NTLIN

Structure, Function and C/C++API

Application Developers Manual

esd electronics 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

Notes

The information in this document has been carefully checked and is believed to be entirely reliable. esd electronics 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 electronics reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance, or design.

All rights to this documentation are reserved by esd electronics. Distribution to third parties, and reproduction of this document in any form, whole or in part, are subject to esd electronics' written approval.

© 2022 esd electronics gmbh, Hannover

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

	This manual contains important information and instructions on safe and efficient handling of the NTLIN. Carefully read this manual before commencing any work and follow the instructions. The manual is a product component, please retain it for future use.

Trademark Notices

CANopen® and CiA® are registered EU trademarks of CAN in Automation e.V.
QNX® is a trademark of QNX Software Systems Limited, and is a registered trademark and/or used in certain jurisdictions.
Windows® is a registered trademark of Microsoft Corporation in the United States and other countries.
Linux® is the registered trademark of Linus Torvalds in the United States and/or other countries.
PCI Express® is a registered trademark of PCI-SIG.

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\LIN\C.2007.21_NTLIN\API\LIN-API_Application_Developers_Manual_en_10.docx
Date of print:	2022-08-09
Document-type number:	DOC0800

Products covered by this document

CAN-Driver / SDK	(Driver) Revision
Linux® Driver (32-/64-Bit)	From 4.1.4
QNX® 6 / QNX® 7 Driver	4.x.y

Hardware	Order No.
CAN-PCI/402-2-FD with LIN Addon CAN-PCIe/402-Slot2-LIN	C.2049.64 with C.2045.12
CAN-PCIe/402-2-FD with LIN Addon CAN-PCIe/402-Slot2-LIN	C.2045.64 with C.2045.12

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
1.0	-	First English LIN-API manual	2022-08-09

Technical details are subject to change without further notice.

Classification of Warning Messages and Safety Instructions

This manual contains noticeable descriptions for a safe use of the NTLIN and important or useful information.

NOTICE

Notice statements are used to notify people on hazards that could result in things other than personal injury, like property damage.

NOTICE
This NOTICE statement contains the general mandatory sign and gives information that must be heeded and complied with for a safe use.

INFORMATION

INFORMATION
Notes to point out something important or useful.

Data Safety

This software can be used to establish a connection to data networks. This may allow attackers to compromise normal function, get illegal access or cause damage.
esd does not take responsibility for any damage caused by the device if operated at any networks. It is the responsibility of the device's user to take care that necessary safety precautions for the device's network interface are in place.

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.

Abbreviations

API	Application Programming Interface
C99	Informal name for ISO/IEC 9899:1999
CAN	Controller Area Network
CPU	Central Processing Unit
CiA	CAN in Automation
FD	Flexible Data
FIFO	First-In-First-Out
HW	Hardware
I/O	Input/Output
LIN	Local Interconnect Network
N/A	Not Applicable
OS	Operating System
PCI	Peripheral Component Interconnect (Computer Bus)
PCIe	Peripheral Component Interconnect Express (Computer Bus)
SDK	Software Development Kit
SoC	System on Chip

Table of Contents

1 Introduction 8
1.1 Scope 8
1.2 Overview 8
1.3 References 8
1.4 Terminology 9
1.5 Description of the LIN Bus 10
1.6 Features 12
2 NTLIN-API and Device Driver 13
2.1 Abstraction Layer 13
2.2 Implementation Details Overview 14
2.2.1 Operating System Integration 14
2.2.2 Interaction 15
2.2.3 Extended Features 15
3 LIN Communication with NTLIN-API 16
3.1 Overview 16
3.2 Bit Rate Configuration 17
3.2.1 Automatic Bit Rate Detection 17
3.3 Interaction 17
3.4 Timestamps 18
3.4.1 Implementation 18
3.4.2 Usage 18
4 API Reference 19
4.1 Initialization and Clean-up 20
4.1.1 linOpen() 20
4.1.2 linClose() 22
4.2 Configuration 23
4.2.1 linIoctl() 23
4.2.2 linIdAdd() 25
4.2.3 linIdDelete() 26
4.3 Blocking Calls 27
4.3.1 linWait() 27
4.4 LIN Master 28
4.4.1 linMasterTxHeader() 28
4.5 LIN Slave 29
4.5.1 linSlaveTxCreate() 29
4.5.2 linSlaveTxUpdate() 30
4.5.3 linSlaveTxDestroy() 31
4.5.4 linSlaveRxTake() 32
5 Data Types 33
5.1 Simple Data Types 33
5.1.1 NTLIN_HANDLE 33
5.1.2 NTLIN_RESULT 34
6 Return Codes 35
6.1 General Return Codes 35
7 Example Source 39
7.1 LIN Master 39
7.2 LIN Slave 41
7.3 LIN Logger 45
8 Order Information 47

List of Tables

Table 1: Supported bus systems 13
Table 2: Parameter Usage Types.. 19
Table 3: Simple C99 data types used by NTLIN-API 33
Table 4: Order information.. 47
Table 5: Available Manuals 47

List of Figures

Figure 1: Example: LIN Cluster with Slaves and Master and Logger 10
Figure 2: Master, Slave and Logger (example with CAN-PCIe/402-2-FD)..................................... 11
Figure 3: LIN Message Interaction 15

1 Introduction

This document describes the software design and the application layer of the cross-platform communication interface for **esd** Local Interconnect Network. The well-structured Application Programming Interface (API) allows an easy integration into any application. The functional range and versatility of the implementation provide all necessary mechanisms to control, configure, and monitor LIN networks.
Within this document the API is referred as **NTLIN-API** and the common implementation as a combination of a device driver, a library NTCAN and a library as **NTLIN**.

1.1 Scope

This document covers the description of the NTLIN architecture which usually consists of an OS and LIN hardware specific device driver, a (shared) NTCAN library and a (shared) library which exports the application interface to integrate LIN I/O into an application.

1.2 Overview

Chapter 1 contains a general overview on the structure of this manual.
Chapter 2 provides general overview about the features of the NTLIN implementation.
Chapter 3 describes the NTLIN concepts and how the API can be integrated into an application to realize a LIN bus-based communication.
Chapter 4 describes the Application Programming Interface (API) with all functions followed by **Chapter 5** which contains the reference to the simple and complex data types used with NTLIN-API.
Chapter 6 is a description of the error codes which are returned by the NTLIN-API functions described in the previous chapters in case of a failure.
Chapter 7 contains three complete, small example applications which demonstrate Master, Slave and Logger communications.
Chapter 8 contains the order information.

1.3 References

(1) esd electronics gmbh, NTCAN-API Part 2: Installation Guide, Revision 4.7, Hannover, 2021
(2) esd electronics gmbh, NTCAN-API Part 1: Application Developers Manual, Revision 5.5, Hannover, 2022

1.4 Terminology

Within this manual you will encounter the following terms:

Event-ID	Identifier of an NTLIN Event.
Frame	A frame consists of a header (provided by the master task) and a response (provided by a slave task).
Header	A header is the first part of a frame; it is always sent by the master task.
LIN	**L**ocal **I**nterconnect **N**etwork A serial bus system (also known as LIN bus) that was originally designed for use in vehicles but is now also used in automation technology.
LIN Board	A LIN board is a hardware which makes one or more physical LIN ports available for use by an application. This is either an **esd** LIN *Interface* or an embedded system with an on-board LIN *Controller*.
LIN Controller	A chip whose hardware processes the LIN bus protocols. This can be a stand-alone chip which is solely dedicated to this function or a *System on Chip* (SoC) which integrates one or more LIN controllers as external interface.
LIN Device	The (logical) application view to a physical LIN port.
LIN Handle	Logical link between the application and a physical LIN port. An application can open several LIN handles to the same or to different LIN ports.
LIN Interface	A LIN interface is a dedicated **esd** hardware which is either connected to a local bus (PCI(e), USB, PC/104, etc.) of a CPU or remotely (Ethernet, Wireless, etc.) to a host system.
LIN Node	All hardware connected to the LIN bus. This can be any hardware with a LIN port ranging from a simple sensor up to a complex control system.
LIN Port	The physical connector to a LIN bus which is handled by a LIN controller.
NTLIN-ID	Identifier (32-Bit) of a LIN Frame message which allows to distinguish between an NTLIN Event and a LIN PID.
PID	Protected Identifier of a LIN request
Response	A frame consists of a header and a response. The response is always sent by the slave task.

1.5 Description of the LIN Bus

The Local Interconnect Network (LIN) bus is a serial bus system designed for use in vehicles as a less performant (and expensive) supplement to the CAN bus with a completely different physical layer.

The LIN bus has a master/slave architecture comprising maximum 16 nodes and works with a single master and up to 15 slaves. The LIN master controls the data transfer by sending requests or commands to the slaves and determines the data rate on the LIN bus.
A LIN cluster consists of the master node and the connected slave nodes. Optionally a logger can be connected to the LIN cluster that records the LIN frames. All nodes contain a slave task. The master node also contains the master task.

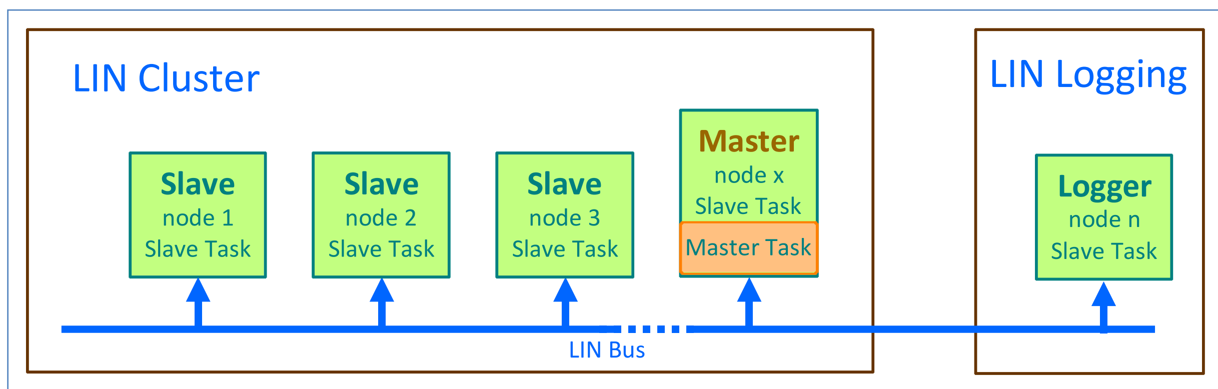


Figure 1: Example: LIN Cluster with Slaves and Master and Logger

The master task determines when which frame is transferred on the bus and provides the header of the frames. The slave tasks provide the data transmitted with the frames.

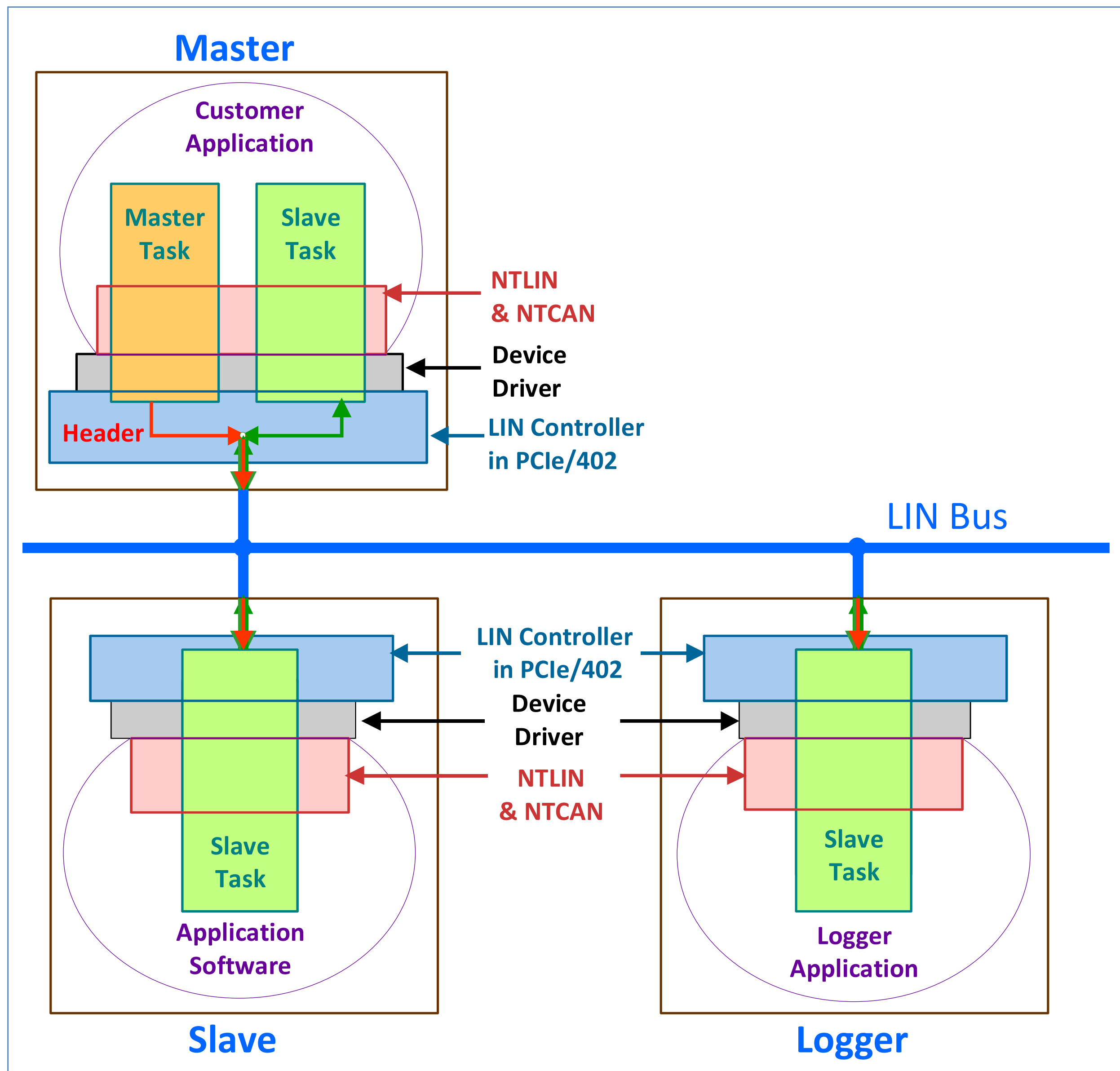


Figure 2: Master, Slave and Logger (example with CAN-PCIe/402-2-FD)

All shown software and hardware layers are used to implement the logical task entities (slave and master respectively).

Dedicated esd boards also provide LIN ports in addition to their CAN ports.
The information content of a LIN frame is like that of a Classical CAN Base Format Message. For this reason, the device driver for the respective esd boards also handle the LIN communication instead of using a dedicated device driver for this.
The NTLIN-API is implemented on top of the NTCAN library instead of accessing the device driver directly.

1.6 Features

The **esd** NTLIN-API is a compact and easy to handle programming interface for integrating the control of LIN based networks in (real-time) applications. The implementation provides the following features*1 which are described in more detail in chapter 2.2.

- ➢ Device driver support of OS specific features
- ➢ Support for Plug & Play and hot-pluggable CAN devices
- ➢ Multitasking/multithreading support
- ➢ Event driven and/or polled LIN I/O
- ➢ LIN message interaction
- ➢ Multiprocessor and multi-core support
- ➢ Firmware update for LIN modules with local operating system
- ➢ Hardware independent LIN node number mapping
- ➢ Common OS independent API on all platforms
- ➢ Sophisticated acceptance filtering
- ➢ Event based status and error indication
- ➢ Timestamps for LIN headers and responses
- ➢ Flexible bit rate configuration
- ➢ LIN bus baud rate detection

*1 Some of the features require special (CAN) hardware or operating system (OS) support. Please refer to (2) (Table 9 in chapter 3.16) for details which features are supported by your hardware/OS combination.

2 NTLIN-API and Device Driver

This chapter contains an overview about the features of the **esd** CAN device drivers and the NTLIN Application Programming Interface (API).

2.1 Abstraction Layer

The NTLIN-API is hardware and OS independent providing the same functionality for the following list of **esd** LIN boards.

Bus	CAN board
PCI	CAN-PCI/402-2-FD with LIN Addon (CAN-PCIe/402-Slot2-LIN)
PCIe	CAN-PCIe/402-2-FD with LIN Addon (CAN-PCIe/402-Slot2-LIN)

Table 1: Supported bus systems

The NTLIN-API is implemented and tested for the operating systems QNX, Linux and Windows® 10/11. The availability for other operating systems depends on the availability of a current esd can driver for the PCIe/402 family for this operating system.
Please contact our support team: support@esd.eu for more information or if you want NTLIN support for other operating systems.

Using the NTLIN-API gives the application developer the possibility to change the **esd** LIN hardware as well as the operating system without the need to change the LIN I/O related parts of the application.

2.2 Implementation Details Overview

This chapter explains several aspects of NTLIN listed in chapter 1.6 in more detail.

2.2.1 Operating System Integration

The actual NTLIN implementation is usually a combination of a library and a working esd CAN driver installation.

Multiprocessor and multi-core processor support

Device driver for operating systems which support more than one processor or core have been developed and tested to support this environment.

Support for multiple LIN ports

Due to the device driver approach driver types of all classes can co-exist on one host and each driver is able to support simultaneously several LIN boards of its device class which have one or more physical CAN ports.

To make the underlying **esd** LIN hardware transparent from the application point of view each physical port is assigned an individual logical net number in the range from 0 to 255. The details about the configuration of the logical net numbers are OS specific and described in the (1).

As the link between a physical LIN port and the application is based on the logical net numbers it is possible to switch easily between different **esd** LIN board types without the need to change the application.

Multitasking / Multithreading support

The NTLIN implementation is not limited to create just one link to a LIN port but allows creating several simultaneous links to the same port with different tasks/processes or even different threads of the same process. The support for this behaviour is based on NTLIN handles. Each handle virtualizes a LIN controller so the underlying physical LIN port can be used by several processes/threads at the same time.

If the host OS supports the handle concept, it is used by the driver otherwise it is emulated.

2.2.2 Interaction

It is possible to run any number processes and/or threads using the same physical LIN port. If there is for example a LIN based control application implementing the LIN master, several other applications implementing LIN slaves and even a monitor tool in another process on the same host using the identical physical LIN port. As shown in the picture below, this feature is supported in all **esd** LIN implementations and is called ***Interaction***.

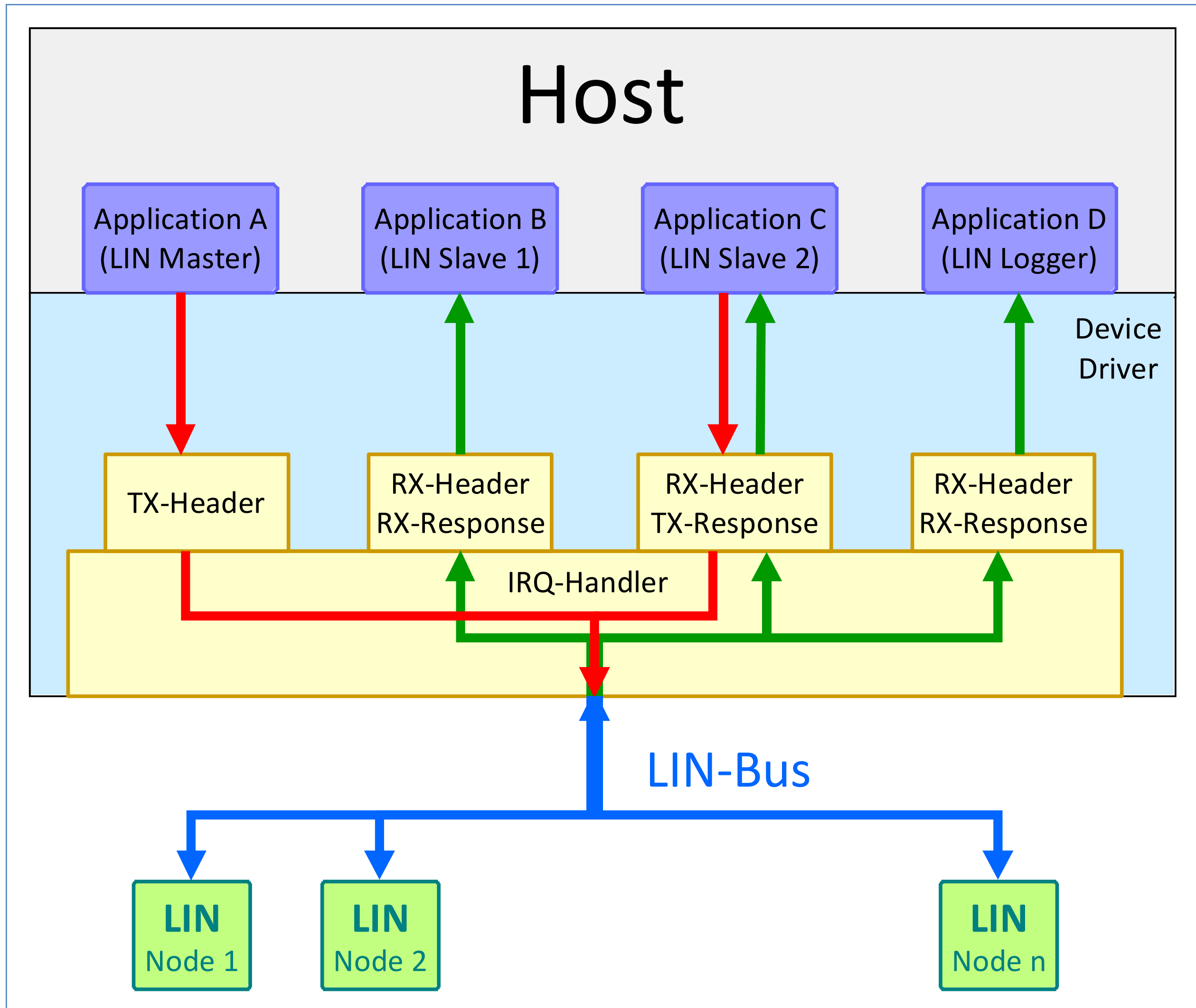


Figure 3: LIN Message Interaction

2.2.3 Extended Features

Timestamp Support

The NTLIN-API supports a timestamp for LIN headers and responses. Depending on the LIN board this timestamp is either captured in software by the device driver in the interrupt handler or, more accurate, applied by the (active) LIN board (refer to chapter 3.4 for more details).

3 LIN Communication with NTLIN-API

This chapter provides an overview of the general functionality of the LIN communication with the NTLIN-API, before the API function calls, data types, etc. will be explained in greater detail in the following chapters.

3.1 Overview

An application which wants to access the LIN bus has to create a logical link to a physical LIN port with ***linOpen()***. This link is represented by an opaque handle of the data type `NTLIN_HANDLE` which is an input parameter for most NTLIN-API calls. A process can use multiple handles to the same or to different physical LIN ports simultaneously.

To distinguish between different physical LIN ports the device driver assigns a logical net number to each port and an application uses this number as an input parameter of ***linOpen()*** to reference the LIN port. This mechanism allows the use of LIN boards with more than one physical LIN port as well as the simultaneous use of several LIN boards of the same or different board type. The process of assigning different logical net numbers to physical LIN ports is hardware and operating system specific (please refer to (1) for further details).

From the application point of view each handle references a virtual LIN controller.

Before any LIN I/O can be performed the bit rate of the physical LIN port has to be configured once by using ***linIoctl().***
In order to prevent two applications from trying to initialise the same physical LIN port with different bit rates, the current configuration can be requested with ***linIoctl()***.

For LIN message transmission the API offers calls to implement master-, slave- and/or logger-functionality.
In order to manage any LIN event the API offers one blocking call ***linWait()*** and several non-blocking calls. A 64-bit high resolution timestamp (see chapter 3.4) is applied to each received LIN event.

The general purpose API call ***linIoctl()*** is available to set or get further device and or/driver configuration options or to request any kind of (diagnostic) information.

In case of an error each API call returns a corresponding error code. The individual error codes will be explained in greater detail in chapter 6.

3.2 Bit Rate Configuration

For bitrate configuration details see the description of the `NTLIN_IOCTL_SET_BAUDRATE` and `NTLIN_IOCTL_GET_BAUDRATE I/O` control commands in chapter 4.2.1, from page 23.

3.2.1 Automatic Bit Rate Detection

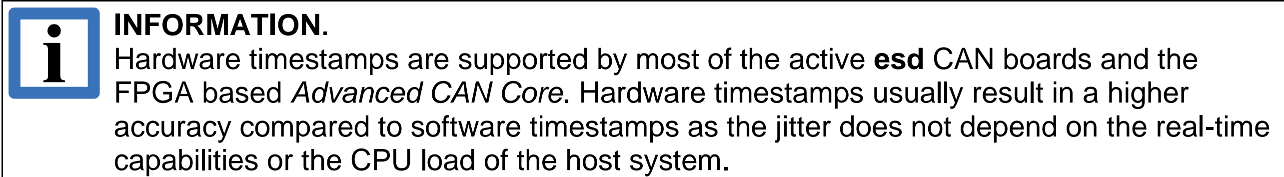
The automatic baud rate detection is not yet implemented. Especially because this would only be useful for slave only nodes.

3.3 Interaction

If a LIN master transmits a LIN header, a LIN controller usually does not receive its own transmitted header. As in a multitasking/multithreading environment it is often required by the application logic or it is at least convenient to receive the LIN header transmitted by a LIN master task, the **esd** LIN driver implements a feature called *Interaction.* which is illustrated in **Figure 3**.

3.4 Timestamps

Most **esd** LIN boards support capturing the time stamp of the moment a LIN header, or a LIN response was received. Depending on the device capabilities the time stamping is performed either in hardware by the LIN board or in software by the driver's interrupt handler using a high-resolution counter of the host CPU.



INFORMATION.
Hardware timestamps are supported by most of the active **esd** CAN boards and the FPGA based *Advanced CAN Core*. Hardware timestamps usually result in a higher accuracy compared to software timestamps as the jitter does not depend on the real-time capabilities or the CPU load of the host system.

3.4.1 Implementation

A timestamp has no default resolution to prevent time consuming calculations in the driver. Instead, the timestamps are realized as 64-bit free-running counter with the most accurate available time stamping source. The application can query the frequency of the time stamping source in order to scale the timestamps online or offline and can query the current timestamp to link them to the absolute system time.

3.4.2 Usage

This chapter summarizes the typical steps to use the timestamp interface (in FIFO Mode):

1. Open LIN handle with ***linOpen().***
2. Somewhere at the beginning of your application request once the following information:
   - ➢ The frequency of the timestamp counter (which is specific for any **esd** LIN board and/or host OS). This can be accomplished by calling ***linIoctl()*** with the `NTLIN_IOCTL_GET_TIMESTAMP_FREQ` command.
   - ➢ The current value of the timestamp counter, in order to correlate received timestamps with your system time. This can be accomplished by calling ***linIoctl()*** with the `NTLIN_IOCTL_GET_TIMESTAMP` command.

4 API Reference

This chapter describes each NTLIN API function logically grouped into the sections

- Initialization and Clean-up.
- Configuration
- Receiving Data
- Transmitting Data
- Miscellaneous

Each API function documentation is structured identically into a description of

1. Syntax
2. Functionality
3. Arguments
4. Return Values
5. Usage
6. Requirements
7. Further References

Arguments

In each function description the arguments are described in a tabular format. A usage type (see table below) in squared brackets is followed by the description of the argument usage in the specific function.

Usage Type	***Meaning***
[in]	Indicates the parameter is input. The function reads from the buffer. The caller provides the buffer and initializes it.
[out]	Indicates the parameter is output. The function writes to the buffer. The caller provides the buffer, and the function initializes it.
[in/out]	Indicates the parameter is input and output. The caller provides the buffer and initializes it. The function both reads from and writes to the buffer.

Table 2: Parameter Usage Types

4.1 Initialization and Clean-up

This section describes the functions available to establish and release a logical link to a physical LIN port.

4.1.1 linOpen()

The function establishes a logical link to a physical LIN port.

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linOpen(
                int            net,           /* net number                    */
                uint32_t       flags,         /* mode flags                    */
                int32_t        logqueuesize,  /* log_queuesize or -1           */
                NTLIN_HANDLE   *handle        /* OUT: Handle                   */
                                );
```

Description:

The function establishes a logical link to a physical LIN port by returning a LIN handle on success which is an input parameter for all NTLIN-API functions described in this chapter. Every LIN handle represents a virtual LIN controller with an individual I/O configuration which is independent from other handles apart from common characteristics of the referenced physical LIN port (e.g.: bit rate).

INFORMATION.
The maximum number of available handles is limited by the driver and operating system specific global or per process limits.

An application can open several handles to the same physical LIN port with different modes of operation or configuration as well as to different physical LIN ports.

Arguments:

net

[in] The logical net number which is assigned to the physical LIN port in the range from 0 to `NTLIN_MAX_NETS`.

flags

[in] This parameter is a bit mask which is reserved for future use.

logqueuesize

[in] Size of the receive queue for LIN frames. The maximum size is limited to the platform specific `NTLIN_MAX_RX_QUEUESIZE`.

**handle*

[out] Pointer to a memory location where the LIN driver will store the LIN handle on success.

Return Values:

On success, the function returns `NTLIN_SUCCESS`. On error, one of the error codes described in chapter 6.

Usage:

The function has to be called before any other function described in this chapter because the returned LIN handle is the input argument for nearly all NTLIN-API functions.

Requirements:
N/A.

See also:
Further information on the handle returned by this function can be found in the description of the data structure `NTLIN_HANDLE` .

4.1.2 linClose()

Close the link to the physical LIN port.

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linClose(
                NTLIN_HANDLE  handle );  /* Handle                      */
```

Description:

The function closes the link to the physical LIN port. As consequence all handle specific resources are released.

NOTICE
Blocking API requests on this handle will return with an error.

Arguments:

handle
[in] LIN handle.

Return Values:

Upon success, `NTLIN_SUCCESS` is returned and one of the error codes described in chapter 6 in case of a failure.

Usage:

N/A.

Requirements:

A valid LIN handle.

See also:

Description of ***linOpen()***.

4.2 Configuration

This section describes the functions available to configure the LIN communication.

4.2.1 linIoctl()

The function performs a variety of control functions on LIN devices.

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linIoctl(
                NTLIN_HANDLE  handle,   /* Handle                         */
                uint32_t      cmd,      /* Command specifier              */
                void *        arg );    /* Ptr to command specific argument*/
```

Description:

This function is a universal entry to configure or request additional LIN I/O configuration. The data type of the input or output data referenced by *arg*, depends on the control command *cmd*. The Usage section contains a list of all supported commands together with their input or output data type.

Arguments:

handle

[in] LIN handle.

cmd

[in] Command.

arg

[in/out] Pointer to *cmd* dependent input or output data.

Return Values:

Upon success, `NTLIN_SUCCESS` is returned or one of the error codes described in chapter 6 in case of a failure

Usage:

List of supported commands with their command specific arguments. If a command does not require an argument, *arg* has to be set to NULL.

`NTLIN_IOCTL_SET_BAUDRATE`	Argument: `uint32_t`	In
Configure the bitrate.		

`NTLIN_IOCTL_GET_BAUDRATE`	Argument: `uint32_t`	Out
Returns the configured bitrate.		

`NTLIN_IOCTL_GET_SERIAL`	Argument: `uint32_t`	Out
The hardware serial number of the LIN board referenced by *handle* is stored at the memory location referenced by *arg* (if supported by the LIN board). As a LIN board can have several physical LIN ports the same serial number is returned for all logical LIN networks related to this board. The serial number is returned in an encoded format. Each of the two upper nibbles of the value represents one of the leading letters of the production lot number (0x0 => 'A' , 0x1 => 'B', …,0xF => 'P'). The remaining 24 bits are the numerical part. <u>Example:</u> The value 0x1D012345 is the serial number BN074565. If reading the serial number is not supported by the device, a 0 is returned which results in the serial number AA000000 according to the encoding described above.		

`NTLIN_IOCTL_GET_TIMESTAMP_FREQ`	Argument: `uint64_t`	Out
The resolution of the timestamp counter (in Hz) of the LIN port referenced by *handle* is stored at the memory location referenced by *arg* if timestamps are supported by LIN hardware, device driver and operating system.		

`NTLIN_IOCTL_GET_TIMESTAMP`	Argument: `uint64_t`	Out
The value of the timestamp counter related to the LIN port referenced by *handle* is stored at the memory location referenced by *arg* if timestamps are supported by LIN hardware, device driver and operating system.		

`NTLIN_IOCTL_GET_NATIVE_OS_HANDLE`	Argument: `Native OS handle type`	Out
The OS specific handle or file descriptor of the LIN device referenced by *handle* is stored at the memory location referenced by *arg* (see description of `NTLIN_HANDLE` for details). The table below contains the native handle type which is returned.		

Operating System	Native OS handle type
Windows	HANDLE
Linux	int
QNX	int
LynxOS	int

`NTLIN_IOCTL_SET_TIMEOUT`	Argument: `uint32_t`	Out
Re-configure the timeout of this handle. The argument is a reference to the previously initialized memory location with the new timeout value in ms. The new value will be used for the next linWait() function. A pending linWait() request is not affected by this change.		

`NTLIN_IOCTL_MASTER_SEL`	Argument: `uint32_t`	Out
Enable/disable the Master pull-up resistor (1K). Enable with a value of ‘1’, disable with a value of ‘0’.		

Requirements:

See also:

Description of ***linOpen()***

4.2.2 linIdAdd()

Enabling a LIN-ID to be received with ***linWait().***

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linIdAdd(
                NTLIN_HANDLE   handle,      /* Handle                       */
                int32_t        id,          /* LIN message identifier      */
                uint32_t       type);
```

Description:

Only LIN ids, enabled by this call, will be received with ***linWait().*** The type value decides about the event type. It is possible to receive LIN-Headers and/or LIN-Responses.

Arguments:

handle

[in] LIN handle.

id

[in] LIN ID

type

[in] One of `NTLIN_FILTER_HEADER`, `NTLIN_FILTER_RESPONSE` or `NTLIN_FILTER_ALL`

Return Values:

Upon success, `NTLIN_SUCCESS` is returned or one of the error codes described in chapter 6 in case of a failure.

Usage:

Requirements:

See also:

Description of ***linWait()***, ***linOpen()***

4.2.3 linIdDelete()

Disabling a LIN-ID to be received with ***linWait()***

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linIdDelete(
                NTLIN_HANDLE  handle,    /* Handle                         */
                int32_t       id,        /* LIN message identifier         */
                uint32_t      type);
```

Description:

LIN ids, enabled by ***linIdAdd()*** can be disabled for receiving with ***linWait().*** The type value decides about the event type. It is possible to disable LIN-Headers and/or LIN-Responses.

Arguments:

handle

[in] LIN handle.

id

[in] LIN ID

type

[in] One of `NTLIN_FILTER_HEADER`, `NTLIN_FILTER_RESPONSE` or `NTLIN_FILTER_ALL`

Return Values:

Upon success, `NTLIN_SUCCESS` is returned or one of the error codes described in chapter 6 in case of a failure.

Usage:

Requirements:

See also:

Description of ***linIdAdd(), linWait(), linOpen()***

4.3 Blocking Calls

This section describes the functions available for blocking calls.

4.3.1 linWait()

Waiting for a LIN event.

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linWait(
                NTLIN_HANDLE handle,      /* Handle                        */
                int32_t      *id,         /* Out: LIN ID                   */
                int32_t      *len,        /* Out: 1...8: Response length   */
                                          /* Out:    -1: Header received   */
                void         *data,       /* Out: 1<=len<=8 data bytes     */
                uint64_t     *timeStamp); /* Out: timestamp                */
```

Description:

The function waits for LIN frames and is the only blocking call.

Arguments:

handle

[in] LIN handle.

**id*

[out] Pointer to a location where the LIN ID is stored.

**len*

[out] NULL or a pointer to a memory location where the length of 1 to 8 bytes of a LIN response is stored. A value of -1 indicates a received header.

**data*

[out] NULL or a pointer to a memory location with a minimal size of 8 bytes to store a LIN-response.

**timestamp*

[out] NULL or a pointer to a memory location where the 64-bit timestamp is stored.

Return Values:

Upon success, `NTLIN_SUCCESS` is returned or one of the error codes described in chapter 6 in case of a failure.

Usage:

Requirements:

N/A.

See also:

Description of ***linIdAdd()***, ***linIdDelete()***, ***linOpen()***.

4.4 LIN Master

This section describes the functions available to implement a LIN master task.

4.4.1 linMasterTxHeader()

Sending a LIN header.

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linMasterTxHeader(
                NTLIN_HANDLE  handle,  /* Handle                           */
                int32_t       id );    /* LIN ID                           */
```

Description:

The function sends a LIN header.

Arguments:

handle

[in] LIN handle.

id

[in] LIN ID

Return Values:

Upon success, `NTLIN_SUCCESS` is returned or one of the error codes described in chapter 6 in case of a failure.

Usage:

The call is intended for a master application which cyclically executes a LIN schedule table.

Requirements:

See also:

Description of ***linOpen()***.

4.5 LIN Slave

This section describes the functions available to implement a LIN slave task.

4.5.1 linSlaveTxCreate()

Creating a Tx response object for a LIN id

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linSlaveTxCreate(
                NTLIN_HANDLE  handle,    /* Handle                          */
                int32_t       id,        /* LIN ID                          */
                uint32_t      flags);    /* Obj flags                       */
```

Description:

The function generates a Tx response object for a LIN ID, which is automatically sent if it is enabled, and the master requests it by sending the appropriate LIN header. After creation, the object is disabled.

Arguments:

handle

[in] LIN handle.

id

[in] LIN id

flags

[in] Object flags

LIN_TX_ONCE – If this flag is set and the object is enabled, the response is only sent once after the last call to ***linSlaveTxUpdate().***

Return Values:

Upon success, `NTLIN_SUCCESS` is returned or one of the error codes described in chapter 6 in case of a failure.

Usage:

The call is intended for slave applications to implement a Tx response object.

Requirements:

See also:

Description of ***linSlaveTxUpdate(), linSlaveTxDestroy(), linOpen()***.

4.5.2 linSlaveTxUpdate()

Updating a Tx object for a LIN id.

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linSlaveTxUpdate(
                NTLIN_HANDLE  handle,    /* Handle                           */
                int32_t       id,        /* LIN ID                           */
                int32_t       len,       /* 0...8, 0 => Disabled             */
                void          *data);    /* 1<=len<=8 data bytes             */
```

Description:

The function updates the data bytes of a Tx response object and sets the transfer *len*.

Arguments:

handle
[in] LIN handle.

id
[in] LIN id

len
[in] Count (1…8) of data bytes or 0 to disable the object.

**data*
[in] Pointer to data bytes (1...8), or NULL if len=0 (disabled)

Return Values:

Upon success, `NTLIN_SUCCESS` is returned or one of the error codes described in chapter 6 in case of a failure.

Usage:

The call is intended for slave applications to implement a Tx response object.

Requirements:

See also:

Description of ***linSlaveTxCreate(), linSlaveTxDestroy(), linOpen()***.

4.5.3 linSlaveTxDestroy()

Destroying a Tx response object for a LIN id

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linSlaveTxDestroy(
                NTLIN_HANDLE  handle,    /* Handle                          */
                int32_t       id);       /* LIN ID                          */
```

Description:

The function destroys a Tx data object for a LIN ID, which is no longer automatically sent.

Arguments:

handle

[in] LIN handle.

id

[in] LIN id

Return Values:

Upon success, `NTLIN_SUCCESS` is returned or one of the error codes described in chapter 6 in case of a failure.

Usage:

The call is intended for slave applications to implement a Tx response object.

Requirements:

See also:

Description of ***linSlaveTxCreate()***, ***linOpen()***.

4.5.4 linSlaveRxTake()

Read the Rx object for a LIN id.

Syntax:

```
EXPORT NTLIN_RESULT CALLTYPE linSlaveRxTake(
              NTLIN_HANDLE  handle,      /* Handle                        */
              int32_t       id,          /* LIN ID                        */
              int32_t       *len,        /* OUT: Data length 0...8 0=NoData*/
              void          *data,       /* OUT: 1 to 8 data bytes        */
              uint64_t      *timestamp); /* OUT: time stamp               */
```

Description:

The function reads the actual data bytes of an Rx object.
Upon return, *len* contains the number data bytes inside the object. The first call of this function for a LIN id creates the RX response object.

Arguments:

handle

[in] LIN handle.

id

[in] LIN id.

**len*

[out] Pointer to a memory location where the driver can store the count of data bytes (0...8) or 0 if the object has never received data.

**data*

[out] Pointer to data bytes (1...8)

**timestamp*

[out] Pointer to a memory location where the 64-bit timestamp is stored.

Return Values:

Upon success, `NTLIN_SUCCESS` is returned or one of the error codes described in chapter 6 in case of a failure.

Usage:

The call is intended for slave applications to implement an Rx response object.

Requirements:

See also:

Description of ***linOpen()***.

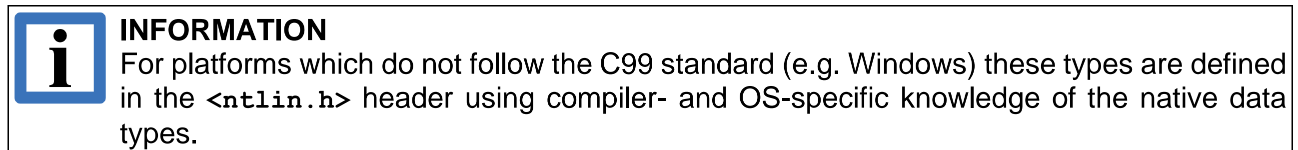
5 Data Types

To stay cross-platform portable with respect to different CPU architectures and compilers the NTLIN-API header **<ntlin.h>** does not use the native standard integer data types of the C language. Instead, the data types in the header **`<stdint.h>`** are supported which defines various integer types and related macros with size constraints.

Specifier	Signing	Bytes	Range
`int8_t`	Signed	1	−128...127
`uint8_t`	Unsigned	1	0...255
`int16_t`	Signed	2	−32,768...32,767
`uint16_t`	Unsigned	2	0...65,535
`int32_t`	Signed	4	−2,147,483,648...2,147,483,647
`uint32_t`	Unsigned	4	0...4,294,967,295
`int64_t`	Signed	8	−9,223,372,036,854,775,808...9,223,372,036,854,775,807
`uint64_t`	Unsigned	8	0...18,446,744,073,709,551,615

Table 3: Simple C99 data types used by NTLIN-API

These data types are part of the ISO/IEC 9899:1999 standard which is also commonly referred to as *C99* standard.



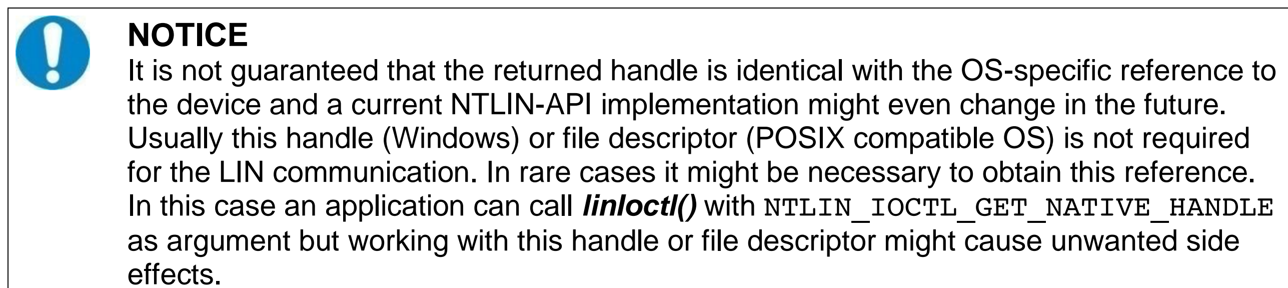
INFORMATION
For platforms which do not follow the C99 standard (e.g. Windows) these types are defined in the **`<ntlin.h>`** header using compiler- and OS-specific knowledge of the native data types.

5.1 Simple Data Types

This section describes the simple data types defined by the NTLIN-API in alphabetical order. They all start with the prefix **`NTLIN_`** with respect to a clean namespace.

5.1.1 NTLIN_HANDLE

The type defines an opaque operating system specific reference to a physical LIN port. This handle is the input or output parameter of all NTLIN-API functions. As an input parameter the handle is validated by the called function. This type should be used in applications instead of the platform specific native type for cross-platform portability.



NOTICE
It is not guaranteed that the returned handle is identical with the OS-specific reference to the device and a current NTLIN-API implementation might even change in the future. Usually this handle (Windows) or file descriptor (POSIX compatible OS) is not required for the LIN communication. In rare cases it might be necessary to obtain this reference. In this case an application can call ***linIoctl()*** with `NTLIN_IOCTL_GET_NATIVE_HANDLE` as argument but working with this handle or file descriptor might cause unwanted side effects.

NOTICE
If an application wants to indicate that a handle is invalid, the portable definition `NTLIN_NO_HANDLE` should be used for this instead of the native representation of an invalid handle.

5.1.2 NTLIN_RESULT

The type defines the operating system specific data type for the return value of every NTLIN-API function. This type should be used in applications instead of an OS specific native type for cross-platform portability.

6 Return Codes

All NTLIN-API functions return a status starting with the prefix 'NTLIN_', which should always be evaluated by the application. If the call returns an error code, the content of all returned values referenced by pointers are undefined and must not be evaluated by the application.
The constants for the returned values are defined in **<ntlin.h>**.
For cross-platform portability an application should refer only to these constants, because each operating system has got its own 'number area' for the numerical values of errors. Therefore, different numerical values are used for the same return status on different operating systems. Furthermore a few constants for errors which are not LIN specific and are usually generated autonomously by the operating system (such as `NTLIN_INVALID_HANDLE`) are mapped to already existing error constants of the operating system to increase the portability.

Below all returned values are listed in a table. The values are divided into the severity categories *Successful*, *Warning* and *Error*. Furthermore, a description of the error reason, a possible solution as well as the NTLIN-API functions which might return this result are part of the description.

6.1 General Return Codes

NTLIN_SUCCESS

No error.

Category	Successful
Cause	The call was terminated without errors. The content of all returned values referenced by pointers are valid and must only be evaluated by the application.
Function	All functions

NTLIN_HANDLE_FORCED_CLOSE

Abortion of a blocking operation.

Category	Warning/Error
Cause	A blocking operation was canceled, because another thread called ***linClose()*** for this handle or the driver was terminated. If a call returns with this error code, the handle is no longer valid.
Solution	If the procedure was unintended by the application, check why handles are being closed.
Function	***linWait()***

NTLIN_INSUFFICIENT_RESOURCES

Insufficient internal resources.

Category	Error
Cause	The operation could not be completed because of insufficient internal resources.
Solution	
Function	***linOpen()***, ***linIdAdd()***

NTLIN_INVALID_DRIVER

Driver and NTLIN library are not compatible.

Category	Error
Cause	The version of the NTLIN library requires a more recent driver version.
Solution	Use a more recent driver version.
Function	***linOpen()***

NTLIN_INVALID_FIRMWARE

Driver and firmware are incompatible.

Category	Error
Cause	The version of the device driver requires a more recent firmware version.
Solution	Update the firmware of the active CAN board.
Function	***linOpen()***

NTLIN_INVALID_HANDLE

Invalid LIN handle.

Category	Error
Cause	An invalid handle was passed to a function call.
Solution	1. Check whether the handle was correctly opened with ***linOpen()***. 2. Check whether the handle was not closed previously with ***linClose()***.
Function	All functions except ***linOpen()***

NTLIN_INVALID_HARDWARE

Driver and hardware are incompatible.

Category	Error
Cause	The version of the device driver is incompatible with the hardware.
Solution	Use another driver version.
Function	***linOpen()***

NTLIN_INVALID_PARAMETER

Invalid parameter.

Category	Error
Cause	An invalid parameter was passed to the library.
Solution	Check all arguments for this call for validity.
Function	All functions.

NTLIN_NET_NOT_FOUND

LIN device not found.

Category	Error
Cause	The logical network number specified when opening ***linOpen()*** does not exist.
Solution	1. Check the logical LIN network number. 2. Check whether the driver, to which this network number should be assigned, was started correctly.
Function	***linOpen()***

NTLIN_NOT_IMPLEMENTED

Command for ***linIoctl()*** is not implemented.

Category	Error
Cause	The argument *cmd* of ***linIoctl()*** is not implemented or supported by the library, device driver or hardware.
Solution	1. Check the *cmd* parameter for validity. 2. If the argument *cmd* is valid, check if a newer driver/library is available which supports this command.
Function	***linIoctl()***

NTLIN_NOT_SUPPORTED

The argument of the call is valid but not supported.

Category	Error
Cause	The argument of the call is valid, but the requested property is not supported because of hardware and/or firmware limitations.
Solution	1. Check all arguments for this call for validity. 2. If the arguments are valid, check if a newer or different firmware for this hardware is available which supports this feature. 3. If the requested feature cannot be supported due to hardware constraints, you might have to use a different **esd** LIN board.
Function	***linIoctl()***, ***linIdAdd()***, ***linIdDelete()***

7 Example Source

This chapter contains complete code examples for using the NTLIN-API.

7.1 LIN Master

```
#include <stdio.h>
#include <time.h>
#include <ntlin.h>

#define CLOCK_MS(ms) do {struct timespec ts; clock_gettime(CLOCK_MONOTONIC, &ts); *(ms) = ts.tv_sec
* 1000 + ts.tv_nsec / 1000000;} while(0)
#define SLEEP_MS_ABS(ms) do {struct timespec ts = {ms/1000, (ms%1000)*1000000};
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &ts, NULL);} while(0)

static uint32_t clockms;

/* Schedule for a 500 ms cycle */
static struct _schedule_slot {
    int32_t id;
    uint32_t delay;
    int32_t  flags;
} sched_main[] = {   /*   Time: ID    */
    { 1,  100 }, /*   0 ms: ID  1 */
    { 5,  100 }, /* 100 ms: ID  5 */
    { -1, 100 }, /* 200 ms: Empty spare slot */
    { 11, 100 }, /* 300 ms: ID 11 */
    { 12, 100 }, /* 400 ms: ID 12 */
    { 0,    0 }, /* Stop mark */
};


static int manageSched(NTLIN_HANDLE hnd, struct _schedule_slot *slot)
{
    NTLIN_RESULT rc = NTLIN_SUCCESS;

    while(slot->delay != 0) {
        if(slot->id >= 0) {
            rc = linMasterTxHeader(hnd, slot->id);
            if(rc != NTLIN_SUCCESS) {
                fprintf(stderr, "linMasterRequest failed: rc=%d\n", rc);
                break;
            }
        }

        clockms += slot->delay;
        SLEEP_MS_ABS(clockms);
        slot ++;
    }

    return rc;
}


int master(int net, int baud)
{
    int rc;
    NTLIN_HANDLE hnd;
    uint32_t sel;

    rc = linOpen(net, 0, 1, &hnd); /* Open LIN handle: net 0, flags = 0, RX Fifo = 1(Not used by
master) */
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linOpen failed: rc=%d\n", rc);
        return rc;
    }

    rc = linIoctl(hnd, NTLIN_IOCTL_SET_BAUDRATE, &baud); /* Set baudrate to 19200 Baud */
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linIoctl NTLIN_IOCTL_SET_BAUDRATE failed: rc=%d\n", rc);
        linClose(hnd);
        return rc;
    }

    sel = 1;
    rc = linIoctl(hnd, NTLIN_IOCTL_MASTER_SEL, &sel);  /* switch on pullup registor, because we are
the LIN master */
```

```
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linIoctl NTLIN_IOCTL_MASTER_SEL failed: rc=%d\n", rc);
        linClose(hnd);
        return rc;;
    }

    CLOCK_MS(&clockms); /* Get monotonic absolute time in ms */
    printf("Actual clock in ms:%d ms\n", clockms);
    do {
        rc = manageSched(hnd, sched_main);
        if(rc != 0) {
            fprintf(stderr, "\nSchedule error...exiting: rc=%d\n", rc);
        }
    } while(rc == 0); /* Master loop */

    sel = 0;
    linIoctl(hnd, NTLIN_IOCTL_MASTER_SEL, &sel);  /* switch off pullup register */
    linClose(hnd);
    return rc;
}
```

7.2 LIN Slave

```
#include <stdio.h>
#include <inttypes.h>
#include <ntlin.h>

static uint64_t tsFreq, tsStart;

enum PR_ID_TYPE {
    PR_ST,
    PR_TX,
    PR_RX,
};

struct _process_data {
    enum PR_ID_TYPE type;
    uint32_t id;
    int32_t  len;
    uint8_t  data[8];
    uint64_t ts;
    uint32_t errors;
    uint32_t touts;
};

static struct _process_data prData1[] = {
    { PR_RX,  1, 8, {0x00,                                      }, 0, 0, 0 },
    { PR_TX,  5, 2, {0x00, 0x00,                                }, 0, 0, 0 },
    { PR_TX, 11, 5, {0x12, 0x23, 0x34, 0x45, 0x67,              }, 0, 0, 0 },
    { PR_ST,  0, 0, {0x00,                                      }, 0, 0, 0 }, /* Stop mark */
};

static struct _process_data prData2[] = {
    { PR_RX,  1, 8, {0x00,                                      }, 0, 0, 0 },
    { PR_RX,  5, 2, {0x00,                                      }, 0, 0, 0 },
    { PR_RX, 11, 5, {0x00,                                      }, 0, 0, 0 },
    { PR_ST,  0, 0, {0x00,                                      }, 0, 0, 0 }, /* Stop mark */
};


static struct _process_data *getPR(int id, struct _process_data *pr)
{
    for(; pr->type != PR_ST; pr ++ ) {
        if(id == pr->id) {
            return pr;
        }
    }

    return pr;
}


static int printFrame(const char *pre, struct _process_data *pr)
{
    NTLIN_RESULT rc;
    int i;
    uint64_t us;

    if(tsFreq) {
        us = (pr->ts - tsStart) * 1000000 / tsFreq;
    } else {
        us = 0;
    }

    if(pr->len == 0) {
        rc = printf("%s e=%3d t=%3d %6" PRIu64 ".%06" PRIu64 " s id=%2d No Data", pre, pr->errors,
pr->touts, us / 1000000, us % 1000000, pr->id);
    } else {
        rc = printf("%s e=%3d t=%3d %6" PRIu64 ".%06" PRIu64 " s id=%2d len=%x data=", pre, pr-
>errors, pr->touts, us / 1000000, us % 1000000, pr->id, pr->len);

        for(i = 0; i < pr->len; i ++) {
            rc += printf("%02x ", pr->data[i]);
        }
    }

    rc += printf("\n");
    return rc;
}
```

```
static void printFrames(struct _process_data *pr)
{
    for(; pr->type != PR_ST; pr ++) {
        switch(pr->type) {
            case PR_TX:
                printFrame("TX    ", pr);
                break;
            case PR_RX:
                printFrame("RX    ", pr);
                break;
            default:
                ;
        }
    }
}


static int manageId(NTLIN_HANDLE hnd, int32_t id, struct _process_data *prData)
{
    NTLIN_RESULT rc = NTLIN_SUCCESS;

    struct _process_data *pr = getPR(id, prData);
    switch(pr->type) {
        case PR_RX:
            {
                rc = linSlaveRxTake(hnd, id, &pr->len, &pr->data, &pr->ts);
                if(rc != NTLIN_SUCCESS) {
                    fprintf(stderr, "linSlaveTake for id %02x failed: rc=%d\n", id, rc);
                } else {
                    printFrames(prData);
                }
            }
            break;

        case PR_TX:
            {
                int i;
                printFrames(prData);

                for(i = 0; i < pr->len; i ++) {  /* Change some data */
                    pr->data[i] ++;
                }

                rc = linSlaveTxUpdate(hnd, id, pr->len, pr->data); /* Update tx data for the next
cycle */
                if(rc != NTLIN_SUCCESS) {
                    fprintf(stderr, "linSlaveUpdate for ID %02x failed: rc=%d\n", id, rc);
                } else {
                    rc = linIoctl(hnd, NTLIN_IOCTL_GET_TIMESTAMP, &pr->ts);  /* Store the time of
out last update */
                    if(rc != NTLIN_SUCCESS) {
                        fprintf(stderr, "linIoctl NTLIN_IOCTL_GET_TIMESTAMP failed: rc=%d\n", rc);
                    }
                }
            }
            break;

        case PR_ST: /* Cannot happen, but removes a possible compiler warning. */
            break;
    }

    return rc;
}


int setupId(NTLIN_HANDLE hnd, struct _process_data *pr)
{
    NTLIN_RESULT rc = NTLIN_SUCCESS;

    if(pr->id < 0 || pr->id > 63) {
        return EINVAL;
    }

    switch(pr->type) {
        case PR_TX:
            /* Setup TX frame */
            rc = linSlaveTxCreate(hnd, pr->id, 0);
            if(rc != NTLIN_SUCCESS) {
                fprintf(stderr, "linSlaveTxCreate for ID %02x failed: rc=%d\n", pr->id, rc);
                break;
            }
```

```
            rc = linSlaveTxUpdate(hnd, pr->id, pr->len, pr->data);
            if(rc != NTLIN_SUCCESS) {
                fprintf(stderr, "linSlaveTxUpdate for ID %02x failed: rc=%d\n", pr->id, rc);
                break;
            }

            rc = linIdAdd(hnd, pr->id, NTLIN_FILTER_HEADER);
            if(rc != NTLIN_SUCCESS) {
                fprintf(stderr, "linIdAdd for ID %02x failed: rc=%d\n", pr->id, rc);
                break;
            }

            break;

        case PR_RX:
            /* Setup RX frame */
            rc = linSlaveRxTake(hnd, pr->id, &pr->len, &pr->data, &pr->ts);
            if(rc != NTLIN_SUCCESS) {
                fprintf(stderr, "linSlaveRxTake for ID %02x failed: rc=%d\n", pr->id, rc);
                break;
            }

            rc = linIdAdd(hnd, pr->id, NTLIN_FILTER_RESPONSE);
            if(rc != NTLIN_SUCCESS) {
                fprintf(stderr, "linIdAdd for ID %02x failed: rc=%d\n", pr->id, rc);
                break;
            }

            break;

        default:
            rc = NTLIN_INVALID_PARAMETER;
    }

    return rc;
}


int slave(int net, int baud, int whichSlave)
{
    int rc;
    NTLIN_HANDLE hnd;
    uint32_t tout;
    int32_t id;
    struct _process_data *prData, *pr;

    switch(whichSlave) {
        case 1:
            prData = prData1;
            break;

        case 2:
            prData = prData2;
            break;

        default:
            return EINVAL;
    }

    rc = linOpen(net, 0, 64, &hnd); /* Open LIN handle: net 0, flags = 0, RX Fifo = 64 */
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linOpen failed: rc=%d\n", rc);
        return rc;
    }

    rc = linIoctl(hnd, NTLIN_IOCTL_SET_BAUDRATE, &baud);
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linIoctl NTLIN_IOCTL_SET_BAUDRATE failed: rc=%d\n", rc);
        linClose(hnd);
        return rc;
    }

    rc = linIoctl(hnd, NTLIN_IOCTL_GET_TIMESTAMP_FREQ, &tsFreq);
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linIoctl NTLIN_IOCTL_GET_TIMESTAMP_FREQ failed: rc=%d\n", rc);
        linClose(hnd);
        return rc;
    }


    rc = linIoctl(hnd, NTLIN_IOCTL_GET_TIMESTAMP, &tsStart);  /* Store the start time */
```

```
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linIoctl NTLIN_IOCTL_GET_TIMESTAMP failed: rc=%d\n", rc);
    }

    tout = 1000;
    rc = linIoctl(hnd, NTLIN_IOCTL_SET_TIMEOUT, &tout);
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linIoctl NTLIN_IOCTL_SET_TIMEOUT failed: rc=%d\n", rc);
        linClose(hnd);
        return rc;
    }

    /* Setup frames */
    for(pr = prData; pr->type != PR_ST; pr ++) {
        rc = setupId(hnd, pr);
        if(rc != NTLIN_SUCCESS) {
            fprintf(stderr, "setupId for ID %02x failed: rc=%d\n", pr->id, rc);
        }
    }

    printf("baudrate=%d\n", baud);
    printFrames(prData);

    do {
        rc = linWait(hnd, &id, 0, 0, 0);
        switch(rc) {
            case NTLIN_TIMEOUT:
                printf("linWait Timeout\n");
                rc = 0;
                break;

            case NTLIN_SUCCESS:
                rc = manageId(hnd, id, prData);
                break;

            default:
                fprintf(stderr, "linWait failed: rc=%d\n", rc);
                continue;
        }
    } while(rc == 0);

    /* Destroy TX frames */
    for(pr = prData; pr->type != PR_ST; pr ++) {
        if(pr->type == PR_TX) {
            rc = linSlaveTxDestroy(hnd, pr->id);
            if(rc != NTLIN_SUCCESS) {
                fprintf(stderr, "linSlaveTxDestroy for ID %02x failed: rc=%d\n", pr->id, rc);
            }
        }
    }

    linClose(hnd);
    return rc;
}
```

7.3 LIN Logger

```
#include <stdio.h>
#include <inttypes.h>
#include <ntlin.h>

static uint64_t tsFreq, tsStart;


void printFrame(int32_t id, int32_t len, uint8_t *data, uint64_t ts)
{
    int i;
    static int response = 0;
    uint64_t us;
    static uint64_t tsLast = 0;

    if(len == -1) {
        us = ts * 1000000 / tsFreq;

        tsLast = ts;

        if(response == 0) {
            printf("\n");
        }

        printf("Hdr:%6" PRIu64 ".%06" PRIu64 " s id=%2d|", us / 1000000, us % 1000000, id);
        response = 0;
        fflush(stdout);
    } else  {
        us = (ts - tsLast) * 1000000 / tsFreq;

        printf("Rsp:%2" PRIu64 ".%06" PRIu64 " s len=%x data=", us / 1000000, us % 1000000, len);
        for(i = 0; i < len; i ++) {
            printf("%02x ", data[i]);
        }

        printf("\n");
        response = 1;
    }
}


int doLog(NTLIN_HANDLE hnd)
{
    NTLIN_RESULT rc = NTLIN_SUCCESS;
    int32_t id;
    int32_t len;
    uint64_t ts;
    uint8_t data[8];

    rc = linWait(hnd, &id, &len, data, &ts);
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linWait failed: rc=%d\n", rc);
        return rc;
    }

    printFrame(id, len, data, ts);
    return rc;
}


int logger(int net, int baud, int id_start, int id_stop)
{
    int rc;
    NTLIN_HANDLE hnd;
    int32_t id;

    rc = linOpen(net, 0, 64, &hnd);
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linOpen failed: rc=%d\n", rc);
        return rc;
    }

    rc = linIoctl(hnd, NTLIN_IOCTL_SET_BAUDRATE, &baud);
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linIoctl NTLIN_IOCTL_SET_BAUDRATE failed: rc=%d\n", rc);
        linClose(hnd);
        return rc;
    }
```

```
    rc = linIoctl(hnd, NTLIN_IOCTL_GET_TIMESTAMP_FREQ, &tsFreq);
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linIoctl NTLIN_IOCTL_GET_TIMESTAMP_FREQ failed: rc=%d\n", rc);
        linClose(hnd);
        return rc;
    }

    rc = linIoctl(hnd, NTLIN_IOCTL_GET_TIMESTAMP, &tsStart);
    if(rc != NTLIN_SUCCESS) {
        fprintf(stderr, "linIoctl NTLIN_IOCTL_GET_TIMESTAMP failed: rc=%d\n", rc);
        linClose(hnd);
        return rc;
    }

    printf("Net=%d, Baudrate=%d Enableds IDs: %2d ... %2d\n", net, baud, id_start, id_stop);
    for(id = id_start; id <= id_stop; id ++) {
        rc =linIdAdd(hnd, id, NTLIN_FILTER_ALL);
        if(rc != NTLIN_SUCCESS) {
            fprintf(stderr, "linIdAdd failed: rc=%d\n", rc);
            linClose(hnd);
            return rc;
        }
    }

    do {
        rc = doLog(hnd);
    } while(rc == 0);

    linClose(hnd);
    return rc;
}
```

8 Order Information

Type	Properties
NTLIN-API	As part of the Windows CAN SDK (esd software development kit) the NTLIN-API is included in delivery of the corresponding LIN hardware. For other operating systems the NTLIN-API is included in the software packages that are delivered with the corresponding LIN hardware.

Table 4: Order information

PDF Manuals
Please download the manual as PDF document from our esd website https://www.esd.eu for free.

Manuals		Order No.
NTLIN-API ME	NTLIN-API: Application Developers Manual	C.2007.21

Table 5: 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.