ELLSI Manual

EtherCAN Low Level Socket Interface

Software Manual

to Product C.2051.xx

esd electronic system design gmbh
Vahrenwalder Str. 207 • 30165 Hannover • Germany
http://www.esd.eu
Phone: +49 (0) 511 3 72 98-0 • Fax: +49 (0) 511 3 72 98-68

N O T E

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. In particular descriptions and technical data specified in this document may not be constituted to be guaranteed product features in any legal sense.

esd reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

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

Document file:	I:\Texte\Doku\MANUALS\PROGRAM\CAN\ELLSI\ELLSI_Manual_en_17.odt
Date of print:	2014-01-14
Document type number:	DOC0800

ELLSI version:	2.0.1

Document History

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

Revision	Chapter	Changes versus previous version	Date
1.6	-	Converted to new manual template / editorial changes	2013-04-17
	2.5.1	Updated statement about sequence numbering. (CAN telegrams have own numbering separated from other telegrams)	
	2.6	Added new command *ELLSI_CMD_UNREGISTER*.	
	2.6.8	Added new sub-commands *ELLSI_IOCTL_CAN_STATUS*, *ELLSI_IOCTL_BUS_STATISTIC*, *ELLSI_IOCTL_GET_TIMESTAMP*, *ELLSI_IOCTL_GET_TIMESTAMP_FREQ* and *ELLSI_IOCTL_GET_SERIAL*.	
	2.6.5	*ellsiExtRegistration* struct was enhanced.	
	2.6.4, 2.6.8.4	*ELLSI_CMD_REGISTER* and *ELLSI_IOCTL_SET_SJA1000_ACMR* now deprecated.	
	2.4	Added chapter “Future protocol changes/enhancements”.	
	3.	Added chapter “ELLSI over WebSocket”.	
1.7	2.6.5	Added flag to “ELLSI_CMD_REGISTERX”	2014-01-14

Technical details are subject to change without further notice.

Table of contents

1. Overview

1.1 Intention

ELLSI offers the possibility to use an esd EtherCAN/2 on all platforms not (yet) supported by esd NTCAN drivers (e.g. Mac OS, PLCs with Ethernet capability, etc.).

For all platforms with an existing NTCAN driver, we suggest to use NTCAN instead of ELLSI for communication with the EtherCAN/2.

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/2 and analysing UDP-datagrams obtained from the ELLSI-server on the esd EtherCAN/2 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/2, 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 than 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/2.

1.4.2 Number of client connections

The number of client connections to the ELLSI server is currently limited to 8, to not overstrain the EtherCAN/2 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. (See also chapter 3., “ELLSI over WebSocket”)

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_REGISTERX* command).

1.4.5 Some thoughts about performance

Here are some proposals to maximize ELLSI performance on an esd EtherCAN/2:

- 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 (use *ELLSI_REGFLAG_CANMAXTHROUGHPUT* in RegisterX telegram to provoke this)
- 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 it, 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.

2.2 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! (i.e. 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 Future protocol changes/enhancements

To stay compatible to future protocol changes an ELLSI client must set reserved values to 0 when sending telegrams to the server and ignore reserved/unknown values from server.
Additionally a client must accept increasing payload lengths from server and ignore the new, not yet known to him, content.
It’s also recommended to avoid *ELLSI_CMD_REGISTER* and to use *ELLSI_CMD_REGISTERX* instead – that includes the client’s supported protocol version (use *ELLSI_IOCTL_CAN_STATUS* to obtain the server’s protocol version).

2.5 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 *0*
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

2.5.1 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 (while CAN telegrams have a separated squence number).

2.6 Commands

2.6.1 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
ELLSI_CMD_UNREGISTER                6
```

2.6.2 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_IOCTL_CAN_STATUS              7
ELLSI_IOCTL_BUS_STATISTIC           8
ELLSI_IOCTL_GET_TIMESTAMP           9
ELLSI_IOCTL_GET_TIMESTAMP_FREQ     10
ELLSI_IOCTL_GET_SERIAL             11
ELLSI_SUBCMD_TXDONE                128
ELLSI_SUBCMD_AUTOACK               256
```

2.6.3 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*

ELLSI_CMD_NOP will always set *lastState* to *0*.

2.6.4 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*

This command is deprecated, please use *ELLSI_CMD_REGISTER**X*** instead. (*ELLSI_CMD_REGISTER* is still supported for backward-compatibility)

lastState is set to *0* for successful registration. All values unequal to *0* stand for a failed registration.

2.6.5 ELLSI_CMD_REGISTERX

This command supersedes the register command described above, it allows the user to have influence on some ELLSI-server parameters and informs the server about the client's protocol version. For that 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*	

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;
      uint16_t    flags;
      uint8_t     clientProtocolVersion;
      uint8_t     netNumber;
      uint32_t    reserved[7];
} 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 *0* 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 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. 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>**
flags	unsigned 16-bit	Ignored when *clientProtocolVersion* is *0*. Bit 0: *netNumber* is valid. Bit 1: maximize CAN throughput (try to send multiple frames in a single telegram, by small delay)
clientProtocolVersion	unsigned 8-bit	Value from *ELLSI_PROTOCOL_VERSION* #define shall be used.
netNumber	unsigned 8-bit	CAN net number on server side that shall be used. (Needs bit in *flags* to be enabled, see above)
reserved[7]	7x unsigned 32-bit	Reserved for future use (set to *0*)

lastState is set to 0 for successful registration. Non-zero values indicate failed registration.

2.6.6 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	

2.6.6.1 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 on server side: *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. See also *ELLSI_IOCTL_GET_TIMESTAMP_FREQ* and *ELLSI_IOCTL_GET_TIMESTAMP*. Must be set to *0* for CAN TX messages

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*)

Note:

The *msg_lost* member does not reflect messages lost by lost ELLSI telegrams – the actual number of lost frames can be much higher.

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 *0* indicates successful completion of the ELLSI-server internal *canSend()*-command, but **does not necessarily indicate a successful transmission of the corresponding CAN frame(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.

2.6.6.2 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.

In short:

TXDONE frames are received like all other CAN frames and identified by a bit in the *len* member.

2.6.7 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/2, 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*

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

2.6.8 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 sub-commands exist:

2.6.8.1 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	

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.

lastState is set to *0* for success, non-zero for failure.

2.6.8.2 ELLSI_IOCTL_CAN_SET_BAUDRATE

By means of this sub-command 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*	

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.

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

2.6.8.3 ELLSI_IOCTL_CAN_GET_BAUDRATE

To read back the currently baud rate set on the EtherCAN/2, send the following telegram to the ELLSI-server:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*0*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*4*
	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*	

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 and the ELLSI-server completed successfully, non-zero means failure.

2.6.8.4 ELLSI_IOCTL_SET_SJA1000_ACMR

Deprecated. Not available in EtherCAN/2.

2.6.8.5 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/2 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	*4*
	subcommand	*ELLSI_IOCTL_GET_LAST_STATE*
	reserved	*0*

As answer you will get a telegram like this:

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

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*, *ELLSI_CMD_CAN_TELEGRAM*, *ELLSI_CMD_HEARTBEAT*, etc.
lastSubcommand	unsigned 32-bit	Last sub-command processed by ELLSI-server: *ELLSI_IOCTL_NOP*, *ELLSI_IOCTL_CAN_ID_ADD*, *ELLSI_IOCTL_CAN_SET_BAUDRATE*, etc.
lastState	32-bit	For states returned by the commands and sub-commands see the corresponding descriptions of commands and sub-commands
lastRxSeq	unsigned 32-bit	The last sequence number the ELLSI-client sent by the appropriate command to the ELLSI-server
reserved	16 bytes	For future protocol extensions

2.6.8.6 ELLSI_IOCTL_CAN_STATUS

CAN_IF_STATUS:

```
typedef struct
{
      uint16_t    hardware;
      uint16_t    firmware;
      uint16_t    driver;
      uint16_t    dll;
      uint32_t    boardstatus;
      uint8_t     boardid[14];
      uint16_t    features;
} CAN_IF_STATUS;
```

Please refer to NTCAN API manual for details. Only the *dll* member has a different meaning with ELLSI: it's the server's ELLSI protocol version.

To request the interface status from the ELLSI-server send the following telegram:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*0*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*4*
	subcommand	*ELLSI_IOCTL_CAN_STATUS*
	reserved	*0*

As answer you will get a telegram like this:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*x*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*4 + sizeof(CAN_IF_STATUS)*
	subcommand	*ELLSI_IOCTL_CAN_STATUS*
	reserved	*0*
Payload	*result (unsigned 32-bit)*	
	CAN_IF_STATUS	

When result is non-zero only the *dll* member (the server's ELLSI protocol version) is valid. The *lastState* value is set to *result*.

2.6.8.7 ELLSI_IOCTL_BUS_STATISTIC

NTCAN_BUS_STATISTIC:

```
typedef struct
{
      uint64_t                timestamp;
      NTCAN_FRAME_COUNT       rcv_count;
      NTCAN_FRAME_COUNT       xmit_count;
      uint32_t                ctrl_ovr;
      uint32_t                fifo_ovr;
      uint32_t                err_frames;
      uint32_t                rcv_byte_count;
      uint32_t                xmit_byte_count;
      uint32_t                aborted_frames;
      uint32_t                reserved[2];
      uint64_t                bit_count;
} NTCAN_BUS_STATISTIC;
```

NTCAN_FRAME_COUNT:

```
typedef struct {
      uint32_t                std_data;
      uint32_t                std_rtr;
      uint32_t                ext_data;
      uint32_t                ext_rtr;
} NTCAN_FRAME_COUNT;
```

Please refer to NTCAN API manual for details.

To request the bus statistics from the ELLSI-server send the following telegram:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*0*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*4*
	subcommand	*ELLSI_IOCTL_BUS_STATISTIC*
	reserved	*0*

As answer you will get a telegram like this:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*x*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*4 + sizeof(NTCAN_BUS_STATISTIC)*
	subcommand	*ELLSI_IOCTL_BUS_STATISTIC*
	reserved	*0*
Payload	*result (unsigned 32-bit)*	
	NTCAN_BUS_STATISTIC	

When *result* is non-zero *NTCAN_BUS_STATISTIC* is not valid. The *lastState* value is set to *result*.

2.6.8.8 ELLSI_IOCTL_GET_TIMESTAMP

To request the current CAN timestamp from the ELLSI-server send the following telegram:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*0*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*4*
	subcommand	*ELLSI_IOCTL_GET_TIMESTAMP*
	reserved	*0*

As answer you will get a telegram like this:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*x*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*12*
	subcommand	*ELLSI_IOCTL_GET_TIMESTAMP*
	reserved	*0*
Payload	*result (unsigned 32-bit)*	
	timestamp (unsigned 64-bit)	

When *result* is non-zero *timestamp* is not valid. The *lastState* value is set to *result*.

2.6.8.9 ELLSI_IOCTL_GET_TIMESTAMP_FREQ

To request the CAN timestamp frequency (in Hz) from the ELLSI-server send the following telegram:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*0*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*4*
	subcommand	*ELLSI_IOCTL_GET_TIMESTAMP_FREQ*
	reserved	*0*

As answer you will get a telegram like this:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*x*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*12*
	subcommand	*ELLSI_IOCTL_GET_TIMESTAMP_FREQ*
	reserved	*0*
Payload	*result (unsigned 32-bit)*	
	timestampFrequency (unsigned 64-bit)	

When *result* is non-zero *timestampFrequency* is not valid. The *lastState* value is set to *result*.

2.6.8.10 ELLSI_IOCTL_GET_SERIAL

To request the device serial number from the ELLSI-server send the following telegram:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*0*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*4*
	subcommand	*ELLSI_IOCTL_GET_SERIAL*
	reserved	*0*

As answer you will get a telegram like this:

Header	**magic**	*ELLSI_MAGIC*
	sequence	*x*
	command	*ELLSI_CMD_CTRL*
	payloadLen	*8*
	subcommand	*ELLSI_IOCTL_GET_SERIAL*
	reserved	*0*
Payload	*result (unsigned 32-bit)*	
	serial (32 bit)	

When *result* is non-zero *serial* is not valid. The *lastState* value is set to *result*. Please refer to NTCAN API manual for details about the serial number format.

2.6.8.11 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.

2.6.9 ELLSI_CMD_UNREGISTER

As UDP is connection-less a “disconnected” client could be recognized only by timeouts. With version 2.0.0 of the ELLSI-server this command has been added to optionally perform a proper “disconnect”.
As the client is usually “cleared” immediately when the server receives this command it’s not valid to request *lastState* afterwards (and the server usually won’t respond to it).

Send this telegram to unregister:

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

3. ELLSI over WebSocket

Beginning with Version 2.0.0 of the ELLSI-server it also supports the WebSocket protocol, which is TCP/IP based.
The UDP Datagrams described here can be imagined as WebSocket messages then – the protocol remains the same, which means:

- The server will still unregister idle clients – although TCP is connection oriented
- The server will still ignore CAN telegrams with wrong sequence number – although TCP guarantees ordered packets
- and so on

The EtherCAN/2 supports using ELLSI over WebSocket parallel to ELLSI over UDP, each of it with its own limit of max clients – it's not recommended to exhaust these limits, see also 1.4.5, "Some thoughts about performance".

4. Order Information

Type	Properties	Order No.
EtherCAN/2	Ethernet-CAN-Gateway (Incl. CAN-DRV-CD Windows/Linux)	C.2051.02
Software		
CAN-DRV-CD Windows/Linux	CAN-DRV-CD CD-ROM Windows & Linux (Incl. Hostdrivers for EtherCAN/2, incl. ELLSI samples)	*

* Current drivers are available for download at www.esd.eu

Table 1: Order information

PDF Manuals
Manuals are available in English and usually in German as well. For availability of English manuals see table below.
Please download the manuals as PDF documents from our esd website www.esd.eu for free.

Manuals		Order No.
ELLSI Manual-ME	ELLSI manual in English – this manual	C.2051.23
CAN-API-ME	NTCAN Part 1: Structure, Function and C/C++ API, Application Developers Manual (English) NTCAN Part 2: Installation, Configuration and Firmware Update, Installation Guide (English)	C.2001.21

Table 2: Available manuals

Printed Manuals
If you need a printout of the manual additionally, please contact our sales team: sales@esd.eu for a quotation. Printed manuals may be ordered for a fee.