Appendix A

# Configuration File

This is an overview of values in the conf.json file. An example can be also found in the project repository.

- `IP_TS`: IP address of the measuring device
- `IP_GW`: IP address of the gateway (DUT)
- `KEY_FILE`: Path to the private key that is used to log in to the boards
- `UGW_CMD` and `UGW_FD_CMD`: Command that starts the user gateway
- `CGW_CMD` and `CGW_FD_CMD`: Command that configures the kernel gateway
- `STRESS_CPUS`: How many threads of synthetic load to run
- `UGW_PRIO`: Priority to set on the user gateway process (default 80)
- `IRQ_PRIO`: Priority to set on CAN interrupt handlers (default 90)
- `OPTS` and `OPTS_FLOOD`: Options for the latester command and options to use during flood test
- `DEST_PATH`: Path where the results of tests should be copied
- `PING_FLOOD_TARGET`: Hostname (or IP) where to direct ping flood during stress tests, can be ommited to disable
- `MAINLINE_DIR`: Location of mainline Linux repository that is pulled before everything else
- `SOURCE_DIR`: Location of Linux source tree that should be built
- `BUILD_DIR`: Location where *make* should put build outputs
- `BUILD_OUT_DIR`: Location where to archive interesting build outputs (zImage and device tree blob)
- `INSTALL_DIR`: Path to the root of the filesystem where to install kernel modules

- `REBOOT_CMD`: The command executed from the server that will reboot the DUT. For example SSH onto the testbed device and send a break signal to DUT's serial port
- `KERNEL_VARIANTS`: An object containing other objects with information about the kernel branches to build. The names of the objects within it are the identifiers that can be used with the `--branches` option of `run_daily_tests.py`. Each inner object has the following fields:
  - `NAME`: The name of the branch how it should appear on the website
  - `CONFIG_FILE`: Location of .config file that should be used for the build
  - `GIT_BRANCH`: Which git branch to checkout before building
- `DTBOCFG_DIR`: Location of *dtbocfg* source code
- `WEB_DIR`: Location of the web
- `OVERVIEW_CONFIGS`: A list of test configurations to show in overview (e.g.: `master-oaat-kern`)

# Appendix B

# Additional Website Screenshots

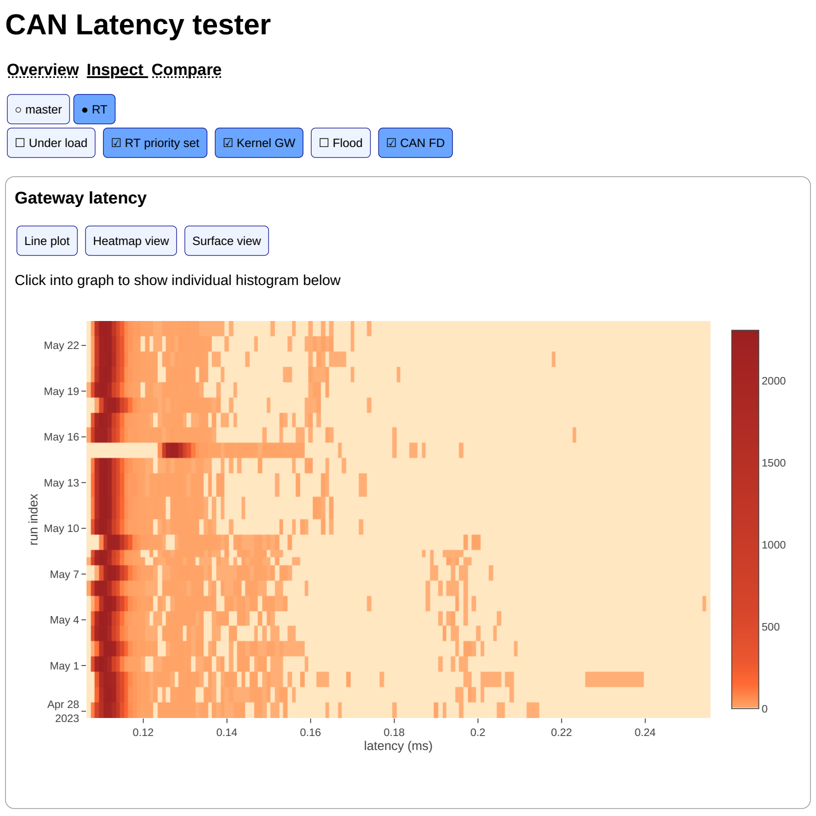
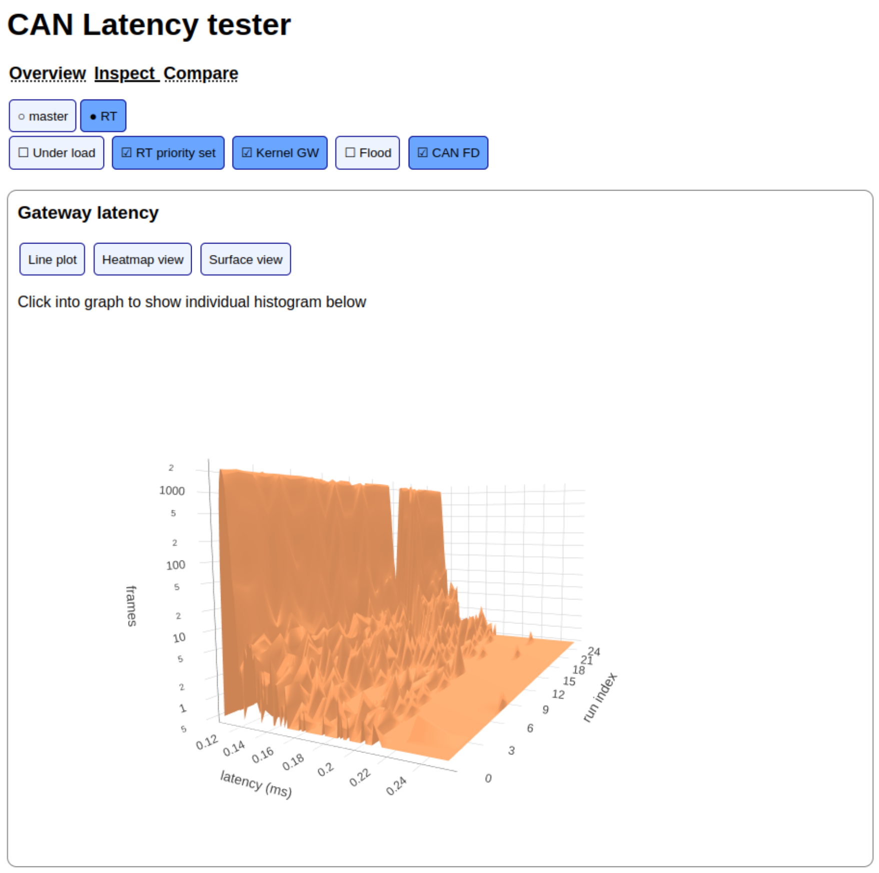


**Figure B.1:** Inspect page – heatmap view



## CAN Latency tester

Overview Inspect Compare

○ master ● RT

☐ Under load ☑ RT priority set ☑ Kernel GW ☐ Flood ☑ CAN FD

### Gateway latency

Line plot Heatmap view Surface view

Click into graph to show individual histogram below

**Figure B.2:** Inspect page – surface view

# Appendix C
# List of Related Software

- `can-latester-automation` - The main repository with code developed as part of this thesis. It contains the scripts that perform the testing, scripts that run on the target devices and the source code of the website. Available at `https://gitlab.fel.cvut.cz/canbus/can-benchmark/can-latester-automation`.
- `can-crossbar-driver` - The repository with the CAN crossbar switch driver also developed as part of this thesis. Available at `https://gitlab.fel.cvut.cz/canbus/can-benchmark/can-crossbar-driver`.
- `can-latester` - The CAN latency measurement software by Michal Sojka, Martin Jeřábek and Matěj Vasilevsky. Minor fixes were contributed as part of this thesis. Available at `https://gitlab.fel.cvut.cz/canbus/can-benchmark/can-latester`.

# Appendix D Bibliography

[1] M. Jeřábek, “FPGA Based CAN Bus Channels Mutual Latency Tester and Evaluation”, Bachelor’s thesis, CTU FEE, Prague, 2016.

[2] M. Vasilevski, “CAN Bus Latency Test Automation for Continuous Testing and Evaluation”, Master’s thesis, CTU FEE, Prague, 2022.

[3] C. Watterson, *Controller Area Network (CAN) Implementation Guide*, Analog Devices Inc., 2017. [Online]. Available: `https://www.analog.com/media/en/technical-documentation/application-notes/AN-1123.pdf` (visited on 2023-05-12).

[4] *LinCAN (CAN bus driver)*. [Online]. Available: `https://ortcan.sourceforge.net/lincan/` (visited on 2023-05-12).

[5] The kernel development community, *SocketCAN - Linux Kernel Documentation*. [Online]. Available: `https://www.kernel.org/doc/html/v6.1/networking/can.html` (visited on 2023-05-14).

[6] The Linux Foundation, *Real-Time Linux Wiki*. [Online]. Available: `https://wiki.linuxfoundation.org/realtime/start` (visited on 2023-05-12).

[7] Open Source Automation Development Lab eG., *OSADL QA Farm on Real-time of Mainline Linux*. [Online]. Available: `https://www.osadl.org/OSADL-QA-Farm-Real-time.linux-real-time.0.html` (visited on 2023-05-20).

[8] *MicroZed APO Datasheet*. [Online]. Available: `https://cw.fel.cvut.cz/b212/_media/courses/b35apo/en/semestral/mz_apo-datasheet-en.pdf` (visited on 2023-05-24).

[9] O. Ille, J. Novák, P. Píša, and M. Vasilevski, “CAN FD open-source IP core”, *CAN Newsletter*, vol. 3/2022, PDF, CAN in Automation, 2022. [Online]. Available: `https://can-newsletter.org/uploads/media/raw/a9abe317ae034be55d99fee4410ad70e.pdf`.

[10] P. Porazil, “tb_orpi-1 schematic”, 2017.

[11] Digi International Inc, *U-Boot Reference Manual*, 2011. [Online]. Available: `https://www.digi.com/resources/documentation/digidocs/PDFs/90000852.pdf` (visited on 2023-05-24).

[12] I. Kawazome, *Device Tree Blob Overlay Configuration File System.* [Online]. Available: `https://github.com/ikwzm/dtbocfg` (visited on 2023-05-19).

[13] M. Jeřábek, “Open-source and Open-hardware CAN FD Protocol Support”, Master’s thesis, CTU FEE, Prague, 2019.

[14] M. Sojka, P. Píša, and Z. Hanzálek, “Performance evaluation of Linux CAN-related system calls”, *2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014)*, pp. 1–8, 2014. DOI: `10.1109/WFCS.2014.6837608`.

[15] E. Howey, *Static vs dynamic websites: what are the differences?* [Online]. Available: `https://www.sanity.io/static-websites/static-vs-dynamic-websites` (visited on 2023-05-21).

[16] Plotly, *Plotly JavaScript Open Source Graphing Library.* [Online]. Available: `https://plotly.com/javascript/` (visited on 2023-05-22).

[17] Nginx, Inc., *nginx documentation.* [Online]. Available: `https://nginx.org/en/docs/` (visited on 2023-05-23).

[18] Mozilla Corporation, *MDN Web Docs.* [Online]. Available: `https://developer.mozilla.org/en-US/` (visited on 2023-05-23).