ELLSI

EtherCAN Low Level Socket Interface

Software Manual

to Product C.2050.xx and C.2051.xx

esd electronic system design gmbh
Vahrenwalder Str. 207 • 30165 Hannover • Germany
www.esd-electronics.com • Fax: 0511/37 29 8-68
Phone: 0511/37 29 80 • International: +49-5 11-37 29 80

NOTE

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. **esd** reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

esd assumes no responsibility for the use of any circuitry other than circuitry which is part of a product of **esd** gmbh.

esd does not convey to the purchaser of the product described herein any license under the patent rights of **esd** gmbh nor the rights of others.

esd electronic system design gmbh
Vahrenwalder Str. 207
30165 Hannover
Germany

Phone: +49-511-372 98-0
Fax: +49-511-372 98-68
E-mail: info@esd.eu
Internet: www.esd.eu

USA / Canada:
esd electronics Inc.
525 Bernardston Road
Suite 1
Greenfield, MA 01301
USA

Phone: +1-800-732-8006
Fax: +1-800-732-8093
E-mail: us-sales@esd-electronics.com
Internet: www.esd-electronics.us

All trademarks, product names, company names or company logos used in this manual are reserved by their respective owners.

Document file:	I:\Texte\Doku\MANUALS\PROGRAM\CAN\ELLSI\Ellsi_Manual_en_15.wpd
Date of print:	2010-02-22

ELLSI version:	1.1.4

Changes in the chapters

The changes in the document listed below affect changes in the firmware as well as changes in the description of facts only.

Chapter	Changes versus previous version
2.2	New chapter '2.2 Port' inserted.

Technical details are subject to change without further notice.

This page is intentionally left blank.

Contents

1. Introduction 7
1.1 Intention 7
1.2 Functional Principle 7
1.3 ELLSI vs. esd NTCAN API 7
1.4 Restrictions 8
1.4.1 ELLSI API 8
1.4.2 Number of Client Connections 8
1.4.3 TCP/IP vs. UDP/IP 8
1.4.4 CAN Interaction 8
1.5 Some Thoughts about Performance 8

2. The ELLSI-Protocol 9
2.1 Data Layout 9
2.2 Byte Order 9
2.3 Header 9
2.3.1 Sequence Numbering 10
2.3.2 Commands 11
2.3.2.1 Numerical Values of Commands 11
2.3.2.2 Numerical Values of Sub-Commands 11
2.3.2.3 ELLSI_CMD_NOP 11
2.3.2.3.1 lastState for ELLSI_CMD_NOP 11
2.3.2.4 ELLSI_CMD_REGISTER 12
2.3.2.4.1 lastState for ELLSI_CMD_REGISTER 12
2.3.2.5 ELLSI_CMD_REGISTERX 13
2.3.2.5.1 ellsiExtRegistration 13
2.3.2.5.2 lastState for ELLSI_CMD_REGISTERX 14
2.3.2.6 ELLSI_CMD_CAN_TELEGRAM 15
2.3.2.6.1 ellsiCMSG_T 16
2.3.2.6.2 lastState for ELLSI_CMD_CAN_TELEGRAM 17
2.3.2.6.3 ELLSI_SUBCMD_TXDONE 17
2.3.2.7 ELLSI_CMD_HEARTBEAT 18
2.3.2.7.1 lastState for ELLSI_CMD_HEARTBEAT 18
2.3.2.8 ELLSI_CMD_CTRL 19
2.3.2.8.1 ELLSI_IOCTL_CAN_ID_ADD/DELETE 19
2.3.2.8.1.1 ellsiCanIdRange 20
2.3.2.8.1.2 lastState for ELLSI_IOCTL_CAN_ID_ADD/DELETE 20
2.3.2.8.2 ELLSI_IOCTL_CAN_SET_BAUDRATE 20
2.3.2.8.2.1 Baud Rate Values 21
2.3.2.8.2.2 lastState for ELLSI_IOCTL_CAN_SET_BAUDRATE 21
2.3.2.8.3 ELLSI_IOCTL_CAN_GET_BAUDRATE 22
2.3.2.8.3.1 lastState for ELLSI_IOCTL_CAN_GET_BAUDRATE 22
2.3.2.8.4 ELLSI_IOCTL_SET_SJA1000_ACMR 23
2.3.2.8.4.1 Mapping of ACR / AMR to CAN ID in a 29-Bit Frame 24
2.3.2.8.4.2 lastState for ELLSI_IOCTL_SET_SJA1000_ACMR 24
2.3.2.8.5 ELLSI_IOCTL_GET_LAST_STATE 25
2.3.2.8.5.1 ellsiLastState 26
2.3.2.8.6 ELLSI_SUBCMD_AUTOACK 26

This page is intentionally left blank.

1. Introduction

1.1 Intention

ELLSI offers the possibility to use an esd EtherCAN on all platforms not (yet) supported by esd NTCAN drivers (e.g. Mac OS, PLCs with Ethernet capability, etc.).

The standard NTCAN over Ethernet functionality is still usable in parallel to ELLSI. However, **for all platforms with an existing NTCAN driver, we suggest to use NTCAN instead of ELLSI for communication with the esd EtherCAN.**

1.2 Functional Principle

We tried to develop ELLSI as simple as possible. We don't provide an API to use ELLSI, but there is some sample code, which should help you to build such an API yourself. Using ELLSI is "simply" assembling UDP-datagrams plus transmitting them to the ELLSI-server on the esd EtherCAN and analysing UDP-datagrams obtained from the ELLSI-server on the esd EtherCAN hardware.

At first the ELLSI-client has to register itself at the ELLSI-server. After this, both sides have to send heartbeat-messages at regular intervals if there is no data exchange.

If the client has not received any data or heartbeat from the server within a given time interval, the client will assume that the server has disappeared. Maybe the network connection is broken, somebody did a reset on the EtherCAN, etc. In consequence of this, the client has to try to register at the server again.

If the server has not seen any data or heartbeat from the client within a given time interval, it assumes the client has disappeared. The server no longer transfers any data and heartbeat to the client then.

After the client has registered itself, it must set a baud rate and enable all CAN IDs it wants to receive data for. Now the client is ready for transmission and reception of CAN telegrams.

To be sure CAN telegrams are sent / received in correct order, there is a sequence number.

1.3 ELLSI vs. esd NTCAN API

esd carefully tried to develop ELLSI as compatible as possible with the standard esd NTCAN API. We would therefore recommend to read the esd NTCAN API documentation in parallel to this document. The esd NTCAN API documentation far often delivers more detailed information about the esd NTCAN philosophy then this document.

1.4 Restrictions

1.4.1 ELLSI API

esd electronics does **not** support and maintain an official API for ELLSI, but you can use the provided examples, in particular *ellsiCommon.c*, *ellsiCommon.h* in combination with *ellsiClnt.c* and *ellsiClnt.h* as a base for your personal ELLSI API. **For all platforms with an existing NTCAN driver, we suggest to use NTCAN instead of ELLSI for communication with the esd EtherCAN**.

1.4.2 Number of Client Connections

The number of client connections to the ELLSI server is currently limited to 5, to not overstrain the EtherCAN hardware.

1.4.3 TCP/IP vs. UDP/IP

At the moment the ELLSI-server only supports UDP. For future versions it is planned to also support TCP-connections.

1.4.4 CAN Interaction

The standard esd NTCAN drivers maintain a feature called interaction. This feature allows CAN messages transmitted on a certain CAN ID on a certain CAN bus also to be received by other processes reading CAN messages on the same physical CAN bus (CAN card). ELLSI does **not** support this feature in the current release. But for future releases it is planned to allow the user to reactivate interaction (as optional parameter for the *ELLSI_CMD_REGISTER* command).

1.5 Some Thoughts about Performance

Here are some proposals to maximize ELLSI performance on an esd EtherCAN:

- Try to send as many CAN TX messages as possible in one ELLSI telegram. Furthermore, the ELLSI server automatically tries to pack multiple CAN RX messages into a single ELLSI telegram to improve the performance.
- Only enable those CAN IDs for reception you're really interested in
- Minimize the number of clients connected to the ELLSI server
- Make use of the auto-acknowledge (*ELLSI_SUBCMD_AUTOACK*) feature wherever it is possible
- If your application allows for this, avoid sending CAN TX telegrams using the TX-DONE feature

2. The ELLSI-Protocol

2.1 Data Layout

The data always consists of a header plus trailing payload data.

The payload data itself consists of the data according to a single command or to n-CAN-telegrams.

Header	*Command data* **or** *n * ellsiCMSG_T*

Thus it is possible to send or receive multiple CAN telegrams at the "same" time. Using this feature you can greatly improve the performance of the esd EtherCAN.

2.2 Port

The default port for the ELLSI UDP server is 2209.

2.3 Byte Order

Attention! All ELLSI-telegram data has to be given (or is given) in **network byte order** (most significant byte first).
E.g. Intel x86 processors host byte order is least significant byte first. So always be aware of your host byte order before assembling ELLSI-telegrams!

2.4 Header

The header mentioned above looks like this (see *ellsiCommon.h*):

```
typedef struct {
  uint32_t magic;
  uint32_t sequence;
  uint32_t command;
  uint32_t payloadLen;
  uint32_t subcommand;
  union {
    int32_t i[8];
    int8_t  c[32];
  } reserved;
} ellsiHeader;
```

Member	***Size***	***Description***
magic	unsigned 32-bit	Magic number: *ELLSI_MAGIC= 0x454c5349* It's mandatory to have this value (switched to network byte order!) in every ELLSI telegram. ELLSI clients should first check this value before doing anything else with a received ELLSI telegram.
sequence	unsigned 32-bit	Sequence number or zero
command	unsigned 32-bit	ELLSI_CMD_* (see *ellsiCommon.h*)
payloadLen	unsigned 32-bit	Length of payload data (in bytes)
subcommand	unsigned 32-bit	ELLSI_SUBCMD_* or ELLSI_IOCTL_* (see *ellsiCommon.h*)
reserved	32 bytes	For future protocol extensions

2.4.1 Sequence Numbering

UDP does not guarantee to receive the datagrams in the same order they were transmitted. In local Ethernets without routing, you normally don't have to bother about this. To avoid sending CAN telegrams in wrong order to the CAN bus, the ELLSI-client can make use of the *sequence*-element. If *sequence* equals zero, the ELLSI- server does not take care of the sequence number and unconditionally will send CAN telegrams to the CAN-bus. If non-zero, the ELLSI-server discards CAN TX telegrams if the sequence number is less or equal to the sequence number of the last CAN TX telegram.

For the other direction, the ELLSI-server will increment the *sequence*-element for every telegram send to the ELLSI-client (regardless if CAN RX data, a response to a previous command or a heartbeat).

2.4.2 Commands

2.4.2.1 Numerical Values of Commands

You can find the following defines in *ellsiCommon.h*:

```
ELLSI_CMD_NOP                    0
ELLSI_CMD_CAN_TELEGRAM           1
ELLSI_CMD_HEARTBEAT              2
ELLSI_CMD_CTRL                   3
ELLSI_CMD_REGISTER               4
ELLSI_CMD_REGISTERX              5
```

2.4.2.2 Numerical Values of Sub-Commands

You can find the following defines in *ellsiCommon.h*:

```
ELLSI_IOCTL_NOP                  0
ELLSI_SUBCMD_NONE                0
ELLSI_IOCTL_CAN_ID_ADD           1
ELLSI_IOCTL_CAN_ID_DELETE        2
ELLSI_IOCTL_CAN_SET_BAUDRATE     3
ELLSI_IOCTL_CAN_GET_BAUDRATE     4
ELLSI_IOCTL_GET_LAST_STATE       5
ELLSI_IOCTL_SET_SJA1000_ACMR     6
ELLSI_SUBCMD_TXDONE              128
ELLSI_SUBCMD_AUTOACK             256
```

2.4.2.3 ELLSI_CMD_NOP

A type of no-operation command.

Header	**magic**	*ELLSI_MAGIC*
	sequence	0
	command	*ELLSI_CMD_NOP*
	payloadLen	0
	subcommand	0
	reserved	0

2.4.2.3.1 lastState for ELLSI_CMD_NOP

ELLSI_CMD_NOP always will set *lastState* to 0.

2.4.2.4 ELLSI_CMD_REGISTER

As the first operation the ELLSI-client has to register itself at the ELLSI-server. Therefore a telegram like this must be set up:

Header	**magic**	*ELLSI_MAGIC*
	sequence	0
	command	*ELLSI_CMD_REGISTER*
	payloadLen	0
	subcommand	*0 or ELLSI_SUBCMD_AUTOACK*
	reserved	0

2.4.2.4.1 lastState for ELLSI_CMD_REGISTER

0 for successful registration. All values unequal to 0 stand for a failed registration.

2.4.2.5 ELLSI_CMD_REGISTERX

Starting with ELLSI Version 1.1.1 (March 2006) there is an extended registration operation, allowing the user to have influence on some ELLSI-server parameters. Therefor you need to setup a telegram like this:

Header	**magic**	*ELLSI_MAGIC*
	sequence	0
	command	*ELLSI_CMD_REGISTERX*
	payloadLen	*sizeof(ellsiExtRegistration)*
	subcommand	*0 or ELLSI_SUBCMD_AUTOACK*
	reserved	0
Payload		*ellsiExtRegistration*

2.4.2.5.1 ellsiExtRegistration

The *ellsiExtRegistration* structure mentioned above looks like this:

```
typedef struct {
  uint32_t heartBeatIntervall;
  uint32_t clientDeadMultiplier;
  uint32_t canTxQueueSize;
  uint32_t canRxQueueSize;
  uint32_t socketSendMaxNTelegrams;
  uint32_t socketSendIntervall;
  uint32_t reserved[8];
} ellsiExtRegistration;
```

Member	***Size***	***Description***
heartBeatIntervall	unsigned 32-bit	ELLSI server heartbeat interval in ms. Use 0 for default value (default is 2500 ms). The valid range is 250 <= x <= 30000.
clientDeadMultiplier	unsigned 32-bit	After *clientDeadMultiplier / 10 * heartBeatTime [ms]* we assume a client as "dead". Use zero for default value. Default is 30 (which is equivalent to a multiplier of 30/10 = 3.0). The valid range is 10 <= x <= 100.
canTxQueueSize	unsigned 32-bit	Size of message queue used for CAN TX telegrams for each(!) of the clients. Use 0 for default (default is 128). The valid range is 1 <= x <= 2048.
canRxQueueSize	unsigned 32-bit	Size of queue used for CAN RX telegrams for each(!) of the clients. Use 0 for default (default is 512). The valid range is 1 <= x <= 2048.
socketSendMaxNTelegrams	unsigned 32-bit	Maximum numbers of CAN RX telegrams to store in a UDP telegram. Use 0 for default. (default is *CAN_READ_MAXLEN*= 50) The valid range is 1 <= x <= *CAN_READ_MAXLEN*.
socketSendIntervall	unsigned 32-bit	Try to collect CAN RX data for up to *socketsendintervall* ms before sending an UDP telegram to the client. Use 0 for default (default is 0 ms). **Not yet implemented!!!**
reserved[8]	8x unsigned 32-bit	Reserved for future use !!! To be set to zero !!!

2.4.2.5.2 lastState for ELLSI_CMD_REGISTERX

0 for successful registration. All values unequal to 0 stand for a failed registration.

2.4.2.6 ELLSI_CMD_CAN_TELEGRAM

ELLSI telegram layout for received CAN telegrams and CAN telegrams to be send:

Header	**magic**	*ELLSI_MAGIC*
	sequence	Sequence # [or 0]
	command	*ELLSI_CMD_CAN_TELEGRAM*
	payloadLen	n * sizeof(*ellsiCMSG_T*)
	subcommand	0 [or *ELLSI_SUBCMD_TXDONE*]
	reserved	0
Payload		*ellsiCMSG_T* #1
		.
		.
		.
		ellsiCMSG_T #n

2.4.2.6.1 ellsiCMSG_T

The *ellsiCMSG_T* data structure of CAN messages mentioned above looks like this:

```
typedef struct {
        uint32_t id;
        uint8_t  len;
        uint8_t  msg_lost;
        uint8_t  reserved[2];
        uint8_t  data[8];
        ellsiCAN_TIMESTAMP timestamp;
} ellsiCMSG_T;
```

Member	***Size***	***Description***
id	unsigned 32-bit	11- or 29-bit CAN ID data was received on or data should be transmitted on
len	unsigned 8-bit	Bit 0-3 : Number of CAN data bytes [0..8] Bit 4 : RTR Bit 5 : TXDONE (see *ELLSI_SUBCMD_TXDONE*) Bit 6-7 : Reserved
msg_lost	unsigned 8-bit	Counter for lost CAN RX messages. Allows the user to detect data overrun: msg_lost = 0 : no lost messages 0 < msg_lost < 255 : # of lost frames = value of msg-lost msg_lost = 255 : # of lost frames > 255
reserved[2]	2x unsigned 8-bit	Only meaningful together with *ELLSI_SUBCMD_TXDONE.* In this case used to allow association of TX-DONE messages with previously sent CAN TX messages (see *ELLSI_SUBCMD_TXDONE*)
data[8]	8x unsigned 8-bit	CAN data bytes
timestamp	64-bit	Time stamp for CAN RX messages. This time stamp is meaningless for CAN TX messages **Not yet supported!!!**

To ease porting applications between ELLSI and NTCAN, this structure is compatible to the *CMSG_T*-structure in the esd NTCAN API (see *ntcan.h*).

2.4.2.6.2 lastState for ELLSI_CMD_CAN_TELEGRAM

lastState, after issuing a CAN TX message using ELLSI_CMD_CAN_TELEGRAM, contains the return value given by the *canSend()*-function of the esd NTCAN API. Concrete, 0 stands for successful completion of *canSend()* and the respective *ELLSI_CMD_CAN_TELEGRAM*-command. All non-zero values will indicate an error condition.

Seeing *lastState* as zero indicates successful completion of the ELLSI-server internal *canSend()*-command, but does not necessarily indicate a successful transmission of the corresponding CAN telegram(s) on the CAN bus, because *canSend()* is a non-blocking function! Therefore, if you are interested in knowing, if the appropriate CAN telegram has been successfully send on the CAN bus, *lastState* will not help you. See *ELLSI_SUBCMD_TXDONE* instead.

2.4.2.6.3 ELLSI_SUBCMD_TXDONE

As mentioned above, requesting the last state of an *ELLSI_CMD_CAN_TELEGRAM* command does not necessarily indicate a successful transmission of a CAN telegram to the CAN bus. If you've the need to know if your CAN telegram was successfully transmitted, don't query *lastState*. Instead, while assembling a CAN TX message using *ELLSI_CMD_CAN_TELEGRAM*, set the headers *subcommand* element to *ELLSI_SUBCMD_TXDONE*. The ELLSI-server then will send you a transfer-done message (TX-DONE message) after successful transmission on the CAN bus. This TX-DONE message is assembled very similar to a "normal" CAN RX telegram.

To distinguish a normal CAN telegram from a TX-DONE telegram, the *length* element in the corresponding *ellsiCMSG_T* is logically ORed with *ELLSI_CMSGT_LEN_TXDONE* (0x20). Additionally, the two reserved bytes in *ellsiCMSG_T* are echoed back! If you e.g. set this two reserved bytes to the two last significant bytes of the sequence number, you will easily be allowed to associate a received TX-DONE to a previously sent CAN telegram .

If additionally to *ELLSI_CMSGT_LEN_TXDONE*, *ELLSI_CMSGT_LEN_TXERROR* is set in the *length* element, this will indicate an error while sending the telegram to the CAN bus. An corresponding error code can be found in the id element of ellsiCMSG_*T*.

2.4.2.7 ELLSI_CMD_HEARTBEAT

Both sides (ELLSI-client and ELLSI-server) have to send heartbeat-messages at regular intervals if there is no data exchange. At the moment this interval is fix 2500 ms. Future releases will add the possibility to change the interval(s) used by the ELLSI-server.

If the client has not seen any data or heartbeat from the server within a given time interval, the client will assume that the server has disappeared. Maybe the network connection is broken, somebody did a reset on the EtherCAN, etc. In consequence of this, the client has to try to register at the server again.

If the server has not seen any data or heartbeat from the client within a given time interval, it assumes the client as disappeared. The ELLSI-server no longer will transfer any data and heartbeat to the client then.

Telegram layout for a heartbeat message:

Header	**magic**	*ELLSI_MAGIC*
	sequence	0
	command	*ELLSI_CMD_HEARTBEAT*
	payloadLen	0
	subcommand	0
	reserved	0

2.4.2.7.1 lastState for ELLSI_CMD_HEARTBEAT

ELLSI_CMD_HEARTBEAT will (contrary to the very similar looking *ELLSI_CMD_NOP command)* **not** set *lastState*!

2.4.2.8 ELLSI_CMD_CTRL

Setting the headers *command* element to *ELLSI_CMD_CTRL*, the client can send special commands to the ELLSI-server. This special commands are specified by setting the headers *subcommand* element.

Currently the following *subcommand* controls exist:

2.4.2.8.1 ELLSI_IOCTL_CAN_ID_ADD/DELETE

By means of *ELLSI_IOCTL_CAN_ID_ADD* the client can enable CAN IDs for reception. Using *ELLSI_IOCTL_CAN_ID_DELETE* the client can disable (previously enabled) IDs, to no longer receive data on this CAN IDs.

The IDs to be enabled or disabled are given in the efficient form of an array of *ellsiCanIdRange* structures. Telegram layout for enabling / disabling CAN IDs:

Header	**magic**	*ELLSI_MAGIC*
	sequence	0
	command	*ELLSI_CMD_CTRL*
	payloadLen	n * sizeof(*ellsiCanIdRange*)
	subcommand	*ELLSI_IOCTL_CAN_ID_ADD* or *ELLSI_IOCTL_CAN_ID_DELETE*
	reserved	0
Payload		*ellsiCanIdRange #1*
		.
		.
		.
		ellsiCanIdRange #n

2.4.2.8.1.1 ellsiCanIdRange

```
typedef struct {
    uint32_t rangeStart;
    uint32_t rangeEnd;
} ellsiCanIdRange;
```

Member	*Size*	*Description*
rangeStart	unsigned 32-bit	Interval start, CAN ID(s) to be enabled for reception / disabled from reception
rangeEnd	unsigned 32-bit	Interval end, CAN ID(s) to be enabled for reception / disabled from reception

The complete range, including *rangeStart* and *rangeEnd* itself, will be enabled or disabled. If *rangeEnd* is less or equal to *rangeStart*, only the CAN ID given by *rangeStart* will be enabled or disabled.

2.4.2.8.1.2 lastState for ELLSI_IOCTL_CAN_ID_ADD/DELETE
0 for success, non-zero for failure.

2.4.2.8.2 ELLSI_IOCTL_CAN_SET_BAUDRATE
By means of this *subcommand* you can set the baud rate to be used on the CAN bus.

Header	**magic**	*ELLSI_MAGIC*
	sequence	0
	command	*ELLSI_CMD_CTRL*
	payloadLen	4
	subcommand	*ELLSI_IOCTL_CAN_SET_BAUDRATE*
	reserved	0
Payload		*baudrate*

2.4.2.8.2.1 Baud Rate Values

baudrate has to be seen as a 32-bit unsigned integer. The predefined baud rates are:

baud rate	CAN bit rate [Kbit/s]
0x0	1000
0x1	666.6
0x2	500
0x3	333.3
0x4	250
0x5	166
0x6	125
0x7	100
0x8	66.6
0x9	50
0xA	33.3
0xB	20
0xC	12.5
0xD	10

If the LSB (bit 31) of parameter *baudrate* is set to '1', the value will be evaluated differently. In this case, the register value for the bit-timing registers BTR0 and BTR1 transmitted in modules with CAN controllers 82C200, SJA1000, 82527 (and all other controllers with this baud rate structure) is defined directly. For further information on this topic, see our esd NTCAN API documentation.

2.4.2.8.2.2 lastState for ELLSI_IOCTL_CAN_SET_BAUDRATE

lastState represents the return value of NTCAN *canSetBaudrate()*, so 0 stands for success and non-zero for failure.

2.4.2.8.3 ELLSI_IOCTL_CAN_GET_BAUDRATE

To read back the currently used CAN bus baud rate, set on the EtherCAN, send the following datagram to the ELLSI-server:

Header	**magic**	*ELLSI_MAGIC*
	sequence	0
	command	*ELLSI_CMD_CTRL*
	payloadLen	0
	subcommand	*ELLSI_IOCTL_CAN_GET_BAUDRATE*
	reserved	0

As answer you will get a telegram like this:

Header	**magic**	*ELLSI_MAGIC*
	sequence	x
	command	*ELLSI_CMD_CTRL*
	payloadLen	4
	subcommand	*ELLSI_IOCTL_CAN_GET_BAUDRATE*
	reserved	0
Payload		*baudrate*

2.4.2.8.3.1 lastState for ELLSI_IOCTL_CAN_GET_BAUDRATE

There should be no reason for anyone to query the *lastState* after an ELLSI_IOCTL_CAN_GET_BAUDRATE.
Nevertheless, if you do it:

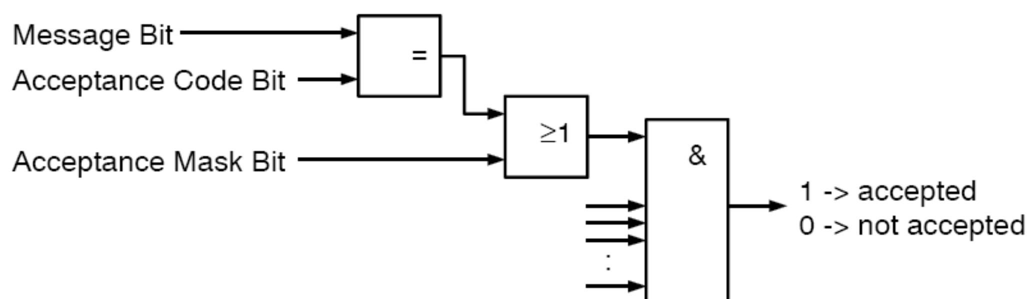
0 means NTCAN *canGetBaudrate()*-function an the ELLSI-server completed successfully, non-zero means failure.

2.4.2.8.4 ELLSI_IOCTL_SET_SJA1000_ACMR

Some platforms using a SJA1000 CAN controller, like the standard esd EtherCAN (with ELLSI version >= 1.1.02), allow setting its hardware acceptance filter. Therefore the user can directly access the SJA1000's ACR (Acceptance Code Register) and the AMR (Acceptance Mask Register):

Header	**magic**	*ELLSI_MAGIC*
	sequence	0
	command	*ELLSI_CMD_CTRL*
	payloadLen	8
	subcommand	*ELLSI_IOCTL_SET_SJA1000_ACMR*
	reserved	0
Payload	**0: ACR**	*SJA1000: ACR0, ARC1, ACR2, ACR3*
	4: AMR	*SJA1000: AMR0, AMR1, AMR2, AMR3*

The acceptance filter is realized by a logical AND-combination of the **A**cceptance **C**ode **R**egister followed by a logical OR-combination with the **A**cceptance **M**ask **R**egister according to the following figure.



The figure above shows that an active bit within the acceptance mask results in a *don't care* condition for the result of the comparison of received message bit and acceptance code bit. It is possible to limit the filter exactly to one 29-bit CAN identifier or one group of 29-bit CAN identifiers.

The following table shows some examples of bit combinations of the acceptance mask and acceptance code and their meaning for the filter (SJA1000 in “Single Filter Configuration”):

Acceptance Code	Acceptance Mask	Filter
`0x00000100`	`0x00000000`	Only 29-bit messages with the CAN identifier 0x100 are stored in the receive FIFO of the handle.
`0x00000100`	`0x000000FF`	All 29-bit CAN messages within the identifier area 0x100-0x1FF are stored in the receive FIFO of the handle.
any	`0x1FFFFFFF`	All 29-bit CAN messages are stored in the receive FIFO of the handle (open mask). **default value**

The combination in the last row of the table above shows the default function for the reception of 29-bit CAN messages, if no mask is configured.

Attention!
The 2 least significant bits of the ACR3 and AMR3 are not used and should be set to 1. Bit #3 in ACR3 and AMR3 corresponds to the RTR-bit of the filtered CAN frame! So, if you don’t want to take care of the RTR bit, you simply have to left-shift ACR and AMR by 3 and logical-OR 0x7 afterwards. E.g. to set the acceptance code to 0x00000100 and the acceptance mask to 0x000000ff (as in the above example), set *0x803 ((0x100 << 3) | 0x7)* in ACR and *0x7ff ((0xff << 3) | 0x7)* into AMR. To allow reception of all 29-bit IDs, write *0xffffffff ((0x1fffffff << 3) | 0x7)* into AMR.

2.4.2.8.4.1 Mapping of ACR / AMR to CAN ID in a 29-Bit Frame

Here you can see, how the two 32-bit registers ACR and AMR are mapped to the according 29 CAN ID bits:

MSB																															**LSB**
ACR0								**ACR1**								**ACR2**								**ACR3**							
AMR0								**AMR1**								**AMR2**								**AMR3**							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00	RTR		

<u>For additional information, especially when using the above filter mechanism on receiving 11-bit standard CAN frames, also consult the according pages in the SJA1000 data sheet!</u>

2.4.2.8.4.2 lastState for ELLSI_IOCTL_SET_SJA1000_ACMR

lastState represents the return value of the IOCTLs done on the EtherCAN platform. So 0 stands for success and non-zero for a failure condition.

2.4.2.8.5 ELLSI_IOCTL_GET_LAST_STATE

ELLSI_IOCTL_GET_LAST_STATE allows to get some information about the last command processed by ELLSI on the EtherCAN module and will most times be used to see, if important commands, like registering the client, setting the baud rate or enabling CAN IDs, etc., reached the ELLSI-server and were successfully processed.

To request the "last state" from the ELLSI-server send the following telegram:

Header	**magic**	*ELLSI_MAGIC*
	sequence	0
	command	*ELLSI_CMD_CTRL*
	payloadLen	0
	subcommand	*ELLSI_IOCTL_GET_LAST_STATE*
	reserved	0

The ELLSI-server will respond with the following telegram:

Header	**magic**	*ELLSI_MAGIC*
	sequence	x
	command	*ELLSI_CMD_CTRL*
	payloadLen	sizeof(*ellsiLastState*)
	subcommand	*ELLSI_IOCTL_GET_LAST_STATE*
	reserved	0
Payload		*ellsiLastState*

2.4.2.8.5.1 ellsiLastState

```
typedef struct {
  uint32_t lastCommand;
  uint32_t lastSubcommand;
  int32_t  lastState;
  uint32_t lastRxSeq;
  uint32_t reserved[4];
} ellsiLastState;
```

Member	***Size***	***Description***
lastCommand	unsigned 32-bit	Last command processed by the ELLSI-server: `ELLSI_CMD_NOP` **or** `ELLSI_CMD_CAN_TELEGRAM` **or** `ELLSI_CMD_HEARTBEAT` **or** `ELLSI_CMD_CTRL` **or** `ELLSI_CMD_REGISTER` **or** `ELLSI_CMD_REGISTERX`
lastSubcommand	unsigned 32-bit	Last sub-command processed by ELLSI-server: `ELLSI_IOCTL_NOP` **or** `ELLSI_IOCTL_CAN_ID_ADD` **or** `ELLSI_IOCTL_CAN_ID_DELETE` **or** `ELLSI_IOCTL_CAN_SET_BAUDRATE` **or** `ELLSI_IOCTL_CAN_GET_BAUDRATE` **or** `ELLSI_IOCTL_GET_LAST_STATE` **or** `ELLSI_SUBCMD_TXDONE` **or** `ELLSI_SUBCMD_AUTOACK`
lastState	32-bit	For states returned by the commands and subcommands see the corresponding descriptions of commands and subcommands
lastRxSeq	unsigned 32- bit	The last sequence number the ELLSI-client send by the appropriate command to the ELLSI-server
reserved	16 bytes	For future protocol extensions

2.4.2.8.6 ELLSI_SUBCMD_AUTOACK

To speed up the procedure of sending a command and afterwards using *ELLSI_IOCTL_GET_LAST_STATE* to request the state of this command, we introduced *ELLSI_SUBCMD_AUTOACK*.

By a disjunction of subcommand with *ELLSI_SUBCMD_AUTOACK*, the ELLSI-server will automatically generate a telegram analogue to the one generated by using the *ELLSI_IOCTL_GET_LAST_STATE* described above.