A Comparison of Linux CAN Drivers and their Applications

Michal Sojka, Pavel Píša, Martin Petera, Ondřej Špinka and Zdeněk Hanzálek
Department of Control Engineering, Faculty of Electrical Engineering
Czech Technical University
Karlovo náměstí 13, 121 35 Prague 2, Czech Republic
sojkam1@fel.cvut.cz, pisa@cmp.felk.cvut.cz, martin.petera@gmail.com, spinkao@fel.cvut.cz, hanzalek@fel.cvut.cz

***Abstract*—The aim of this paper is to introduce LinCAN, a CAN driver system for Linux, developed at the Department of Control Engineering of the Czech Technical University in Prague, and to provide a thorough comparison with SocketCAN, which is the most common CAN solution for Linux nowadays. Thorough timing analysis and performance comparison with Socket CAN are presented, with several case-studies and applications of LinCAN shown in the end.**

LinCAN has been developed since 2003 and supports many CAN controllers from various manufacturers. It is designed with emphasis on strict real-time properties and reliability, making it ideally suitable for networked control systems (as is also demonstrated in the case-studies). LinCAN is also portable to other Operating Systems and can be used even system-less (without any OS) on less-powerful microcontrollers.

A timing analysis and performance tests of both drivers were performed using various types of load with several recent Linux kernels. Obtained results indicate that LinCAN seems better suited for hard real-time applications, its performance being either better or on-par with SocketCAN in presented tests.

Both LinCAN and SocketCAN drivers are completely open-source as well as our testing tools, so any researcher interested in our results is welcome to download all relevant source codes, check our testing methodology in detail and perhaps recreate our results or extend them by performing other test, providing valuable feedback and independent verification of our work.

I. Introduction

The Controller Area Network (CAN) has been widely used, not only in automotive or industrial applications, for quite a long time and is well-known among the embedded systems community. Linux, especially with its real-time extensions merged into the mainline recently, is gaining popularity and is the Operating System (OS) of choice in many embedded applications, emphasizing an arousing need for a sound CAN solution for this OS.

CAN is a differential serial two (or three, including a common ground, which is highly recommended) wire bus with deterministic medium access algorithm and non-destructive packet arbitration. This makes it ideally suitable for time-critical applications, where communication bandwidth is not the prime factor, while strictly defined real-time properties are vital. A typical CAN-based network consists of multiple simple microcontroller-based nodes and one or a few of more sophisticated nodes, which implement higher-level control algorithms (as is shown in some of the case-studies in section V). These more complex nodes usually involve some kind of Operating System, often being based on Linux.

There are several CAN drivers available for Linux, but one of them clearly stands out, setting a standard - the SocketCAN [1], primarily developed by Volkswagen Research. This driver is already included in the Linux kernel mainline, making it the obvious choice among developers. SocketCAN supports a variety of most common CAN controllers from different manufacturers, making it suitable for a large pool of hardware platforms.

SocketCAN is built around standard Linux networking infrastructure, which is the main factor determining its inherent assets and shortcomings. The main asset of this approach is that SocketCAN is simple to use, making the CAN communication no different to normal TCP/IP communication from the Application Interface (API) point of view. The very same functions are being used to handle the device and datapackets, the only notable difference being the data-holding structure, which includes the header of the CAN datapacket and is restricted to eight bytes of data.

The drawback of this architecture is that it is hard to achieve desired real-time properties and reliability. SocketCAN is using the same data handling infrastructure as Ethernet and other networking devices (Bluetooth etc.), which makes it prone to negative influence by other communication (as is explained in section III-B).

The Linux networking infrastructure is designed with the goal of achieving the highest possible data throughput for IP and similar high-bandwidth traffic, which is almost opposite to the purpose of CAN, designed for low-bandwidth communications with guaranteed precise timing. One can therefore ask whether and how the usage of Linux networking layer influences communication latencies and reliability in case of SocketCAN.

LinCAN [2] [3] is being developed by the authors of this paper and other contributors since 2003 at the Department of Control Engineering of the Czech Technical University of Prague. It is a brainchild of Pavel Píša, who is also the lead developer of the project. It was designed with a different philosophy in mind. As well as SocketCAN, it is a versatile driver, supporting many different CAN controllers, but the emphasis was put on strictly defined real-time properties and reliability, sacrificing common API to a little extent.

LinCAN is a character device driver with its own queuing infrastructure, completely independent on other devices, designed for specific needs of time-critical CAN communication. The obvious setback is that the common Linux networking interface cannot be used. In order to provide a simpler API for LinCAN and, above all, a common API for various CAN drivers, VCA (Virtual CAN API) library was developed along with LinCAN in the course of OCERA (Open Components for Embedded Real-time Applications) project [4]. This library is compatible with SocketCAN and other drivers and provides a desirable common API, making applications independent on any specific CAN driver. It is therefore possible to convert an VCA-based application from SocketCAN to LinCAN (or vice versa) merely by changing a single line in the makefile, or even to compile both options in and switch among SocketCAN and LinCAN in run-time.

LinCAN was also designed to be easily portable to other Operating Systems (for example RTEMS [5] - a RTEMS port of LinCAN is included in the plans for future development), or even to be used directly as a part of the firmware for simple system-less CAN nodes (which is already implemented, as mentioned in section V).

Both drivers, SocketCAN as well as LinCAN, are open-source. LinCAN is available under the GNU-GPL 2 or later + exception license, while SocketCAN is available under GNU-GPL 2. Therefore, it is simple to obtain detailed technical information if needed. LinCAN is distributed as an out-of-tree driver and currently there are no plans to push it to the mainline, while the SocketCAN is already included in the mainline kernel.

In order to evaluate the performance of both drivers, we have developed a tool for measuring packet Round-Trip Times (RTT) on CAN and conducted several experiments to evaluate RTT under different system loads and other conditions, focusing on the two drivers mentioned above. The experiments and our findings are presented in section IV.

This paper is organized as follows: In the next section, most significant work, related to ours, is summarized. Section III briefly describes the internal structure of both LinCAN and SocketCAN. Section IV presents our testing methodology and results. Case studies and applications of our LinCAN driver are shown in section V, followed by conclusions.

II. Related Work

There are many papers focused on CAN performance analysis, timing, mean and worst-case latency analysis etc., but as far as we know, there is currently no reliable comparison of Linux CAN drivers, which is very important for embedded system developers who are planning to use Linux in their applications. Our work was mostly inspired by [6], which provides a comprehensive analysis of message response times on CAN. [6] states that one of the main problems causing unpredictable latencies is the fact that vast majority of CAN drivers does not support any kind of preferential treatment of high-priority messages, which could cause high-priority messages having to wait until earlier, lower-priority messages are transmitted. This is partially solved in our LinCAN driver by the possibility to implement priority queues, as described in section III-A. [6] also provides very good statistical analysis of datapacket latencies and testing methodology, which influenced the design of our own test cases.

Our approach was also influenced by works [7], [8] and [9]. [7] introduces an interesting time-triggered communication protocol with flexible time-frames, also providing good timing analysis and determining the major sources of latencies. In [8], another task oriented real-time CAN protocol is presented and evaluated. Scheduling of real-time traffic on CAN using servers with real-time properties is shown in [9]. This paper was interesting mainly because of timing evaluation and testing methodology. Somewhat similar topic is also dealt with in [10] and [11]. These papers are useful to gain a systematic insight into the real-time scheduling and communication problems.

III. Internal Structure of the Drivers

This section gives a very brief insight into the basic functional principles of both drivers, especially the RX and TX paths, which are the major factors influencing the timing properties. More detailed informations are available, along with source codes, at respective project websites [2] and [1].

A. LinCAN

LinCAN driver represents all CAN devices as character devices `/dev/canX`. Application software can use the driver directly through the standard UN*X interface represented by standard set of five functions: `open()`, `read()`, `write()`, `ioctl()` and `close()`. Alternatively, a simple middleware, represented by the VCA library, can be used (which is recommended, because it simplifies the API and also allows for simple portability to other CAN drivers if needed). The heart of the driver is a flexible queuing infrastructure [2], which allows simultaneous driver usage by programs from the userspace, IRQ handlers and RT-Linux programs. This queue is pre-allocated in the memory and both inserting and extracting a member from the queue takes constant time. For every user (i.e. for every corresponding file descriptor in the userspace or RT-Linux application), there is a dedicated queue for message transmission and another one for message reception.

Each queue has a fixed number of preallocated slots to hold received messages or messages waiting for transmission. By default, each queue can hold up to 64 messages. The queuing infrastructure does not support any message ordering (for example by their priority), but there is a possibility to allocate dedicated message queues for different priorities, allowing preferential treatment of high-priority messages. Also, each queue can be assigned an acceptance mask, allowing to filter accepted messages according their identifier.

1) Transmit path: The `write()` system call is used to transmit a message. This call executes the `can_write()` function, which stores the message into corresponding transmit queue and wakes the CAN controller instantly, in order to

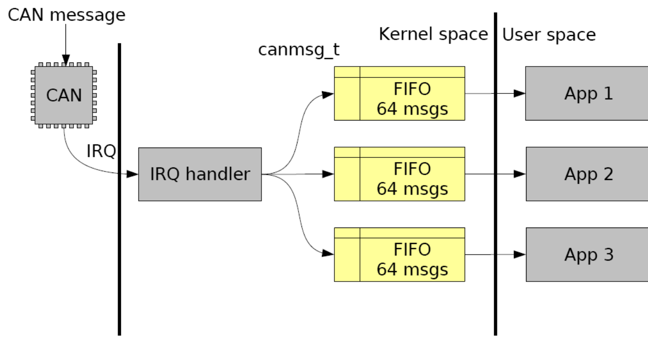

Fig. 1. LinCAN RX path

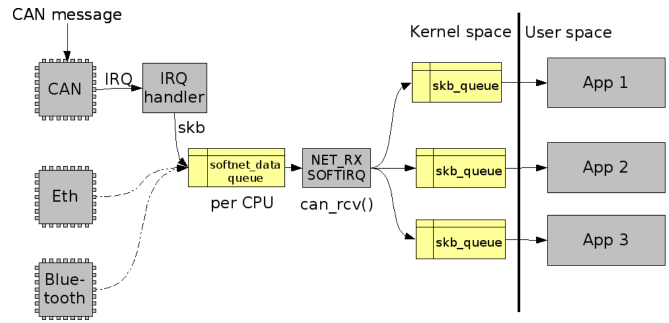

Fig. 2. SocketCAN RX path

proceed. If the controller was already busy (in case there were other messages waiting in the transmit queues), the message is simply added to corresponding queue, to be processed as soon as possible.

2) Receive path: A message reception is heralded by a hardware interrupt. The interrupt handler reads the message from the message box of the CAN controller and stores it directly to all receive queues, connected to the particular device. All applications, waiting in the `read()` system call, are woken up instantly. Important thing is that the message is taken away from the message box directly in the interrupt handler, preventing it from being overwritten by another incoming message (in case of a burst reception of messages). Of course, it is important to assign a reasonably high priority to the interrupt handler. The RX path of the LinCAN driver is shown in Figure 1.

B. SocketCAN

SocketCAN is built around standard Linux networking infrastructure. As of kernel version 2.6.31, most of it is already merged into the mainline. Beside the drivers for most common CAN controllers, SocketCAN also implements several higher-level protocols (BCM, ISO-TP and raw) and provides some useful userspace utilities. The application software can access its functions using BSD sockets, which is a common method in network programming, not limited solely to the UN*X world.

1) Transmit path: Outgoing messages are sent using `sendmsg()` (or similar) system call. This call passes the message to the `can_send()`, function, which stores it into corresponding device queue (by calling `dev_queue_xmit()`). This function uses a queuing discipline (qdisc) to store the message in the TX queue of the driver. The default queue length is 10 messages. When a new message is enqueued, `qdisc_run()` is called to process the queue. If the driver is not busy (by transmitting another message) it can be processed immediately, otherwise it is appended to the qdisc's own queue for corresponding CPU (see `__netif_reschedule()`) and processed later in `NET_TX_SOFTIRQ`.

2) Receive path: Incoming message handling depends on specific chip driver in SocketCAN. The RX interrupt handler either reads the message from the device entirely (for example the *MSCAN* driver) and stores it into the `softnet_data` queue, or (in case of *SJA1000* or *MPC5200*, for example) it only stores a notification into that queue, and the message is read from the device later, by `NET_RX_SOFTIRQ`. In either case, `NET_RX_SOFTIRQ` is called for further processing, placing the messages into corresponding `skb_queues`. It is important to note that the `softnet_data` queue is shared among all network traffic (as shown in Figure 2), including Ethernet, Bluetooth etc. Therefore, Ethernet or other network traffic can significantly affect latencies or even cause a loss of CAN messages (in case of *SJA1000* or *MPC5200* and others, where incoming message it not read directly by the RX interrupt handler, and therefore can be overwritten before it could be read by `NET_RX_SOFTIRQ`). This might seem somewhat irrelevant for embedded applications, but it is worth to note that even in embedded control systems it is not uncommon to use, for instance, CAN and Ethernet simultaneously. For example, in one of the case studies presented in this paper (see section V-B), Ethernet is used to transmit telemetry data to a monitoring station, while CAN is used to interconnect distributed nodes of the embedded control system itself. In this real-life application, both Ethernet and CAN are used extensively and simultaneously.

The problem with many CAN controllers is that their built-in buffers are relatively small in capacity (often capable to handle only a handful of datapackets, like four or five), or even missing at all, so the controller is only able to hold very limited number messages. So if there is a burst of CAN messages, it is very hard for SocketCAN to properly handle all of them, without missing a packet (in case the message is read from the device by `NET_RX_SOFTIRQ`), even when other networking devices are not receiving any data at all. The situation naturally gets worse with increasing data input from other devices, causing SocketCAN to lose datapackets more frequently because they are being overwritten by other packets before they can be read from the device, not to mention any real-time properties, which naturally cannot be guaranteed at all.

It was determined by the SocketCAN developers [12] that the amount of time each received message, or notification of received message (depending on specific implementation), spends in the `softnet_data` queue (including the processing times of the IRQ handler and `NET_RX_SOFTIRQ`) is

about 70% of the whole processing time for each message (the time from message reception until its arrival to the target application). It is clear that this queuing mechanism is the major source of latencies introduced to the process. It is therefore important to mention again that LinCAN does not use this mechanism and takes advantage of its own tailor-made flexible queuing infrastructure to improve its real-time behaviour.

Since CAN is meant primarily for time-critical applications, this property represents a significant setback for the SocketCAN architecture.

IV. Performance Analysis

As can be seen from previous section, both TX and especially RX paths are significantly more complicated in case of SocketCAN (compared to LinCAN), and therefore a natural question arises, whether and how exactly does this affect its performance. This was the main motivation for following series of tests. Because CAN applications usually require precise timing more than anything else, our tests are focused primarily on timing analysis of both LinCAN and SocketCAN.

A. Testing Setup

To answer this question, we have designed a testbed, allowing us to compare both drivers under exactly the same conditions. The testbed was based on a Pentium 4 based PC with a Kvaser PCIcan-Q (also reffered to as Kvaser PCIcan 4xHS) [13] adapter (four-channel PCI card based on SJA1000 CAN controller). The CPU was a single core with Hyper-Threading (HT).

There is an “official” Linux driver available for the Kvaser PCIcan-Q adapter, but it was excluded from the tests because is seems quite antiquated (last revision is dated back to 2005) and it is a problem to even compile it with the latest kernels.

In most experiments, we measured the Round-Trip Time, i.e. the time it takes a message to travel to another node and back again (the other node was, in our case, another channel on the PCIcan-Q card). To measure the RTT, we developed a tool called *canping*, which does exactly what the ordinary Ethernet ping does - sends a message forth and back again, precisely measuring the time in between. Contrary to the Ethernet ping, the *canping* tool is able to process multiple parallel pings simultaneously. The *canping* tool source code is available in our Git repository [14], so anybody interested in our tests is welcome to download it and check our testing methodology in detail, and perhaps recreate our tests using another hardware, which would bring interesting results and verification of our work - any feedback would be highly appreciated.

The *canping* tool sends CAN messages according to command line switches and measures the time, elapsed until receiving a response. The next message is sent after a response to the previous one is received. Measured times are statistically processed and histograms can be generated using these data. To access the CAN driver, *canping* uses our VCA (Virtual CAN API) middleware, which can be obtained from [15].

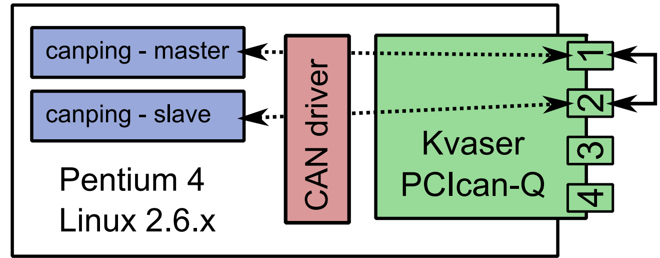

Fig. 3. Testing setup

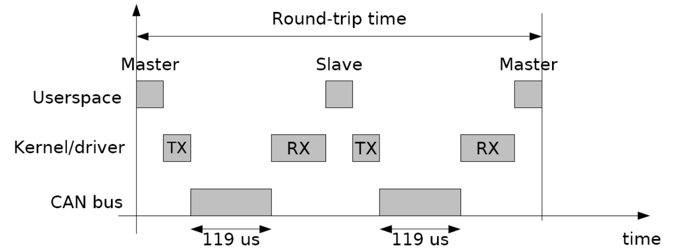

Fig. 4. Timing diagram of the RRT measurement sequence

In its physical form, the testbed was created by simply interconnecting channels no. 1 and 2 of the CAN adapter card. Two instances of *canping* were run – one in the slave mode to provide a “mirror” for incoming messages, generated by the second (master) instance of the program. Both instances were run with real-time priorities and had their entire memory spaces locked (`mlockall()`), in order to prevent swapping. The testbed structure can be seen in Figure 3.

Scheduling policy `SCHEDFIFO` (i.e. the real-time policy) was used for the CAN IRQ handler, as well as for both instances of the *canping* tool. Because the maximum priority of other tasks (including IRQ handlers) was set to 50, both instances of *canping* were issued priority 60 (the higher the number the higher the priority). The CAN IRQ handler was issued a standard priority of 50 in most of the tests, or a “boosted” priority of 60 in some of them (denoted “boosted IRQ prio” in the graphs).

During each experiment, 10 000 messages were sent. The experiments were repeated using different kernel versions, most recent at the time. Both Real-Time and standard kernels were used, to provide an interesting comparison in their influence to message latencies.

CAN baudrate was set to 1 Mbit/s and all messages carried 8 fixed bytes of data during all tests. The time length of all messages was verified using an oscilloscope and confirmed to be $112\,\mu$s, every message being followed by 7 recessive end of frame bits and 3 bits of inter-frame space. Therefore, the shortest (theoretical) time, needed to send the message forth and back, is $2 * (112 + 7) + 3 = 241\mu$s. The whole “pinging” process is purely sequential, as shown in Figure 4. Therefore, it is sufficient to use a single computer as a testbed without introducing any additional delays; “pinging” between two computers would bring no benefits due to inherently sequential character of the process.

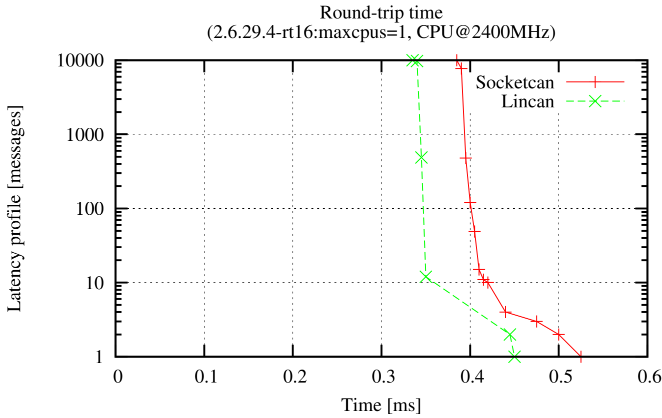

Fig. 5. Round-Trip Time in unloaded system, 2.4 GHz

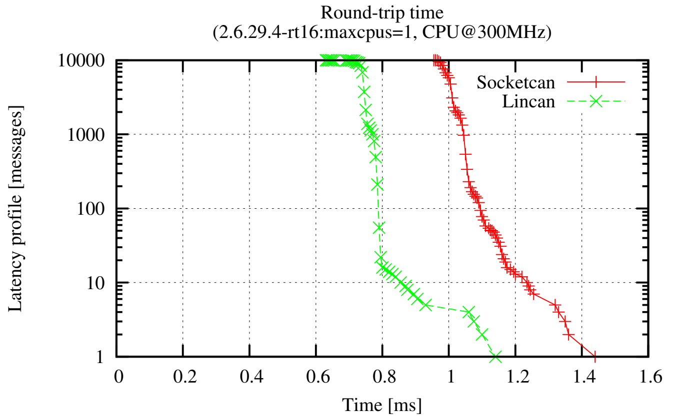

Fig. 6. Round-Trip Time in unloaded system, 300 MHz

The computer didn't run any unnecessary services, such as cron. Also the irqbalance daemon was disabled, so the interrupts were always catch by the first virtual CPU (remember Hyper-Threading). Since there were some differences in the results, depending on whether hyper-threading was used or not, we also performed all tests on kernels with `maxcpus=1` command line parameter (in order to disable Hyper-Threading). Such test are marked by the `:maxcpus=1` suffix to the kernel version.

All tests were run at full CPU clock speed, which was 2.4 GHz, as well as at the lowest possible setting (300 MHz) in an attempt to emulate a typical embedded computing platform, much slower than a desktop PC.

The tests were run on various Linux kernels, stable and experimental alike, the latest available at the time. Real-Time kernels (denoted by the suffix `-rt`) were obtained by applying the rt-preempt patch at a standard, mainline kernel. No other extensions, such as RT-Linux or RTAI, were used. At the time the tests were performed, the latest stable revision of rt-kernel available was 2.6.29, hence this revision was used in most of the tests.

SocketCAN revision 1009 from trunk at SocketCAN SVN was used in all tests. The only exception was the kernel 2.6.31-rc7, for which the version in the mainline could be used, and therefore was adopted. LinCAN version 2009-06-11 was used, which can be obtained from the LinCAN CVS repository [2].

B. Results

The results are displayed using the so-called latency profiles. A latency profile is a cumulative histogram with reversed vertical axis, displayed in logarithmic scale. This way, it is possible to see the exact number of datapackets with worst latencies in the bottom right section of the graph.

The graph in Figure 5 shows measured RTT in an unloaded system. It can be seen that LinCAN exhibits somewhat lower latencies than SocketCAN (both mean and worst case), which is an expected result, due to higher overhead of SocketCAN. Another observation is that the worst-case driver overhead is approximately the same as the message transmission time. For LinCAN, the worst-case overhead is $450 - 241 = 209\,\mu$s, for SocketCAN it is $525 - 241 = 284\,\mu$s.

With the CPU frequency lowered to 300 MHz, the overhead becomes much larger for both LinCAN and SocketCAN (Figure 6).

Since the SocketCAN shares the RX path with other networking devices, the next experiment intends to measure how other network traffic influences CAN datapacket latency. Ethernet load to the CAN testbed was generated using another computer. The Ethernet card was Intel `e100` at 100 Mbps. Following load types were tried: flood ping, flood ping with 64 kB datagrams and TCP load (high bandwidth SSH traffic). It turned out that the worst influence was caused by the flood ping with 64 kB datagrams (as shown in Figure 7). With default system configuration, both drivers exhibit additional latency of approximately 300 μs.

However, with the real-time (`-rt`) kernels, most interrupt handlers run in thread context and it is possible to set the priority of these IRQ threads. Therefore, we increased the priority of the IRQ thread handling the CAN card interrupts (the interrupt line was not shared with any other peripherals). The results are also depicted in Figure 7 (by the lines marked as "boosted IRQ prio"). As can be observed, LinCAN latency decreases to the same values that were measured without any load. On the other hand, SocketCAN latency remains uninfluenced by this change at all. There is a significant difference between the two contenders in this case.

SocketCAN has another parameter that can be used to tune the packet reception – `netdev_budget` sysctl. By this parameter, one can tune how many received packets is processed in a single run of `NET_RX_SOFTIRQ`. The default value of this parameter is 300 and as can be seen again in Figure 7, playing around with this value has no measurable effect on the SocketCAN latency. Further investigation showed that with our tests there were rarely more than 10 pending packets, and since softirqs are re-executed up to 10 times (if there are others pending at the time one finishes execution), even lowering the `netdev_budget` to 1 had no effect.

For the sake of completeness, we also tested whether

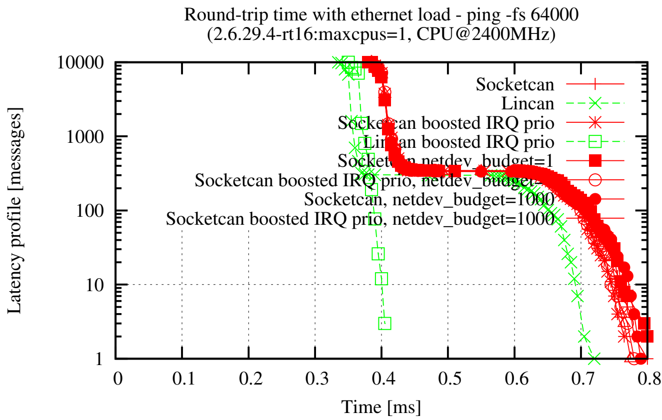

Fig. 7. Round-Trip Time with Ethernet load

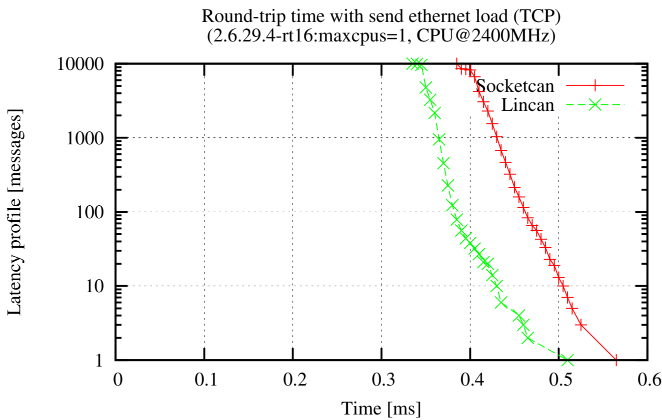

Fig. 8. Round-Trip Time with Ethernet transmission

Ethernet transmission could influence the CAN packet latency. As can be seen in Figure 8, there is almost no influence by this type of load.

Also, there is almost no influence by a non-realtime CPU load, which was generated by the `hackbench` tool [16] (Figure 9).

The worst-case latency was usually very high for both drivers with non-realtime kernels, as can be seen in Figure 10. In this particular case, SocketCAN suffered one 22 ms delay, but in other experiments this happened to LinCAN as well. We didn't investigate the source of this latency in detail, but it was most likely caused by the `ath5k` driver, because after unloading this driver, this latency "peaks" never appeared again.

After "cleaning up" all unnecessary modules from the kernel, the latencies went down for LinCAN, but not so much for SocketCAN (as shown in 11). The reason is that LinCAN does all its RX processing in the hard-IRQ context, while SocketCAN employs a soft-IRQ which is shared by all network devices.

Another item of interest is the performance analysis of virtual CAN drivers. Virtual driver can be used to test CAN ap-

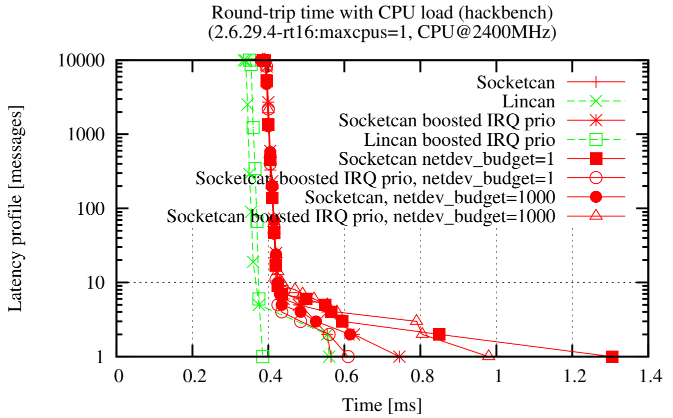

Fig. 9. Round-Trip Time with CPU loaded by *hackbench -pipe 20*

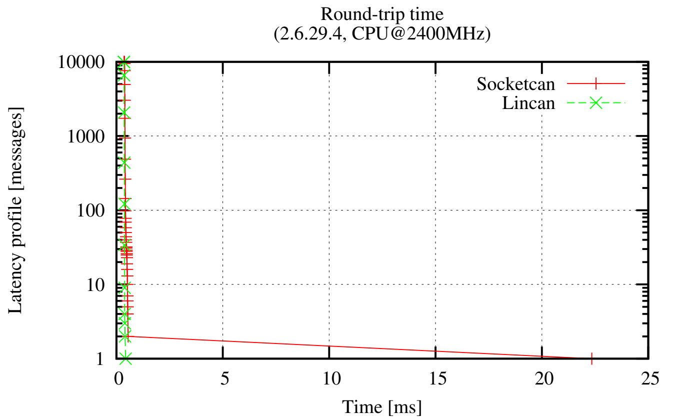

Fig. 10. Round-Trip Time in unloaded system, 2.4 GHz, non-rt kernel

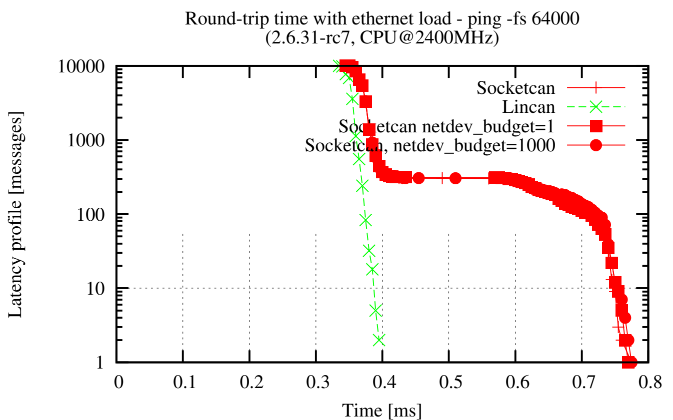

Fig. 11. Round-Trip Time with Ethernet load, 2.4 GHz, non-rt kernel, no unnecessary modules

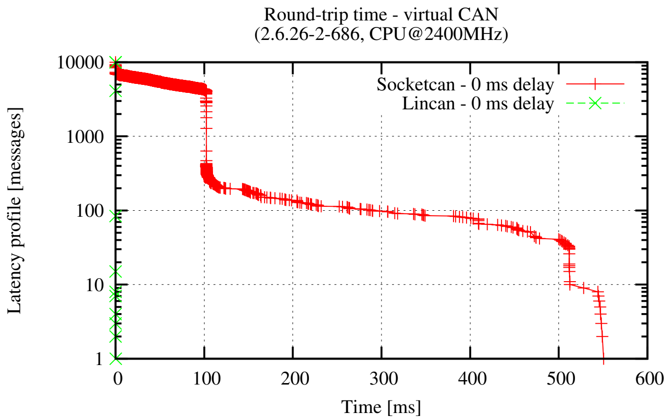

Fig. 12. Round-Trip Time with virtual CAN driver, Debian kernel 2.6.26-2-686

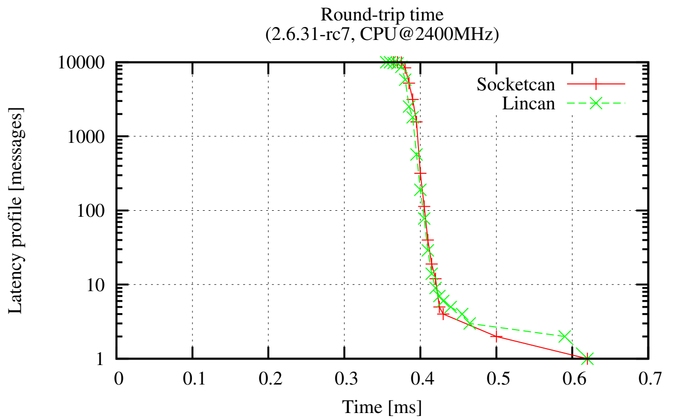

Fig. 14. Round-Trip Time with 2.6.31-rc7 kernel and unloaded system

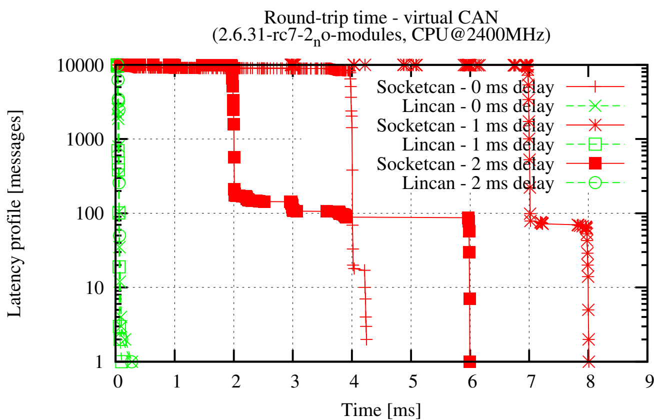

Fig. 13. Round-Trip Time with virtual CAN driver, 2.6.31-rc7 kernel

Fig. 15. The “Regionova” train, running on LinCAN

plications without any real CAN hardware available. It seems there are some problems with implementation of this feature in SocketCAN. Under Debian kernel 2.6.26-2-686, which was configured with `PREEMPT_NONE=y`, SocketCAN latencies raised up to 550 ms (Figure 12), which is a particularly poor performance, inexplainable solely by “normal” SocketCAN overhead.

With 2.6.31-rc7 kernel, the results seem also a bit strange for SocketCAN (Figure 13). For all other kernels the results were more or less as expected - worst-case latencies were in order of several hundreds of microseconds.

It is fair to mention that the Linux network stack is being continuously improved and in 2.6.31-rc7 kernel, the performance of SocketCAN was only slightly worse than that of LinCAN (see Figure 14).

Complete results of all our tests are published at our web page [17]. As was said earlier, anybody interested in our tests is welcome to take them further or reproduce our results. All source codes, configuration files etc. are available in our Git repository [18].

V. Case Studies

LinCAN has many industrial, as well as academical, applications to its credit. As far as we know, it is (or was) successfully used in trains, an ultra-light aircraft, underground gas tanks, an unmanned helicopter, several walking robots, a milling machine and multiple academic projects. There may be more projects involving LinCAN we are not aware of, since it is distributed under relatively free license and nobody using LinCAN is obliged to let us know, although we naturally appreciate any feedback. Let us mention at least some of the most interesting case-studies in this section.

Unicontrols a. s. is the major industrial user of the LinCAN driver. Their most significant application of LinCAN are the new generation of “Regionova” trains (see Figure 15), designed for the Czech Railways. These are small trains intended for short regional routes, and are currently in active service with the Czech Railways (and possibly other users in the future). The Unicontrols company also used LinCAN in a major project on control of underground natural gas tanks. The lenght of the bus in this case is several kilometers.

On-Track s. r. o. is also a locomotive manufacturer, who is using LinCAN in their new-generation of locomotives,

Fig. 16. The ProAir autogyro

Fig. 17. Cocpit view

developed for Serbian Railways. Interestingly enough, the layout of the CAN controllers in this case was optimized for use with LinCAN.

LinCAN is also used in a quite interesting walking robot Spejbl [19], which is a networked hard real-time system, consisting of 14 CAN nodes.

Other industrial users include following companies: MPL AG Elektronikunternehmen, Technologic Systems, Ingeneria Almudi S. L., BFAD GmbH & Co.KG, Applied Biosystems and PiKRON. LinCAN is also used at the Universita degli Studi di Parma, Italy, Universidad Politecnico di Valencia, Spain, LanZhou university, China, Slowak Academy of Sciences and Czech Technical University in Prague, Czech Republic.

Let us introduce two interesting case-studies in more detail. The first one will be on integrated avionics system for ultra-light aircraft and the second on an unmanned rotorcraft. Both projects were held at the Department of Control Engineering of the Czech Technical University in Prague.

A. *Integrated Avionics System for Ultra-Light Aircraft*

The goal of this project [20] was to develop affordable, yet robust and reliable modern avionics system for ultra-light aircraft. Because of relative benevolence of the rules governing ultra-light aircraft design and operation, the certification process of such system is great deal simpler and cheaper compared to the certification of avionics intended for "full-scale" vehicles.

The system was developed by our department specifically for the new-generation ultra-light autogyro produced by a local company ProAir (Figures 16 and 17), but can be easily adopted for any type of aircraft. It is a distributed data acquisition system, providing pilot with important flight data, such as barometric altitude, true altitude above the terrain (in the range of 0-6 meters), variometer, air speed, main rotor RPM, navigation and engine data (engine RPM, water and oil temperature, oil pressure, fuel level etc.). All data are displayed on a Multi-Function Screen (MFS) and some of them are also provided acoustically to the pilot's headphones (for example the true altitude above ground is reported in form of beeps of variable length, to help the pilot handle the landing approach). The system naturally incorporates plenty of safety functions, such as ground proximity warning, steep descent warning, low fuel/oil/water and fuel/oil/water leak warning, exceeding of rotor/engine speed, engine temperature warning etc. All warnings are displayed on the MFS and also reported acoustically into pilot's headphones. The avionics is also equipped with a self-diagnostic system and Flight Data Recorder.

Fig. 18. A RAMA-based rotorcraft in flight

The avionics is designed as a distributed, hierarchical data acquisition system, with independent intelligent sensors distributed around the airplane, connected to CAN, and a central node, called the Main Flight Computer, providing high-level services. LinCAN, along with VCA, is used as the CAN interface middleware in the Main Flight Computer. At the time when this application was designed, no other driver for Linux was readily available, not to mention proven. LinCAN already had two years of development and testing to its credit, and was thoroughly tested and known to work flawlessly on the selected hardware, making it the obvious choice.

B. *RAMA - an Open-Source UAV Control System*

The RAMA system [21], [22], [23] (Figure 18), developed at our department, is an open-source universal control system for Unmanned Aerial Vehicles (UAVs), which could be fitted to any kind of UAV (either a rotorcraft or a fixed-wing aircraft). As an academic project, it is designed primarily for research applications.

RAMA consists of the *Airborne Part* (AP) and the *Ground Station* (GS), comprised of a laptop computer and a RC (Radio Control) transmitter. The laptop is used only to visualize and

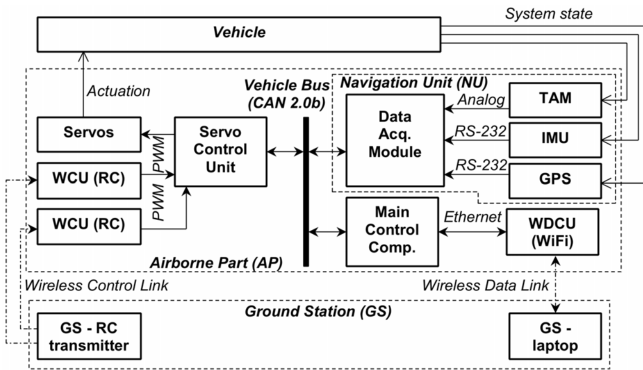

Fig. 19. Control system block diagram

record the on-line telemetry of the vehicle, while a model RC set is used to issue commands to the airborne control system, and also allows one to re-take full manual control of the drone in case of a control system failure. The airborne part of the system is designed as a networked hierarchical distributed system. It consists of several basically independent functional blocks, interconnected via the *Vehicle Bus* (VB), as can be seen from the block diagram (Figure 19). Those blocks are the *Navigation Unit* (NU) (comprised of sensors and their management), *Main Control Computer* (MCC), *Wireless Data Communication Unit* (WDCU), two *Wireless Control Units* (WCU) (for redundancy) and the *Servo Control Unit* (SCU).

This structure allows a separate design and testing of each component, also simplifying any possible future extensions of the system. The nodes may be added or exchanged without affecting the rest of the system very much, as all nodes are interconnected only via a single interface and a single medium, the Vehicle Bus. Of course, integration work and testing must follow any modification, but it is simplified by a great deal due to the modular structure of the system.

The core member of the RAMA system is the Main Control Computer (MCC), where the control and communication algorithms run. It is based on the MPC5200 processor and runs Linux 2.6.29-rt kernel with busybox. The Main Control Computer is connected to the Ground Station via the Wireless Data Communication Unit (WDCU), which is currently an *IEEE 802.11g* (WiFi) module. The WiFi communication is non-critical, as it is used only for the telemetry. The commands for the autopilot are issued via the Wireless Control Units (WCU) (currently a model RC set), which is much more reliable and has a lot wider range than WiFi. RAMA is able to operate without the data communication with the Ground Station (GS), as it serves only for the on-line system monitoring and telemetry recording.

The sensor data are acquired and pre-processed by the *Navigation Unit* (NU). This unit includes the *Inertial Measurement Unit* (IMU), *Three-Axis Magnetometer* (TAM), *Global Positioning System* (GPS) receiver and the *Data Acquisition Module* (DAM). All sensors are connected to the DAM, the purpose of which is to synchronously sample all the data, pre-process it (filtering, unit conversion, etc.) and send it to other control system nodes. DAM also provides the *Time Synchronization Message* (TSM) for the rest of the system, the purpose of which is to synchronize all system nodes.

The actuators (the servomotors driving the control surfaces of the vehicle) are controlled by the *Servo Control Unit* (SCU). All actuators and both Wireless Control Units are connected to this node. Its purpose is to control the actuators, sample their positions and also to sample the control sticks' movement on the RC transmitter (provided by the signals from the Wireless Control Units).

The *Controller Area Network (CAN) 2.0b* [24], running at 1 Mbps, is used as the Vehicle Bus. Its utilization is approx. 15%, although this mean number is somewhat misleading. RAMA is designed as a time-triggered system, with all system nodes sampling and transmitting data at the same time point, determined by a time synchronisation message. This results in a strongly uneven bus utilization, when the bus is inactive most of the time, but in each sampling point a burst of messages is sent from various nodes, resulting in a momental 100% bus utilization.

The Wireless Data Communication Unit (WDCU) is a standard embedded WiFi device, configured as an access point. The Ground Station laptop is the client - multiple Ground Stations are allowed, the RAMA system has an telemetry server and is able to broadcast data to multiple clients simultaneously. The communication data rate is 54 Mbps, which is more than enough for the telemetry, which is a soft real-time task anyway. The communication is encrypted, using the standard 128-bit WEP algorithm.

The RAMA project originally used SocketCAN as the CAN driver for the MCC (because LinCAN was not available for MPC5200 at the time), but persistent problems with occasional packet loss made us to find an alternate solution. It was found that these losses were caused by the CAN datapacket bursts, which occur in every sampling point, when all system nodes are feverishly transmitting new data.

This issue couldn't be cured even by assigning the highest real-time priority to the CAN controller interrupt or in any other way, and it only got worse as the Ethernet traffic increased. The problem is that the SocketCAN driver for MPC5200 is implemented so that in the interrupt handler, only notification of an incoming message is stored into the `softnet_data` queue, while the message itself is extracted from the hardware buffer of the CAN controller only after that notification gets processed by `NET_RX_SOFTIRQ`. This is probably done in order to reduce the size of the `softnet_data` queue, but proved to be a rather unfortunate solution. Since the hardware buffer of the controller can only hold five messages, a burst of six or more messages at 1 Mbit/s rate is likely to cause a data loss, because the driver is unable to process incoming data at sufficient rate, which eventually leads to the hardware buffer overflow.

Therefore, it was decided to migrate the project to LinCAN, which required implementing the LinCAN driver for MPC5200. This was done and successfully tested by running similar tests to these shown in section IV on MPC5200

platform. LinCAN emerged as a clear winner and the RAMA project migrated to VCA and LinCAN instead of Socket CAN. Thanks to the VCA middleware, it is still possible to migrate back to SocketCAN merely by changing a single parameter in the makefile, which might be convenient in the future. The datapacket losses never occurred again since the migration to LinCAN. This is because the LinCAN driver extracts each incoming message from the hardware buffer right in the interrupt handler, preventing the buffer overflows suffered by the SocketCAN.

VI. Conclusions

The experiments, shown in this paper, demonstrated that the character-device based LinCAN driver, with its dedicated infrastructure, specially designed for critical real-time applications, is clearly superior to the socket-based SocketCAN driver in all respects. The latest evolutions of SocketCAN are clearly improving and the results could be very close, as the last test demonstrated. In some cases, there can be some reliability concerns with SocketCAN, because it can lose datapackets under certain circumstances (as was shown in section V-B).

Overhead caused by the usage of standard Linux networking infrastructure in SocketCAN was quantified. Our conclusion is that the difference could be significant and using SocketCAN for a hard real-time application could be tricky, although it can be suitable for many other applications. On the other hand, LinCAN driver with properly assigned real-time priority is perfectly suitable for critical real-time applications, and is used so in many applications (as was demonstrated by the case-studies).

This work can also be useful for following purposes:

1) Users can approximately assume timing properties of a particular CAN-on-Linux solution.
2) Testing tools, developed in the course of this work, can be used for benchmarking other hardware and to determine suitability of a particular CAN solution for any intended application.
3) Driver developers can test and improve their drivers using our testing tools and find factors contributing to the worst-case delays in RX/TX paths.

Acknowledgments

This work was supported by the Ministry of Education of the Czech Republic under projects 1M0567 and ME10039.

References

[1] "The SocketCAN poject website," http://developer.berlios.de/projects/socketcan.

[2] P. Píša, "Linux/RT-Linux CAN driver (LinCAN)," 2010. [Online]. Available: http://freshmeat.net/projects/lincan

[3] Pavel Píša, "LinCAN technical report," 2005. [Online]. Available: http://cmp.felk.cvut.cz/ pisa/can/doc/lincandoc-0.3.pdf

[4] "Open Components for Embedded Real-time Applications (OCERA)," 2006. [Online]. Available: http://www.ocera.org

[5] "Real-Time Executive for Multiprocessor Systems (RTEMS)," 2010. [Online]. Available: http://www.rtems.com

[6] H. Zeng, M. D. Natale, P. Giusto, and A. Sangiovanni-Vincentelli, "Statistical Analysis of Controller Area Network Message Response Times," in *IEEE Symposium on Industrial Embedded Systems*. Lausanne, Switzerland: Ecole Polytechnique Federale de Lausanne, 2009.

[7] L. Almeida, J. Fonseca, and P. Fonseca, "A flexible time triggered communication system based on the controller area network," in *Proceedings of the FeT'99Fieldbus Systems and their Applications Conference*, Germany, 1999.

[8] J. L. Campos, J. J. Gutierrez, and M. G. Harbour, "CAN-RT-TOP: Real-Time Task-Oriented Protocol over CAN for Analyzable Distributed Applications," in *In 3rd International Workshop on Real-Time Networks (formerly RTLIA)*, Catania, Sicily (Italy), 2004.

[9] T. Nolte, M. Nolin, and H. Hansson, "Real-time server-based communication for CAN," *IEEE Transactions on Industrial Informatics*, vol. 1(3), pp. 192–201, 2005.

[10] P. Balbastre, I. Ripoll, and A. Crespo, "Minimum Deadline Calculation for Periodic Real-Time Tasks in Dynamic Priority Systems," *IEEE Transactions on Computers*, vol. 57, no. 1, pp. 96–109, 2008.

[11] G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic Scheduling for Flexible Workload Management," *IEEE Transactions on Computers*, vol. 51, no. 3, pp. 289–302, 2002.

[12] "Socketcan users mailing list," http://old.nabble.com/Re%3A-Socketcan-and-LinCAN-benchmarks-p25256685.html.

[13] "Kvaser Company web site," 2010. [Online]. Available: http://www.kvaser.com

[14] P. Píša, "canping tool Git repository," 2009. [Online]. Available: http://rtime.felk.cvut.cz/gitweb/canping.git

[15] P. Píša, M. Sojka, and F. Vacek, "Virtual CAN API middleware web site," 2006. [Online]. Available: http://ocera.cvs.sourceforge.net /viewvc/ocera/ocera/components/comm/can/canvca/libvca/

[16] "The hackbench tool project website," 2009. [Online]. Available: http://devresources.linux-foundation.org/craiger/hackbench

[17] M. Sojka, "Testing results website," 2009. [Online]. Available: http://rtime.felk.cvut.cz/can/benchmark/1/

[18] M. Sojka, "CAN benchmark Git repository," 2009. [Online]. Available: http://rtime.felk.cvut.cz/gitweb/can-benchmark.git

[19] M. Peca, M. Sojka, and Z. Hanzálek, "Spejbl – The Biped Walking Robot," in *7th IFAC International Conference on Fieldbuses and networks in industrial and embedded systems*. Toulouse: Universite Toulouse, 2007, pp. 63–70.

[20] O. Špinka, J. Krákora, M. Sojka, and Z. Hanzálek, "Low-Cost Avionics System for Ultra-Light Aircraft," in *11th IEEE International Conference on Emerging Technologies and Factory Automation Proceedings, Prague, Czech Republic*, 2006, pp. 102–109.

[21] O. Špinka, O. Holub, and Z. Hanzálek, "Low-Cost Reconfigurable Control System for Small UAVs," *IEEE Transactions on Industrial Electronics*, vol. 56, 2010.

[22] O. Špinka, Š. Kroupa, and Z. Hanzálek, "Control System for Unmanned Aerial Vehicles," in *5th IEEE International Conference on Industrial Informatics, Vienna, Austria*, vol. ISBN 1-4244-0864-4, 2007, pp. 455–460.

[23] O. Špinka, "RAMA, an open-source UAV Autopilot - Project website," http://rtime.felk.cvut.cz/helicopter, 2007.

[24] K. Etschberger, *Controller Area Network*. IXXAT Press, 2001.