UART-based LIN-bus Support for Linux with SocketCAN Interface

Pavel Píša[1]
pisa@cmp.felk.cvut.cz
Rostislav Lisový[1]
lisovy@gmail.com
Oliver Hartkopp[2]
oliver.hartkopp@volkswagen.de
Michal Sojka[1]
sojkam1@fel.cvut.cz

[1] Czech Technical University in Prague, Department of Control Engineering
Karlovo náměstí 13, 121 35 Praha 2, Czech Republic

[2] Volkswagen Group Research
Brieffach 1777, 38436 Wolfsburg, Germany

Abstract

The LIN-bus (Local Interconnect Network) is a vehicle bus standard or computer networking bus-system used within current automotive network architectures to control slave peripherals for which CAN bus is too expensive or complex. Concept of LIN frames and identifiers has its roots in CAN-bus however data bytes are serialized in asynchronous serial communication format usual for UARTs.

The article focuses on LIN-bus support implementation and integration into Linux based systems with attempt to offer portable solution with minimal hardware dependencies. This is possible thanks to the compatibility of common UART hardware with LIN-bus serial format.

The developed `slLIN` protocol driver is implemented as a Linux TTY line discipline and uses only common Linux UART serial line discipline API. The solution does not require to implement specialized driver for each architecture or serial interface hardware and is highly portable.

The interface from application to `slLIN` is based on CAN protocol family network API (same as `SocketCAN` uses). This approach was chosen because LIN-bus is usually found together with CAN-bus infrastructure in applications.

The portability of the implementation has been tested on common PC serial port and MPC5200 hardware against third party fully-functional LIN device. A utility for frame sequences configuration is also part of the implementation.

1 Introduction

Main focus of the work is to provide portable implementation of the LIN-bus (Local Interconnect Network) support for Linux based systems.

The article starts with short overview of the LIN-bus origin, concept and relation to Controller Area Network (CAN) communication infrastructure. Thereafter technical terms used by standard are defined.

As shown later, it is possible to use serial port (UART) combined with a simple logic level converter to interface with the LIN bus.

Next section describes a Linux LIN-bus driver (`slLIN`) implemented as serial line discipline which can be attached to most of low level UART drivers already available for all Linux kernel supported architectures.

Interoperability tests of the designed solution are presented and analyzed. Full interoperability has been achieved when master task is controlled by `slLIN`.

The obstacles to implement standalone slave node by `slLIN` without Linux UART drivers API extension is analyzed and goals for future work in this area are described.

2 LIN-bus (Local Interconnect Network)

The LIN-bus (Local Interconnect Network) is a low cost vehicle bus standard which has been designed to provide simple and low cost bidirectional communication infrastructure for simple peripherals connected to ECUs (Electronic/Engine Control Units). It has been standardized to complement CAN-bus based interconnection of more complex ECUs.

Controller Area Network (CAN) is networking standard originally developed for in vehicle ECUs interconnection. It is dominating communication bus solution in current vehicles designs and most of the contemporary microcontrollers targetting industrial or automotive area integrate a CAN controller. The number of inputs and outputs from each ECU grows significantly as number of peripherals (end-point actuators and sensors) connected to them increases. I.e. connection of knobs on a steering wheel, lights. The use of bus technology is logical solution to tame the wires pile up. However using of CAN equipped microcontroller in these peripherals would be too expensive.

The LIN-bus is response to the demand for low cost peripherals bus. Its introduction and the first standart preparation goes back to late 1990s. Actual standard revision is freely available from LIN Consortium[1].

The bytes are serialized on the wire same as common asynchronous serial communication does (i.e. RS-232). Such serial communication is supported even by the cheapest microcontrollers. LIN data transfers are organized to frames which content is differentiated by 6-bit identifier. Identifier is part of the frame header which is followed by up to 8 data bytes and check sum byte. Only one node schedules and broadcasts frame headers in a given LIN-bus network. But responses containing data bytes and check sum can be generated by any node including node sending header. Thus network data interchange is similar to CAN-bus except a limitation that all communications order and frames sequence is controlled/(time) triggered by single node. But it is usually desired setup for couple of simple peripherals connected to a single superordinate ECU unit. Mechnisms to allow wake master node by peripheral node is defined in actual standard as well but these topics are out of the scope of this paper and `slLIN` driver implementation.

The frame header starts with break character. Break character consists of logic 0 transmission for time longer than 11 bit transfer intervals which is longer than logic 0 steady state for all other regular start bit (0) and stop bit (1) delimited serial line 8-bit characters. Break character is recognized by all units regardless of their notion of actual transfer phase and that way the synchronization of all units and reset of the incorrect state caused by lost bit or byte is ensured.

The header then follows by synchronization character 0x55 and already mentioned 6-bit identifier protected by two parity bits.

2.1 Glossary

This section introduces the terms which are essential to understand this document. More complete glossary can be found in LIN specification[1].

Cluster
A cluster is defined as the LIN bus wire and all the *nodes* connected to it.

Frame
All transmitted information is packed into frames; a frame consist of a *header* and a *response*.

Header
A header is the first part of a *frame* and contains a *protected identifier*. It is always sent by the *master task*.

Master node
The master node is a *node* that contains a *master task*. Besides that, master node also contains zero or more *slave tasks*.

Master task
The master task is responsible for sending all *headers* on the bus, i.e. it controls the timing

on the bus. Its implementation is usually based on a schedule table.

Note: The current slLIN's task model does not correspond to the task model of LIN specification. In slLIN, it is possible to have multiple Linux tasks (or processes) that serve as master tasks.

Node
Loosely speaking, a node is a LIN device.

Protected identifier
An eight-bit value containing the frame identifier together with its two parity bits.

Response
Response is the second part of the frame. It is transmitted after the *header*.

Slave node
A node that contains *slave task(s)* only, i.e. it does not contain a *master task*.

Slave task
The slave task listens to all *headers* on the bus. Depending on the identifier of the received header it either publishes a frame *response*, or it receives the responses published by another slave task, or it ignores the header.

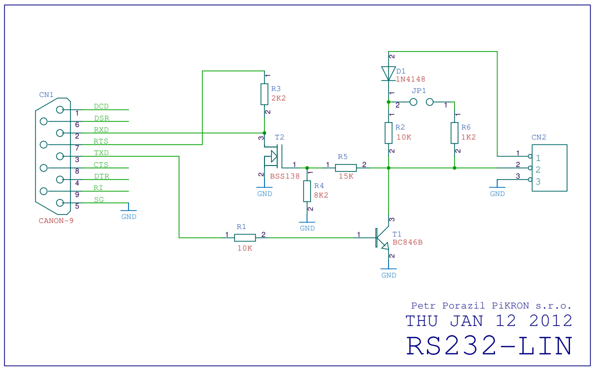


FIGURE 1: *RS232 to LIN logic level converter*

2.2 LIN Physical Layer and Interfacing to the UART

The physical LIN layer is as simple as possible. It uses only single wire. Logic 0 level of serialized bits is mapped to low voltage on wire which is achieved by connection of the link to the ground by the transmitter. Voltage level corresponding to logic 1 level is held on link by pull-up resistor connected to an ECU power supply.

The trasmitter can be implemented as single transistor. Logic level receive requires single comparator with treshold around half of the supply voltage for minimal setup. More sophisticated trensceivers (i.e. TJA1021T) are usually used due to faults detection and wakeup function requirements.

The aim of the presented work is to provide simple solution – that is why a simple logic level converter for interfacing LIN-bus to RS232 port has been designed. A schematic of the converter is shown in Figure 1.

3 SocketCAN and TTY Line Discipline Based LIN Driver

As described in the previous section, LIN-bus combines UART style byte transfers with CAN style data framing and identification. Because of these similarities it is quite logical to think about LIN support integration into existing subsystems (i.e. CAN or UART) of the Linux kernel.

The standard Linux API for CAN based communication is SocketCAN CAN protocol family (PF_CAN) subsystem and associated CAN controller drivers. The SocketCAN description can be found in actual Linux kernel mainline as well as in articles [2] and [3].

LIN-bus driver can be implemented as SocketCAN compatible controller driver for specific UART hardware. That would allow to achieve optimal implementation for given UART hardware and utilize special LIN extended functionality if provided by hardware. However this solution has disadvantage that the driver has to be ported to each individual serial interface/UART type. Other problem is that there is registered driver for most of the serial interface types and driver unbind would be required before LIN specific driver can be used.

slLIN approach is not to implement alternative UART driver but to reuse already existing UART drivers. The low level driver API is available to modules which implement (own/new/alternative) serial/TTY line discipline. Discipline can be selected runtime accordind to intended use of the UART instance.

3.1 TTY Line Discipline

TTY line discipline is a code that implements a specific protocol on an UART interface. The TTY line discipline interacts with the Linux TTY (terminal) subsystem.

To use the protocol, the line discipline needs to be attached to a TTY by passing its identifier (defined in `include/linux/tty.h`) to `ioctl(fd, TIOCSETD, &tty_disc_nr)` system call.

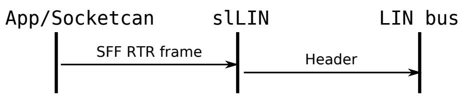
4 slLIN Setup and Application Interface

From high-level view `slLIN` operates as follows. After loading the kernel module and attaching the TTY line discipline to an existing UART interface, a new network interface, e.g. `sllin0`, is created (note that the number of the created `slLIN` interface may be different). From the application's point of view, this interface presents the traffic received from LIN bus (i.e. UART RX) as CAN traffic and transforms the CAN frames sent to it by applications into LIN frames on the bus.

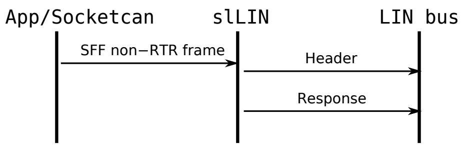
4.1 Master Mode

In Master mode, `slLIN` operates according to the following rules. Each rule is illustrated with a simple sequence diagram.

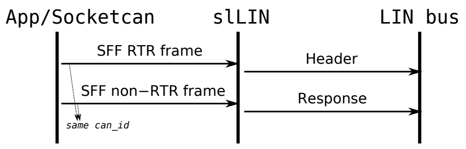
1. LIN header is sent to the LIN-bus after receiving SFF RTR CAN frame from an application. (*LIN id = can_id*).



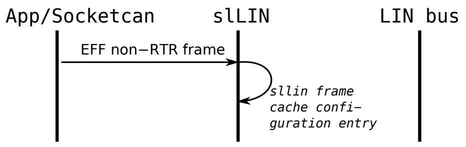
2. LIN header immediatelly followed by LIN response is sent to the LIN bus after receiving SFF non-RTR CAN frame from an application.
(*LIN id = can_id; LIN response = can_frame.data*).



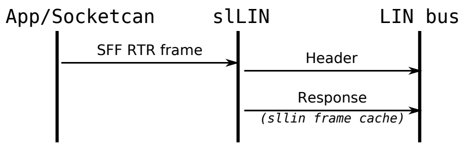
3. LIN response is sent to the LIN-bus (LIN-header is sent due to reception of SFF RTR CAN frame) after receiving SFF non-RTR CAN frame.
(*can_id of both frames must be the same; LIN response = can_frame.data*).



4. A frame is stored in a *frame cache* (see Section 4.4) after receiving EFF non-RTR CAN frame. This operation is controlled by the flags in `can_id` of the frame.

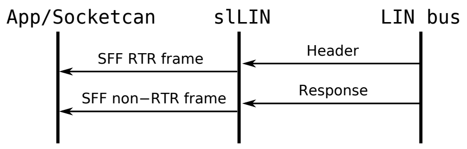


5. LIN response from correctly configured *frame cache* is sent to the LIN-bus upon sending the LIN header due to the reception of SFF RTR CAN frame.



4.2 Slave Mode

Slave mode enables monitoring of the LIN-bus which means that intercepted LIN frames are sent to `sllin0` interface in the form of CAN frames. Currently, slave mode is not finished and more functionality needs to be added to use it for implementation of real LIN slave tasks.



4.3 Error Reporting

Errors from `slLIN` are reported to applications by sending CAN frames with flags which are part of `can_id`. List of the individual error flags follows (they are also defined in `linux/lin_bus.h`).

`LIN_ERR_RX_TIMEOUT` Reception of the LIN response timed out

`LIN_ERR_CHECKSUM` Calculated checksum does not match the received data

`LIN_ERR_FRAMING` Framing error

4.4 Frame Cache

`slLIN` integrates a so called *frame cache*. For each LIN ID, it is possbile to store up to 8 bytes of data. Frame cache is currently used in Master mode only, but it is planned to be used in slave mode as well.

`slLIN` can send LIN response based on the data stored in the frame cache immediately after transmission of the LIN header. This can be configured for each LIN ID separately by sending EFF CAN frames where CAN ID consists out of LIN ID and cobination of following flags:

`LIN_CACHE_RESPONSE` Sets that slave response will be sent from frame cache

`LIN_CHECKSUM_EXTENDED` Sets extended checksum for LIN frame with particular ID

Example To store `0xab` data byte to be used as LIN response for LIN ID 0x5, it is necessary to send EFF (i.e. LIN configuration) non-RTR CAN frame with CAN ID set to `0x5 | LIN_CACHE_RESPONSE` and data `0xab`.

4.5 Communication Examples

SocketCAN project provided `cangen` utility `slLIN`

SFF RTR CAN frame, LIN response from PCAN-LIN slave

```
$ cangen sllin0 -r -I 1 -n 1 -L 0

$ candump sllin0
  sllin0    1  [0] remote request
  sllin0    1  [2] 00 00
```

SFF non-RTR CAN frame

```
$ cangen sllin0 -I 7 -n 1 -L 2 -D f00f

$ candump sllin0
  sllin0    7  [2] F0 0F
  sllin0    7  [2] F0 0F
```

SFF RTR CAN frame without response (ERR_RX_TIMEOUT)

```
$ cangen sllin0 -r -I 8 -n 1 -L 0

$ candum sllin0
  sllin0    8  [0] remote request
  sllin0      2000  [0]

$ ip -s link show dev sllin0
14: sllin0: <NOARP,UP,LOWER_UP> mtu 16 ... link/can
    RX: bytes  packets  errors  ...
    2          4        1       ...
    TX: bytes  packets  errors  ...
    0          4        0       ...
```

EFF non-RTR CAN frame to configure frame cache

```
# (LIN_CACHE_RESPONSE | 0x8) == 0x108
$ cangen sllin0 -e -I 0x108 -n 1 -L 2 -D beef
$ candump sllin0
  sllin0       108  [2] BE EF

# Try RTR CAN frame with ID = 8 again
# (there is no active slave task)
$ cangen sllin0 -r -I 8 -n 1 -L 0
$ candump sllin0
  sllin0    8  [0] remote request
  sllin0    8  [2] BE EF
```

Slave mode

```
$ insmod ./sllin.ko master=0
$ ...
$ candump -t d sllin0
 (000.000000)  sllin0    2  [0] remote request
 (001.003734)  sllin0    1  [0] remote request
 (000.000017)  sllin0    1  [2] 08 80
 (000.996027)  sllin0    2  [0] remote request
 (001.003958)  sllin0    1  [0] remote request
 (000.000017)  sllin0    1  [2] 08 80
 (000.996049)  sllin0    2  [0] remote request
 (001.003930)  sllin0    1  [0] remote request
 (000.000016)  sllin0    1  [2] 08 80
```

There is no response (only RTR CAN frame) to the fist LIN header with ID 2 in the sequence. The second LIN header (RTR frame with ID 1) is followed by a response (non-RTR CAN frame with the same ID). The schedule repeats again with ID 2 from this point.

4.6 Configuration

A dedicated utility (`lin_config`) was developed to simplify `slLIN` configuration. It is able to:

1. Attach `slLIN` line discipline to particular UART device
2. Configure BCM (SocketCAN Broadcast Manager) to periodically send LIN headers (according to LIN schedule table)
3. Configure `slLIN` frame cache

The configuration is obtained from an XML file. The format of this XML file is the same as the one generated by the official PCAN-LIN configuration tool.

The described utility is also able to configure the PEAK PCAN-LIN device (from the same XML configuration file).

The usage is as follows:

`./lin_config` [*OPTIONS*] *SERIAL_INTERFACE*

SERIAL_INTERFACE is only mandatory argument. It selects a target serial/LIN interface to configure. The format of the argument is *CLASS* : *PATH* where

CLASS
: defines the device class – it is either `sllin` or `pcanlin` (when not set, default is 'sllin')

PATH
: is path to the serial interface, e.g `/dev/ttyS0`

The next general options are recognized:

-c *FILE*
: Path to XML configuration file in PCLIN format If this parameter is not set, file 'config.pclin' is used

PCAN-LIN specific options:

-f
: Store the active configuration into internal flash memory

-r
: Execute only reset of a device

slLIN specific options:

-a
: Attach sllin TTY line discipline to particular `SERIAL_INTERFACE`

Examples

Configure the device with the configuration from `config.pclin`

```
./lin_config sllin:/dev/ttyS0
```

Reset the device

```
./lin_config -r pcanlin:/dev/ttyS0
```

After invoking `lin_config` and successful configuration of `slLIN`, the configuration utility switches to background and runs as a daemon. This behaviour is necessary because of the preservation of the BCM and TTY line discpline configuration. To detach the `slLIN` line discipline, it is necessary to kill the running daemon.

5 Tests

`slLIN` was developed and tested on IBM PC compatible computer, however its proper functionality was also tested on MPC5200-based (PowerPC) embedded board. Results of our tests are reported in the following sections.

5.1 Communication with of-the-shelf LIN Devices

Proper behavior in a real-world environment was tested in conjunction with PCAN-LIN device. PCAN-LIN was configured by using the tools delivered with the device. Two different setups were used:

- PCAN-LIN as Slave node, `slLIN` in Master mode – in this setup PCAN-LIN correctly responded to LIN headers sent by `slLIN`.
- PCAN-LIN as Master node, `slLIN` in Slave mode – PCAN-LIN device in master mode was sending LIN headers and LIN headers with corresponding LIN responses. `slLIN` was reading this traffic and converting to CAN frames.

5.2 Proper Timing Verification

Timing properties of `slLIN` were observed on an oscilloscope. Waveforms captured on the LIN-bus are shown in Figures 2 and 3. It can be seen that there are no extensive delays caused by problems in `slLIN` implementation. Note, however, that these experiments were conducted on otherwise unloaded system. Huge amount of test has been run even on i686 PC where data interchange has been run with Peak's PCAN-LIN device as slave which can monitor and detect incorrect timing inside frame transfers.

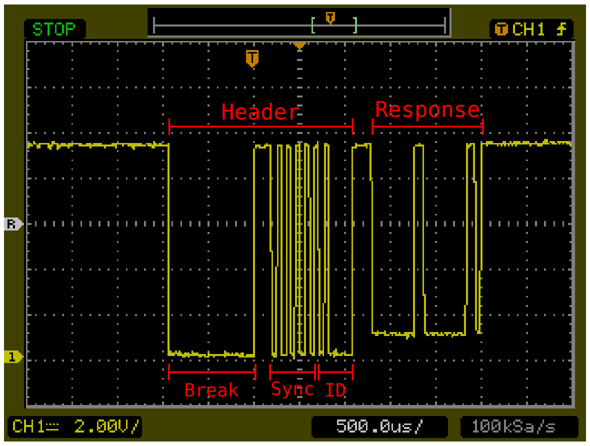


FIGURE 2: *Master: MPC5200 with `slLIN`; Slave: PCAN-LIN*

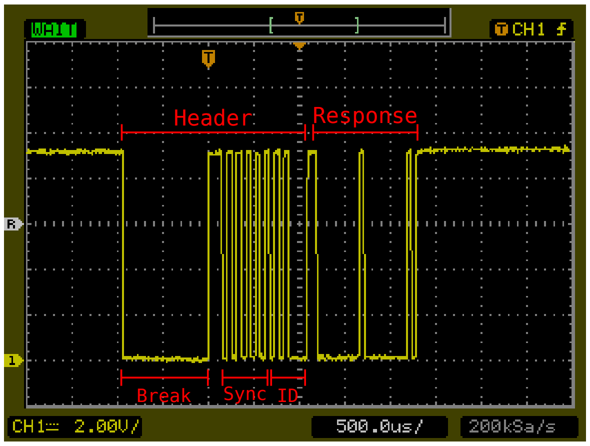


FIGURE 3: *Master: MPC5200 with `slLIN`; Slave: MPC5200 with `slLIN`*

6 Implementation and Difficulties

This section describes the difficulties that were encountered during the development. Then selected approach to generate break character is described. The problem to control Rx FIFO trigger level for response only/slave transfer is left for future work and communication with serial drivers maintainers.

6.1 Experienced Problems

First prototype of `slLIN` was programmed for user-space. At first this seemed to be much easier than to implement a TTY discipline, however we were experiencing some problems. Those are briefly mentioned in the following paragraphs.

- It is possible to generate UART-break by calling `tcsendbreak()` system call. The result was a break signal lasting hundreds of milliseconds, whereas about 700 μs long break signal is needed for LIN when operating at 19200 bauds.

  A possible way for generating break signal of the appropriate length would be to lower UART baud rate and send a normal character of value 0x00 using the changed speed. The baud rate can be changed by `cfsetospeed(struct termios *termios_p, speed_t speed)` (and `tcsetattr()`) system call but it does not allow to use arbitrary value for `speed_t`, only a predefined values can be used. This means that it is not possible to decrease the baud rate to 2/3 of the current baud rate.

  When tried to use half baud rate for sending break, the break signal was still to long.

- Slave implementation faces another fundamental problem. Common UART chips signal a receive event when either RX FIFO is filled up to the certain level or after a timeout (typically one to three characters long) elapses. FIFO RX trigger level can be configured for some UART chips but there is no standard API (neither in the kernel nor in the user space) to set the level. Linux serial drivers set the level to a fixed value. Even worse, the most common 16C550 based chips cannot be told to set the RX FIFO trigger level to one character. The only solution is to disable RX and TX FIFOs completely. But again, there is no API to ask for that in Linux serial drivers.

6.2 Break Signal Generation

There are two possible ways how to generate correct LIN break signal in a *user-space* program:

- Baud rate can be decreased by setting `custom_divisor` field in `struct serial_struct` structure, which is obtained by calling `ioctl(tty_fd, TIOCGSERIAL, &sattr)` (see the file `tty_lin_master/main.c` at line 60). This approach works with PC UARTs, however it is deprecated and may not work with every UART controller.
- Alternatively, baud rate can also be decreased by setting `struct termios2` structure, which is obtained by calling `ioctl(tty_fd, TCGETS2, &tattr)` (see file `tty_lin_master/main.c` at line 95).

`slLIN` is implemented in *kernel-space*. It generates the correct break character by direct switching of TX line state for the necessary amount of time. The time interval is measured by `usleep_range()` function.

Alternate implemented solution is similar to the user-space approach. Break signal can be generated by changing the baud rate. This can be achieved by setting `struct ktermios` belonging to the particular TTY (`struct tty_struct`).

7 Conclusion

Portable Linux LIN-bus driver (`slLIN`) has been implemented. Implantation is based on SocketCAN introduced and based network API/protocol family (PF_CAN). Hardware is controlled through architecture/UART type independent low level Linux UART API through `slLIN` serial line discipline.

`slLIN` is currently capable of operating as a LIN-bus *master node*. The full support for sending LIN headers and or LIN headers and associated LIN responses is implemented and tested. All traffic is fully monitored and all respones are delivered to applications according to PF_CAN filters under application control. Local *slave tasks* responses can be sent directly by applications in response to RTR SFF CAN frames or can be supplied by the *frame cache*.

Transfers schedule table is not implemented directly by the `slLIN` driver. Timing can be controlled by master controlling user-space application by sending RTR SFF CAN frames to `slLIN` at appropriate time instants or SocketCAN Broadcast Manager (BCM) can be configured to control transfers timing from kernel-space. The utility `lin_config` has been implemented to simplifies transfer schedule and *frame cache* configuration.

Slave mode enables monitoring of the LIN-bus but sending *slave task* responses by *slave node* is not solved. To fully implement slave mode in a way that is independent on the underlaying driver requires extension of a low lever kernel UART API to allow control the RX FIFO of serial controllers. This will have to be discussed with Linux TTY layer maintainers. On the other hand, slave implementation has been tested on MPC5200 UART which can be managed to behave as required for `slLIN` slave configuration.

The portability of the `slLIN` to different architectures has been demonstrated by running the driver on i686 PC 16C550 UART and on PowerPC MPC5200 board. Interoperability with Peak LIN professional offers has been successfully verified.

The actual information about `slLIN` development can be found at the DCE FEE CTU page[4]. Source code of the driver and `lin_config` utility is available from Rtime GIT repository[5].

8 Acknowledgment

This work was financially supported by Volkswagen AG and Ministry of Education of the Czech Republic under the Project ME 10039.

References

[1] LIN consortium: *LIN Specification Package, Revision 2.2* [online], December 2010, Available: `http://www.lin-subbus.de/`.

[2] Hartkopp, O: *The CAN networking subsystem of the Linux kernel*, in *Proceedings of the 13th International CAN Conference (iCC'12)*, Hambach Castle, Germany: CiA, March 2012, pp. 05-10 – 05-16.

[3] Kleine-Budde, M: *SocketCAN – The official CAN API of the Linux kernel*, in *Proceedings of the 13th International CAN Conference (iCC'12)*, Hambach Castle, Germany CiA, March 2012, pp. 05-17 – 05-22.

[4] *CAN and LIN pages – Rtime server at DCE FEE CTU* [online], 2012, Available: `http://rtime.felk.cvut.cz/can/lin-bus/`.

[5] *Linux LIN and slLIN Driver GIT repository* [online], 2012, Available: `http://rtime.felk.cvut.cz/gitweb/linux-lin.git`.