sllin – TTY discipline for UART-LIN device implementation

P. Píša, R. Lisový, M. Sojka
Czech Technical University in Prague

August 27, 2012
Version f510653

Abstract

This document describes `sllin`– TTY line discipline for GNU/Linux operating system that allows it to communicate over LIN bus interfaced via serial port (UART) and a simple logic level converter.

The API and ABI defined in this document are subjects to future changes.

Contents

1 Introduction

The LIN-bus (Local Interconnect Network) is a vehicle bus standard or computer networking bus-system used within current automotive network architectures.

It is possible to use serial port (UART) combined with simple logic level converter to interface with the LIN bus.

`sllin` is TTY line discipline implemented for GNU/Linux operating system that handles the interchange of LIN messages between the underlying UART driver and the CAN bus subsystem. This way, the same API as the one used for CAN networking can be used for communication with LIN devices.

1.1 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.

[1] To be downloaded from `http://www.lin-subbus.de/`

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.

1.2 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.

2 Using sllin

2.1 Compilation

To successfully compile `sllin`, it is necessary to have the source code of Linux kernel actually running on the computer.

Compilation of `sllin` code can be configured by changing the values of two preprocessor symbols:

- `DEBUG` – when *defined*, `sllin` logs debugging information into kernel log.
- `BREAK_BY_BAUD` – when *defined*, `sllin` generates break signal by changing the baud rate (and transmitting 0x00 character). Otherwise, the break signal is generated by manual setting of TX signal for short period of time. Both approaches are described in more detail in section 3.2.

To compile `sllin`, run

```
$ make
```

2.2 Loading the module

The `sllin` module can be loaded as any other Linux kernel module by `modprobe` or `insmod` utilities. Run time behavior of the module can be changed by setting the following module parameters:

maxdev
: Optional parameter.
Possible values: unsigned int.
Defines the maximum number of `sllin` interfaces. Default is `maxdev` = 4.
When `maxdev` < 4, `maxdev` = 4.

master
: Optional parameter.
Possible values: 0 or 1.
Determines whether the LIN interface will be in master or slave mode (1 = master, 0 = slave).
Default is `master` = 1.

baudrate
Optional parameter.
Possible values: unsigned int.
Determines the baudrate of the LIN bus.
Default is `baudrate` = 19200.

2.3 Attaching sllin to the UART interface

There are several ways how `sllin` could be attached to the particular UART interface:

1. Writing `sllin`-specific utility
2. Using modified `slcan_attach` utility – it is sufficient to change only the number identifying `slcan` TTY discipline to the one of `sllin` TTY discipline.
3. Using `ldattach` utility from `util-linux` package

The last possibility is the easiest one and is used in the following examples. The only disadvantage is that the user has to know the number identifying `sllin` TTY discipline (which may vary in time, as new line disciplines are added to the Linux kernel).

To attach `sllin` to `/dev/ttyS0` interface, use

```
# ldattach 25 /dev/ttyS0
```

This opens the device and attaches particular line discipline. The program then goes into the background keeping the device open to keep the line discipline attached.

To *detach* `sllin` from TTY, it is necessary to kill `ldattach`.

Initialization of sllin network interface After successful attachment of `sllin` to an UART interface, it is necessary to activate the newly created network interface:

```
# ip link set sllin0 up

# ip link show dev sllin0
11: sllin0: <NOARP,UP,LOWER_UP> mtu 16 qdisc pfifo_fast state UNKNOWN qlen 10
    link/can
```

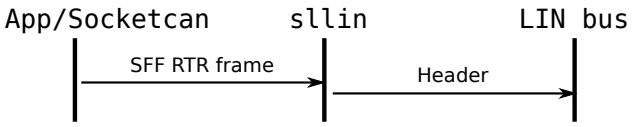
2.4 Sllin operation

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.

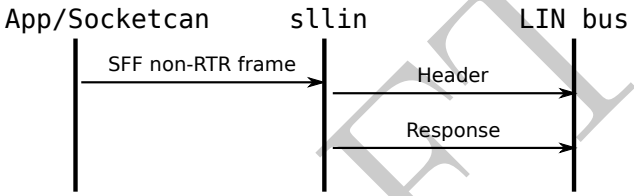
2.4.1 Master mode

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

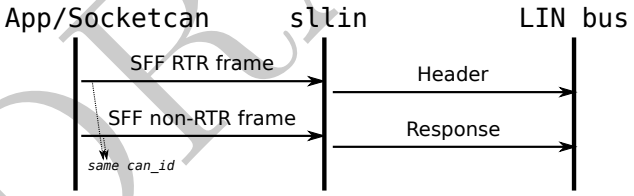
1. LIN header is sent to the LIN-bus after receiving SFF RTR CAN frame from an application.
   (*LIN id = can_id*).



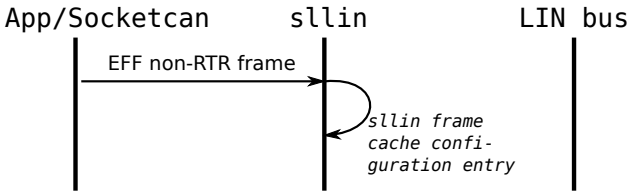
2. LIN header immediatelly followed by LIN response is sent to the LIN bus after receiving SFF non-RTR CAN frame from an application.
   (*LIN id = can_id; LIN response = can_frame.data*).



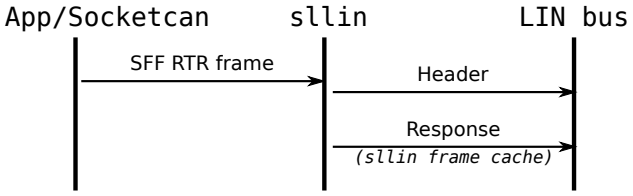
3. LIN response is sent to the LIN-bus (LIN-header is sent due to reception of SFF RTR CAN frame) after receiving SFF non-RTR CAN frame.
   (*can_id of both frames must be the same; LIN response = can_frame.data*).



4. A frame is stored in a *frame cache* (see Section 2.4.4) after receiving EFF non-RTR CAN frame. This operation is controlled by the flags in `can_id` of the frame.

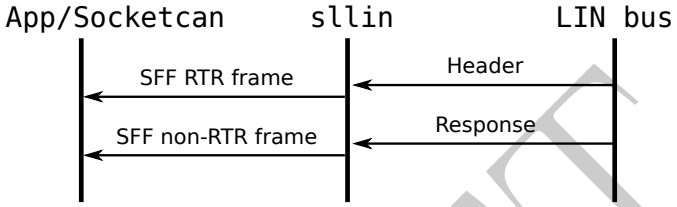


5. LIN response from correctly configured *frame cache* is sent to the LIN-bus upon sending the LIN header due to the reception of SFF RTR CAN frame.



2.4.2 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.



2.4.3 Error reporting

Errors from `sllin` are reported to applications by sending CAN frames with flags which are part of `can_id`. Individual error flags are listed in table 2.1 (they are also defined in `linux/lin_bus.h`).

Error flag	**Meaning**
`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

Table 2.1: Error flags used by `sllin`

2.4.4 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 particular flags (these are listed in table 2.2).

Flag	Meaning
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

Table 2.2: Flags used when configuring `sllin` frame cache

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

2.4.5 Examples

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 qdisc pfifo_fast state UNKNOWN qlen 10
    link/can
    RX: bytes  packets  errors  dropped overrun mcast
    2          4        1       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    0          4        0       0       0       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 one LIN header without response on the bus (= only RTR
# CAN frame) and another LIN header followed by a response (= RTR
# + non-RTR CAN frame with the same ID)
```

2.4.6 Configuration

A dedicated utility was developed[1] 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

[1] `https://rtime.felk.cvut.cz/gitweb/linux-lin.git/tree/HEAD:/lin_config`

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:

```
Usage: ./lin_config [OPTIONS] <SERIAL_INTERFACE>

'lin_config' is used for configuring sllin -- simple LIN device
  implemented as a TTY line discipline for arbitrary UART interface
  (works only on built-in interfaces -- not on USB to RS232
  convertors).
  This program is able to configure PCAN-LIN (RS232 configurable
  LIN node) as well.

SERIAL_INTERFACE is in format CLASS:PATH
  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

General options:
 -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:
 ./lin_config sllin:/dev/ttyS0        (Configure the device with the
                                       configuration from 'config.pclin')
 ./lin_config -r pcanlin:/dev/ttyS0   (Reset the device)
```


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.

3 Implementation details

This section mentions the problems that were encountered during the development and that are relevant for future work. After that, the currently implemented approaches are described.

3.1 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.

3.2 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 `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 `lin_master/main.c` at line 95).

`sllin` is implemented in *kernel-space*. It generates the correct break signal by manually controlling TX line for the necessary amount of time. The time interval is measured by `usleep_range()` function.

Another possible solution is similar to the user-space approach. Break signal is generated by changing the baud rate. This is done by setting `struct ktermios` belonging to the particular TTY (`struct tty_struct`).

4 Hardware

To connect an RS232 interface to LIN-bus, it is necessary to use logic level converter. Possible implementation of such a converter is shown in Figure 4.1.

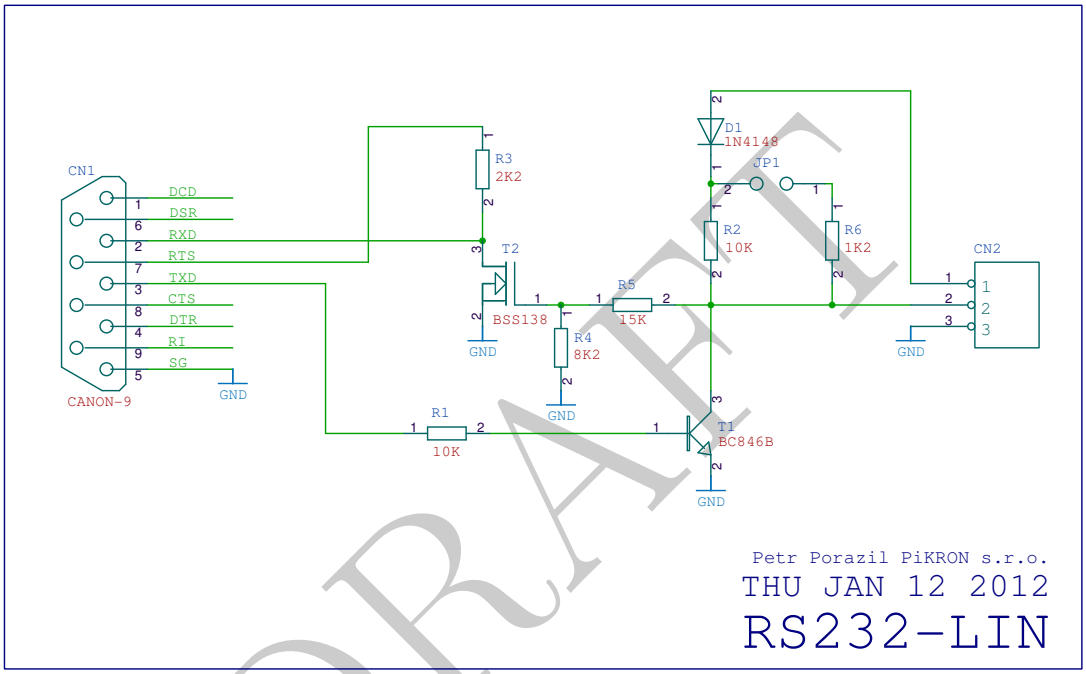


Figure 4.1: RS232 to LIN logic level converter

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 observation

Timing properties of `sllin` were observed on an oscilloscope. Waveforms captured on the LIN-bus are shown in Figures 5.1 and 5.2. It can be seen that there are no extensive delays caused by incorrect `sllin` implementation. Note, however, that these experiments were conducted on otherwise unloaded system.

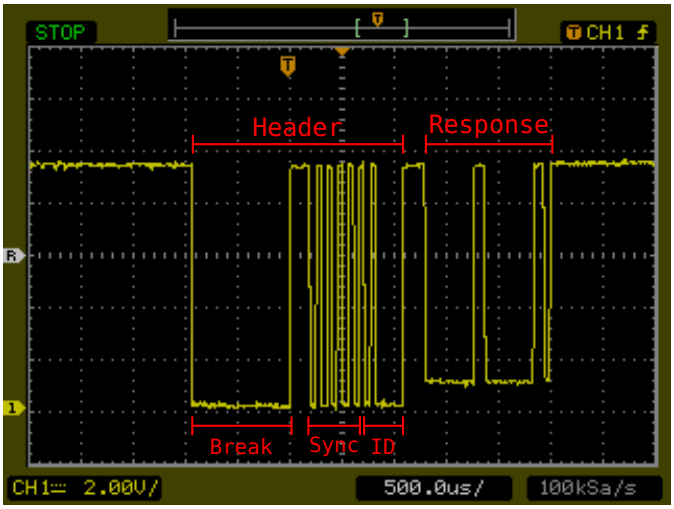


Figure 5.1: Master: MPC5200 with sllin; Slave: PCAN-LIN

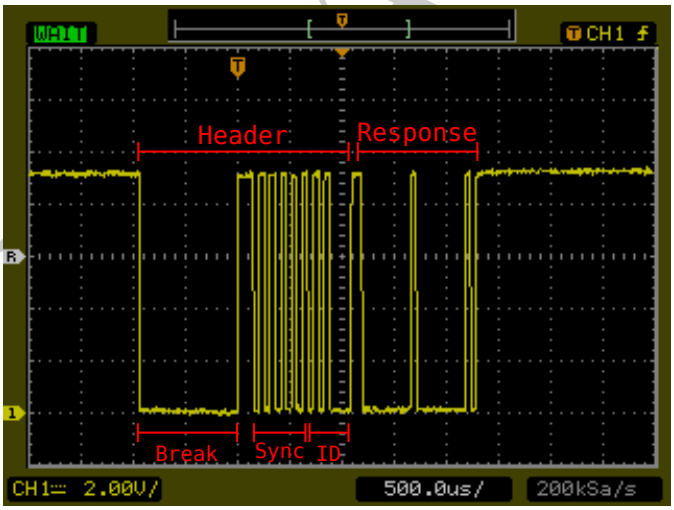


Figure 5.2: Master: MPC5200 with sllin; Slave: MPC5200 with sllin

6 Conclusion

This document described the prototype implementation of `sllin`. Due to technical difficulties encountered during development and limited project time, `sllin` is not (yet) a complete and fully working solution for interfacing LIN bus from Linux.

`sllin` is currently capable of operating as a *master node* – i.e. either sending only LIN headers or LIN headers and associated LIN responses.

Schedule table is not implemented in `sllin`. Timing is controlled in the application by sending RTR SFF CAN frames to `sllin` at appropriate times.

Slave mode enables monitoring of the LIN-bus. Full featured *slave node* and support for *slave task* were not implemented. To fully implement slave mode in a way that is independent on the under laying driver would require adding a kernel API for controlling the RX FIFO of serial controllers. This will have to be discussed with Linux TTY layer maintainers.

6.1 Acknowledgment

This work was financially supported by Volkswagen AG. The authors would like to thank Oliver Hartkopp for his feedback on this work.

Bibliography

[1] LIN consortium, "LIN specification package, revision 2.2." [Online]. Available: http://www.lin-subbus.de/