# ELLSI

## EtherCAN Low Level Socket Interface

## Software Manual

For Products EtherCAN/2 (C.2051.02) and EtherCAN/3-FD (C.2055.62)

## Notes

The information in this document has been checked carefully and is considered to be entirely reliable. esd electronics makes no warranty of any kind regarding the material in this document and assumes no responsibility for any errors that may appear in this document. In particular, the descriptions and technical data specified in this document may not be constituted to be guaranteed product features in any legal sense.

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



**esd electronics gmbh**
Vahrenwalder Str. 207
30165 Hannover
Germany

Tel.: +49-511-37298-0
Fax: +49-511-37298-68
E-Mail: info@esd.eu
Internet: www.esd.eu

**Links**
esd electronics gmbh assumes no liability or guarantee for the content of Internet pages to which this document refers directly or indirectly. Visitors follow links to websites at their own risk and use them in accordance with the applicable terms of use of the respective websites.

**Trademark Notices**
Windows® is a registered trademark of Microsoft Corporation in the United States and other countries.
All other trademarks, product names, company names or company logos used in this manual are reserved by their respective owners.

# Document Information

| | |
|---|---|
| Document file: | I:\Texte\Doku\MANUALS\PROGRAM\CAN\ELLSI\ELLSI_Manual_en_19.docx |
| Date of print: | 2025-02-18 |
| Document-type number: | DOC0800 |

| | | |
|---|---|---|
| EtherCAN/2 | Firmware version: | 2.0.1 |
| EtherCAN/3-FD | Firmware version: | 1.0.2 |
| ELLSI | Protocol Version | 1.3.0 |

**Document History**
The changes in the document listed below affect changes in the hardware as well as changes in the description of the facts, only.

| Rev. | 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, 2.6.9 | 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 |
| 1.8 | 2.6.5 | Added flag ELLSI_REGFLAG_ENABLE_INTERACTION to . Named other available flags on (ELLSI_REGFLAG_NETVALID and ELLSI_REGFLAG_CANMAXTHROUGHPUT) | 2016-05-27 |
| 1.9 | 2.6.7 | Added ELLSI_CMD_CANX_TELEGRAM for CAN FD support | 2025-02-18 |
| | 2.6.5 | Added flag ELLSI_REGFLAG_CAN_FD for CAN FD support | |
| | | | |

Technical details are subject to change without further notice.

### Typographical Conventions

Throughout this manual the following typographical conventions are used to distinguish technical terms.

| Convention | Example |
|---|---|
| File and path names | **`/dev/null`** or **`<stdio.h>`** |
| Function names | ***open()*** |
| Programming constants | `NULL` |
| Programming data types | `uint32_t` |
| Variable names | *Count* |

### Number Representation

All numbers in this document are base 10 unless designated otherwise. Hexadecimal numbers have a prefix of 0x, and binary numbers have a prefix of 0b. For example, 42 is represented as 0x2A in hexadecimal and 0b101010 in binary.

# Table of Contents

# 1 Overview

## 1.1 Intention

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

**For all platforms with an existing NTCAN driver, it is recommended to use NTCAN instead of ELLSI for communication with all EtherCAN devices.**

## 1.2 Functional Principle

ELLSI has been designed to be as simple as possible. There is no prebuilt API for using ELLSI, but example code is available to serve as a foundation for implementing a custom API.

The basic usage of ELLSI consists of:

1. Creating an ELLSI telegram on the client to configure communication
2. Transmitting the telegram via UDP to the ELLSI server on the EtherCAN device
3. Receiving and analyzing ELLSI telegrams, which the server sends back via UDP

Alternatively, the EtherCAN/3-FD also supports communication over TCP. This provides a more reliable connection but requires additional effort, as the ELLSI telegrams must be reassembled from the TCP data stream.

**Connection Setup and Communication**
Registration of the ELLSI Client
The client must first register with the ELLSI server.

Heartbeat Mechanism

- If no data transmission occurs, both the client and server must send heartbeat messages at regular intervals.
- If the client does not receive any data or heartbeat messages from the server within a defined time period, it assumes that the server is no longer reachable (e.g., due to a network failure or a reset of the EtherCAN device). In this case, the client must re-register.
- The server behaves similarly: If it does not receive any data or heartbeat messages from the client within a defined time period, it assumes that the client is no longer active and stops transmitting data to it.

Client Configuration
After successful registration, the client must:

- Set the baud rate
- Activate the desired CAN IDs for which it wants to receive data

Once these steps are completed, the client is ready to send and receive CAN telegrams.

**Ensuring Correct Message Order**
To ensure that CAN telegrams are processed in the correct order, each telegram is assigned a sequence number. This guarantees that the messages are correctly sorted and interpreted.

# 1.3 ELLSI vs. esd NTCAN API

Care has been taken to make ELLSI as compatible as possible with the standard esd NTCAN API. It is therefore recommended to read the esd NTCAN API documentation in parallel to this document. The esd NTCAN API documentation often contains 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, it is recommended to use NTCAN instead of ELLSI for communicating with the EtherCAN devices.

## 1.4.2 Network Protocols

Depending on the EtherCAN device you use, multiple network protocols are supported.

**UDP**

- UDP is supported by EtherCAN, EtherCAN/2 and EtherCAN/3-FD. It is a fast, but unreliable communication, which allows transmitting the maximum throughput.

**TCP**

- TCP is supported by EtherCAN/3-FD. It is a connection-oriented protocol that guarantees reliable, ordered, and error-checked delivery of data by establishing a connection before transmitting and managing data integrity throughout the transfer.

**WebSocket**

- WebSocket is supported by EtherCAN/2 and EtherCAN/3-FD. It operates over TCP and provides continues, two-way communication. The EtherCAN/2 includes a HTTP WebSockets on top of the ELLSI WebSocket which can be used to self-host your own website. This feature has been discontinued on EtherCAN/3-FD for security reasons.

## 1.4.3 Number of Client Connections

The number of client connections to the ELLSI server is limited based on the platform you are one.

- EtherCAN
  - 8 connections over UDP
- EtherCAN/2
  - 8 connections over UDP and WebSockets each
- EtherCAN/3-FD
  - 4 connections over UDP, TCP or WebSockets

## 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 supports this function, but it must be activated as an optional parameter for the *ELLSI_CMD_REGISTERX* command.

### 1.4.5 CAN Ports

On devices with more than one CAN port, ELLSI allows the user to select the CAN port as an optional parameter for the *ELLSI_CMD_REGISTERX* command.

### 1.4.6 Some Thoughts about Performance

The following suggestions are provided to maximize ELLSI performance on EtherCAN devices:

- 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* or n * *ellsiCMSG_X* |

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 devices.

## 2.2 Port

The default port for the ELLSI server is **2209**.

## 2.3 Byte Order

**Note:**
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 with 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 the server and ignore the new, not yet known to him, content.
It’s also recommended to avoid and to use instead – that includes the client’s supported protocol version (use  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*<br>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 sequence 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
ELLSI_CMD_CANX_TELEGRAM         7
```

## 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_IOCTL_CAN_SET_BAUDRATEX       12
ELLSI_IOCTL_CAN_GET_BAUDRATEX       13
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* |

**Note:**
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 set up 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).<br>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.<br>Default is 30 (which is equivalent to a multiplier of 30/10 = 3.0).<br>The valid range is 10 <= x <= 100. |
| *canTxQueueSize* | unsigned 32-bit | Size of message queue used for CAN TX telegrams<br>Use *0* for default (default is 128).<br>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.<br>(default is *CAN_READ_MAXLEN*= 50)<br>The valid range is<br>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.<br>Use *0* for default (default is 0 ms). |
| *flags* | unsigned 16-bit | Ignored when *clientProtocolVersion* is *0*.<br><br>ELLSI_REGFLAG_NETVALID (Bit 0):<br>- *netNumber* is valid.<br><br>ELLSI_REGFLAG_CANMAXTHROUGHPUT (Bit 1):<br>- Maximize CAN throughput (try to send multiple frames in a single telegram, by small delay).<br><br>ELLSI_REGFLAG_CAN_FD (Bit 2):<br>- CAN FD operation enabled. Received frames are send in the *ellsiCMSG_X* format. Only available on EtherCAN/3-FD.<br><br>ELLSI_REGFLAG_ENABLE_INTERACTION (Bit 4):<br>- Enable CAN interaction on this registered connection (see also esd NTCAN API manual) |
| *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 sent:

| | | |
|---|---|---|
| **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]<br>Bit 4 : RTR<br>Bit 5 : TXDONE (see *ELLSI_SUBCMD_TXDONE*)<br>Bit 6-7 : Reserved |
| *msg_lost* | unsigned 8-bit | Counter for lost CAN RX messages. Allow the user to detect data overrun on server side:<br>*msg_lost* = 0 : no lost messages<br>0 < *msg_lost* < 255 : # of lost frames = value of *msg_lost*<br>*msg_lost* = 255 : # of lost frames ≥ 255 |
| *reserved[2]* | 2x unsigned 8-bit | Only meaningful together with *ELLSI_SUBCMD_TXDONE.*<br>In this case used to allow association of TX-DONE messages with previously sent CAN TX messages<br>(see *ELLSI_SUBCMD_TXDONE*) |
| *data[8]* | 8x unsigned 8-bit | CAN data bytes |
| *timestamp* | 64-bit | Time stamp for CAN RX messages.<br>See also *ELLSI_IOCTL_GET_TIMESTAMP_FREQ* and *ELLSI_IOCTL_GET_TIMESTAMP*.<br>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 sent 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/X* is logically ORed with *ELLSI_CMSG_LEN_TXDONE* (0x20). Additionally, the two reserved bytes in *ellsiCMSG_T/X* are echoed back! If you e.g. set these 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_CANX_TELEGRAM

This ELLSI telegram is similar the *ELLSI_CMD_CAN_TELEGRAM* command. But instead of *ellsiCMSG_T* it uses *ellsiCMSG_X* which has a data field of 64 bytes for CAN FD. Only available on EtherCAN/3-FD.

| | | |
|---|---|---|
| **Header** | **magic** | *ELLSI_MAGIC* |
| | **sequence** | Sequence# [or *0*] |
| | **command** | *ELLSI_CMD_CANX_TELEGRAM* |
| | **payloadLen** | n * sizeof(*ellsiCMSG_X*) |
| | **subcommand** | *0* [or *ELLSI_SUBCMD_TXDONE*] |
| | **reserved** | *0* |
| **Payload** | *ellsiCMSG_X #1* | |
| | ⋮ | |
| | *ellsiCMSG_X #n* | |

### 2.6.7.1 ellsiCMSG_X

The *ellsiCMSG_X* 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[64];
      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]<br>Bit 4 : RTR<br>Bit 5 : TXDONE (see *ELLSI_SUBCMD_TXDONE*)<br>Bit 6 : Reserved<br>Bit 7 : CAN FD |
| *msg_lost* | unsigned 8-bit | Counter for lost CAN RX messages. Allow the user to detect data overrun on server side:<br>*msg_lost* = 0 : no lost messages<br>0 < *msg_lost* < 255 : # of lost frames = value of *msg_lost*<br>*msg_lost* = 255 : # of lost frames ≥ 255 |
| *reserved[2]* | 2x unsigned 8-bit | Only meaningful together with *ELLSI_SUBCMD_TXDONE.*<br>In this case used to allow association of TX-DONE messages with previously sent CAN TX messages<br>(see *ELLSI_SUBCMD_TXDONE*) |
| *data[64]* | 8x unsigned 8-bit | CAN data bytes |
| *timestamp* | 64-bit | Time stamp for CAN RX messages.<br>See also *ELLSI_IOCTL_GET_TIMESTAMP_FREQ* and *ELLSI_IOCTL_GET_TIMESTAMP*.<br>Must be set to *0* for CAN TX messages |

To ease porting applications between ELLSI and NTCAN, this structure is compatible to the *CMSG_X*-structure in the esd NTCAN API. (see *ntcan.h*). The other information to this command is equal to the *ELLSI_CMD_CAN_TELEGRAM* command.

### 2.6.8 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 fixed to 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 devices, 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 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*.

## 2.6.9 ELLSI_CMD_CTRL

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

Currently the following sub-commands exist:

### 2.6.9.1 ELLSI_IOCTL_CAN_ID_ADD/DELETE

By means of *ELLSI_IOCTL_CAN_ID_ADD* the client can enable specific 11-bit 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.

29-bit IDs can only be enabled or disabled as a whole. To enable or disable 29bit IDs add or delete an Id with the 30th bit set.

There are also special IDs which enable specific events. This can be used to be notified when the CAN bus state changes. Please refer to the NTCAN API Manual for more information.

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.

### 2.6.9.2 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.

### 2.6.9.3 ELLSI_IOCTL_CAN_GET_BAUDRATE

To read back the currently baud rate set on the EtherCAN devices, 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 an 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.

### 2.6.9.4 ELLSI_IOCTL_SET_SJA1000_ACMR

Deprecated. Not available for EtherCAN/2 and EtherCAN/3-FD.

### 2.6.9.5 ELLSI_IOCTL_GET_LAST_STATE

*ELLSI_IOCTL_GET_LAST_STATE* allows you to get some information about the last command processed by ELLSI on the EtherCAN devices 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 an 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 |

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

### 2.6.9.7 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                rcv_count_fd;
      uint32_t                xmit_count_fd;
      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 an 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*.

### 2.6.9.8 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 an 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)*<br>*timestamp (unsigned 64-bit)* | |

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

### 2.6.9.9 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 an 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*.

### 2.6.9.10 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 an 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)*<br>*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.

### 2.6.9.11 ELLSI_IOCTL_CAN_SET_BAUDRATEX

By means of this sub-command you can set the CAN baud rate. It offers more possibilities to set the baudrate than the *ELLSI_IOCTL_CAN_SET_BAUDRATE* sub-command. It can be used to set baud rate for CAN FD. Only available on EtherCAN/3-FD.

NTCAN_BAUDRATE_X:

```
typedef struct {
    uint16_t            mode;           /* Mode word                          */
    uint16_t            flags;          /* Control flags                      */
    NTCAN_TDC_CFG       tdc;            /* TDC configuration parameters       */
    NTCAN_BAUDRATE_CFG  arb;            /* Bit rate during arbitration phase  */
    NTCAN_BAUDRATE_CFG  data;           /* Bit rate during data phase         */
} NTCAN_BAUDRATE_X;
```

NTCAN_TDC_CFG:

```
typedef struct {
    uint8_t tdc_mode;               /* TDC Mode                               */
    uint8_t ssp_offset;             /* SSP Offset                             */
    int8_t  ssp_shift;              /* SSP Shift                              */
    uint8_t tdc_filter;             /* TDC Filter                             */
} NTCAN_TDC_CFG;
```

NTCAN_BAUDRATE_CFG:

```
typedef struct {
    union {
        uint32_t idx;               /* esd bit rate table index               */
        uint32_t rate;              /* Numerical bit rate                     */
        uint32_t btr_ctrl;          /* BTR register (CAN Controller layout)   */
        struct {
            uint16_t brp;           /* Bit rate pre-scaler                    */
            uint16_t tseg1;         /* TSEG1 register                         */
            uint16_t tseg2;         /* TSEG2 register                         */
            uint16_t sjw;           /* SJW register                           */
        } btr;
    } u;
} NTCAN_BAUDRATE_CFG;
```

Please refer to NTCAN API manual for details.

To set the baudrate via this command send the following telegram:

| | | |
|---|---|---|
| **Header** | **magic** | *ELLSI_MAGIC* |
| | **sequence** | *0* |
| | **command** | *ELLSI_CMD_CTRL* |
| | **payloadLen** | *4* |
| | **subcommand** | *ELLSI_IOCTL_CAN_SET_BAUDRATEX* |
| | **reserved** | *0* |
| **Payload** | *NTCAN_BAUDRATE_X* | |

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

### 2.6.9.12 ELLSI_IOCTL_CAN_GET_BAUDRATEX

To read back the currently baud rate set on the EtherCAN devices, 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_BAUDRATEX* |
| | **reserved** | *0* |

As an answer you will get a telegram like this:

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

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

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

Only available on EtherCAN/3-FD.

### 2.6.9.13 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.

## 2.6.10 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

ELLSI on EtherCAN/2 and EtherCAN/3-FD 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

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 1.4.6, "

# 4 Order Information

## 4.1 Associated Hardware Articles

| Type | Properties | Order No. |
|---|---|---|
| EtherCAN/3-FD | CAN FD-Ethernet-Gateway<br>(incl. CAN-DRV-CD Windows/Linux) | C.2055.62 |
| EtherCAN/2 | CAN-Ethernet-Gateway<br>(incl. CAN-DRV-CD Windows/Linux) | C.2051.02 |
| **Software:** | | |
| CAN-DRV-CD Windows/Linux | CAN-DRV-CD CD-ROM Windows & Linux<br>(Incl. EtherCAN SDK with ELLSI examples) | * |

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

**Table 1:** Order information

## 4.2 Manuals

**PDF Manuals**
For the availability of the manuals see the table below.
Please download the manuals as PDF documents from our esd website https://www.esd.eu for free.

| Manuals | | Order No. |
|---|---|---|
| ELLSI-ME | ELLSi software manual in English (this manual) | C.2051.23 |
| CAN-API-ME | NTCAN-API: Application Developers Manual<br>NTCAN-API: Driver Installation Guide | C.2001.21 |

**Table 2:** Available Manuals