embeddedworld
Exhibition&Conference

Continuous CAN Bus Subsystem Latency Evaluation and Stress Testing on GNU/Linux-Based Systems

Pavel Píša, Pavel Hronek, Matěj Vasilevski
Department of Control Engineering, FEE
Czech Technical University in Prague
Prague, Czech Republic
pisa@felk.cvut.cz, hronepa1@fel.cvut.cz,
matej.vasilevski@gmail.com

Jiří Novák
Department of Measurement, FEE
Czech Technical University in Prague
Prague, Czech Republic
jnovak@fel.cvut.cz

***Abstract*—The control of vehicles, industrial, and or motion systems requires complex controllers with defined response times. Many high-level control systems utilize complete POSIX-based operating systems capable of processing data from cameras, LIDARs, and other sensors by running complex machine-learning algorithms. The fully preemptive variant of the kernel with long-term testing, as proceeded by example on OSADL Quality Assurance Real-Time Farm, provides enough guarantee for the timing of the control algorithms running on the single system. However, the ability to deliver commands in time to other components is limited by latencies caused by communication hardware and related operating system drivers. CAN and CAN FD are often used for this communication. Long-term evaluation of latencies caused by system load, driver implementation, and other factors is critical to assess system reliability. Monitoring and checking the evolution of latencies in the mainline Linux kernel source tree allows for resolving problems before they propagate to production system updates. The daily evaluation is the service of our continuous CAN/CAN FD latency testing system. CTU CAN FD IP core is used for precise timestamping on the generator side. At the time of the conference, we present data from daily testing of multiple kernel configurations running more than one year already. All developed components are available in the source form and can be reused by carmakers and other CAN bus users.**

Keywords—Linux; kernel; CAN bus; CAN FD; latency; drivers

I. Introduction

The Czech Technical University in Prague team has supported companies in industrial and automotive communications for over 30 years. The CAN communication boards with a coprocessor for CAN open communication were designed in the nineties. Later, many generations of CAN and CAN FD bus and automotive ETHERNET analyzers, error injection systems, and other tools for carmakers such as SkodaAuto, Porsche, and Volkswagen have been delivered. CAN and CAN FD IP cores and other FPGA solutions have been designed for general and special purposes. As for the software side, CAN/CAN FD drivers for own and generally available hardware have been developed for Linux kernel, NuttX, RTEMS, and Windows operating systems. The LinCAN character driver-based framework [1] predates later widely adapted Linux SockteCAN [2] by multiple years. When SocketCAN gained momentum, bit-timing support, and some drivers were provided to the SocketCAN mainline.

The analysis and evaluation of communication and response latencies for control systems have been taken as the critical part of the provided support and projects, and the tool to evaluate CTU's LinCAN has been designed in parallel with the subsystem and extended later to support even SocketCAN. Then, in 2011, Volkswagen Group Research worked on an in-kernel Linux-based CAN gateway. Because previous work started by Pavel Pisa has been well known in the community, the Department of Control Engineering CTU FEE has been contracted to provide a system for evaluating the latencies of the CAN gateway. The experiments measured the latency of the gateway, the time spent between receiving a CAN message on one bus, processing it, and transmitting the message on a second bus. Such a gateway can be used, for example, in a car, where it separates different networking subsystems. One subsystem can contain electronic control units (ECUs) that control the engine, and the other subsystem consists of an 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 continued in 2014, comparing the performance of multiple interfaces used to access a CAN bus on GNU/Linux systems. To evaluate 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. The x86 PC computer

equipped with a Kavser PCI quad-CAN SJA1000-based card has been used for traffic generation and latency measurement.

The new CTU CAN FD IP core [3] has been conceived to extend analyzers for SkodaAuto to support CAN FD protocol (around 2015). The IP core is equipped with a 64-bit clock counter common to all CAN FD interfaces implemented on a given FPGA-based system. Integration to Intel FPGA-based PCIe card as Intel and AMD/Xilinx ARM-based SoCs has been developed. The Xilinx Zynq-based system is used for the Linux kernel continuous latency evaluation system. The 100 MHz FPGA clock frequency corresponds to the 10 ns time resolution of timestamps attached to the received CAN/CAN FD frames.

II. CAN Latency Measurement HW Setup

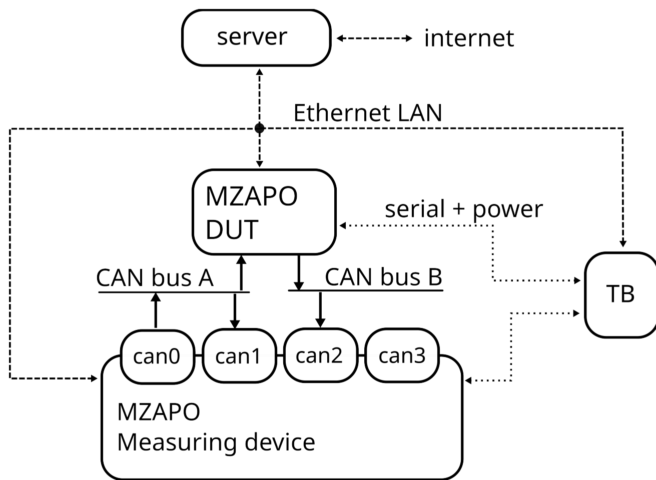

Figure 1: Latency tester HW setup

The two systems are used to evaluate gateway latency. The system under test (DUT) has a simple role – receive messages from CAN bus A and instantly send them to the second CAN bus B. The other system, measuring one, generates CAN/CAN FD frames on CAN bus A and receives frames on both buses with precise timestamps attached. Its 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 measurement principle. On the device under test, one controller is connected to internal 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 Fig. 1. The two educational kits MZ_APO [4] based on AMD/Xilinx Zynq SoCs on MicroZed SBC board are used in our case. However, there are minimal requirements for the device under the test, which is two CAN/CAN FD interfaces. Even a system with a single CAN interface can be evaluated with a modified test procedure when a message is replied to/echoed by a message with a different CAN ID. The measuring device utilizes advanced CTU CAN FD IP core timestamping, but another system with multiple CAN channels and hardware timestamping can also be used.

The operation of both devices needs to be coordinated during measurement, and the results need to be stored and processed. An external server running in the CTU FEE virtualization farm is connected to the devices via Ethernet and provides a root filesystem over NFS for both systems. It also controls booting and actual measurements over SSH. The advantage of using an additional server 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. The NFS exported device root filesystem is updated to include matching kernel modules and the kernel and bitstream ITS image server over TFTP after each kernel build.

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. It allows both low-level and panic kernel messages to be captured and the MZ_APO boards to be reset remotely. The MZ_APO kit USB to serial console circuitry monitors break condition, and when active for longer than 2 seconds, it activates the Zynq hard reset signal. The server, however, can be in a different location than the boards, making direct connection impractical. That has been solved by adding another testbed control device (TB) – a combination of the ARM64 OrangePi Zero Plus board equipped with an expansion board designed by Ing. Petr Porazil of PiKRON.com. Besides a USB hub for providing serial consoles to the MZ_APO boards, it includes power switches that allow the power of two other boards to be switched on or off remotely. It also means that a single power supply can power all three devices. The TB is connected to the same Ethernet LAN as the other devices.

III. Latency Measurement Software

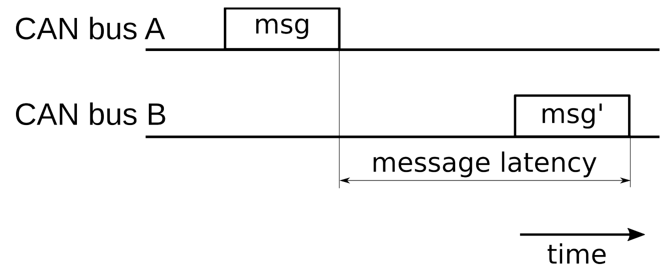

Figure 2: CAN message latency after retransmissions by gateway

The measurement itself is carried out by the can-latester software [5]. 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 allows testing the CAN frame processing latencies of systems equipped with only one CAN interface, like the ESP32C3 MCU. In this case, the DUT retransmits each frame onto the same bus, just with a higher priority/CAN ID.

In gateway mode, the measurement system sends 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 (DUT) 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 source of variance in measured latencies. The measuring device subtracts the hardware-captured timestamps recorded at the start of the frame reception by can1 and can2 interfaces and computes the latency as shown in Fig. 2. Using the timestamp from can1 ensures we know the exact time when the frame 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.

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 suitable for use in flood mode since it has to wait for the medium to become available (end of the current frame), which slightly affects the latency. 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 do not start their time counters at the same time. Therefore, synchronization of timestamps is performed before the measurement. All interfaces are temporarily connected by the CAN crossbar switch when a single synchronization frame is sent, and received frame timestamps are used as a reference base for measurement. This step is unnecessary for CTU CAN FD IP core with a single hardware time counter for all interfaces, but it is still performed to ensure portability to other interfaces with timestamping. 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 [6].

IV. Latency Testing Automation

The test system is orchestrated by automation scripts written in Python language. The top-level one is called *run_daily_tests.py* and is run daily by the cron daemon. The top-level script calls multiple utility ones. The *build_linux.py* is called to build a Linux kernel for each test run.

The latest Linux kernel versions from specified branches pulled from the kernel.org repository are built. Currently, two variants are built; one version is the latest state of the master branch, and the second is the current development version, which includes 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 and processes these results for presentation on the project web pages. The scripts that handle the automation and the website source code are available from the project GIT repository [7].

The U-boot ITS image is built from the DUT kernel and programmable logic bitstream and copied to the directory accessible from the boards over TFTP protocol. The boot is controlled by a small script/sequence of U-boot commands, which is loaded using the TFTP protocol. The DUT and measurement boards are distinguished from the script by comparison of the Ethernet address of the device. The measurement device loads a stable/known to work well kernel. Its CTU CAN FD driver is patched by a hardware timestamping support patch, which has not yet reached the mainline.

The test system resets the boards over TB serial line connections and waits till both systems respond to the test attempt to connect the SSH port. The board root filesystem is based on the Debian Linux distribution, but they initialize through the init-overlay script [8], which maps the overlay filesystem with tmpfs layered over base NFS read-only export. This way, no changes or damage can propagate to the exported root filesystem, and multiple devices can boot from the same export.

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 the latester and ugw - user mode gateway, do not handle). Additionally, the irq threads of the measuring device's receiving CAN interfaces must be given higher priority than the transmitting interface. Otherwise, the system sometimes did not read out frames quickly enough, leading to buffer overflows. All these steps are automated in a script called init-device.sh, which is included in the repository. 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*.

Configuring the FPGA design uses the synthesized FPGA design (firmware) and a device tree overlay file (DTBO) that describes the devices in the design to the operating system. The design file is copied under */lib/firmware/* directory. The device tree overlay file is compiled into a device tree overlay (DTBO) using the DTC tool (part of the Linux kernel repository). The DTBO is then loaded using the *dtbocfg* kernel module. The module first needs to be loaded (modprobe dtbocfg), and then a directory under */sys/kernel/config/device-tree/overlays* is created, for example, *my-overlay*. The DTBO file is copied inside this new directory under dtbo name, and overlay is activated by writing "1" into a *status* file in the same directory. The correct firmware is automatically loaded into the FPGA as specified in the device tree fragment (e.g. firmware-name = "system.bit.bin";).

When devices boot, the *run_test.py* script is executed for each 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 script. When all configurations for different loads, priority, and gateway implementation are tested, then *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 independently from the command line, although the main script imports them as modules and calls their functions. The option to run scripts manually was helpful during development when some phases of the testing failed.

An additional script called *build_web.py* needs to be executed manually only when some configuration changes.

All scripts use the *conf.json config* file, which is by default 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 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 next options that are set on and off are **flood** (send CAN messages as fast as possible) **kern** (kernel gateway is used instead of user mode one), **stress** (stress CPU and memory of the dut system), and **fd** (CAN FD messages are used). When testing an RT kernel, there is an additional **rt** option to elevate CAN IRQ thread priorities that does not apply to master. These options are passed in a list as the parameters or on the command line.

V. WEB PRESENTATION

The web interface was designed to be a statically hosted site. All pages are generated and updated daily beforehand and can be served directly by a server like Apache or NGINX without processing on the server side. Dynamic elements are implemented on the client side in JavaScript.

The main benefits of the website being static are simplicity, greater loading speed, and better security, thanks to a smaller attack surface on the server. The downside is the need to load a whole series of measured data for processing on the client side. However, latency trend visualization for consecutive development kernel versions is a typical use case for monitoring, and the ability to analyze individual latency profiles would required to load data on the request later anyway. On the typical contemporary network, loading a whole year of data series is fast enough, so a more complex server-side would not be justified.

The plots are drawn using a JavaScript library called *plotly*. It is a feature-rich and easy-to-use graphing library with a comprehensive offer of plot types. Its use for data series representation requires little code and provides a 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 with a mouse to display data point values in a tooltip [9]. It is a bit heavy at 3.4 MB of uncompressed minified JavaScript, although fortunately, there is an option to create a custom version, see *https://github.com/plotly/plotly.js/blob/master/CUSTOM_BUNDLE.md*. That includes only the required chart types, which shrinks the size to about 1.5 MB. That might still be a little more than optimal, 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 under 1 MB.

An alternative could have been generating static SVGs beforehand with *matplotlib*. The obvious downside is 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. The closest library with a similar interactive features set is probably *vis.js*, which has the benefit of being modularized by default, and *Apexcharts*. Many 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*, 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 the implementation of custom interactive features.

The single histogram result files typically represent 500 B to 5 KiB, with the average for the RT branch after running for two months being 1979 B. These are not served individually; instead, the merged files that contain the whole series have slightly lower overhead (JSON object keys). It has been expected that even if the histograms in the series are concatenated, that would make about 705 KiB for each test series after a year of testing—the actual results after little more than one year of daily operation range from 200 kB for smooth series of unload system to 11 MB for actually misbehaving RT kernel under full load conditions. Whole Web presentation data are compressed to about 5 MB after year for the daily run. The archive of this size is transferred to GitLab pages for the final presentation. With today's internet speeds, this amount of data with a maximal series of about 10 MB is acceptable.

VI. THE OBSERVED RESULTS AFTER YEAR OF OPERATION

The system was prepared during the last quarter of 2022, and after some experimental rounds, it was set up for continuous operation at the start of April 2023. The gateway process's priority and latencies' measurement under the load (*stress --cpu 2 --vm 2* and ping flood on the DUT) has been adjusted, and the system runs under constant conditions from 25 April 2023. Typical latencies for the unloaded mainline Linux kernel are in a range of 0.1 ms. The maximal latencies measured on loaded mainline kernel are usually around 1 ms for in kernel CAN gateway and around 3 ms when CAN frame retransmission is realized by the user space program (ugw) even that RT priority (SCHED_RR 80) is set for the program. However, such behavior is expected for the mainline kernel with non-preemptible kernel sections, huge networking interrupts, and packet processing in the kernel tasklets and

bottom halves, which can block CAN processing for a long time.

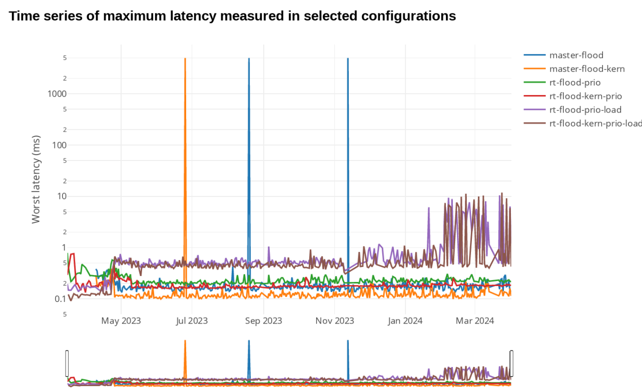

Figure 3: CAN latency tester overview page

The main aim of the latency testing is to monitor the PREEMP_RT Linux kernel variant behavior and evolution. With more than ten years of experience with running and tuning real-time enabled Linux kernel on many ARM-based systems, our expectation is that about 0.2 ms (4 kHz with safe margin) is a realistic expectation for the timing of control loops where no sampling period is skipped on 32-bit ARM systems. We have used Zynq-based systems for motion control tasks, including complex PMSM servo systems control. The *cyclictest -m -Sp99 -i200 -h400* confirms our expectations even on the latest PREEMP_RT Linux kernels. The Linux kernel networking stack is known to be problematic from a real-time point of view. It uses the bottom halves-based processing and mixes short CAN/CAN FD packets with complex TCP/IP traffic. However, many projects rely on SocketCAN as a subsystem used in closed control loops, so knowledge of its behavior and some statistical long-term assessment should be considered before such use.

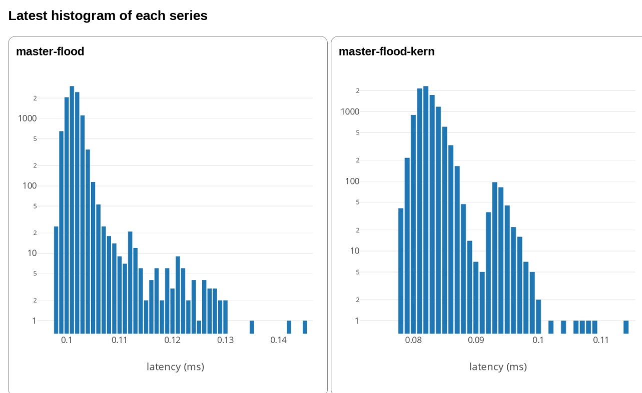

Figure 4: CAN Latency tester histograms for individual daily tests

We can observe that unloaded PREEMP_RT Linux kernels can keep the maximal latency around 0.2 ms. There are some glitches up to 0.45 ms and some samples that could be caused by some problems connected to RT patches evolution and integration during kernel updates. The continuous testing on Zynq systems allows us to report some critical errors early, which slip into PREMP_RT kernel development. It was the case of breakage of VFP support for the whole ARM 32-bit family in 6.3 development cycle. The report has been quickly resolved by the PREEMP_RT development team (ARM: vfp: Fixes for PREEMP_RT patch by Ard Biesheuvel and Sebastian Andrzej Siewior).

We observe a significant statistical increase in the latencies of the PREEMP_RT, which corresponds to the update from the 6.7.0-rt6 to the 6.8.0-rc1-rt1 version. If the situation does not improve by the start of the 6.9 series, we plan to investigate further the cause and consult the state with the expert team working on PREEMP_RT development and integration into the mainline.

VII. Conclusion

The system for monitoring CAN communication latencies in the Linux kernel has been designed and automated to the level that it already served for a year. The measuring system is based on the CTU's own CAN controller design with precise timestamping, but latency testing automation is implemented as portable even to other controllers with hardware timestamping. At least three independent CAN interfaces with some circuitry, which allow the connecting of all buses together for initial timestamp hardware offset calibration, are ideal for the Linux-supported measurement system. The system can be used for latency assessment of almost arbitrary devices under the test (DUT), primarily focusing on Linux-based systems with at least two CAN/CAN FD interfaces. However, the use of our latency tester system has been demonstrated even on small Espressif ESP32C3 systems running the NuttX system, which has only a single CAN interface supported by the NuttX driver from our other development project. We are preparing a completely new CAN/CAN FD subsystem for the RTEMS operating system used by ESA and NASA, and we plan to add this stack assessment to our list of daily rounds as well. Our preliminary results are very promising, with latencies under 0.1 ms under the load on the same hardware. Our driver infrastructure allows ideal utilization of a limited number of hardware Tx buffers for multiple messages priority classes with a mechanism to keep FIFO order for a given communication stream yet preventing a link-level messages arbitration priority inversion case. The CAN subsystem for RTEMS design article will be published at the International CAN Conference 2024 [10].

We believe that each serious industrial or carmaker user of the Linux kernel-based systems who use it with CAN/CAN FD communication for more than pure traffic recording for offline use should care and have at least statistical data for in-system latencies. Our setup is fully documented and available from the CTU CAN GitLab project. We have negotiated cooperation with Open Source Automation Development Lab (OSADL) eG in the area of monitoring CAN latencies and their hunting. There are more ongoing related projects run by members of the informal CTU Open Technologies Research Education and Exchange Services group [11] [12].

Acknowledgment

The project has been supported by part from the Josef Bozek National Center of Competence for Surface Vehicles, Support programme for applied research, experimental development and innovation of Technology Agency of the Czech Republic, ID TN01000026

References

[1] CTU, “LinCAN driver”, OCERA Real-Time CAN (OrtCAN) project https://ortcan.sourceforge.net/lincan/, [Online; accessed 2024-04-01].

[2] Linux Foundation, “SocketCAN – Controller Area Network”, https://docs.kernel.org/networking/can.html, [Online; accessed 2024-04-01]

[3] CTU, FEE, Department of Measurement, “CTU CAN FD IP Core – Datasheet”, https://canbus.pages.fel.cvut.cz/, [Online; accessed 2024-04-01]

[4] CTU, PiKRON.com, “Education Kit MicroZed APO”, https://cw.fel.cvut.cz/wiki/courses/b35apo/en/documentation/mz_apo/start, [Online; accessed 2024-04-01]

[5] CTU, FEE, “CAN Bus Latency Tester (Benchmarking Utilities)”, https://gitlab.fel.cvut.cz/canbus/can-benchmark/can-latester/, [Online; accessed 2024-04-01]

[6] M. Vasilevski, “CAN Bus Latency Test Automation for Continuous Testing and Evaluation”, master's thesis, 2022, CTU, FEE, https://dspace.cvut.cz/bitstream/handle/10467/101450/F3-DP-2022-Vasilevski-Matej-vasilmat.pdf, [Online; accessed 2024-04-01]

[7] CTU, FEE, “The automation of CAN latency Quality Assurance reports generation”, https://gitlab.fel.cvut.cz/canbus/can-benchmark/can-latester-automation, [Online; accessed 2024-04-01]

[8] init-overlay script from “Utilities and Configurations for Raspberry Pi GNU/Linux”, https://github.com/ppisa/rpi-utils/, [Online; accessed 2024-04-01]

[9] Plotly, “Plotly JavaScript Open Source Graphing Library”, https://plotly.com/javascript/, [Online; accessed 2024-04-01]

[10] M. Lenc, P. Pisa, “Scheduling of CAN frame transmission when multiple FIFOs with assigned priorities are used in RTOS drivers”, CAN in Automation, Germany, 2024, [in print]

[11] CTU, FEE, “CAN bus CTU FEE Projects List”, https://canbus.pages.fel.cvut.cz/, [Online; accessed 2024-04-01]

[12] CTU, FEE, “Open Technologies Research Education and Exchange Services Knowledge Base”, https://gitlab.fel.cvut.cz/otrees/org/-/wikis/knowbase, [Online; accessed 2024-04-01]

[13] CTU, FEE, “CAN Latency Tester”, daily output, https://canbus.pages.fel.cvut.cz/can-latester/, [Online; accessed 2024-04-01]