PLIN API Documentation

PEAK LIN Application Programming Interface

Table of Contents

1 PLIN-API Documentation

Welcome to the documentation of PLIN API, a PEAK Application

Programming interface for the LIN Bus and Hardware.

In the following chapters you will find all the

information needed to take

advantage of this API .

Last Update: 04.04.2022

2 Introduction

Welcome to the documentation to PLIN-API.

PLIN stands for PEAK LIN Applications and it is a system for the development and use of LIN busses. It is a helpful and extensive product, directed to developers and end-users. The PLIN-API is the Programming Interface to the PLIN system which allow the real-time connection of Windows applications to the LIN busses physically connected to the PC.

In this Chapter

Topics	Description
PLIN Basics (⊠ see page 2)	This section contains an introduction to PLIN.
PLIN-Client and API Basics (⊠ see page 3)	Information and process flow with a PLIN client.
License Regulations (⊠ see page 3)	License regulations to this software.
Contact information (⊠ see page 4)	Contact information - PEAK-System Technik GmbH.

2.1 PLIN Basics

The communication between the PC and an external LIN hardware through a LIN bus is done using a Windows Services called "*PLIN Manager*" (PLinMng.exe). It forms the basis for the interaction between the LIN software and the LIN-PC hardware. The service manages all resources and data flowing between software and hardware.

The interface to the user, like in the PCAN system , are so-called PLIN Clients. With their help, the LIN bus will accessed and its internal resources managed. The PLIN system allows the connection of multiple clients to a hardware.

Unlike the PCAN system, the PLIN system does not support the use of nets. A client connects directly with a hardware. Within the PCAN system, the nets are a kind of extension of the CAN bus in the PC. Since a part of the protocol stack of LIN must be processed in the hardware (in real-time), for a LIN communication is always a hardware needed. Furthermore, the development of a purely virtual LIN bus, using simulation within a PC, is not possible. If there are clients connected to a hardware (eg. PCAN-USB Pro), the communication will be interrupted if that hardware is plugged off.

The following rules are used to dealing with PLIN clients and hardware:

- A PLIN client can be connected to several hardware.
- A hardware supplies several PLIN clients.
- When a client sends, another client will gets the message/response only after the message has physically appeared on the bus.
- LIN frames can be received from all connected clients, when those frames passed the client acceptance filter.
- Each client has a receive queue, in which messages are waiting to be processed.
- A physical LIN bus corresponds to a hardware. If an adapter contains multiple LIN bus connectors (eg. PCAN-USB pro with 2x-LIN and 2x-CAN), for each LIN-Bus will be a hardware (hardware handle) available.
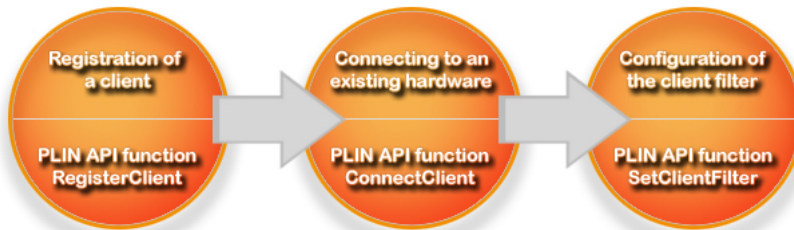
See Also

Getting Started (⊠ see page 5)

PLIN-Client and API Basics (⧉ see page 3)

2.2 **PLIN-Client and API Basics**

Context between the software, the PLIN client and the LIN hardware. The following scheme shows the basic procedures for a application to connect to a PLIN system:

To explain the interaction with the PLIN system and its behavior, the following rules are to be observed:

- If several clients are connected to a hardware, the hardware can be as whole only a LIN-master or LIN-slave.

- The clients or their applications above, have free access to all settings of the hardware. The applications are responsible for the regulation of access the hardware and its settings.

- It is not needed to have a registered client to query the hardware settings, since requests do not affect the hardware.

- A internal message forwarding process (PC level), directly from a client to another, does not exists. The messages sent, eg. using LIN_Write (⧉ see page 118) (), will always go to the hardware. The messages read using LIN_Read (⧉ see page 117)(), will always come from the hardware.

See Also

PLIN Basics (⧉ see page 2)

Getting Started (⧉ see page 5)

2.3 **License Regulations**

Namings for products in this manual, that are registered trademarks, are not separately marked. Therefore the missing of the ® sign does not implicate, that the naming is a free trade name. Furthermore the used names do not indicate patent rights or anything similar. PEAK-System Technik GmbH makes no representation or warranties with respect to the use of enclosed software or the contents of this manual, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, PEAK-System Technik GmbH reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

See Also

2.4 Contact Information

This software is a product of:

PEAK-System Technik GmbH

Otto-Röhm-Str. 69

64293 Darmstadt, Germany

Info:	info@peak-system.com
Support:	support@peak-system.com
Web:	www.peak-system.com

3 Getting Started

The following chapters are intended to provide a basic overview of the LIN environment within the PLIN API. It will be shown how a PLIN Client is to be created and configured, how to handle connections to a LIN hardware and its configuration, and other LIN concepts.

In this Chapter

Topics	Description
The LIN Client (🔲 see page 5)	Shows information about the handling with a PLIN client.
Configuring the Hardware (🔲 see page 7)	Shows information about the handling an configuration of a LIN Hardware.
The Frame Entry (🔲 see page 10)	Shows information about the data used to configure a LIN Table.
The Schedule Slot (🔲 see page 12)	Shows information about the Schedule table used by a LIN Hardware in Master operation.
The Receive Message (🔲 see page 14)	Shows information about the data contained in a received LIN frame.
The Handling of Schedule Tables (🔲 see page 20)	Shows information about how to work with Schedule tables within a Master.
The Keep-Alive Message (🔲 see page 23)	Shows information about the special frame Keep-Alive.
The Automatic Baud Rate Detection (🔲 see page 24)	Shows information about the process of baud rate detection.
The LIN-Bus Communication (🔲 see page 24)	Shows information about the LIN bus communication.
Processing Event Frames (🔲 see page 21)	Shows information about handling with Event Frames.

3.1 The LIN Client

A client in the PLIN system represents a user instance interacting with the PLIN environment. A client is the key for using the PLIN API because it is used to grant access to the API functionality. The clients registered in the system are identified using a unique handle.

Generally, only one PLIN client is used/registered within a windows application, but this is not a restriction. An application can registers more than one client when needed.

The process to get a valid PLIN client consists in: register the client, search for available hardware, connect the client to a hardware and configure it for communication. This last step depends on the way how a hardware is initialized.

In this Chapter

Topics	Description
Creating a Client (🔲 see page 6)	This section contains a tutorial for creating clients.

Selecting a Hardware (⊡ see page 6)	This section contains Information about locating and selecting a LIN hardware for connection.
Configuring a Client (⊡ see page 7)	This section contains Information about configuring basic parameters of a client.

3.1.1 Creating a Client

A PLIN client is an entity registered in the PLIN system which allows an user application to communicates on a LIN bus. The registration process is done using the function LIN_RegisterClient (⊡ see page 109) which allocates the necessary information needed for the interaction User-API-Hardware.

A client is identified twice on the PLIN system, one time through a name or label given by the user of the PLIN API at the moment of registration and, the most important, a unique handle which is returned by the function LIN_RegisterClient (⊡ see page 109) when a client is successfully registered. This handle will be used for almost all calls to the API and for this reason it should be saved by the client application as long as a PLIN client is needed. When a PLIN client is not needed anymore, the opposite function, LIN_RemoveClient (⊡ see page 110), should be called in order to liberate the resources occupied by that client. It is not necessary to disconnect a client (eg. using LIN_DisconnectClient (⊡ see page 111)) before calling the LIN_RemoveClient (⊡ see page 110) function. The disconnection is intrinsically done.

See Also

Selecting a Hardware (⊡ see page 6)

LIN_RegisterClient (⊡ see page 109) (.NET: RegisterCleint)

LIN_RemoveClient (⊡ see page 110) (.NET: RemoveClient (⊡ see page 54))

LIN_DisconnectClient (⊡ see page 111) (.NET: DisconnectClient (⊡ see page 55))

3.1.2 Selecting a Hardware

After having a client registered in the PLIN system, a hardware can be connected using the LIN_ConnectClient (⊡ see page 110) function, in order to participate in a communication. A client must use the API to get connected to a hardware present in the system.

The way to identify a hardware in the PLIN system is analog to the clients, using handles. The PLIN system assigns a handle, beginning with 1, to each LIN hardware registered in the system. A client application can ask for the available hardware, using the function LIN_GetAvailableHardware (⊡ see page 120) which returns an array of handles from all LIN hardware available for connection.

In many cases it is needed to physically identify a hardware before a client get connected, eg. when a computer has more than one LIN devices connected to different LIN busses. In this case, a client application can use the function LIN_IdentifyHardware (⊡ see page 127) to identify a hardware. Calling this function with a valid hardware handle as parameter causes the owner of that handle to blink its LED several times, leaving it recognizable.

The last step selecting a hardware is to connect the wanted hardware by using the function LIN_ConnectClient (⊡ see page 110). Completes the function successfully, means that the caller (PLIN client handle) was successfully associated to the hardware represented by the given hardware handle.

A client can also disconnect a hardware connected before, using the opposed function LIN_DisconnectClient (⊡ see page 111). This function receives two handles, one representing the client requesting a separation and the other representing the hardware to disconnect. In similar way as by connecting, a client application can use the function LIN_IdentifyHardware (⊡ see page 127) to identify the hardware to disconnect.

See Also

Configuring a Client (⊞ see page 7)

LIN_ConnectClient (⊞ see page 110) (.NET: ConnectClient (⊞ see page 55))

LIN_DisconnectClient (⊞ see page 111) (.NET: DisconnectClient (⊞ see page 55))

LIN_GetAvailableHardware (⊞ see page 120) (.NET: GetAvailableHardware (⊞ see page 67))

LIN_IdentifyHardware (⊞ see page 127) (.NET: IdentifyHardware (⊞ see page 81))

3.1.3 **Configuring a Client**

The settings process for a PLIN client consists in to adjust the different client parameters according with the needs of the client application (user), and configuring the Message Filter of the client which allows the reception of message frames from the LIN bus.

To adjust the client parameters is only needed that a client is registered in the PLIN system (a valid client handle is needed). There is currently only one parameter that can be set and it is "clpReceiveStatusFrames". This parameter allows a client to get status information about the LIN bus through message frames placed on its receive queue, called status frames. The parameter is by default active when a client is registered using the function LIN_RegisterClient (⊞ see page 109).

To adjust the Message Filter of a client it is needed that it is registered in the PLIN system and connected to at least one hardware. Since a client can connect multiple hardware, it has a Message Filter for each connected hardware. Setting a client filter will be done always as pair of handles. In this way, a client can have connected the hardware 2 and 3, by example, and only allows the ID 3 to pass through the hardware 2 and the ID 7 to pass through the hardware 3. There are two functions available in the PLIN API which configure message filters and they are LIN_SetClientFilter (⊞ see page 115) and LIN_RegisterFrameId (⊞ see page 127). The principal different is that the first one, LIN_SetClientFilter (⊞ see page 115), replace the existing filter for the new one, while the second function increases the filter.

See Also

Creating a Client (⊞ see page 6)

Selecting a Hardware (⊞ see page 6)

TLINClientParam (⊞ see page 36)

LIN_RegisterClient (⊞ see page 109) (.NET: RegisterClient (⊞ see page 53))

LIN_RegisterFrameId (⊞ see page 127) (.NET: RegisterFrameId (⊞ see page 82))

LIN_SetClientFilter (⊞ see page 115) (.NET: SetClientFilter (⊞ see page 62))

3.2 **Configuring the Hardware**

A LIN Hardware can be configured as Slave or Master. In order to configure a hardware it is necessary to have a client registered in the PLIN system, connected to the hardware to be initialized and configured to get messages from the LIN bus. Information about how to register and connect a PLIN client can be found in The LIN Client (⊞ see page 5) chapter.

The key for setting the working mode of a LIN hardware is the function LIN_InitializeHardware (⊞ see page 119). This function allows a client application to set the communication speed and the mode of the hardware. There are 2 possible modes, Slave and Master. Because the master can also uses a Schedule table, it is possible to talk about two kind of master states, Simple Master (Scheduler suspended) and Advanced Master (Scheduler running).

3

In this Chapter

Topics	Description
Programming a LIN Slave (⊠ see page 8)	This section contains Information about programming a LIN hardware as Slave.
Programming a LIN Master (⊠ see page 8)	This section contains Information about programming a LIN hardware as Simple-Master.
Programming a LIN Advanced-Master (⊠ see page 9)	This section contains Information about programming a LIN hardware as Advanced-Master.

3.2.1 Programming a LIN Slave

A client can configure a Hardware as Slave using the mode "modSlave" and a baud rate within the function LIN_InitializeHardware (⊠ see page 119). The baud rate is an integer value. The conversion into baud rate divider is hidden at this level. Note that if the hardware was already initialized, it will be re-initialized, maybe corrupting current Frame transfers. The function LIN_GetStatus (⊠ see page 142) can be used in order to check if a hardware is initialized or not. The data transfer in a Slave is determined by setting its LIN frames.

When the LIN bus stays more than 4 seconds without data traffic (idle), the network goes in the sleep state. A client application can send a Wake-Up impulse to a hardware working as Slave using the function LIN_XmtWakeUp (⊠ see page 140). The hardware sends then a single data byte (F0h) to generate a suitable impulse at the configured baud rate.

Reading

The client can get messages received by the Slave. If a received message meets the configured filter of a client (the filter allows the message to pass through), the Slave stores the message in the receive queue of the client. From there the messages can be read, using the functions LIN_Read (⊠ see page 117) and LIN_ReadMulti (⊠ see page 117).

Writing

A client using a connection to a Hardware configured as Slave can not directly send messages to the LIN bus. A Slave must be configured using the function LIN_SetFrameEntry (⊠ see page 128). A client application can configure all data Frames for a hardware (Slave), from ID 0 to 63. The configuration in the hardware can be read again using the function LIN_GetFrameEntry (⊠ see page 129). More information about a Frame Entry can be found in The LIN Frame Entry (⊠ see page 10) chapter. The data sent from a Slave, as Publisher, will be updated using the function LIN_UpdateByteArray (⊠ see page 130).

See Also

The LIN Frame Entry (⊠ see page 10)

LIN_InitializeHardware (⊠ see page 119) (.NET: InitializeHardware (⊠ see page 66))

LIN_GetStatus (⊠ see page 142) (.NET: GetStatus (⊠ see page 97))

LIN_SetFrameEntry (⊠ see page 128) (.NET: SetFrameEntry (⊠ see page 83))

LIN_GetFrameEntry (⊠ see page 129) (.NET: GetFrameEntry (⊠ see page 84))

LIN_UpdateByteArray (⊠ see page 130) (.NET: UpdateByteArray (⊠ see page 85))

LIN_XmtWakeUp (⊠ see page 140) (.NET: XmtWakeUp (⊠ see page 96))

3.2.2 Programming a LIN Master

A client can configure a Hardware as Master using the mode "modMaster" and a baud rate within the function

LIN_InitializeHardware (see page 119). The baud rate is an integer value. The conversion into baud rate divider is hidden at this level. Note that if the hardware was already initialized, it will be re-initialized, maybe corrupting current Frame transfers. The function LIN_GetStatus (see page 142) can be used in order to check if a hardware is initialized or not. The Scheduler is initialized as suspended, after the initialization as Master of the hardware.

When the LIN bus stays more than 4 seconds without data traffic (idle), the network goes in the sleep state. Because the hardware is a Master itself, the client application can wakes up the bus by sending frames again using the LIN_Write (see page 118) function.

Reading

The client can get messages received by the Master. If a received message meets the configured filter of a client (the filter allows the message to pass through), the Master stores the message in the receive queue of the client. From there the messages can be read, using the functions LIN_Read (see page 117) and LIN_ReadMulti (see page 117).

Writing

A Simple Master must not be configured In order to send data frames. A client application connected to a Simple Master can send directly data frames into the LIN bus using the function LIN_Write (see page 118). The complete timing behavior on the bus will be determined by calling the LIN_Write (see page 118) function. The content of the message structure passed to the function determines the data flow.

Each call to LIN_Write (see page 118) generates a Header on the LIN-Bus. A Publisher Frame will generate a Response field too. The member "Length" pretends the length of the transmitted response. A Subscriber Frame receives the response from the Slave. The member "Length" pretends the estimated length of the response. An AutoLength Frame receives the response from the Slave. The member "Length" is not relevant but should range from 1 to 8.

LIN_Write (see page 118) sends a message exact as it is got. The FrameId will not be modified. Use the function LIN_GetPID (see page 146) in order to determine the parity bits of the ID. The Checksumtype 'cstCustom' is only valid for the LIN_Write (see page 118) function on Publisher Frames. Using this type of checksum, the checksum will be send passed by the frame member "Checksum". Otherwise the Checksum is computed by Hardware.

See Also

LIN_InitializeHardware (see page 119) (.NET: InitializeHardware (see page 66))

LIN_GetStatus (see page 142) (.NET: GetStatus (see page 97))

LIN_Write (see page 118) (.NET: Write (see page 65))

LIN_GetPID (see page 146) (.NET: GetPID (see page 102))

3.2.3 **Programming a LIN Advanced-Master**

A client can configure a Hardware as Master using the mode "modMaster" and a baud rate within the function LIN_InitializeHardware (see page 119). The baud rate is an integer value. The conversion into baud rate divider is hidden at this level. Note that if the hardware was already initialized, it will be re-initialized, maybe corrupting current Frame transfers. The function LIN_GetStatus (see page 142) can be used in order to check if a hardware is initialized or not. The Scheduler is initialized as suspended, after the initialization as Master of the hardware.

The principal difference between a Simple and an Advanced Master is that the Advanced Master can independently process a Schedule Table. This make possible for the Master to transport data in a precise time, not depending from a client application (calling the function LIN_Write (see page 118)), between all participants, because the Hardware determines the timing. The principal functions to work out the Schedule tables are:

- LIN_SetSchedule (see page 133): A client application can use this function to pass a Schedule Table to the hardware (Master). To learn more about how many tables and how many slots a hardware supports, please consult the manual of the hardware.

- LIN_StartSchedule (see page 138): This function is used to start processing a Schedule Table. This process start

always from the first entry of the table. While the schedule is being worked out, it is not possible to send frames using the function LIN_Write (⊡ see page 118).

- LIN_SuspendSchedule (⊡ see page 139): This function is used to stop processing a Schedule Table. While the processing of the schedule is suspended, it is possible for a client application to send frames using the function LIN_Write (⊡ see page 118).

- LIN_ResumeSchedule (⊡ see page 139): This function is used to resume the processing of a schedule which was suspended before. The process start at the point where it was stopped.

Reading

The client can get messages received by the Master. If a received message meets the configured filter of a client (the filter allows the message to pass through), the Master stores the message in the receive queue of the client. From there the messages can be read, using the functions LIN_Read (⊡ see page 117) and LIN_ReadMulti (⊡ see page 117).

Writing

Basically, the master can operates in two states:

▶Scheduler Suspended

A client application connected to the Master sends data frames using the function LIN_Write (⊡ see page 118). The calling of this function determines the timing behavior on the bus. The values and options of the message passed to the function control the data flow.

▶Scheduler Running

The timing behavior on the bus is provided by the current Schedule Table. The values and options of the Frame Entries Table control the data flow. The Frame Table determines how the Slave responses to an ID. While the Master is running is not possible to directly send messages to the LIN bus using the function LIN_Write (⊡ see page 118).

The Frame Table will be used to control the data flow, because the Schedule Table does not contain any information about frame lengths or directions. All configurations of a Frame Entry, with exception of the Single Shot flag, can be used to configure the frames. The configuration of the Frames is done using the function LIN_SetFrameEntry (⊡ see page 128) and LIN_GetFrameEntry (⊡ see page 129). The data sent from a Master, as Publisher, will be updated on the hardware using the function LIN_UpdateByteArray (⊡ see page 130).

See Also

LIN_InitializeHardware (⊡ see page 119) (.NET: InitializeHardware (⊡ see page 66))

LIN_GetStatus (⊡ see page 142) (.NET: GetStatus (⊡ see page 97))

LIN_Write (⊡ see page 118) (.NET: Write (⊡ see page 65))

LIN_SetSchedule (⊡ see page 133) (.NET: SetSchedule (⊡ see page 88))

LIN_StartSchedule (⊡ see page 138) (.NET: StartSchedule (⊡ see page 93))

LIN_SuspendSchedule (⊡ see page 139) (.NET: SuspendSchedule (⊡ see page 94))

LIN_ResumeSchedule (⊡ see page 139) (.NET: ResumeSchedule (⊡ see page 95))

LIN_SetFrameEntry (⊡ see page 128) (.NET: SetFrameEntry (⊡ see page 83))

LIN_UpdateByteArray (⊡ see page 130) (.NET: UpdateByteArray (⊡ see page 85))

3.3 **The LIN Frame Entry**

A LIN Frame Entry (represented by the structure TLINFrameEntry (⊡ see page 29)) is used to configure the LIN Table. This table contains 64 entries, from ID 0 to ID 63. When a hardware is configured as Slave (slave-mode), the entries in the LIN Table determine how the hardware behaves, when a LIN Master sends a header with a determined ID.

When the hardware is configured as Master (master-mode), occur analogically the same, but only in the case when the schedule of the hardware is active.

A LIN Frame Entry is defined by the following elements:

▶ *Frame ID*

Is used as index for the frame being configured (ID 0 to 63).

▶ *Length*

Determines the length of the Response-Field. The length can be a value between 1 and 8.

▶ *Direction*

Determines the direction of the data flow. The direction is defined as TLINDirection (▣ see page 42) and the possible values are:

- *dirDisabled*: The ID will be ignored. No send or receive is carried out.
- *dirPublisher*: The hardware sends a Response-Field.
- *dirSubscriber*: The hardware receives a Response-Field.
- *dirSubscriberAutolength*: The hardware receives a Response-Field without evaluating the length or timeout.

▶ *Checksum Type*

Configures the type of the checksum. The checksum type is defined as TLINChecksumType (▣ see page 43) and the possible values are:

- *cstCustom*: This value is not allowed within a Frame Entry.
- *cstClassic*: The checksum corresponds to the classic model.
- *cstEnhanced*: The checksum corresponds to the enhanced model.
- *cstAuto*: The checksum will be recognized automatically by the hardware.

▶ *Flags*

The flags allow the configuration of special options. The following flags are available:

- *FRAME_FLAG_RESPONSE_ENABLE*: Enables/disables in a frame with direction dirPublisher, the sending of the Response-Field.
- *FRAME_FLAG_SINGLE_SHOT (Slave only)*: Configures a frame with direction dirPublisher to send the Response-Field only when the data was actualized. A single shot response will be retransmitted until transmission was successful. The internal update flag will be cleared when the transmission was error-free. Otherwise pending responses could be lost.
- FRAME_FLAG_IGNORE_INIT_DATA: Suppress the Initial Data associated with the Frame Entry.

▶ *Initial Data*

The data is a buffer with a length of 1 to 8 bytes, containing the data associated with the Frame Entry.

The LIN-Frame Entry permits several configuration scenarios, although not all combinations are valid. The following table shows, regarding the Frame Direction, which other values are allowed:

Direction	dirDisabled	dirPublisher	dirSubscriber	dirSubscriberAutolength
Length	1-8	1-8	1-8	1-8
Checksum Type	-/-	cstClassic cstEnhanced	cstClassic cstEnhanced cstAuto	cstClassic cstEnhanced cstAuto

Flags	-/-	FRAME_FLAG_RESPONSE_ENABLE FRAME_FLAG_SINGLE_SHOT (Slave only)	-/-	-/-
Initial Data	-/-	Yes	-/-	-/-

See Also

TLINFrameEntry (⧉ see page 29)

TLINDirection (⧉ see page 42)

TLINChecksumType (⧉ see page 43)

LIN_SetFrameEntry (⧉ see page 128) (.NET: SetFrameEntry (⧉ see page 83))

LIN_GetFrameEntry (⧉ see page 129) (.NET: GetFrameEntry (⧉ see page 84))

Definitions (⧉ see page 150) (.NET: Constants (⧉ see page 106))

3.4 The LIN Schedule Slot

When the hardware is used as Master (master-mode), it offers the possibility to deal with a self-standing Schedule table. The table is generated as an array of single slots (represented by the structure TLINScheduleSlot (⧉ see page 30)).

The hardware works out the table from index X to X+1. The element 0 of the array contains the first slot of the table.

A Schedule-Slot contains the following elements:

▶ *Type*

Represents the type of a Slot. The Type is defined as TLINSlotType (⧉ see page 41) and can be one of the following values:

- *sltUnconditional*: An unconditional frame.
- *sltEvent*: An event frame.
- *sltSporadic*: A sporadic frame.
- *sltMasterRequest*: A diagnose master request frame.
- *sltSlaveResponse*: A diagnose slave response frame.

▶ *Delay*

This is the time spacing between between this slot and the next one, expressed in milliseconds.

▶ *Frame Id*

An array with Frame Id(s), without parity bits, associated with a slot.

▶ *Count Resolve*

For a slot of type sltEvent: Contains the number of Schedule table to resolve the collision.

For a slot of type sltSporadic: The frame Id count for an sporadic frame.

▶ *Handle*

Slot-Handle to be used for breakpoints.

The values of a Slot must be configured depending on its type:

Unconditional Slots

Element	Description
Delay	Defines the lapse of time to the next Slot.
Frame Id (item 0)	Contains the sending ID, without parity bits. Index 1 to 7 have no meaning.
Handle	Used for breakpoints.

Event Slots

Element	Description
Delay	Defines the lapse of time to the next Slot.
Frame Id (Item 0)	Contains the sending ID, without parity bits. Index 1 to 7 have no meaning. The given ID is the Collision Frame.
Count Resolve	Contains the number of Schedule table to resolve the collision.
Handle	Used for breakpoints.

Sporadic Slots

Element	Description
Delay	Defines the lapse of time to the next Slot.
Frame Id	Contains a list of IDs from this slot. Index 0 has the highest priority.
Count Resolve	The count of sporadic IDs from 1 to 8.
Handle	Used for breakpoints.

Diagnose Master Request slot

Element	Description
Delay	Defines the lapse of time to the next Slot.
Frame Id	The slot has implicit the ID 60. The Index 0 can be set to 60 to avoid unnecessary queries from the graphical output.
Count Resolve	It is here not relevant.
Handle	Used for breakpoints.

Diagnose Slave Response Slot

Element	Description
Delay	Defines the lapse of time to the next Slot.
Frame Id	The slot has implicit the ID 61. The Index 0 can be set to 61 to avoid unnecessary queries from the graphical output.
Count Resolve	It is here not relevant.
Handle	Used for breakpoints.

See Also

TLINScheduleSlot (⧉ see page 30)

TLINSlotType (⧉ see page 41)

LIN_SetSchedule (⧉ see page 133) (.NET: SetSchedule (⧉ see page 88))

3.5 The LIN Receive Message

The messages that a hardware sends and/or receives can be read from each connected client . The PLIN system forwards a received message to the receive queue of each connected client, if the filter of the client accepts the message. An user can read the messages from the receive queue of a client using the function LIN_Read () (.NET: Read ()) or using LIN_ReadMulti () (.NET: ReadMulti ()).

The interpretation of a received message is the same for both a Master and a Slave. In slave-mode, an external Master is responsible for sending the Frame-Header. In master-mode, the hardware itself is responsible for sending the Header.

A Receive Message contains the following elements:

▶ *Type*

Represents the type of a received Message. The Type is defined as TLINMsgType () and can be one of the following values:

- *mstStandard*: A message with data.
- *mstBusSleep*: A message meaning that the bus state has changed to Bus-Sleep (The bus is recesive for at least 4 seconds). **Only** the Timestamp and Hardware Handle within this message is valid.
- *mstBusWakeUp*: A message meaning that a Wake-Up Impulse was received while the hardware was in Sleep state (data byte F0h). **Only** the Timestamp and Hardware Handle within this message is valid.
- *mstAutobaudrateTimeOut*: A message meaning that a Baud-Rate Recognition process started before, got a timeout and was unsuccessful. **Only** the Timestamp and Hardware Handle within this message is valid.
- *mstAutobaudrateReply*: A message that is a response to a Baud-rate Recognition process started before. The Timestamp of this message is valid. The data bytes 0 (lsb) to 3 (msb) contain information about the duration of 8 data bits in microseconds (the Sync Byte is measured).
- *mstOverrun*: A message indicating that the receive buffer of the hardware has been read out too late. The Timestamp of this message is valid. The data bytes 0 (lsb) to 1 (msb) contain the current count of overrun errors.
- *mstQueueOverrun*: A message indicating that USB data packages are being lost or the sequence of those packages has changed. The Timestamp of this message is valid. The data bytes 0 (lsb) to 1 (msb) contain the current count of queue-overrun errors.

▶ *Frame Id*

Valid only for messages of type *mstStandard*. Represents the ID of the received frame with parity bits.

▶ *Length*

Represents the length of the data bytes contained within the received message. The length can be a value between 0 and 8.

▶ *Direction*

Valid only for messages of type *mstStandard*. Represents the direction of a received Message. The Direction of the message shows how the message must be interpreted.The Type is defined as TLINDirection () and can be one of the following values:

- *dirPublisher*: A received message is a publisher message. See Message as Publisher ().
- *dirSubscriber*: A received message is a subscriber message. See Message as Subscriber ().
- *dirSubscriberAutoLength*: A received message is a subscriber message with automatic length. See Message as Subscriber-AutoLength ().

▶ *Checksum Type*

Valid only for messages of type *mstStandard*. Gives information about the type of checksum expected within the message. The checksum type is defined as TLINChecksumType () and the possible values are:

- *cstCustom*: The checksum has a custom checksum (only Publisher Frames sent by LIN_Write (⊡ see page 118) in master-mode).

- *cstClassic*: The checksum corresponds to the classic model.

- *cstEnhanced*: The checksum corresponds to the enhanced model.

- *cstAuto*: The checksum will be recognized automatically by the hardware.

▶ *Data*

Represents the data associated with the message. The length of this data can vary between 0 and 8 bytes length. The interpretation of this data bytes depends on the type of the message:

- *mstStandard*: Data bytes (maximum 8 bytes) associated with the the message.

- *mstBusSleep*: Not used. Invalid data.

- *mstBusWakeUp*: Not used. Invalid data.

- *mstAutobaudrateTimeOut*: Not used. Invalid data.

- *mstAutobaudrateReply*: Only bytes from 0 to 3 are valid.

- *mstOverrun*: Only bytes from 0 to 1 are valid.

- *mstQueueOverrun*: Only bytes from 0 to 1 are valid.

▶ *Checksum*

Valid only for messages of type *mstStandard*. Represents the calculated checksum of the received message, which must be in concordance with the Checksum Type.

▶ *Error Flags*

Valid only for messages of type *mstStandard*. Offer detailed information about the actually bus behavior from where the message comes. The error flags are represented by the type TLINMsgErrors (⊡ see page 34).

▶ *Time Stamp*

Represents the time, in microseconds, of the falling edge of the LIN-Break.

▶ *Hardware Handle*

Indicates the handle of the Hardware from where the message comes.

See Also

Message as Publisher (⊡ see page 15)

Message as Subscriber (⊡ see page 17)

Message as Subscriber-AutoLength (⊡ see page 18)

TLINDirection (⊡ see page 42)

TLINMsgType (⊡ see page 40)

LIN_Read (⊡ see page 117) (.NET: Read (⊡ see page 63))

LIN_ReadMulti (⊡ see page 117) (.NET: ReadMulti (⊡ see page 64))

3.5.1 **Message as Publisher**

When a Publisher Frame is read means that the hardware itself is the sender of the Response field.

▶ *Frame Id*

Contains the frame ID with parity bits.

▶ *Length*

Indicates the length of the response that should be sent.

▶ *Checksum Type*

Gives information about the type of checksum sent within the message.

▶ *Data*

An array which contains the data bytes of the message.

▶ *Checksum*

Contains the checksum of the message.

▶ *Error Flags*

Contains a value of type TLINMsgErrors (⤢ see page 34). This value is a flag value and can be a combination of the following values:

Flag	Description
MSG_ERR_INCONSISTENT_SYNC	The Synch-Byte was received with a value different than 55h.
MSG_ERR_ID_PARITY_BIT0	The received ID has an error in the Parity-Bit 0.
MSG_ERR_ID_PARITY_BIT1	The received ID has an error in the Parity-Bit 1.
MSG_ERR_SLAVE_NOT_RESPONDING	A Slave did not responded. See **Peculiarities and special cases for further information.**
MSG_ERR_TIMEOUT	The hardware did not send the complete data bytes and/or the checksum.
MSG_ERR_CHECKSUM	The received checksum is wrong.
MSG_ERR_GND_SHORT	Bus short circuit to GND.
MSG_ERR_VBAT_SHORT	Bus short circuit to VBAT.
MSG_ERR_OTHER_RESPONSE	The data of the message came from another participant.

▶ *Time Stamp*

Contains the time, in microseconds, of the falling edge of the LIN-Break.

▶ *Hardware Handle*

Indicates the handle of the Hardware, originator of the message.

Peculiarities and special cases for further information:

- When the error flag, MSG_ERR_TIMEOUT was set, means that at least the Checksum was not sent. It is possible too the sending of that data bytes was incomplete. In this case, the field Checksum contains the count of the actually bytes sent.

- The field "*Checksum Type*" contains the kind of the checksum sent. This type can be cstClassic or cstEnhanced. A checksum error will be shown through the flag "MSG_ERR_CHECKSUM".

- Publisher Frames can conflict with other participants (eg. Event-Frame). Even the Response itself can be controlled/suppressed through Frame-Flags. The possible cases are:

  1. The hardware has no data to send. This can have two possible reasons, that Response is disabled (FRAME_FLAG_RESPONSE_ENABLE = 0) or that Response is enabled, single shot is configured and there is no new data ready to be send. The possible constellations are:

     - No other participants sends: The message read contains the flags MSG_ERR_SLAVE_NOT_RESPONDING and MSG_ERR_TIMEOUT.

     - Another participant sends its Frame (eg. Event-Frame): The message read contains the flag MSG_ERR_OTHER_RESPONSE because the data came from another participant.

     - Two other participants send their frames and collide (eg. Event-Frames): The message read contains the flags MSG_ERR_OTHER_RESPONSE and MSG_ERR_TIMEOUT. This because the data came from another

participants and because the collision, the participants canceled the sending process.

2. The hardware has data to send. This can have two possible reasons, that Response is enable and single shot is not configured, or that Response is enabled, single shot is configured and there is new data to be sent. The possible constellations are:

- The hardware sends it response with checksum: MSG_ERR_CHECKSUM could denote an checksum error.

- While sending, the hardware cancels (eg. collision by Event-Frame): The message contains the flag MSG_ERR_TIMEOUT. The sending process was aborted. The field Checksum contains the count of bytes sent.

See Also

LIN Receive Message (⊡ see page 14)

Message as Subscriber (⊡ see page 17)

Message as Subscriber-AutoLength (⊡ see page 18)

3.5.2 **Message as Subscriber**

When a Subscriber Frame is read means that the hardware itself was one of the receivers of the Response field.

▶*Frame Id*

Contains the frame ID with parity bits.

▶*Length*

Indicates the length of the response estimated by the hardware.

▶*Checksum Type*

Gives information about the type of checksum expected within the message

▶*Data*

An array which contains the data bytes of the message.

▶*Checksum*

Contains the checksum of the message.

▶*Error Flags*

Contains a value of type TLINMsgErrors (⊡ see page 34). This value is a flag value and can be a combination of the following values:

Flag	Description
MSG_ERR_INCONSISTENT_SYNC	The Synch-Byte was received with a value different than 55h.
MSG_ERR_ID_PARITY_BIT0	The received ID has an error in the Parity-Bit 0.
MSG_ERR_ID_PARITY_BIT1	The received ID has an error in the Parity-Bit 1.
MSG_ERR_SLAVE_NOT_RESPONDING	The counterpart (the Publisher) not sent any data.
MSG_ERR_TIMEOUT	The counterpart (the Publisher) not send the complete data bytes and/or the checksum.
MSG_ERR_CHECKSUM	The received checksum is wrong.
MSG_ERR_GND_SHORT	Bus short circuit to GND.
MSG_ERR_VBAT_SHORT	Bus short circuit to VBAT.

▶*Time Stamp*

Contains the time, in microseconds, of the falling edge of the LIN-Break.

▶ *Hardware Handle*

Indicates the handle of the Hardware, originator of the message.

Peculiarities and special cases for further information:

- When the error flag, MSG_ERR_TIMEOUT was set, means that at least the Checksum was not received. It is possible too that received data bytes are incomplete. In this case, the field Checksum contains the count of the actually received bytes.

- The field "*Checksum Type*" must be considered in relation to the configured value. In this context apply the following rules:

  - If the Type is configured as cstClassic, the message contains the type cstClassic. A Checksum-Error will be show through the flag MSG_ERR_CHECKSUM.

  - If the Type is configured as cstEnhanced, the message contains the type cstEnhanced. A Checksum-Error will be show through the flag MSG_ERR_CHECKSUM.

  - If the Type is configured as cstAuto, the message contains the type cstClassic if a classic sum was recognized. If the recognition of the sum results in ctsEnhanced, then the message contains the type cstEnhanced. If none of the both described types can be recognized, the field "*Checksum Type*" contains the type cstAuto and the flag MSG_ERR_CHECKSUM is set.

See Also

3.5.3 **Message as Subscriber-AutoLength**

When a Subscriber-AutoLength Frame is read means that the hardware itself was one of the receivers of the Response field. In contrast to the Subscriber and Publisher Frames, a timeout evaluation will not take place. The last received data byte is interpreted as a checksum.

▶ *Frame Id*

Contains the frame ID with parity bits.

▶ *Length*

Indicates the length of the response. See lower description of Peculiarities.

▶ *Checksum Type*

Gives information about the type of checksum expected within the message

▶ *Data*

An array which contains the data bytes of the message.

▶ *Checksum*

Contains the checksum of the message.

▶ *Error Flags*

Contains a value of type TLINMsgErrors (⊠ see page 34). This value is a flag value and can be a combination of the following values:

Flag	Description
MSG_ERR_INCONSISTENT_SYNC	The Synch-Byte was received with a value different than 55h.
MSG_ERR_ID_PARITY_BIT0	The received ID has an error in the Parity-Bit 0.
MSG_ERR_ID_PARITY_BIT1	The received ID has an error in the Parity-Bit 1.
MSG_ERR_SLAVE_NOT_RESPONDING	The counterpart (the Publisher) not sent any data.
MSG_ERR_CHECKSUM	The received checksum is wrong.
MSG_ERR_GND_SHORT	Bus short circuit to GND.
MSG_ERR_VBAT_SHORT	Bus short circuit to VBAT.

▶ *Time Stamp*

Contains the time, in microseconds, of the falling edge of the LIN-Break.

▶ *Hardware Handle*

Indicates the handle of the Hardware, originator of the message.

Peculiarities and special cases for further information:

- The flag MSG_ERR_TIMEOUT does not exists within this message, because the length is not specified but calculated. Thus,it is not possible to do a reliable timeout analysis.

- If the hardware receives more than a data byte, than the last received byte will be treated as checksum.

- A more detailed Response Resolution is shown in the following table:

MSG_ERR_SLAVE_NOT_RESPONDING	Length	Meaning
1	x	nothing received
0	0	1 byte of data received
0	1	1 byte of data received and Checksum received.
0	2	2 byte of data received and Checksum received.
0	3	3 byte of data received and Checksum received.
0	4	4 byte of data received and Checksum received.
0	5	5 byte of data received and Checksum received.
0	6	6 byte of data received and Checksum received.
0	7	7 byte of data received and Checksum received.
0	8	8 byte of data received and Checksum received.

- The field "*Checksum Type*" must be considered in relation to the configured value. In this context apply the following rules:

  - If the Type is configured as cstClassic, the message contains the type cstClassic. A Checksum-Error will be show through the flag MSG_ERR_CHECKSUM.

  - If the Type is configured as cstEnhanced, the message contains the type cstEnhanced. A Checksum-Error will be show through the flag MSG_ERR_CHECKSUM.

  - If the Type is configured as cstAuto, the message contains the type cstClassic if a classic sum was recognized. If the recognition of the sum results in ctsEnhanced, then the message contains the type cstEnhanced. If none of the both described types can be recognized, the field "*Checksum Type*" contains the type cstAuto and the flag MSG_ERR_CHECKSUM is set.

See Also

LIN Receive Message (🔲 see page 14)

Message as Publisher (🔲 see page 15)

Message as Subscriber (🔲 see page 17)

3.6 Handling of Schedule Tables by the Hardware

The LIN-Master can independently process a Schedule Table. During the operation of the schedule is not possible to send direct messages to the bus using the function LIN_Write (⧉ see page 118). The Schedule Table determines the time behavior of the Bus while the Scheduler is working. The Frame Entry Table (FrameId) determines how the IDs are constructed (the direction, length, etc).

There are 5 types of Slots in disposal to construct a Schedule Table, which is an array of single Slots. The hardware dealt with the table from index **X** to index **X+1**. The element 0 of the array contains the first Slot of the table. The Scheduler starts the process again dealing with the first Slot when the last Slot contained in the array was processed. The Slots contain the IDs without parity bits. The parity bits will be designated by the hardware at runtime. The smallest delay from one slot to another is 4 milliseconds. This corresponds to a frame of length 1 by 20 KBit/s. The biggest delay that can be configured is 65535 milliseconds.

Slot Types

  ▶**Unconditional Slots**

The hardware sends always a Header with the ID "FrameId [0]". It will take "Delay" milliseconds to the next entry.

  ▶**Event Slots**

The hardware sends always a Header with the ID "FrameId [0]". It will take "Delay" milliseconds to the next entry. The ID is the collision-ID of the event frame. In this slot are basically three types of Response possible:

- If no Slave sends, the client application receives the Frame without response.

- If a Slave sends, the client application receives the Frame with the response from the slave.

- If two or more Slaves send, a collision occurs. The client application receives a Frame with Timeout. The hardware recognizes the collision, and automatically branche to the Schedule Table indicated in the slot member "CountResolve". This Resolve-Schedule contains the Unconditional IDs for the resolution of the collision.

  ▶**Sporadic Slots**

The hardware sends a Header only when a Frame of the list contains new data. The update of the data is done using the function LIN_UpdateByteArray (⧉ see page 130). It will take "Delay" milliseconds to the next entry. A sporadic frame is similar to a single shot in slave mode. The response will be retransmitted until success. The internal update flag will be cleared when the transmission was error-free. Otherwise pending responses could be lost. In this slot are basically three types of Frames possible:

- If there is no ID with new data, then no Frame will be sent. The client application does not receive a message.

- If there is an ID with new data, those will be sent together with a Header, so that the client application gets a message.

- If more than one ID have new data, the ID with the smaller List Index will be sent. "FrameId [0]" has the highest priority and "FrameId[7]" the lowest. If the Scheduler passes through this Slot in the next iteration, it will be sent again until all requests are processed according to their priority.

  ▶**Diagnose Master Request Slot**

The hardware sends a Header only when the ID 60 contains new data. This behavior is similar to the Sporadic slot. It will take "Delay" milliseconds to the next entry. A Header will not be be sent if there is no new data. In this case a client application will not receives any message. The response will be retransmitted until success. The internal update flag will be cleared when the transmission was error-free. Otherwise pending responses could be lost.

  ▶**Diagnose Slave Response Slot**

The hardware sends always a Header with the ID 61. It will take "Delay" milliseconds to the next entry. If there is new diagnostic data within a Slave, then the client application receives a message with data. If no Slave is asked to send data, then the message received by the client application contains no response.

See Also

TLINScheduleSlot (⧉ see page 30)

LIN_UpdateByteArray (⧉ see page 130) (.NET: UpdateByteArray (⧉ see page 85))

3.7 Processing Event Frames

Event Frames increases the responsitivity of LIN Slaves without assigning too much of the bus bandwidth for the polling of multiple Slaves with seldom occurring events, like push buttons. All Slave nodes involved to the event frame have to be configured with the same frame settings for frame length and checksum type. The event frame reserves the first data byte for an identification of the transmitting node. This is typically the protected identifier of the associated unconditional frame.

Two slaves sharing ID 9 as event frame:

Slave-1:

Frame Id	Remap	Direction	Length	Checksum Type	Flags
12	→	dirPublisher	8	cstEnhanced	enable
11		dirDisabled	1	cstAuto	
10	→	dirPublisher	4	cstClassic	enable
9	→	dirPublisher	4	cstClassic	enable, single-shot
8	→	dirSubscriber	5	cstClassic	

Slave-2:

Frame Id	Remap	Direction	Length	Checksum Type	Flags
12	→	dirSubscriber	8	cstEnhanced	
11	→	dirPublisher	4	cstClassic	enable
10	→	dirDisabled	1	cstAuto	
9	→	dirPublisher	4	cstClassic	enable, single-shot
8	→	dirSubscriber	5	cstClassic	

A part of the configuration for both slave nodes is described in the table above. ID 9 is used for the event frame. A slave node shall only transmit a response if at least one of the signals carried out in the frame is updated. If none of the slave nodes respond to the header, the frame is silent and the header is ignored. If both slave nodes respond to the header, a collision will occur. This collision is detected by the master switching its schedule table to a resolving schedule. The resolving schedule contains the unconditional frames 10 (for slave 1) and 11 (for slave 2). Additional unconditional frames can be sent in the resolving schedule too.

Schedule processing example
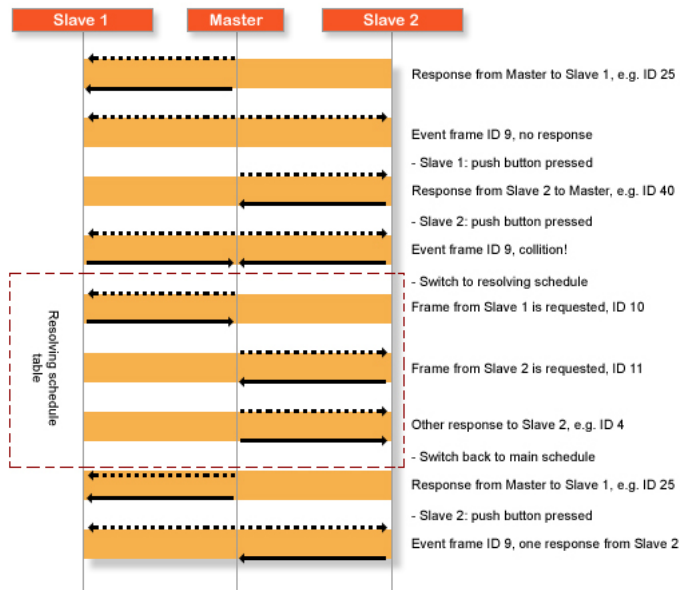
A schedule table contains different frames for response transfers like ID 25, ID 40, etc. and an event frame ID 9. The first time the event frame header is sent, no slave will respond to it. The slot is empty and will be ignored. After other frames the event frame will be triggered a second time. Since the last event frame, both slaves have new data pending for transmission, a collision will occur. The master then branches to the resolving schedule directly after the event slot responsible for the collision. Returning from the resolving schedule will resume the main schedule with the slot subsequent to the event frame where the collision occurred. Triggering the event frame a third time, only slave 2 will respond to it.

Setting up the example with PLIN-API

For this example is assumed that 2 application exist, one for each Slave. Both have all initialization and configuration needed and they have a button, which will be used to call the function LIN_UpdateByteArray (see page 130).

1. Set up FrameId 9 with LIN_SetFrameEntry (see page 128) using the values listed bellow. This step must be done for both slave, **Slave 1** and **Slave 2**:
   - Direction: dirPublisher
   - Length: 4
   - Type: cstClassic
   - Flags: FRAME_FLAG_RESPONSE_ENABLE **Or** FRAME_FLAG_SINGLE_SHOT

2. Set up FrameId 10 with LIN_SetFrameEntry (see page 128) using the values listed bellow and remap the publisher response of ID 10 to ID 9, with LIN_SetResponseRemap (see page 147). This step must be done for **Slave 1** only:
   - Direction: dirPublisher
   - Length: 4
   - Type: cstClassic
   - Flags: FRAME_FLAG_RESPONSE_ENABLE

3. Set up FrameId 11 with LIN_SetFrameEntry (see page 128) using the values listed bellow and remap the publisher response of ID 11 to ID 9, with LIN_SetResponseRemap (see page 147). This step must be done for **Slave 2** only:
   - Direction: dirPublisher
   - Length: 4
   - Type: cstClassic
   - Flags: FRAME_FLAG_RESPONSE_ENABLE

4. Update the ID 9 from both Slaves, using the function LIN_UpdateByteArray (see page 130), eg. using a button.

Both slaves will update ID 9 with LIN_UpdateByteArray (⊠ see page 130)! Data byte 0 differs between the slaves to identify the publisher if one single response is transmitted.

See Also

LIN_SetFrameEntry (⊠ see page 128) (.NET: SetFrameEntry (⊠ see page 83))

LIN_UpdateByteArray (⊠ see page 130) (.NET: UpdateByteArray (⊠ see page 85))

LIN_GetResponseRemap (⊠ see page 149) (.NET: GetResponseRemap (⊠ see page 104))

LIN_SetResponseRemap (⊠ see page 147) (.NET: SetResponseRemap (⊠ see page 103))

3.8 Using the Keep-Alive Message

The Keep-Alive message is part of the Master. It can be periodically sent by the hardware when the scheduler is suspended. This message serves to suppress the network-change in the sleep state, when the client application analyses carried out data with the function LIN_Write (⊠ see page 118), or when the client application does not trigger any action who demands traffic on the LIN bus. The Keep-Alive message is the only place where a hardware, in master mode, overlaps the states suspended and running.

If the Scheduler is suspended, the entire LIN communication takes place via the function LIN_Write (⊠ see page 118). The content of the messages define the data flow.

If the Scheduler is active, the Schedule Table provides the timing and the Frame Table controls the data flow.

The Keep-Alive message overlaps these two states. The hardware sends the message only in the suspended state. This would represent an automated LIN_Write (⊠ see page 118). The values for the control of the data flow, however, come from the Frame Table.

The Keep-Alive message is started by the function LIN_StartKeepAlive (⊠ see page 131). Since this function only receives as parameter the ID and the period of a message, the message takes it corresponding settings from the Frame Table according with the given ID.

If the scheduler is restarted, it discontinues sending Keep-Alive messages and LIN_Write (⊠ see page 118) messages from the send queue.

The function LIN_SuspendKeepAlive (⊠ see page 132) suspends the sending of the Keep-Alive message. The function LIN_ResumeKeepAlive (⊠ see page 132) continues sending with the configured values.

The Keep-Alive message can contain a time period which is smaller than the smallest time period a message can physically be sent. Because the hardware itself, in a suspended state, does not overtake, the period is dynamically adjusted. LIN messages from the transmit queue have also a higher priority than the Keep-Alive messages, and therefore Keep-Alive message can not bring the LIN bus to a standstill.

If the Keep-Alive message is a Pubilsher Frame, then the response can be changed using the function LIN_UpdateByteArray (⊠ see page 130).

See Also

LIN_StartKeepAlive (⊠ see page 131) (.NET: StartKeepAlive (⊠ see page 86))

LIN_ResumeKeepAlive (⊠ see page 132) (.NET: ResumeKeepAlive (⊠ see page 88))

LIN_UpdateByteArray (⊠ see page 130) (.NET: UpdateByteArray (⊠ see page 85))

LIN_Write (⊠ see page 118) (.NET: Write (⊠ see page 65))

3.9 **Automatic Baud Rate Detection**

The function LIN_StartAutoBaud (⊡ see page 141) can be used to start an automatic baud rate recognition process, when the hardware supports this feature. It is necessary to have at least one client connected to the hardware. This recognition process yields always an answer from the hardware.

In order to use this feature, the hardware must be in an not initialized state (the hardware was not initialized using the function LIN_InitializeHardware (⊡ see page 119) and the Hardware-Mode current configured is modNone). The algorithm for baud rate recognition needs a error-free Break and Sync field.

When the recognition process fails, the client who started the process will receive a message of type "mstAutobaudrateTimeOut". Note that this message is relevant only the timestamp.

When the recognition process was successful, all connected clients will receive a message of type "mstAutobaudrateReply". The timestamp of the message gives information about the moment when the baud rate was recognized. A 32 bit integer value (the data bytes from 0 to 3) contains information about the times measured on the Sync byte of a Header. This value contains the time for 8 data bits in microseconds.

See Also

LIN_InitializeHardware (⊡ see page 119) (.NET: InitializeHardware (⊡ see page 66))

LIN_StartAutoBaud (⊡ see page 141) (.NET: StartAutoBaud (⊡ see page 97))

3.10 **LIN-Bus Communication**

As shown within the The LIN Client (⊡ see page 5), it is possible, through simple ways and without extensive hardware configurations, as master to communicate with other slaves. For this, the client application use the function LIN_Write (⊡ see page 118). Every message sent with a call to LIN_Write (⊡ see page 118) generates a Header on the bus. In contrast to the scheduler or slave operation, the values of the message passed to the function determine the data flow. The Frame Table has no influence on the LIN_Write (⊡ see page 118) function.

There are the following relationships within the LIN Message given to the LIN_Write (⊡ see page 118) function:

- The Frame ID of the message will be sent with all eight bits. The function LIN_GetPID (⊡ see page 146) can be used to determinate the ID with the parity bits.
- Direction, Length and Checksum Type can be combined as show in the table of LIN Frame Entry (⊡ see page 10) excepting the Direction "Disabled".
- There is an extra Checksum Type for publishers frames available: "cstCustom". Using "cstClassic" or "cstEnhanced" causes the hardware to calculate the sum to send. Using "cstCustom" within a message indicates that the sum to be sent is the checksum contained in message given to the LIN_Write (⊡ see page 118) function.

See Also

The LIN Client (⊡ see page 5)

LIN Frame Entry (⊡ see page 10)

LIN_Write (⊡ see page 118) (.NET: Write (⊡ see page 65))

LIN_GetPID (⊡ see page 146) (.NET: GetPID (⊡ see page 102))

4 Reference

This section contains information about the data types (structures, types, defines, enumerations) and API functions and classes which are contained in the PLIN API.

In this Chapter

Topics	Description
Structures (☒ see page 25)	Lists the defined structures.
Types (☒ see page 32)	Lists the defined types.
Namespaces (☒ see page 49)	Lists the defined namespaces.
Functions (☒ see page 107)	List the defined functions.
Definitions (☒ see page 150)	Lists the defined values.

4.1 Structures

The PLIN API defines the following structures:

Name	Description
TLINVersion (☒ see page 25)	Define a version information.
TLINMsg (☒ see page 26)	Defines a LIN message.
TLINRcvMsg (☒ see page 27)	Defines a received LIN message.
TLINFrameEntry (☒ see page 29)	Defines a LIN frame entry.
TLINScheduleSlot (☒ see page 30)	Defines a LIN Schedule slot.
TLINHardwareStatus (☒ see page 31)	Defines a LIN Hardware status data.

4.1.1 TLINVersion

Defines a version information.

Syntax

Pascal

```
TLINVersion = record
    Major: Smallint;
    Minor: Smallint;
    Revision: Smallint;
    Build: Smallint;
end;
```

C++

```
typedef struct
{
```

```
        short Major;
        short Minor;
        short Revision;
        short Build;
}TLINVersion;
```

C#

```
[StructLayout(LayoutKind.Sequential)]
public struct TLINVersion
{
    public short Major;
    public short Minor;
    public short Revision;
    public short Build;
}
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential)> _
Public Structure TLINVersion
    Public Major As Short
    Public Minor As Short
    Public Revision As Short
    Public Build As Short
End Structure
```

Fields

Name	Description
Major	Major part of a version number.
Minor	Minor part of a version number.
Revision	Revision part of a version number.
Build	Build part of a version number.

4.1.2 TLINMsg

Defines a LIN Message to be sent.

Syntax

Pascal

```
TLINMsg = record
    FrameId: Byte;
    Length: Byte;
    Direction: TLINDirection;
    ChecksumType: TLINChecksumType;
    Data: array[0..7] of Byte;
    Checksum: Byte;
end;
```

C++

```
typedef struct
{
    BYTE FrameId;
    BYTE Length;
    TLINDirection Direction;
    TLINChecksumType ChecksumType;
    BYTE Data[8];
    BYTE Checksum;
}TLINMsg;
```

C#

```csharp
[StructLayout(LayoutKind.Sequential)]
public struct TLINMsg
{
    public byte FrameId;
    public byte Length;
    [MarshalAs(UnmanagedType.U1)]
    public TLINDirection Direction;
    [MarshalAs(UnmanagedType.U1)]
    public TLINChecksumType ChecksumType;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 8)]
    public byte[] Data;
    public byte Checksum;
}
```

Visual Basic

```vbnet
<StructLayout(LayoutKind.Sequential)> _
Public Structure TLINMsg
    Public FrameId As Byte
    Public Length As Byte
    <MarshalAs(UnmanagedType.U1)> _
    Public Direction As TLINDirection
    <MarshalAs(UnmanagedType.U1)> _
    Public ChecksumType As TLINChecksumType
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=8)> _
    Public Data As Byte()
    Public Checksum As Byte
End Structure
```

Fields

Name	Description
FrameId	Frame ID (6 bit) + Parity (2 bit).
Length	Frame length (1..8).
Direction	Frame Direction.
ChecksumType	Frame Checksum type.
Data	Data bytes (0..7).
Checksum	Frame Checksum.

See Also

TLINDirection (🗗 see page 42)

TLINChecksumType (🗗 see page 43)

4

4.1.3 **TLINRcvMsg**

Defines a received LIN message.

Syntax

Pascal

```pascal
TLINRcvMsg = record
    MsgType: TLINMsgType;
    FrameId: Byte;
    Length: Byte;
    Direction: TLINDirection;
    ChecksumType: TLINChecksumType;
    Data: array[0..7] of Byte;
```

```
    Checksum: Byte;
    ErrorFlags: TLINMsgErrors;
    TimeStamp: UInt64;
    hHw: HLINHW;
end;
```

C++

```cpp
typedef  struct
{
    TLINMsgType Type;
    BYTE FrameId;
    BYTE Length;
    TLINDirection Direction;
    TLINChecksumType ChecksumType;
    BYTE Data[8];
    BYTE Checksum;
    TLINMsgErrors ErrorFlags;
    unsigned __int64 TimeStamp;
    HLINHW hHw;
} TLINRcvMsg;
```

C#

```csharp
[StructLayout(LayoutKind.Sequential)]
public struct TLINRcvMsg
{
    [MarshalAs(UnmanagedType.U1)]
    public TLINMsgType Type;
    public byte FrameId;
    public byte Length;
    [MarshalAs(UnmanagedType.U1)]
    public TLINDirection Direction;
    [MarshalAs(UnmanagedType.U1)]
    public TLINChecksumType ChecksumType;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 8)]
    public byte[] Data;
    public byte Checksum;
    [MarshalAs(UnmanagedType.I4)]
    public TLINMsgErrors ErrorFlags;
    public UInt64 TimeStamp;
    public HLINHW hHw;
}
```

Visual Basic

```vb
<StructLayout(LayoutKind.Sequential)> _
Public Structure TLINRcvMsg
    <MarshalAs(UnmanagedType.U1)> _
    Public Type As TLINMsgType
    Public FrameId As Byte
    Public Length As Byte
    <MarshalAs(UnmanagedType.U1)> _
    Public Direction As TLINDirection
    <MarshalAs(UnmanagedType.U1)> _
    Public ChecksumType As TLINChecksumType
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=8)> _
    Public Data As Byte()
    Public Checksum As Byte
    <MarshalAs(UnmanagedType.I4)> _
    Public ErrorFlags As TLINMsgErrors
    Public TimeStamp As ULong
    Public hHw As ULong
End Structure
```

Fields

Name	Description
Type	Frame type.
FrameId	Frame ID (6 bit) + Parity (2 bit).
Length	Frame length (1..8).

Direction	Frame Direction.
ChecksumType	Frame Checksum type.
Data	Data bytes (0..7).
Checksum	Frame Checksum.
ErrorFlags	Frame error flags.
TimeStamp	Timestamp in microseconds.
hHw	Handle of the Hardware which received the message.

See Also

TLINMsgType (see page 40)

TLINDirection (see page 42)

TLINChecksumType (see page 43)

TLINMsgErrors (see page 34)

HLINHW (see page 34)

4.1.4 **TLINFrameEntry**

Defines a LIN frame entry.

Syntax

Pascal

```pascal
TLINFrameEntry = record
    FrameId: Byte;
    Length: Byte;
    Direction: TLINDirection;
    ChecksumType: TLINChecksumType;
    Flags: Word;
    InitialData: array[0..7] of Byte;
end;
```

C++

```cpp
typedef struct
{
    BYTE FrameId;
    BYTE Length;
    TLINDirection Direction;
    TLINChecksumType ChecksumType;
    WORD Flags;
    BYTE InitialData[8];
} TLINFrameEntry;
```

C#

```csharp
[StructLayout(LayoutKind.Sequential)]
public struct TLINFrameEntry
{
    public byte FrameId;
    public byte Length;
    [MarshalAs(UnmanagedType.U1)]
    public TLINDirection Direction;
    [MarshalAs(UnmanagedType.U1)]
    public TLINChecksumType ChecksumType;
    public ushort Flags;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 8)]
```

```
    public byte[] InitialData;
}
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential)> _
Public Structure TLINFrameEntry
    Public FrameId As Byte
    Public Length As Byte
    <MarshalAs(UnmanagedType.U1)> _
    Public Direction As TLINDirection
    <MarshalAs(UnmanagedType.U1)> _
    Public ChecksumType As TLINChecksumType
    Public Flags As UShort
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=8)> _
    Public InitialData As Byte()
End Structure
```

Fields

Name	Description
FrameId	Frame ID (without parity)
Length	Frame length (1..8).
Direction (⧉ see page 42)	Frame Direction.
ChecksumType (⧉ see page 43)	Frame Checksum type.
Flags (⧉ see page 10)	Frame flags.
InitialData	Data bytes (0..7).

See Also

TLINDirection (⧉ see page 42)

TLINChecksumType (⧉ see page 43)

LIN_SetFrameEntry (⧉ see page 128)

SetFrameEntry (⧉ see page 83)

4.1.5 **TLINScheduleSlot**

Defines a LIN Schedule slot.

Syntax

Pascal

```
TLINScheduleSlot = record
    &Type: TLINSlotType;
    Delay: Word;
    FrameId: array[0..7] of Byte;
    CountResolve: Byte;
    Handle: Longword;
end;
```

C++

```
typedef struct
{
    TLINSlotType Type;
    WORD Delay;
    BYTE FrameId[8];
    BYTE CountResolve;
    DWORD Handle;
```

```
} TLINScheduleSlot;
```

C#

```csharp
[StructLayout(LayoutKind.Sequential)]
public struct TLINScheduleSlot
{
    [MarshalAs(UnmanagedType.U1)]
    public TLINSlotType Type;
    public ushort Delay;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 8)]
    public byte[] FrameId;
    public byte CountResolve;
    public uint Handle;
}
```

Visual Basic

```vbnet
<StructLayout(LayoutKind.Sequential)> _
Public Structure TLINScheduleSlot
    <MarshalAs(UnmanagedType.U1)> _
    Public Type As TLINSlotType
    Public Delay As UShort
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=8)> _
    Public FrameId As Byte()
    Public CountResolve As Byte
    Public Handle As UInteger
End Structure
```

Fields

Name	Description
Type	Slot Type.
Delay	Slot Delay in Milliseconds.
FramId	Frame IDs (without parity).
CountResolve	ID count for sporadic frames. Resolve schedule number for Event frames
Handle	Slot handle. This value is read-only.

See Also

TLINSlotType (⊿ see page 41)

4.1.6 **TLINHardwareStatus**

Defines a LIN Hardware status data.

Syntax

Pascal

```pascal
TLINHardwareStatus = record
    Mode: TLINHardwareMode;
    Status: TLINHardwareState;
    FreeOnSendQueue: Byte;
    FreeOnSchedulePool: Word;
    ReceiveBufferOverrun: Word;
end;
```

C++

```cpp
typedef struct
{
    TLINHardwareMode Mode;
    TLINHardwareState Status;
    BYTE FreeOnSendQueue;
```

```
    WORD FreeOnSchedulePool;
    WORD ReceiveBufferOverrun;
} TLINHardwareStatus;
```

C#

```csharp
[StructLayout(LayoutKind.Sequential)]
public struct TLINHardwareStatus
{
    [MarshalAs(UnmanagedType.U1)]
    public TLINHardwareMode Mode;
    [MarshalAs(UnmanagedType.U1)]
    public TLINHardwareState Status;
    public byte FreeOnSendQueue;
    public ushort FreeOnSchedulePool;
    public ushort ReceiveBufferOverrun;
}
```

Visual Basic

```vbnet
<StructLayout(LayoutKind.Sequential)> _
Public Structure TLINHardwareStatus
    <MarshalAs(UnmanagedType.U1)> _
    Public Mode As TLINHardwareMode
    <MarshalAs(UnmanagedType.U1)> _
    Public Status As TLINHardwareState
    Public FreeOnSendQueue As Byte
    Public FreeOnSchedulePool As UShort
    Public ReceiveBufferOverrun As UShort
End Structure
```

Fields

Name	Description
Mode	Node state.
Status	Bus state.
FreeOnSendQueue	Count of free places in the Transmit Queue.
FreeOnSchedulePool	Free slots in the Schedule pool.
ReceiveBufferOverrun	USB receive buffer overrun counter.

See Also

TLINHardwareMode (see page 44)

TLINHardwareState (see page 44)

4.2 Types

The PLIN API defines the following types:

Name	Description
HLINCLIENT (see page 33)	Represents a LIN Client handle.
HLINHW (see page 34)	Represents a LIN Hardware handle.
TLINMsgErrors (see page 34)	Represents the Error flags for LIN received messages.
TLINClientParam (see page 36)	Represents the Client-Parameters used within the functions GetClientParam (see page 58) and SetClientParam (see page 57).
TLINHardwareParam (see page 37)	Represents the Hardware-Parameters used within the functions GetHardwareParam (see page 73) and SetHardwareParam (see page 68).
TLINMsgType (see page 40)	Represents the Type of a received LIN message.

TLINSlotType (⊡ see page 41)	Represents the Type of a LIN Schedule Slot.
TLINDirection (⊡ see page 42)	Represents the Direction-Type of a LIN message.
TLINChecksumType (⊡ see page 43)	Represents the Checksum-Type of LIN a message.
TLINHardwareMode (⊡ see page 44)	Represents the Operation Mode of a LIN Hardware.
TLINHardwareState (⊡ see page 44)	Represents the Status of a LIN Hardware.
TLINScheduleState (⊡ see page 45)	Represents the Status of the active LIN Schedule of a LIN Hardware.
TLINError (⊡ see page 46)	Represents the LIN error codes for the API functions.

4.2.1 HLINCLIENT

Represents a LIN Client handle. This handle is returned by the LIN function RegisterClient (⊡ see page 53).

Syntax

Pascal

```
HLINCLIENT = Byte;
```

C++

```
typedef BYTE HLINCLIENT;
```

C#

```
using HLINCLIENT = System.Byte;
```

Visual Basic

```
Imports HLINCLIENT = System.Byte
```

Remarks

.NET Framework programming languages:

An alias is used to represent a Client handle under Microsoft .NET in order to originate an homogeneity between all programming languages listed above.

Aliases are defined in the Peak.Lin (⊡ see page 50) Namespace for C# and VB .NET. However, including a namespace does not include the defined aliases.

If it is wished to work with aliases, those must be copied to the working file, right after the inclusion of the Peak.Lin (⊡ see page 50) Namespace. Otherwise, just use the native type, which in case of the HLINCLIENT is Byte.

C#

```
using System;
using Peak.Lin;
using HLINCLIENT = System.Byte;
using HLINHW = System.UInt16;
```

Visual Basic

```
Imports System
Imports Peak.Lin
Imports HLINCLIENT = System.Byte
Imports HLINHW = System.UInt16
```

See Also

RegisterClient (⊡ see page 53)

4.2.2 **HLINHW**

Represents a LIN Hardware handle. The available LIN hardware handles are returned by the LIN function GetAvailableHardware (see page 67).

Syntax

Pascal

```
HLINHW = Word;
```

C++

```
typedef WORD HLINHW;
```

C#

```
using HLINHW = System.UInt16;
```

Visual Basic

```
Imports HLINHW = System.UInt16
```

Remarks

.NET Framework programming languages:

An alias is used to represent a Hardware handle under Microsoft .NET in order to originate an homogeneity between all programming languages listed above.

Aliases are defined in the Peak.Lin (see page 50) Namespace for C# and VB.NET. However, including a namespace does not include the defined aliases.

If it is wished to work with aliases, those must be copied to the working file, right after the inclusion of the Peak.Lin (see page 50) Namespace. Otherwise, just use the native type, which in case of the HLINHW is UInt16.

C#

```
using System;
using Peak.Lin;
using HLINCLIENT = System.Byte;
using HLINHW = System.UInt16;
```

Visual Basic

```
Imports System
Imports Peak.Lin
Imports HLINCLIENT = System.Byte
Imports HLINHW = System.UInt16
```

See Also

GetAvailableHardware (see page 67)

4.2.3 **TLINMsgErrors**

Represents the Error flags for LIN received messages. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z4}
TLINMsgErrors = (
    InconsistentSynch = $01,
    IdParityBit0 = $02,
    IdParityBit1 = $04,
    SlaveNotResponding = $08,
    Timeout = $10,
    Checksum = $20,
    GroundShort = $40,
    VBatShort = $80,
    SlotDelay = $100,
    OtherResponse = $200
);
```

C++

```
#define TLINMsgErrors INT32

#define MSG_ERR_INCONSISTENT_SYNC 0x1
#define MSG_ERR_ID_PARITY_BIT0 0x2
#define MSG_ERR_ID_PARITY_BIT1 0x4
#define MSG_ERR_SLAVE_NOT_RESPONDING 0x8
#define MSG_ERR_TIMEOUT 0x10
#define MSG_ERR_CHECKSUM 0x20
#define MSG_ERR_GND_SHORT 0x40
#define MSG_ERR_VBAT_SHORT 0x80
#define MSG_ERR_SLOT_DELAY 0x100
#define MSG_ERR_OTHER_RESPONSE 0x200
```

C#

```
[Flags]
public enum TLINMsgErrors : int
{
    InconsistentSynch = 0x1,
    IdParityBit0 = 0x2,
    IdParityBit1 = 0x4,
    SlaveNotResponding = 0x8,
    Timeout = 0x10,
    Checksum = 0x20,
    GroundShort = 0x40,
    VBatShort = 0x80,
    SlotDelay = 0x100,
    OtherResponse = 0x200
}
```

Visual Basic

```
<Flags()> _
Public Enum TLINMsgErrors As Integer
    InconsistentSynch = &H1
    IdParityBit0 = &H2
    IdParityBit1 = &H4
    SlaveNotResponding = &H8
    Timeout = &H10
    Checksum = &H20
    GroundShort = &H40
    VBatShort = &H80
    SlotDelay = &H100
    OtherResponse = &H200
End Enum
```

Values

Name	Value	Description
InconsistentSynch MSG_ERR_INCONSISTENT_SYNC	1	Error on Synchronization field.

IdParityBit0 MSG_ERR_ID_PARITY_BIT0	2	Wrong parity Bit 0.
IdParityBit1 MSG_ERR_ID_PARITY_BIT1	4	Wrong parity Bit 1.
SlaveNotResponding MSG_ERR_SLAVE_NOT_RESPONDING	8	Slave not responding error.
Timeout MSG_ERR_TIMEOUT	16	A timeout was reached.
Checksum MSG_ERR_CHECKSUM	32	Wrong checksum.
GroundShort MSG_ERR_GND_SHORT	64	Bus shorted to ground.
VBatShort MSG_ERR_VBAT_SHORT	128	Bus shorted to VBat.
SlotDelay MSG_ERR_SLOT_DELAY	256	A slot time (delay) was too small.
OtherResponse MSG_ERR_OTHER_RESPONSE	512	Response was received from other station.

4.2.4 **TLINClientParam**

Represents the Client-Parameters used within the functions GetClientParam (☑ see page 58) and SetClientParam (☑ see page 57). According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z2}
TLINClientParam = (
    clpName = 1,
    clpMessagesOnQueue = 2,
    clpWindowHandle = 3,
    clpConnectedHardware = 4,
    clpTransmittedMessages = 5,
    clpReceivedMessages = 6,
    clpReceiveStatusFrames = 7,
    clpOnReceiveEventHandle = 8,
    clpOnPluginEventHandle = 9,
    clpLogStatus = 10,
    clpLogConfiguration = 11
);
```

C++

```
#define TLINClientParam WORD

#define clpName 1
#define clpMessagesOnQueue 2
#define clpWindowHandle 3
#define clpConnectedHardware 4
#define clpTransmittedMessages 5
#define clpReceivedMessages 6
#define clpReceiveStatusFrames 7
#define clpOnReceiveEventHandle 8
#define clpOnPluginEventHandle 9
```

```
#define clpLogStatus 10
#define clpLogConfiguration 11
```

C#

```csharp
public enum TLINClientParam : ushort
{
    clpName = 1,
    clpMessagesOnQueue = 2,
    clpWindowHandle = 3,
    clpConnectedHardware = 4,
    clpTransmittedMessages = 5,
    clpReceivedMessages = 6,
    clpReceiveStatusFrames = 7,
    clpOnReceiveEventHandle = 8,
    clpOnPluginEventHandle = 9,
    clpLogStatus = 10,
    clpLogConfiguration = 11,
}
```

Visual Basic

```vb
Public Enum TLINClientParam As UShort
    clpName = 1
    clpMessagesOnQueue = 2
    clpWindowHandle = 3
    clpConnectedHardware = 4
    clpTransmittedMessages = 5
    clpReceivedMessages = 6
    clpReceiveStatusFrames = 7
    clpOnReceiveEventHandle = 8
    clpOnPluginEventHandle = 9
    clpLogStatus = 10
    clpLogConfiguration = 11
End Enum
```

Values

Name	Value	Description
clpName	1	Client Name.
clpMessagesOnQueue	2	Unread messages in the Receive Queue.
clpWindowHandle	3	Registered windows handle (information purpose).
clpConnectedHardware	4	Handles of the connected Hardware.
clpTransmittedMessages	5	Number of transmitted messages.
clpReceivedMessages	6	Number of received messages.
clpReceiveStatusFrames	7	Status of the property "Status Frames".
clpOnReceiveEventHandle	8	Handle of the Receive event.
clpOnPluginEventHandle	9	Handle of the Hardware plug-in event.
clpLogStatus	10	Debug-Log activation status
clpLogConfiguration	11	Configuration of the debugged information (see Definitions (see page 150) for LOG_FLAG_***)

4.2.5 **TLINHardwareParam**

Represents the Hardware-Parameters used within the functions GetHardwareParam (see page 73) and SetHardwareParam (see page 68). According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z2}
TLINHardwareParam = (
    hwpName = 1,
    hwpDeviceNumber = 2,
    hwpChannelNumber = 3,
    hwpConnectedClients = 4,
    hwpMessageFilter = 5,
    hwpBaudrate = 6,
    hwpMode = 7,
    hwpFirmwareVersion = 8,
    hwpBufferOverrunCount = 9,
    hwpBossClient = 10,
    hwpSerialNumber = 11,
    hwpVersion = 12,
    hwpType = 13,
    hwpQueueOverrunCount = 14,
    hwpIdNumber = 15,
    hwpUserData = 16
    hwpBreakLength = 17,
    hwpLinTermination = 18,
    hwpFlashMode = 19,
    hwpScheduleActive 20,
    hwpScheduleState 21,
    hwpScheduleSuspendedSlot 22
);
```

C++

```
#define TLINHardwareParam WORD

#define hwpName 1
#define hwpDeviceNumber 2
#define hwpChannelNumber 3
#define hwpConnectedClients 4
#define hwpMessageFilter 5
#define hwpBaudrate 6
#define hwpMode 7
#define hwpFirmwareVersion 8
#define hwpBufferOverrunCount 9
#define hwpBossClient 10
#define hwpSerialNumber 11
#define hwpVersion 12
#define hwpType 13
#define hwpQueueOverrunCount 14
#define hwpIdNumber 15
#define hwpUserData 16
#define hwpBreakLength 17
#define hwpLinTermination 18
#define hwpFlashMode 19
#define hwpScheduleActive 20
#define hwpScheduleState 21
#define hwpScheduleSuspendedSlot 22
```

C#

```
public enum TLINHardwareParam : ushort
{
    hwpName = 1,
    hwpDeviceNumber = 2,
    hwpChannelNumber = 3,
    hwpConnectedClients = 4,
    hwpMessageFilter = 5,
    hwpBaudrate = 6,
    hwpMode = 7,
    hwpFirmwareVersion = 8,
    hwpBufferOverrunCount = 9,
    hwpBossClient = 10,
```

4

```
        hwpSerialNumber = 11,
        hwpVersion = 12,
        hwpType = 13,
        hwpQueueOverrunCount = 14,
        hwpIdNumber = 15,
        hwpUserData = 16,
        hwpBreakLength = 17,
        hwpLinTermination = 18,
        hwpFlashMode = 19,
        hwpScheduleActive = 20,
        hwpScheduleState = 21,
        hwpScheduleSuspendedSlot = 22,
}
```

Visual Basic

```
Public Enum TLINHardwareParam As UShort
        hwpName = 1
        hwpDeviceNumber = 2
        hwpChannelNumber = 3
        hwpConnectedClients = 4
        hwpMessageFilter = 5
        hwpBaudrate = 6
        hwpMode = 7
        hwpFirmwareVersion = 8
        hwpBufferOverrunCount = 9
        hwpBossClient = 10
        hwpSerialNumber = 11
        hwpVersion = 12
        hwpType = 13
        hwpQueueOverrunCount = 14
        hwpIdNumber = 15
        hwpUserData = 16
        hwpBreakLength = 17
        hwpLinTermination = 18
        hwpFlashMode = 19
        hwpScheduleActive = 20
        hwpScheduleState = 21
        hwpScheduleSuspendedSlot = 22
End Enum
```

Values

Name	Value	Description
hwpName	1	Hardware name.
hwpDeviceNumber	2	Index of the owner device.
hwpChannelNumber	3	Channel Index on the owner device (0 or 1).
hwpConnectedClients	4	Handles of the connected clients.
hwpMessageFilter	5	Message filter.
hwpBaudrate	6	Baud rate.
hwpMode	7	Master status.
hwpFirmwareVersion	8	Lin firmware version (Text with the form xx.yy where: xx = major version. yy = minor version).
hwpBufferOverrunCount	9	Receive buffer overrun Counter.
hwpBossClient	10	Registered master client.
hwpSerialNumber	11	Serial number of a hardware.
hwpVersion	12	Version of a hardware.
hwpType	13	Type of a hardware.
hwpQueueOverrunCount	14	Receive queue buffer overrun counter.
hwpIdNumber	15	Identification number for a hardware.

4

hwpUserData	16	User data on a hardware.
hwpBreakLength	17	Number of bits used as break field in a LIN frame.
hwpLinTermination	18	LIN Master termination status (independently of the hardware mode (⧉ see page 44)).
hwpFlashMode	19	Maintenance mode (flash mode) for firmware update.
hwpScheduleActive	20	Number of the schedule currently active.
hwpScheduleState	21	Operation state of a schedule (see schedule states (⧉ see page 45)).
hwpScheduleSuspendedSlot	22	Handle of the executing slot of a suspended schedule.

4.2.6 **TLINMsgType**

Represents the Type of a received LIN message. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z1}
TLINMsgType = (
    mstStandard = 0,
    mstBusSleep = 1,
    mstBusWakeUp = 2,
    mstAutobaudrateTimeOut = 3,
    mstAutobaudrateReply = 4,
    mstOverrun = 5,
    mstQueueOverrun = 6,
    mstClientQueueOverrun = 7
);
```

C++

```
#define TLINMsgType BYTE

#define mstStandard 0
#define mstBusSleep 1
#define mstBusWakeUp 2
#define mstAutobaudrateTimeOut 3
#define mstAutobaudrateReply 4
#define mstOverrun 5
#define mstQueueOverrun 6
#define mstClientQueueOverrun 7
```

C#

```
public enum TLINMsgType : byte
{
    mstStandard = 0,
    mstBusSleep = 1,
    mstBusWakeUp = 2,
    mstAutobaudrateTimeOut = 3,
    mstAutobaudrateReply = 4,
    mstOverrun = 5,
    mstQueueOverrun = 6,
    mstClientQueueOverrun = 7,
}
```

Visual Basic

```
Public Enum TLINMsgType As Byte
    mstStandard = 0
    mstBusSleep = 1
```

```
    mstBusWakeUp = 2
    mstAutobaudrateTimeOut = 3
    mstAutobaudrateReply = 4
    mstOverrun = 5
    mstQueueOverrun = 6
    mstClientQueueOverrun = 7
End Enum
```

Values

Name	Value	Description
mstStandard	0	Standard LIN Message.
mstBusSleep	1	Bus Sleep status message.
mstBusWakeUp	2	Bus WakeUp status message.
mstAutobaudrateTimeOut	3	Auto-baudrate Timeout status message.
mstAutobaudrateReply	4	Auto-baudrate Reply status message.
mstOverrun	5	Bus Overrun status message.
mstQueueOverrun	6	Queue overrun status message
mstClientQueueOverrun	7	Client's receive queue overrun status message

4.2.7 **TLINSlotType**

Represents the Type of a LIN Schedule Slot. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z1}
TLINSlotType = (
    sltUnconditional = 0,
    sltEvent = 1,
    sltSporadic = 2,
    sltMasterRequest = 3,
    sltSlaveResponse = 4
);
```

C++

```
#define TLINSlotType BYTE

#define sltUnconditional 0
#define sltEvent 1
#define sltSporadic 2
#define sltMasterRequest 3
#define sltSlaveResponse 4
```

C#

```
public enum TLINSlotType : byte
{
    sltUnconditional = 0,
    sltEvent = 1,
    sltSporadic = 2,
    sltMasterRequest = 3,
    sltSlaveResponse = 4,
}
```

4

Visual Basic

```
Public Enum TLINSlotType As Byte
    sltUnconditional = 0
    sltEvent = 1
    sltSporadic = 2
    sltMasterRequest = 3
    sltSlaveResponse = 4
End Enum
```

Values

Name	Value	Description
sltUnconditional	0	Unconditional frame.
sltEvent	1	Event frame.
sltSporadic	2	Sporadic frame.
sltMasterRequest	3	Diagnostic Master Request frame.
sltSlaveResponse	4	Diagnostic Slave Response frame.

4.2.8 **TLINDirection**

Represents the Direction-Type of a LIN message. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z1}
TLINDirection = (
    dirDisabled = 0,
    dirPublisher = 1,
    dirSubscriber = 2,
    dirSubscriberAutoLength = 3
);
```

C++

```
#define TLINDirection BYTE

#define dirDisabled 0
#define dirPublisher 1
#define dirSubscriber 2
#define dirSubscriberAutoLength 3
```

C#

```
public enum TLINDirection : byte
{
    dirDisabled = 0,
    dirPublisher = 1,
    dirSubscriber = 2,
    dirSubscriberAutoLength = 3,
}
```

Visual Basic

```
Public Enum TLINDirection As Byte
    dirDisabled = 0
    dirPublisher = 1
    dirSubscriber = 2
    dirSubscriberAutoLength = 3
```

```
End Enum
```

Values

Name	Value	Description
dirDisabled	0	Direction disabled.
dirPublisher	1	Direction is Publisher.
dirSubscriber	2	Direction is Subscriber.
dirSubscriberAutoLength	3	Direction is Subscriber (detect Length).

4.2.9 **TLINChecksumType**

Represents the Checksum-Type of LIN a message. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z1}
TLINChecksumType = (
    cstCustom = 0,
    cstClassic = 1,
    cstEnhanced = 2,
    cstAuto = 3
);
```

C++

```
#define TLINChecksumType BYTE

#define cstCustom 0
#define cstClassic 1
#define cstEnhanced 2
#define cstAuto 3
```

C#

```
public enum TLINChecksumType : byte
{
    cstCustom = 0,
    cstClassic = 1,
    cstEnhanced = 2,
    cstAuto = 3,
}
```

Visual Basic

```
Public Enum TLINChecksumType As Byte
    cstCustom = 0
    cstClassic = 1
    cstEnhanced = 2
    cstAuto = 3
End Enum
```

Values

Name	Value	Description
cstCustom	0	Custom checksum.

cstClassic	1	Classic checksum (ver 1.x).
cstEnhanced	2	Enhanced checksum.
cstAuto	3	Detect checksum.

4.2.10 **TLINHardwareMode**

Represents the Operation Mode of a LIN Hardware. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z1}
TLINHardwareMode = (
    modNone = 0,
    modSlave = 1,
    modMaster = 2
);
```

C++

```
#define TLINHardwareMode BYTE

#define modNone 0
#define modSlave 1
#define modMaster 2
```

C#

```
  public enum TLINHardwareMode : byte
{
    modNone = 0,
    modSlave = 1,
    modMaster = 2,
}
```

Visual Basic

```
Public Enum TLINHardwareMode As Byte
    modNone = 0
    modSlave = 1
    modMaster = 2
End Enum
```

Values

Name	Value	Description
modNone	0	Hardware is not initialized.
modSlave	1	Hardware working as Slave.
modMaster	2	Hardware working as Master.

4.2.11 **TLINHardwareState**

Represents the Status of a LIN Hardware. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z1}
TLINHardwareState = (
    hwsNotInitialized = 0,
    hwsAutobaudrate = 1,
    hwsActive = 2,
    hwsSleep = 3,
    hwsShortGround = 6,
    hwsVBatMissing = 7
);
```

C++

```
#define TLINHardwareState BYTE

#define hwsNotInitialized 0
#define hwsAutobaudrate 1
#define hwsActive 2
#define hwsSleep 3
#define hwsShortGround 6
#define hwsVBatMissing 7
```

C#

```
public enum TLINHardwareState : byte
{
    hwsNotInitialized = 0,
    hwsAutobaudrate = 1,
    hwsActive = 2,
    hwsSleep = 3,
    hwsShortGround = 6,
    hwsVBatMissing = 7,
}
```

Visual Basic

```
Public Enum TLINHardwareState As Byte
    hwsNotInitialized = 0
    hwsAutobaudrate = 1
    hwsActive = 2
    hwsSleep = 3
    hwsShortGround = 6
    hwsVBatMissing = 7
End Enum
```

Values

Name	Value	Description
hwsNotInitialized	0	Hardware is not initialized.
hwsAutobaudrate	1	Hardware is detecting the baud rate.
hwsActive	2	Hardware (bus) is active.
hwsSleep	3	Hardware (bus) is in sleep mode.
hwsShortGround	6	Hardware (bus-line) shorted to ground.
hwsVBatMissing	7	Hardware (USB adapter) external voltage supply missing.

4.2.12 TLINScheduleState

Represents the Status of the active LIN Schedule of a LIN Hardware. According with the programming language, this type

can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z1}
TLINScheduleState = (
    schNotRunning = 0,
    schSuspended = 1,
    schRunning = 2
);
```

C++

```
#define TLINScheduleState BYTE

#define schNotRunning 0
#define schSuspended 1
#define schRunning 2
```

C#

```
public enum TLINScheduleState : byte
{
    schNotRunning = 0,
    schSuspended = 1,
    schRunning = 2,
}
```

Visual Basic

```
Public Enum TLINScheduleState As Byte
    schNotRunning = 0
    schSuspended = 1
    schRunning = 2
End Enum
```

Values

Name	Value	Description
schNotRunning	0	No schedule Is running.
schSuspended	1	A schedule Is currently suspended.
schRunning	2	A schedule Is currently running.

4.2.13 **TLINError**

Represents the LIN error codes for the API functions. According with the programming language, this type can be a group of defined values or an enumeration.

Syntax

Pascal

```
{$Z4}
TLINError = (
    errOK = 0,
    errXmtQueueFull = 1,
    errIllegalPeriod = 2,
    errRcvQueueEmpty = 3,
    errIllegalChecksumType = 4,
    errIllegalHardware = 5,
```

```
    errIllegalClient = 6,
    errWrongParameterType = 7,
    errWrongParameterValue = 8,
    errIllegalDirection = 9,
    errIllegalLength = 10,
    errIllegalBaudrate = 11,
    errIllegalFrameID = 12,
    errBufferInsufficient = 13,
    errIllegalScheduleNo = 14,
    errIllegalSlotCount = 15,
    errIllegalIndex = 16,
    errIllegalRange = 17,
    errIllegalHardwareState  = 18,
    errIllegalSchedulerState  = 19,
    errIllegalFrameConfiguration =  20,
    errScheduleSlotPoolFull = 21,
    errIllegalSchedule = 22,
    errIllegalHardwareMode = 23,
    errOutOfResource = 1001,
    errManagerNotLoaded = 1002,
    errManagerNotResponding = 1003,
    errMemoryAccess = 1004,
    errNotImplemented = $FFFE,
    errUnknown = $FFFF
);
```

C++

```
#define TLINError DWORD

#define errOK 0
#define errXmtQueueFull 1
#define errIllegalPeriod 2
#define errRcvQueueEmpty 3
#define errIllegalChecksumType 4
#define errIllegalHardware 5
#define errIllegalClient 6
#define errWrongParameterType 7
#define errWrongParameterValue 8
#define errIllegalDirection 9
#define errIllegalLength 10
#define errIllegalBaudrate 11
#define errIllegalFrameID 12
#define errBufferInsufficient 13
#define errIllegalScheduleNo 14
#define errIllegalSlotCount 15
#define errIllegalIndex 16
#define errIllegalRange 17
#define errIllegalHardwareState  18
#define errIllegalSchedulerState  19
#define errIllegalFrameConfiguration 20
#define errScheduleSlotPoolFull 21
#define errIllegalSchedule 22
#define errIllegalHardwareMode 23
#define errOutOfResource 1001
#define errManagerNotLoaded 1002
#define errManagerNotResponding 1003
#define errMemoryAccess 1004
#define errNotImplemented 0xFFFE
#define errUnknown 0xFFFF
```

C#

```
public enum TLINError : int
{
    errOK = 0,
    errXmtQueueFull = 1,
    errIllegalPeriod = 2,
    errRcvQueueEmpty = 3,
    errIllegalChecksumType = 4,
    errIllegalHardware = 5,
    errIllegalClient = 6,
```

```
    errWrongParameterType = 7,
    errWrongParameterValue = 8,
    errIllegalDirection = 9,
    errIllegalLength = 10,
    errIllegalBaudrate = 11,
    errIllegalFrameID = 12,
    errBufferInsufficient = 13,
    errIllegalScheduleNo = 14,
    errIllegalSlotCount = 15,
    errIllegalIndex = 16,
    errIllegalRange = 17,
    errIllegalHardwareState  = 18,
    errIllegalSchedulerState  = 19,
    errIllegalFrameConfiguration =  20,
    errScheduleSlotPoolFull = 21,
    errIllegalSchedule = 22,
    errIllegalHardwareMode = 23,
    errOutOfResource = 1001,
    errManagerNotLoaded = 1002,
    errManagerNotResponding = 1003,
    errMemoryAccess = 1004,
    errNotImplemented = 0xFFFE,
    errUnknown = 0xFFFF,
}
```

Visual Basic

```
Public Enum TLINError As Integer
    errOK = 0
    errXmtQueueFull = 1
    errIllegalPeriod = 2
    errRcvQueueEmpty = 3
    errIllegalChecksumType = 4
    errIllegalHardware = 5
    errIllegalClient = 6
    errWrongParameterType = 7
    errWrongParameterValue = 8
    errIllegalDirection = 9
    errIllegalLength = 10
    errIllegalBaudrate = 11
    errIllegalFrameID = 12
    errBufferInsufficient = 13
    errIllegalScheduleNo = 14
    errIllegalSlotCount = 15
    errIllegalIndex = 16
    errIllegalRange = 17
    errIllegalHardwareState  = 18
    errIllegalSchedulerState  = 19
    errIllegalFrameConfiguration =  20
    errScheduleSlotPoolFull = 21
    errIllegalSchedule = 22
    errIllegalHardwareMode = 23
    errOutOfResource = 1001
    errManagerNotLoaded = 1002
    errManagerNotResponding = 1003
    errMemoryAccess = 1004
    errNotImplemented = &HFFFE
    errUnknown = &HFFFF
End Enum
```

Remarks

The error codes are, internally, sorted in ranges. Currently exist 3 regions of Error/Return Codes.

- **API Return Codes:** This range is represented by values from **0** to **1000**. This values are the actual return codes of the PLIN API functions, which signalize the success of a function call or the cause of it fail. Not used values in this range are reserved for future use.

- **DLL Interaction Codes:** This range is represented by values from **1001** to **2000:** This values are used to represent errors occurred within the DLL-Services interaction (*PLinApi.dll* and *PLinMng.exe*). Not used values in this range are reserved for future use.

- **Exception Codes:** This range is represented by values arranged from the maximum value of a WORD (**65535**) and down

to **65000**. This values are used to represent unexpected errors as an unfinished function or an unknown error. Not used values in this range are reserved for future use.

Values

Name	Value	Description
errOK	0	No error. Success.
errXmtQueueFull	1	Transmit Queue is full.
errIllegalPeriod	2	Period of time is invalid.
errRcvQueueEmpty	3	Client Receive Queue is empty.
errIllegalChecksumType	4	Checksum type is invalid.
errIllegalHardware	5	Hardware handle is invalid.
errIllegalClient	6	Client handle is invalid.
errWrongParameterType	7	Parameter type is invalid.
errWrongParameterValue	8	Parameter value is invalid.
errIllegalDirection	9	Direction is invalid.
errIllegalLength	10	Length is outside of the valid range.
errIllegalBaudrate	11	Baudrate is outside of the valid range.
errIllegalFrameID	12	ID is outside of the valid range.
errBufferInsufficient	13	Buffer parameter is too small.
errIllegalScheduleNo	14	Scheduler Number is outside of the valid range.
errIllegalSlotCount	15	Slots count is bigger than the actual number of available slots.
errIllegalIndex	16	Array index is outside of the valid range.
errIllegalRange	17	Range of bytes to be updated is invalid.
errIllegalHardwareState	18	The hardware is initialized and it should not, or is not initialized and it should.
errIllegalSchedulerState	19	Bad state of the scheduler.
errIllegalFrameConfiguration	20	Bad frame configuration.
errScheduleSlotPoolFull	21	The global pool for schedule slots is full.
errIllegalSchedule	22	There is no schedule present.
errIllegalHardwareMode	23	Operation not allowed by the current hardware mode.
errOutOfResource	1001	PLIN Manager does not have enough resources for the current task.
errManagerNotLoaded	1002	The PLIN Device Manager is not running.
errManagerNotResponding	1003	The communication to the LIN Device Manager was interrupted.
errMemoryAccess	1004	A "MemoryAccessViolation" exception occurred within an API method.
errNotImplemented	65534	An API method is not implemented.
errUnknown	65535	An internal error occurred within the LIN Device Manager.

4.3 Namespaces

PEAK-System offers a .NET Framework compatible programming environment. The following namespaces are available to be used:

Namespaces

	Name	Description
{}	Peak	Contains all namespaces that are part of the managed programming environment from PEAK-System.
{}	Peak.Can	Contains types and classes used to handle with CAN devices from PEAK-System.
{}	Peak.Lin (⊡ see page 50)	Contains types and classes used to handle with LIN devices from PEAK-System.
{}	Peak.RP1210A	Contains types and classes used to handle with CAN devices from PEAK-System through the TMC Recommended Practices 1210, version A, as known as RP1210(A).

4.3.1 Peak.Lin

The **Peak.Lin** namespace contains types and classes used to handle with LIN devices. This represent the managed use of the PLIN API from PEAK-System.

Aliases

	Alias	Description
●	HLINCLIENT (⊡ see page 33)	Represents a LIN Client handle.
●	HLINHW (⊡ see page 34)	Represents a LIN Hardware handle.

Classes

	Class	Description
⬧	PLinApi (⊡ see page 51)	Defines a class which represents the PLIN API.

Structures

	Structure	Description
◆	TLINVersion (⊡ see page 25)	Define a version information.
◆	TLINMsg (⊡ see page 26)	Defines a LIN message.
◆	TLINRcvMsg (⊡ see page 27)	Defines a received LIN message.
◆	TLINFrameEntry (⊡ see page 29)	Defines a LIN frame entry.
◆	TLINScheduleSlot (⊡ see page 30)	Defines a LIN Schedule slot.
◆	TLINHardwareStatus (⊡ see page 31)	Defines a LIN Hardware status data.

Enumerations

	Enumeration	Description
🗗	TLINMsgErrors (⊡ see page 34)	Represents the Error flags for LIN received messages.

🔩	TLINClientParam (⧉ see page 36)	Represents the Client-Parameters used within the functions GetClientParam (⧉ see page 58) and SetClientParam (⧉ see page 57).
🔩	TLINHardwareParam (⧉ see page 37)	Represents the Hardware-Parameters used within the functions GetHardwareParam (⧉ see page 73) and SetHardwareParam (⧉ see page 68).
🔩	TLINMsgType (⧉ see page 40)	Represents the Type of a received LIN message.
🔩	TLINSlotType (⧉ see page 41)	Represents the Type of a LIN Schedule Slot.
🔩	TLINDirection (⧉ see page 42)	Represents the Direction-Type of a LIN message.
🔩	TLINChecksumType (⧉ see page 43)	Represents the Checksum-Type of LIN a message.
🔩	TLINHardwareMode (⧉ see page 44)	Represents the Operation Mode of a LIN Hardware.
🔩	TLINHardwareState (⧉ see page 44)	Represents the Status of a LIN Hardware.
🔩	TLINError (⧉ see page 46)	Represents the LIN error codes for the API functions.

4.3.1.1 **PLinApi**

Define a class which represents the PLIN API.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```
public class PLinApi
```

Visual Basic

```
Public Class PLinApi
```

Remarks

The PLinApi class is a static class to work within the .NET Framework from Microsoft. It collects and implements the LIN API functions. each method is called just like the API function with the exception that the prefix "LIN_" is not used. The structure and functionality of the methods and API functions is the same.

For purposes of simplification, there are values defined as constants, which correspond to certain defines of the API.

See Also

Methods (⧉ see page 51)

Constants (⧉ see page 106)

4

4.3.1.1.1 **Methods**

The methods of the PLinApi (⧉ see page 51) class are divided in 5 groups of functionality:

Information

	Method	Description
🔹🅢	GetClientParam (⧉ see page 58)	Gets a Client parameter.
🔹🅢	GetClientFilter (⧉ see page 62)	Gets the filter corresponding to a given Client-Hardware pair.
🔹🅢	GetAvailableHardware (⧉ see page 67)	Gets an array containing the handles of the current Hardware available in the system.

⚡S	GetHardwareParam (☒ see page 73)	Gets a Hardware parameter.
◦S	IdentifyHardware (☒ see page 81)	Physically identifies a LIN Hardware.
◦S	GetFrameEntry (☒ see page 84)	Gets the configuration of a LIN Frame from a given Hardware.
◦S	GetSchedule (☒ see page 90)	Gets the slots of a Schedule from a Hardware.
◦S	GetStatus (☒ see page 97)	Retrieves current status information from a Hardware.
◦S	GetVersion (☒ see page 99)	Returns the PLIN-API DLL version.
◦S	GetVersionInfo (☒ see page 100)	Returns a string containing Copyright information.
◦S	GetErrorText (☒ see page 100)	Gets a description text for a LIN-Error code.
◦S	GetPID (☒ see page 102)	Calculates a Frame ID with Parity.
◦S	GetTargetTime (☒ see page 102)	Gets the time used by the LIN-USB adapter.
◦S	CalculateChecksum (☒ see page 98)	Calculates the checksum of a LIN Message.
◦S	GetResponseRemap (☒ see page 104)	Gets the publisher remap table from a given hardware.
◦S	GetSystemTime (☒ see page 105)	Gets the current system time.

Client

	Method	Description
◦S	RegisterClient (☒ see page 53)	Registers a Client at the LIN Manager.
◦S	RemoveClient (☒ see page 54)	Removes a Client from the Client list of the LIN Manager.
◦S	ConnectClient (☒ see page 55)	Connects a Client to a Hardware.
◦S	DisconnectClient (☒ see page 55)	Disconnects a Client from a Hardware.
◦S	ResetClient (☒ see page 56)	Flushes the Receive Queue of the Client and resets its counters.
◦S	SetClientParam (☒ see page 57)	Sets a Client parameter to a given value.
◦S	SetClientFilter (☒ see page 62)	Sets the filter of a Client.
◦S	RegisterFrameId (☒ see page 82)	Modifies the filter of a Client.

Configuration

	Method	Description
◦S	InitializeHardware (☒ see page 66)	Initializes a Hardware with a given Mode and Baud rate.
◦S	ResetHardware (☒ see page 80)	Flushes the queues of the Hardware and resets its counters.
◦S	ResetHardwareConfig (☒ see page 81)	Deletes the current configuration of the Hardware and sets its defaults.
⚡S	SetHardwareParam (☒ see page 68)	Sets a Hardware parameter to a given value.
◦S	SetFrameEntry (☒ see page 83)	Configures a LIN Frame in a given Hardware.
◦S	SetSchedule (☒ see page 88)	Configures the slots of a Schedule in a Hardware.
◦S	SetScheduleBreakPoint (☒ see page 92)	Sets a 'breakpoint' on a slot from a Schedule in a Hardware.
◦S	DeleteSchedule (☒ see page 91)	Removes all slots contained by a Schedule of a Hardware.
◦S	UpdateByteArray (☒ see page 85)	Updates the data of a LIN Frame for a given Hardware.
◦S	SetResponseRemap (☒ see page 103)	Sets the publisher response remap.

Control

	Method	Description
⇒◦§	StartKeepAlive (▣ see page 86)	Sets a Frame as Keep-Alive frame for a Hardware and starts to send.
⇒◦§	SuspendKeepAlive (▣ see page 87)	Suspends the sending of a Keep-Alive frame in a Hardware.
⇒◦§	ResumeKeepAlive (▣ see page 88)	Resumes the sending of a Keep-Alive frame in a Hardware.
⇒◦§	StartSchedule (▣ see page 93)	Activates a Schedule in a Hardware.
⇒◦§	SuspendSchedule (▣ see page 94)	Suspends an active Schedule in a Hardware.
⇒◦§	ResumeSchedule (▣ see page 95)	Restarts a configured Schedule in a Hardware.
⇒◦§	StartAutoBaud (▣ see page 97)	Starts a process to detect the Baud rate of the LIN bus on a Hardware.

Communication

	Method	Description
⇒◦§	Read (▣ see page 63)	Reads the next message/status information from a Client's Receive Queue.
⇒◦§	ReadMulti (▣ see page 64)	Reads several received messages.
⇒◦§	Write (▣ see page 65)	Transmits a message to a LIN Hardware.
⇒◦§	XmtWakeUp (▣ see page 96)	Sends a Wake-Up message impulse.

4.3.1.1.1.1 RegisterClient

Registers a Client at the PLIN Manager. Creates a Client handle and allocates the Receive Queue (only one per Client). The hWnd parameter can be zero for DOS Box Clients. The Client does not receive any messages until RegisterFrameId (▣ see page 82) or SetClientFilter (▣ see page 62) is called.

Namespace: Peak.Lin (▣ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_RegisterClient")]
public static extern TLINError RegisterClient(
    string strName,
    IntPtr hWnd,
    out HLINCLIENT hClient
)
```

Visual Basic

```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_RegisterClient")> _
Public Shared Function RegisterClient( _
    ByVal strName As String, _
    ByVal hWnd As Integer, _
    ByRef hClient As HLINCLIENT) As TLINError
End Function
```

Parameters

Parameters	Description
strName	Name of the Client. See LIN_MAX_NAME_LENGTH (▣ see page 106)
hWnd	Window handle of the Client (only for information purposes).
hClient	Client handle buffer.

Returns

The return value is an TLINError (▣ see page 46) code. *errOK* is returned on success, otherwise one of the following codes

are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errOutOfResource*

See Also

RemoveClient (⊡ see page 54)

ConnectClient (⊡ see page 55)

PLinApi (⊡ see page 51)

Other languages: LIN_RegisterClient (⊡ see page 109)

4.3.1.1.1.2 **RemoveClient**

Removes a Client from the Client list of the LIN Manager. Frees all resources (receive queues, message counters, etc.). If the Client was a Boss-Client for one or more Hardware, the Boss-Client property for those Hardware will be set to INVALID_LIN_HANDLE (⊡ see page 106).

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_RemoveClient")]
public static extern TLINError RemoveClient(
    HLINCLIENT hClient
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_RemoveClient")> _
Public Shared Function RemoveClient( _
    ByVal hClient As HLINCLIENT) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient*

See Also

RegisterClient (⊡ see page 53)

PLinApi (⊡ see page 51)

Other languages: LIN_RemoveClient (⊡ see page 110)

4.3.1.1.1.3 ConnectClient

Connects a Client to a Hardware. The Hardware is assigned by its Handle.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_ConnectClient")]
public static extern TLINError ConnectClient(
    HLINCLIENT hClient,
    HLINHW hHw
)
```

Visual Basic

```vbnet
<DllImport("plinapi.dll", EntryPoint:="LIN_ConnectClient")> _
Public Shared Function ConnectClient( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).
hHw	Handle of the Hardware to be connected. (One handle from the list of available hardware, returned by the function GetAvailableHardware (⊡ see page 67)).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

RegisterClient (⊡ see page 53)

DisconnectClient (⊡ see page 55)

GetAvailableHardware (⊡ see page 67)

PLinApi (⊡ see page 51)

Other languages: LIN_ConnectClient (⊡ see page 110)

4.3.1.1.1.4 DisconnectClient

Disconnects a Client from a Hardware. This means: no more messages will be received by this Client from this Hardware.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_DisconnectClient")]
```

```
public static extern TLINError DisconnectClient(
    HLINCLIENT hClient,
    HLINHW hHw
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_DisconnectClient")> _
Public Shared Function DisconnectClient( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).
hHw	Handle of the Hardware to be disconnected. (One handle from the list of connected hardware of the client above. See GetClientParam (⊡ see page 58)).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

ConnectClient (⊡ see page 55)

RemoveClient (⊡ see page 54)

GetClientParam (⊡ see page 58)

PLinApi (⊡ see page 51)

Other languages: LIN_DisconnectClient (⊡ see page 111)

4.3.1.1.1.5 ResetClient

Flushes the Receive Queue of the Client and resets its counters.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_ResetClient")]
public static extern TLINError ResetClient(
    HLINCLIENT hClient);
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_ResetClient")> _
Public Shared Function ResetClient( _
    ByVal hClient As HLINCLIENT) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊠ see page 53).

Returns

The return value is an TLINError (⊠ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient*

See Also

RegisterClient (⊠ see page 53)

PLinApi (⊠ see page 51)

Other languages: LIN_ResetClient (⊠ see page 112)

4.3.1.1.1.6 SetClientParam

Sets a Client parameter to a given value.

Namespace: Peak.Lin (⊠ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_SetClientParam")]
public static extern TLINError SetClientParam(
    HLINCLIENT hClient,
    TLINClientParam wParam,
    int dwValue
)
```

Visual Basic

```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_SetClientParam")> _
Public Shared Function SetClientParam( _
    ByVal hClient As HLINCLIENT, _
    ByVal wParam As TLINClientParam, _
    ByVal dwValue As Integer) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊠ see page 53).
wParam	The TLINClientParam (⊠ see page 36) parameter to be set. Allowed are: • clpReceiveStatusFrames • clpLogStatus • clpLogConfiguration
dwValue	Value to be set. 0 to deactivate it, otherwise to active it.

Returns

The return value is an TLINError (⊠ see page 46) code. *errOK* is returned on success, otherwise one of the following codes

are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient*

Remarks

The value of the parameter **hClient** is not important when configuring the Log functionality (*clpLogStatus*, *clpLogConfiguration*). Logging doesn't depend on a connected client since this is an API-wide operation. The value of **hClient** for this case should be set to **0** for clarity, though passing a valid handle also works. The handle is just ignored.

See Also

TLINClientParam (🗗 see page 36)

GetClientParam (🗗 see page 58)

Log File Generation (🗗 see page 152)

PLinApi (🗗 see page 51)

Other languages: LIN_SetClientParam (🗗 see page 112)

4.3.1.1.1.7 **GetClientParam**

Gets a Client parameter.

Overload

	Name	Description
🔹🅂	GetClientParam (HLINCLIENT, TLINClientParam, out int, ushort) (🗗 see page 58)	Gets a Client parameter whose type is a signed 32 bits value.
🔹🅂	GetClientParam (HLINCLIENT, TLINClientParam, string, ushort) (🗗 see page 59)	Gets a Client parameter whose type is a null terminated string.
🔹🅂	GetClientParam (HLINCLIENT, TLINClientParam, HLINHW[], ushort) (🗗 see page 60)	Gets a Client parameter whose type is an array of LIN Hardware handles.

See Also

PLinApi (🗗 see page 51)

Peak.Lin (🗗 see page 50)

Other languages: LIN_GetClientParam (🗗 see page 113)

4.3.1.1.1.7.1 **GetClientParam (HLINCLIENT, TLINClientParam, out int, ushort)**

Gets a Client parameter whose type is a signed 32 bits value.

Namespace: Peak.Lin (🗗 see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetClientParam")]
```

```
public static extern TLINError GetClientParam(
    HLINCLIENT hClient,
    TLINClientParam wParam,
    out int pBuff,
    ushort wBuffSize
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetClientParam")> _
Public Shared Function GetClientParam( _
    ByVal hClient As HLINCLIENT, _
    ByVal wParam As TLINClientParam, _
    ByVal pBuff As Integer, _
    ByVal wBuffSize As Integer) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊠ see page 53).
wParam	The TLINClientParam (⊠ see page 36) parameter to be retrieved. Allowed are: • clpMessagesOnQueue • clpWindowHandle • clpTransmittedMessages • clpReceivedMessages • clpReceiveStatusFrames • clpLogStatus • clpLogConfiguration
pBuff	Buffer for an integer value.
wBuffSize	Buffer size in bytes. It must be set to 0 or to the size of *pBuff*.

Returns

The return value is an TLINError (⊠ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient, errBufferInsufficient*

Remarks

The value of the parameter **hClient** is not important when retrieving Log information (*clpLogStatus*, *clpLogConfiguration*). Logging doesn't depend on a connected client since this is an API-wide operation. The value of **hClient** for this case should be set to **0** for clarity, though passing a valid handle also works. The handle is just ignored.

See Also

TLINClientParam (⊠ see page 36)

SetClientParam (⊠ see page 57)

PLinApi (⊠ see page 51)

Other languages: LIN_GetClientParam (⊠ see page 113)

4.3.1.1.1.7.2 GetClientParam (HLINCLIENT, TLINClientParam, string, ushort)

Gets a Client parameter whose type is a null terminated string.

Namespace: Peak.Lin (see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetClientParam")]
public static extern TLINError GetClientParam(
    HLINCLIENT hClient,
    TLINClientParam wParam,
    [MarshalAs(UnmanagedType.LPStr, SizeParamIndex = 3)]
    string pBuff,
    ushort wBuffSize
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetClientParam")> _
Public Shared Function GetClientParam( _
    ByVal hClient As HLINCLIENT, _
    ByVal wParam As TLINClientParam, _
    <MarshalAs(UnmanagedType.LPStr, SizeParamIndex:=3)> _
    ByVal pBuff As String, _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (see page 53).
wParam	The TLINClientParam (see page 36) parameter to be retrieved. Allowed are: • clpName
pBuff	Buffer for a string value.
wBuffSize	Size in bytes of the *pBuff* buffer. See LIN_MAX_NAME_LENGTH (see page 106)

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient, errBufferInsufficient*

See Also

TLINClientParam (see page 36)

SetClientParam (see page 57)

PLinApi (see page 51)

Other languages: LIN_GetClientParam (see page 113)

4.3.1.1.1.7.3 GetClientParam (HLINCLIENT, TLINClientParam, HLINHW[], ushort)

Gets a Client parameter whose type is an array of LIN Hardware handles.

Namespace: Peak.Lin (see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetClientParam")]
public static extern TLINError GetClientParam(
    HLINCLIENT hClient,
    TLINClientParam wParam,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 3)]
    HLINHW[] pBuff,
    ushort wBuffSize
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetClientParam")> _
Public Shared Function GetClientParam( _
    ByVal hClient As HLINCLIENT, _
    ByVal wParam As TLINClientParam, _
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=3)> _
    ByVal pBuff As HLINHW(), _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).
wParam	The TLINClientParam (⊡ see page 36) parameter to be retrieved. Allowed are: • clpConnectedHardware
pBuff	Buffer for an array of HLINHW (⊡ see page 34) handles.
wBuffSize	The size in bytes of the *pBuff* buffer. See **Remarks** for more information.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient, errBufferInsufficient*

Remarks

Asking the amount of hardware connected by a specific client return an array of HLINHW (⊡ see page 34). The array buffer must at least have a length of **"Total handles + 1" multiplied by the size of HLINHW (⊡ see page 34)**, where "Total handles" is the actual count of connected hardware.

The first position of the array (position 0) contains the amount of hardware (*n*) handles returned in the buffer. The connected hardware handles are returned beginning at the position 1 to *n*.

By example, in the case a client is connected to 3 Hardware, the *pBuff* buffer must have at least a size of 8 bytes (*(3+1)*sizeof(HLINHW (⊡ see page 34))*). So, the *pBuff*[0] contain the value 3 (3 handles returned) and the positions [1], [2] and [3] the returned handles.

See Also

TLINClientParam (⊡ see page 36)

SetClientParam (⊡ see page 57)

PLinApi (⊡ see page 51)

4.3.1.1.1.8 SetClientFilter

Sets the filter of a Client and modifies the filter of the connected Hardware.

Namespace: Peak.Lin (see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_SetClientFilter")]
public static extern TLINError SetClientFilter(
    HLINCLIENT hClient,
    HLINHW hHw,
    [MarshalAs(UnmanagedType.I8)]
    UInt64 iRcvMask
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_SetClientFilter")> _
Public Shared Function SetClientFilter( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    <MarshalAs(UnmanagedType.I8)> _
    ByVal iRcvMask As ULong) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (see page 58)).
iRcvMask	Filter. Each bit corresponds to a Frame ID (0..63).

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

Remarks

This method replaces any configured filter for the given client within the given hardware instead of expands it. To expand a Message Filter, use the method RegisterFrameId (see page 82).

See Also

GetClientFilter (see page 62)

GetClientParam (see page 58)

PLinApi (see page 51)

Other languages: LIN_SetClientFilter (see page 115)

4.3.1.1.1.9 GetClientFilter

Gets the filter corresponding to a given Client-Hardware pair.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_GetClientFilter")]
public static extern TLINError GetClientFilter(
    HLINCLIENT hClient,
    HLINHW hHw,
    [MarshalAs(UnmanagedType.I8)]
    out UInt64 pRcvMask
)
```

Visual Basic

```vbnet
<DllImport("plinapi.dll", EntryPoint:="LIN_GetClientFilter")> _
Public Shared Function GetClientFilter( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    <MarshalAs(UnmanagedType.I8)> _
    ByRef pRcvMask As ULong) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⊡ see page 58)).
pRcvMask	Filter buffer. Each bit corresponds to a Frame ID (0..63).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

SetClientFilter (⊡ see page 62)

GetClientParam (⊡ see page 58)

PLinApi (⊡ see page 51)

Other languages: LIN_GetClientFilter (⊡ see page 116)

4.3.1.1.1.10 Read

Reads the next message/status information from a Client's Receive Queue. The message will be written to the given buffer.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_Read")]
public static extern TLINError Read(
    HLINCLIENT hClient,
```

```
    out TLINRcvMsg pMsg
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_Read")> _
Public Shared Function Read( _
    ByVal hClient As HLINCLIENT, _
    ByRef pMsg As TLINRcvMsg) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⧉ see page 53).
pMsg	A TLINRcvMsg (⧉ see page 27) buffer.

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded, errManagerNotResponse, errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errRcvQueueEmpty*

See Also

ReadMulti (⧉ see page 64)

TLINRcvMsg (⧉ see page 27)

PLinApi (⧉ see page 51)

Other languages: LIN_Read (⧉ see page 117)

4.3.1.1.1.11 ReadMulti

Reads several received messages. This function is the equivalent to a multiple call of the Read (⧉ see page 63) method.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_ReadMulti")]
public static extern TLINError ReadMulti(
    HLINCLIENT hClient,
    [In, Out]
    TLINRcvMsg[] pMsgBuff,
    int iMaxCount,
    out int pCount
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_ReadMulti")> _
Public Shared Function ReadMulti( _
    ByVal hClient As HLINCLIENT, _
    <[In](), Out()> _
    ByVal pMsgBuff As TLINRcvMsg(), _
    ByVal iMaxCount As Integer, _
    ByRef pCount As Integer) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (🔲 see page 53).
pMsgBuff	Buffer for an array of TLINRcvMsg (🔲 see page 27) messages. The size of this buffer must be at least iMaxCount * sizeof(TLINRcvMsg (🔲 see page 27)) bytes.
iMaxCount	Desired amount of messages to be read.
pCount	The actual count of messages read.

Returns

The return value is an TLINError (🔲 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction: *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*

API Return: *errWrongParameterValue, errIllegalClient, errRcvQueueEmpty*

Remarks

The read buffer must be an array of 'iMaxCount' entries (must have at least a size of iMaxCount * sizeof(TLINRcvMsg (🔲 see page 27)) bytes). The size 'iMaxCount' of the array is the maximum amount of messages that can be received. The real number of read messages will be returned in 'pCount'.

ReadMulti is like a multiple call of the Read (🔲 see page 63) function. The return value of ReadMulti corresponds to the return value of the last call of Read (🔲 see page 63). This it means, if all messages were read from the queue but the count of read messages is less than '*iMaxCount*', the error errRcvQueueEmpty is returned. If the count of read messages is just the same amount of desired messages, the return value is errOK.

See Also

Read (🔲 see page 63)

TLINRcvMsg (🔲 see page 27)

PLinApi (🔲 see page 51)

Other languages: LIN_ReadMulti (🔲 see page 117)

4.3.1.1.1.12 **Write**

Transmits a message to a LIN Hardware.

Namespace: Peak.Lin (🔲 see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_Write")]
public static extern TLINError Write(
    HLINCLIENT hClient,
    HLINHW hHw,
    ref TLINMsg pMsg
)
```

Visual Basic

```vbnet
<DllImport("plinapi.dll", EntryPoint:="LIN_Write")> _
Public Shared Function Write( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
```

```
    ByRef pMsg As TLINMsg) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (🔲 see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (🔲 see page 58)).
pMsg	A write buffer.

Returns

The return value is an TLINError (🔲 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalDirection, errIllegalLength, errXmtQueueFull*

Remarks

The Client 'hClient' transmits a message 'pMsg' to the Hardware 'hHw'. The message is written into the Transmit Queue of the Hardware.

See Also

TLINMsg (🔲 see page 26)

GetClientParam (🔲 see page 58)

PLinApi (🔲 see page 51)

Other languages: LIN_Write (🔲 see page 118)

4.3.1.1.1.13 InitializeHardware

Initializes a Hardware with a given mode and baud rate.

Namespace: Peak.Lin (🔲 see page 50)

Syntax

C#
```
[DllImport("plinapi.dll", EntryPoint = "LIN_InitializeHardware")]
public static extern TLINError InitializeHardware(
    HLINCLIENT hClient,
    HLINHW hHw,
    [MarshalAs(UnmanagedType.U1)]
    TLINHardwareMode byMode,
    ushort wBaudrate
)
```

Visual Basic
```
<DllImport("plinapi.dll", EntryPoint:="LIN_InitializeHardware")> _
Public Shared Function InitializeHardware( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    <MarshalAs(UnmanagedType.U1)> _
    ByVal byMode As TLINHardwareMode, _
    ByVal wBaudrate As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⧉ see page 53).
hHw	Handle of the Hardware to initialize. (One handle from the list of available hardware, returned by the function GetAvailableHardware (⧉ see page 67)).
byMode	Hardware mode. See TLINHardwareMode (⧉ see page 44).
wBaudrate	The baud rate to configure the speed of the hardware. See LIN_MIN_BAUDRATE (⧉ see page 106) and LIN_MAX_BAUDRATE (⧉ see page 106).

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalBaudrate*

Remarks

If the Hardware was initialized by another Client, the function will re-initialize the Hardware. All connected clients will be affected. It is the job of the user to manage the setting and/or configuration of Hardware, e.g. by using the Boss-Client (⧉ see page 70) parameter of the Hardware.

See Also

GetAvailableHardware (⧉ see page 67)

TLINHardwareMode (⧉ see page 44)

Constants (⧉ see page 106)

PLinApi (⧉ see page 51)

Other languages: LIN_InitializeHardware (⧉ see page 119)

4.3.1.1.1.14 **GetAvailableHardware**

Gets an array containing the handles of the current Hardware available in the system.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetAvailableHardware")]
public static extern TLINError GetAvailableHardware(
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 1)]
    HLINHW[] pBuff,
    ushort wBuffSize,
    out ushort pCount
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetAvailableHardware")> _
Public Shared Function GetAvailableHardware( _
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=1)> _
    ByVal pBuff As HLINHW(), _
    ByVal wBuffSize As UShort, _
```

```
    ByRef pCount As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
pBuff	Buffer for an array of HLINHW (⊡ see page 34) handles.
wBuffSize	Size of the buffer in bytes (pBuff.Length * sizeof(HLINHW (⊡ see page 34)))
pCount	The count of available Hardware handles returned in the array 'pBuff'.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errBufferInsufficient*

Remarks

To only get the amount of hardware present in a system, call this function with an empty array ('pBuff' with length 0**)** and wBuffSize equal to **zero**. Than, the method returns errOK and the count of available hardware will be written in 'pCount'.

See Also

HLINHW (⊡ see page 34)

PLinApi (⊡ see page 51)

4.3.1.1.1.15 SetHardwareParam

Sets a Hardware parameter to a given value.

Overload

	Name	Description
⇥🇸	SetHardwareParam (HLINCLIENT, HLINHW, TLINHardwareParam, ref UInt64, ushort) (⊡ see page 69)	Sets a Hardware parameter whose type is an unsigned 64 bits value.
⇥🇸	SetHardwareParam (HLINCLIENT, HLINHW, TLINHardwareParam, ref HLINCLIENT, ushort) (⊡ see page 70)	Sets a Hardware parameter whose type is a LIN Client handle.
⇥🇸	SetHardwareParam (HLINCLIENT, HLINHW, TLINHardwareParam, ref int, ushort) (⊡ see page 71)	Sets a Hardware parameter whose type is a signed 32 bits value.
⇥🇸	SetHardwareParam (HLINCLIENT, HLINHW, TLINHardwareParam, byte[], ushort) (⊡ see page 72)	Sets a Hardware parameter whose type is an array of bytes.

See Also

PLinApi (⊡ see page 51)

Peak.Lin (⊡ see page 50)

Other languages: LIN_SetHardwareParam (◩ see page 121)

4.3.1.1.1.15.1 SetHardwareParam (HLINCLIENT, HLINHW, TLINHardwareParam, ref UInt64, ushort)

Sets a Hardware parameter whose type is an unsigned 64 bits value.

Namespace: Peak.Lin (◩ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_SetHardwareParam")]
public static extern TLINError SetHardwareParam(
    HLINCLIENT hClient,
    HLINHW hHw,
    TLINHardwareParam wParam,
    ref UInt64 pBuff,
    ushort wBuffSize
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_SetHardwareParam")> _
Public Shared Function SetHardwareParam( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal wParam As TLINHardwareParam, _
    ByRef pBuff As ULong, _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (◩ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (◩ see page 58)).
wParam	The TLINHardwareParam (◩ see page 37) parameter to be set. Allowed are: • hwpMessageFilter
pBuff	Buffer for a unsigned 64 bits integer value. Each bit corresponds to a Frame ID (0..63).
wBuffSize	Size in bytes of the 'pBuff' buffer. Should be at least 8 byte in length.

Returns

The return value is an TLINError (◩ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

GetHardwareParam (◩ see page 73)

TLINHardwareParam (◩ see page 37)

PLinApi (◩ see page 51)

4

Other languages: LIN_SetHardwareParam (⊡ see page 121)

4.3.1.1.1.15.2 **SetHardwareParam (HLINCLIENT, HLINHW, TLINHardwareParam, ref HLINCLIENT, ushort)**

Sets a Hardware parameter whose type is a LIN Client handle.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_SetHardwareParam")]
public static extern TLINError SetHardwareParam(
    HLINCLIENT hClient,
    HLINHW hHw,
    TLINHardwareParam wParam,
    ref HLINCLIENT pBuff,
    ushort wBuffSize
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_SetHardwareParam")> _
Public Shared Function SetHardwareParam( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal wParam As TLINHardwareParam, _
    ByRef pBuff As HLINCLIENT, _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⊡ see page 58)).
wParam	The TLINHardwareParam (⊡ see page 37) parameter to be set. Allowed are: • hwpBossClient
pBuff	Buffer for a HLINCLIENT (⊡ see page 33) handle.
wBuffSize	Size in bytes of the 'pBuff' buffer. Should have at least a length of sizeof(HLINCLIENT (⊡ see page 33)).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient, errIllegalHardware*

Remarks

The Boss-Client parameter is actually an user-controlled Access Permission. Setting this value **does not causes changes** in the operation of the LIN hardware or restricts other clients of doing a configuration, but gives the user the possibility to implement an "Access Protection" to avoid undesired configuration processes.

The principle is very simple: An application checks this parameter using the method GetHardwareParam (⊡ see page 73). The returned value is the handle of the Client who is registered as the Boss for that hardware in the PLIN system, i.e. this Client owns the permission to modify the hardware. If the return value is null (Nothing in VisualBasic), nobody has control

over the configuration of the hardware. By default, the Boss Client is configured to be the Client that initializes the hardware for the first time. To clear a configured Boss Client, a value of 0 within the buffer (pBuff) must be passed to the method SetHardwareParam (▣ see page 68).

It stays free to the user to decide how to react when checking this parameter.

Take in count that making changes in the configuration of the hardware with a client which is not the Boss-Client, will still work. The PLIN system **does not check** for Client permissions.

See Also

GetHardwareParam (▣ see page 73)

TLINHardwareParam (▣ see page 37)

PLinApi (▣ see page 51)

4.3.1.1.1.15.3 SetHardwareParam (HLINCLIENT, HLINHW, TLINHardwareParam, ref int, ushort)

Sets a Hardware parameter whose type is a signed 32 bits value.

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_SetHardwareParam")]
public static extern TLINError SetHardwareParam(
    HLINCLIENT hClient,
    HLINHW hHw,
    TLINHardwareParam wParam,
    ref int pBuff,
    ushort wBuffSize
)
```

Visual Basic

```vbnet
<DllImport("plinapi.dll", EntryPoint:="LIN_SetHardwareParam")> _
Public Shared Function SetHardwareParam( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal wParam As TLINHardwareParam, _
    ByRef pBuff As Integer, _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (▣ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (▣ see page 58)).
wParams	The TLINHardwareParam (▣ see page 37) parameter to be set. Allowed are: • hwpIdNumber
pBuff	Buffer for a signed 32 bits integer value.
wBuffSize	Size in bytes of the 'pBuff' buffer. Should be at least 4 byte in length.

Returns

The return value is an TLINError (▣ see page 46) code. *errOK* is returned on success, otherwise one of the following codes

are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

GetHardwareParam (see page 73)

TLINHardwareParam (see page 37)

PLinApi (see page 51)

Other languages: LIN_SetHardwareParam (see page 121)

4.3.1.1.1.15.4 SetHardwareParam (HLINCLIENT, HLINHW, TLINHardwareParam, byte[], ushort)

Sets a Hardware parameter whose type is an array of bytes.

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_SetHardwareParam")]
public static extern TLINError SetHardwareParam(
    HLINCLIENT hClient,
    HLINHW hHw,
    TLINHardwareParam wParam,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
    byte[] pBuff,
    ushort wBuffSize
)
```

Visual Basic

```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_SetHardwareParam")> _
Public Shared Function SetHardwareParam( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal wParam As TLINHardwareParam, _
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=4)> _
    ByVal pBuff As Byte(), _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (see page 58)).
wParam	The TLINHardwareParam (see page 37) parameter to be set. Allowed are: • hwpUserData
pBuff	Buffer for a byte array containing user data.
wBuffSize	Size in bytes of the *pBuff* buffer with a maximum length of LIN_MAX_USER_DATA (see page 106)

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

GetHardwareParam (see page 73)

TLINHardwareParam (see page 37)

PLinApi (see page 51)

Other languages: LIN_SetHardwareParam (see page 121)

4.3.1.1.1.16 GetHardwareParam

Gets a Hardware parameter.

Overload

	Name	Description
▪️S	GetHardwareParam (HLINHW, TLINHardwareParam, out int, ushort) (see page 73)	Gets a Hardware parameter whose type is a signed 32 bits value.
▪️S	GetHardwareParam (HLINHW, TLINHardwareParam, string, ushort) (see page 75)	Gets a Hardware parameter whose type is a null terminated string.
▪️S	GetHardwareParam (HLINHW, TLINHardwareParam, byte[], ushort) (see page 76)	Gets a Hardware parameter whose type is a an array of bytes, or an array of LIN Client handles.
▪️S	GetHardwareParam (HLINHW, TLINHardwareParam, TLINVersion, ushort) (see page 77)	Gets a Hardware parameter whose type is a TLINVersion (see page 25) structure.
▪️S	GetHardwareParam (HLINHW, TLINHardwareParam, out UInt64, ushort) (see page 78)	Gets a Hardware parameter whose type is an unsigned 64 bits value.
▪️S	GetHardwareParam (HLINHW, TLINHardwareParam, out HLINCLIENT, ushort) (see page 79)	Gets a Hardware parameter whose type is a LIN Client handle.

See Also

PLinApi (see page 51)

Peak.Lin (see page 50)

Other languages: LIN_GetHardwareParam (see page 123)

4.3.1.1.1.16.1 GetHardwareParam (HLINHW, TLINHardwareParam, out int, ushort)

Gets a Hardware parameter whose type is a signed 32 bits value.

Namespace: Peak.Lin (see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_GetHardwareParam")]
public static extern TLINError GetHardwareParam(
    HLINHW hHw,
    TLINHardwareParam wParam,
    out int pBuff,
    ushort wBuffSize
)
```

Visual Basic

```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_GetHardwareParam")> _
Public Shared Function GetHardwareParam( _
    ByVal hHw As HLINHW, _
    ByVal wParam As TLINHardwareParam, _
    ByRef pBuff As Integer, _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (☑ see page 67)).
wParam	The TLINHardwareParam (☑ see page 37) parameter to be retrieved. Allowed are: • hwpDeviceNumber • hwpChannelNumber • hwpBaudrate • hwpMode • hwpBufferOverrunCount • hwpSerialNumber • hwpVersion • hwpType • hwpQueueOverrunCount • hwpIdNumber
pBuff	Buffer for an integer value.
wBuffSize	Buffer size in bytes. It must be set to 0 or to the size of *pBuff*.

Returns

The return value is an TLINError (☑ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalHardware, errBufferInsufficient*

See Also

SetHardwareParam (☑ see page 68)

TLINHardwareParam (☑ see page 37)

PLinApi (☑ see page 51)

Other languages: LIN_GetHardwareParam (☑ see page 123)

4.3.1.1.1.16.2 GetHardwareParam (HLINHW, TLINHardwareParam, string, ushort)

Gets a Hardware parameter whose type is a null terminated string.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#
```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_GetHardwareParam")]
public static extern TLINError GetHardwareParam(
    HLINHW hHw,
    TLINHardwareParam wParam,
    [MarshalAs(UnmanagedType.LPStr, SizeParamIndex = 3)]
    string pBuff,
    ushort wBuffSize
)
```

Visual Basic
```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_GetHardwareParam")> _
Public Shared Function GetHardwareParam( _
    ByVal hHw As HLINHW, _
    ByVal wParam As TLINHardwareParam, _
    <MarshalAs(UnmanagedType.LPStr, SizeParamIndex:=3)> _
    ByVal pBuff As String, _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (⊡ see page 67)).
wParam	The TLINHardwareParam (⊡ see page 37) parameter to be retrieved. Allowed are: • hwpName
pBuff	Buffer for a string value.
wBuffSize	Size in bytes of the *pBuff* buffer. See LIN_MAX_NAME_LENGTH (⊡ see page 106)

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalHardware, errBufferInsufficient*

See Also

SetHardwareParam (⊡ see page 68)

TLINHardwareParam (⊡ see page 37)

Constants (⊡ see page 106)

PLinApi (⊡ see page 51)

Other languages: LIN_GetHardwareParam (⊡ see page 123)

4.3.1.1.1.16.3 **GetHardwareParam (HLINHW, TLINHardwareParam, byte[], ushort)**

Gets a Hardware parameter whose type is a an array of bytes.

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_GetHardwareParam")]
public static extern TLINError GetHardwareParam(
    HLINHW hHw,
    TLINHardwareParam wParam,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 3)]
    byte[] pBuff,
    ushort wBuffSize
)
```

Visual Basic

```vbnet
<DllImport("plinapi.dll", EntryPoint:="LIN_GetHardwareParam")> _
Public Shared Function GetHardwareParam( _
    ByVal hHw As HLINHW, _
    ByVal wParam As TLINHardwareParam, _
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=3)> _
    ByVal pBuff As Byte(), _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (▣ see page 67)).
wParam	The TLINHardwareParam (▣ see page 37) parameter to be retrieved. Allowed are: • hwpConnectedClients • hwpUserData
pBuff	Buffer for an array of bytes (or HLINCLIENT (▣ see page 33) handles).
wBuffSize	Size in bytes of the *pBuff* buffer. See **Remarks** for more information.

Returns

The return value is an TLINError (▣ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalHardware, errBufferInsufficient*

Remarks

• Asking the user data saved on a hardware returns the complete data which requires a buffer with a length of minimum LIN_MAX_USER_DATA (▣ see page 106).

• Asking the amount of clients connected to a specific hardware return an array of HLINCLIENT (▣ see page 33). The array buffer must at least have a length of **"Total handles + 1" multiplied by the size of HLINCLIENT (▣ see page 33)**, where "Total handles" is the actual count of connected clients. The first position of the array (position 0) contains the amount of client (*n*) handles returned in the buffer. The connected client handles are returned beginning at the position 1 to *n*. By example, in the case a hardware has 3 clients connected, the *pBuff* buffer must have at least a size of 4 bytes (*(3+1)*sizeof(HLINCLIENT (▣ see page 33))*). So, the *pBuff*[0] contain the value 3 (3 handles returned) and the positions [1], [2] and [3] the returned handles.

See Also

SetHardwareParam (⧉ see page 68)

TLINHardwareParam (⧉ see page 37)

PLinApi (⧉ see page 51)

Other languages: LIN_GetHardwareParam (⧉ see page 123)

4.3.1.1.1.16.4 GetHardwareParam (HLINHW, TLINHardwareParam, TLINVersion, ushort)

Gets a Hardware parameter whose type is a TLINVersion (⧉ see page 25) structure.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetHardwareParam")]
public static extern TLINError GetHardwareParam(
    HLINHW hHw,
    TLINHardwareParam wParam,
    [MarshalAs(UnmanagedType.LPStruct, SizeParamIndex = 3)]
    TLINVersion pBuff,
    ushort wBuffSize
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetHardwareParam")> _
Public Shared Function GetHardwareParam( _
    ByVal hHw As HLINHW, _
    ByVal wParam As TLINHardwareParam, _
    <MarshalAs(UnmanagedType.LPStruct, SizeParamIndex:=3)> _
    ByVal pBuff As TLINVersion, _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (⧉ see page 67)).
wParam	The TLINHardwareParam (⧉ see page 37) parameter to be retrieved. Allowed are: • hwpFirmwareVersion
pBuff	Buffer for a TLINVersion (⧉ see page 25) structure.
wBuffSize	Size in bytes of the 'pBuff' buffer. Should have at least a length of sizeof(TLINVersion (⧉ see page 25)).

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalHardware, errBufferInsufficient*

See Also

SetHardwareParam (⧉ see page 68)

TLINHardwareParam (see page 37)

PLinApi (see page 51)

Other languages: LIN_GetHardwareParam (see page 123)

4.3.1.1.1.16.5 GetHardwareParam (HLINHW, TLINHardwareParam, out UInt64, ushort)

Gets a Hardware parameter whose type is an unsigned 64 bits value.

Namespace: Peak.Lin (see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetHardwareParam")]
public static extern TLINError GetHardwareParam(
    HLINHW hHw,
    TLINHardwareParam wParam,
    out UInt64 pBuff,
    ushort wBuffSize
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetHardwareParam")> _
Public Shared Function GetHardwareParam( _
    ByVal hHw As HLINHW, _
    ByVal wParam As TLINHardwareParam, _
    ByRef pBuff As ULong, _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (see page 67)).
wParam	The TLINHardwareParam (see page 37) parameter to be retrieved. Allowed are: • hwpMessageFilter
pBuff	Buffer for a unsigned 64 bits integer value. Each bit corresponds to a Frame ID (0..63).
wBuffSize	Size in bytes of the 'pBuff' buffer. Should be at least 8 byte length.

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalHardware, errBufferInsufficient*

See Also

SetHardwareParam (see page 68)

TLINHardwareParam (see page 37)

PLinApi (see page 51)

Other languages: LIN_GetHardwareParam (⊠ see page 123)

4.3.1.1.1.16.6 GetHardwareParam (HLINHW, TLINHardwareParam, out HLINCLIENT, ushort)

Gets a Hardware parameter whose type is a LIN Client handle.

Namespace: Peak.Lin (⊠ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetHardwareParam")]
public static extern TLINError GetHardwareParam(
    HLINHW hHw,
    TLINHardwareParam wParam,
    out HLINCLIENT pBuff,
    ushort wBuffSize
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetHardwareParam")> _
Public Shared Function GetHardwareParam( _
    ByVal hHw As HLINHW, _
    ByVal wParam As TLINHardwareParam, _
    ByRef pBuff As HLINCLIENT, _
    ByVal wBuffSize As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (⊠ see page 67)).
wParam	The TLINHardwareParam (⊠ see page 37) parameter to be retrieved. Allowed are: • hwpBossClient
pBuff	Buffer for a HLINCLIENT (⊠ see page 33) handle.
wBuffSize	Size in bytes of the 'pBuff' buffer. Should have at least a length of sizeof(HLINCLIENT (⊠ see page 33)).

Returns

The return value is an TLINError (⊠ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalHardware, errBufferInsufficient*

Remarks

The Boss-Client parameter is actually an user-controlled Access Permission. Setting this value **does not causes changes** in the operation of the LIN hardware or restricts other clients of doing a configuration, but gives the user the possibility to implement an "Access Protection" to avoid undesired configuration processes.

The principle is very simple: An application check this parameter. If the return value is null (Nothing in VisualBasic), nobody has control over the configuration of the hardware. If the return value is nonzero, the return value is the handle of the Client who is registered as the Boss in the PLIN system, what it means, this client owns the permission to modify the hardware.

It stays free to the user to decide how to react when checking this parameter.

Take in count that making changes in the configuration of the hardware with a client which is not the Boss-Client, will still work. The PLIN system **does not check** for Client permissions.

See Also

SetHardwareParam (🖻 see page 68)

TLINHardwareParam (🖻 see page 37)

PLinApi (🖻 see page 51)

Other languages: LIN_GetHardwareParam (🖻 see page 123)

4.3.1.1.1.17 ResetHardware

Flushes the queues of the Hardware and resets its counters.

Namespace: Peak.Lin (🖻 see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_ResetHardware")]
public static extern TLINError ResetHardware(
    HLINCLIENT hClient,
    HLINHW hHw
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_ResetHardware")> _
Public Shared Function ResetHardware( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (🖻 see page 53).
hHw	Handle of the Hardware to reset. (One handle from the list of connected hardware of the client above. See GetClientParam (🖻 see page 58)).

Returns

The return value is an TLINError (🖻 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

    DLL Interaction: *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*

    API Return: *errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

InitializeHardware (🖻 see page 66)

GetClientParam (🖻 see page 58)

PLinApi (🖻 see page 51)

Other languages: LIN_ResetHardware (🖻 see page 125)

4.3.1.1.1.18 ResetHardwareConfig

Deletes the current configuration of the Hardware and sets its defaults.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_ResetHardwareConfig")]
public static extern TLINError ResetHardwareConfig(
    HLINCLIENT hClient,
    HLINHW hHw
)
```

Visual Basic

```vbnet
<DllImport("plinapi.dll", EntryPoint:="LIN_ResetHardwareConfig")> _
Public Shared Function ResetHardwareConfig( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⧉ see page 53).
hHw	Handle of the Hardware to reset. (One handle from the list of connected hardware of the client above. See GetClientParam (⧉ see page 58)).

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction: *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*

API Return: *errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

ResetHardware (⧉ see page 80)

GetClientParam (⧉ see page 58)

PLinApi (⧉ see page 51)

Other languages: LIN_ResetHardwareConfig (⧉ see page 126)

4.3.1.1.1.19 IdentifyHardware

Phisically identifies a LIN Hardware (a channel on a LIN Device) by blinking its associated LED.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_IdentifyHardware")]
public static extern TLINError IdentifyHardware(
    HLINHW hHw
```

```
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_IdentifyHardware")> _
Public Shared Function IdentifyHardware( _
    ByVal hHw As HLINHW) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware to identify. (One handle from the list of available hardware, returned by the function GetAvailableHardware (⊡ see page 67)).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalHardware*

See Also

GetAvailableHardware (⊡ see page 67)

PLinApi (⊡ see page 51)

Other languages: LIN_IdentifyHardware (⊡ see page 127)

4.3.1.1.1.20 RegisterFrameId

Modifies the filter of a Client and, eventually, the filter of the connected Hardware.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_RegisterFrameId")]
public static extern TLINError RegisterFrameId(
    HLINCLIENT hClient,
    HLINHW hHw,
    byte bFromFrameId,
    byte bToFrameId
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_RegisterFrameId")> _
Public Shared Function RegisterFrameId( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal bFromFrameId As Byte, _
    ByVal bToFrameId As Byte) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).

hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⊡ see page 58)).
bFromFrameId	First ID of the frame range.
bToFrameId	Last ID of the frame range.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalFrameID*

Remarks

The messages with FrameID *'bFromFrameId'* to '*bToFrameId'* (including both limits) will be received. By example, calling this method with the values '*bFromFrameId'* = 1 and '*bToFrameId'* = 4, register the Frames 1,2,3 and 4.

This method expands the filter instead of replaces it. Continuing with the example above, If a client with an already configured filter (to receive the IDs 1 to 4), calls this method again with 'bFromFrameId' = 5 and 'bToFrameId' = 7, then its filter let messages from ID 1 to ID 7 to go through.

See Also

SetClientFilter (⊡ see page 62)

PLinApi (⊡ see page 51)

Other languages: LIN_RegisterFrameId (⊡ see page 127)

4.3.1.1.1.21 **SetFrameEntry**

Configures a LIN Frame in a given Hardware.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_SetFrameEntry")]
public static extern TLINError SetFrameEntry(
    HLINCLIENT hClient,
    HLINHW hHw,
    ref TLINFrameEntry pFrameEntry
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_SetFrameEntry")> _
Public Shared Function SetFrameEntry( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByRef pFrameEntry As TLINFrameEntry) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).

| hHw | Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⧉ see page 58)). |
| pFrameEntry | Buffer for a TLINFrameEntry (⧉ see page 29) structure. |

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

| **DLL Interaction:** | *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess* |
| **API Return:** | *errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalFrameID, errIllegalLength* |

See Also

TLINFrameEntry (⧉ see page 29)

GetClientFilter (⧉ see page 62)

PLinApi (⧉ see page 51)

4.3.1.1.1.22 GetFrameEntry

Gets the configuration of a LIN Frame from a given Hardware.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetFrameEntry")]
public static extern TLINError GetFrameEntry(
    HLINHW hHw,
    ref TLINFrameEntry pFrameEntry
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetFrameEntry")> _
Public Shared Function GetFrameEntry( _
    ByVal hHw As HLINHW, _
    ByRef pFrameEntry As TLINFrameEntry) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (⧉ see page 67)).
pFrameEntry	Buffer for a TLINFrameEntry (⧉ see page 29) structure. See **Remarks**.

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

| **DLL Interaction:** | *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess* |
| **API Return:** | *errWrongParameterValue, errIllegalHardware, errIllegalFrameID* |

Remarks

The member *'FrameId'* of the structure TLINFrameEntry (🔲 see page 29) given within the parameter *'pFrameEntry'* must be set to the ID of the frame, whose configuration should be returned.

See Also

TLINFrameEntry (🔲 see page 29)

SetFrameEntry (🔲 see page 83)

GetAvailableHardware (🔲 see page 67)

PLinApi (🔲 see page 51)

Other languages: LIN_GetFrameEntry (🔲 see page 129)

4.3.1.1.1.23 **UpdateByteArray**

Updates the data of a LIN Frame for a given Hardware.

Namespace: Peak.Lin (🔲 see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_UpdateByteArray")]
public static extern TLINError UpdateByteArray(
    HLINCLIENT hClient,
    HLINHW hHw,
    byte bFrameId,
    byte bIndex,
    byte bLen,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
    byte[] pData
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_UpdateByteArray")> _
Public Shared Function UpdateByteArray( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal bFrameId As Byte, _
    ByVal bIndex As Byte, _
    ByVal bLen As Byte, _
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=4)> _
    ByVal pData As Byte()) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (🔲 see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (🔲 see page 58)).
bFrameId	ID of the Frame to be updated.
bIndex	Index where the update data starts (0..7).
bLen	Count of Data bytes to be updated.
pData	Data buffer with a size of at least '*bLen*'.

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalFrameID, errIllegalLength, errIllegalIndex, errIllegalRange*

See Also

SetFrameEntry (⧉ see page 83)

GetFrameEntry (⧉ see page 84)

PLinApi (⧉ see page 51)

Other languages: LIN_UpdateByteArray (⧉ see page 130)

4.3.1.1.1.24 **StartKeepAlive**

Sets the Frame 'bFrameId' as Keep-Alive frame for the given Hardware and starts to send it periodically.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_StartKeepAlive")]
public static extern TLINError StartKeepAlive(
    HLINCLIENT hClient,
    HLINHW hHw,
    byte bFrameId,
    ushort wPeriod
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_StartKeepAlive")> _
Public Shared Function StartKeepAlive( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal bFrameId As Byte, _
    ByVal wPeriod As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⧉ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⧉ see page 58)).
bFrameId	ID of the Keep-Alive Frame.
wPeriod	Keep-Alive interval in milliseconds (at least 4 ms).

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalFrameID, errIllegalPeriod, errIllegalSchedulerState, errIllegalFrameConfiguration*

See Also

SuspendKeepAlive (⊞ see page 87)

ResumeKeepAlive (⊞ see page 88)

PLinApi (⊞ see page 51)

4.3.1.1.1.25 SuspendKeepAlive

Suspends the sending of a Keep-Alive frame in the given Hardware.

Namespace: Peak.Lin (⊞ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_SuspendKeepAlive")]
public static extern TLINError SuspendKeepAlive(
    HLINCLIENT hClient,
    HLINHW hHw
)
```

Visual Basic

```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_SuspendKeepAlive")> _
Public Shared Function SuspendKeepAlive( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊞ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⊞ see page 58)).

Returns

The return value is an TLINError (⊞ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

StartKeepAlive (⊞ see page 86)

ResumeKeepAlive (⊞ see page 88)

PLinApi (⊞ see page 51)

4.3.1.1.1.26 ResumeKeepAlive

Resumes the sending of a Keep-Alive frame in the given Hardware.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_ResumeKeepAlive")]
public static extern TLINError ResumeKeepAlive(
    HLINCLIENT hClient,
    HLINHW hHw
)
```

Visual Basic

```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_ResumeKeepAlive")> _
Public Shared Function ResumeKeepAlive( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⊡ see page 58)).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue,* *errIllegalClient,* *errIllegalHardware,* *errIllegalSchedulerState,* *errIllegalFrameConfiguration*

See Also

StartKeepAlive (⊡ see page 86)

SuspendKeepAlive (⊡ see page 87)

PLinApi (⊡ see page 51)

4.3.1.1.1.27 SetSchedule

Configures the slots of a Schedule in a given Hardware.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

4

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_SetSchedule")]
public static extern TLINError SetSchedule(
    HLINCLIENT hClient,
    HLINHW hHw,
    int iScheduleNumber,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
    [In, Out] TLINScheduleSlot[] pSchedule,
    int iSlotCount
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_SetSchedule")> _
Public Shared Function SetSchedule( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal iScheduleNumber As Integer, _
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=4)> _
    <[In](), Out()> _
    ByVal pSchedule As TLINScheduleSlot(), _
    ByVal iSlotCount As Integer) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊞ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⊞ see page 58)).
iScheduleNumber	Schedule number. See LIN_MIN_SCHEDULE_NUMBER (⊞ see page 106) and LIN_MAX_SCHEDULE_NUMBER (⊞ see page 106)
pSchedule	Buffer for an array of TLINScheduleSlot (⊞ see page 30) structures. See **Remarks**.
iSlotCount	Count of Slots contained in the '*pSchedule*' buffer.

Returns

The return value is an TLINError (⊞ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue*, *errIllegalClient*, *errIllegalHardware*, *errIllegalScheduleNo*, *errIllegalSlotCount*, *errScheduleSlotPoolFull*

Remarks

The member '*Handle*' from each Schedule Slot will be updated by the hardware when this function successfully completes. This handle can be used to set up a "break point" on a schedule, using the method SetScheduleBreakPoint (⊞ see page 92).

Example:

The "main" schedule includes an event frame with a corresponding collision-resolve-schedule. Now the user can set a breakpoint e.g. on the first slot of this collision-resolving-schedule. During normal scheduling without a collision the breakpoint will never be reached. When a collision occurs within the event frame, the scheduler will branch to the resolving-schedule but will suspend itself before the first slot of the resolving-schedule is processed.

Now the user can analyse the received data and retrieve the reason for the collision.

See Also

GetSchedule (⊞ see page 90)

DeleteSchedule (⊞ see page 91)

StartSchedule (⧉ see page 93)

SuspendSchedule (⧉ see page 94)

ResumeSchedule (⧉ see page 95)

SetScheduleBreakPoint (⧉ see page 92)

PLinApi (⧉ see page 51)

Other languages: LIN_SetSchedule (⧉ see page 133)

4.3.1.1.1.28 GetSchedule

Gets the slots of a Schedule from a given Hardware.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_GetSchedule")]
public static extern TLINError GetSchedule(
    HLINHW hHw,
    int iScheduleNumber,
    [In, Out]
    TLINScheduleSlot[] pScheduleBuff,
    int iMaxSlotCount,
    out int pSlotCount
)
```

Visual Basic

```vbnet
<DllImport("plinapi.dll", EntryPoint:="LIN_GetSchedule")> _
Public Shared Function GetSchedule( _
    ByVal hHw As HLINHW, _
    ByVal iScheduleNumber As Integer, _
    <[In](), Out()> _
    ByVal pScheduleBuff As TLINScheduleSlot(), _
    ByVal iMaxSlotCount As Integer, _
    ByRef pSlotCount As Integer) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (⧉ see page 67)).
iScheduleNumber	Schedule number. See LIN_MIN_SCHEDULE_NUMBER (⧉ see page 106) and LIN_MAX_SCHEDULE_NUMBER (⧉ see page 106)
pScheduleBuff	Buffer for an array of TLINScheduleSlot (⧉ see page 30) structures.
iMaxSlotCount	Maximum number of slots to be read.
pSlotCount	Real number of slots read returned in the array 'pScheduleBuff'.

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalHardware, errIllegalScheduleNo, errIllegalSlotCount*

See Also

SetSchedule (🔲 see page 88)

DeleteSchedule (🔲 see page 91)

StartSchedule (🔲 see page 93)

SuspendSchedule (🔲 see page 94)

ResumeSchedule (🔲 see page 95)

SetScheduleBreakPoint (🔲 see page 92)

PLinApi (🔲 see page 51)

Other languages: LIN_GetSchedule (🔲 see page 135)

4.3.1.1.1.29 DeleteSchedule

Removes all slots contained by a Schedule of a given Hardware.

Namespace: Peak.Lin (🔲 see page 50)

Syntax

C#
```
[DllImport("plinapi.dll", EntryPoint = "LIN_DeleteSchedule")]
public static extern TLINError DeleteSchedule(
    HLINCLIENT hClient,
    HLINHW hHw,
    int iScheduleNumber
)
```

Visual Basic
```
<DllImport("plinapi.dll", EntryPoint:="LIN_DeleteSchedule")> _
Public Shared Function DeleteSchedule( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal iScheduleNumber As Integer) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (🔲 see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (🔲 see page 58)).
iScheduleNumber	Schedule number. See LIN_MIN_SCHEDULE_NUMBER (🔲 see page 106) and LIN_MAX_SCHEDULE_NUMBER (🔲 see page 106)

Returns

The return value is an TLINError (🔲 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue,* *errIllegalClient,* *errIllegalHardware,* *errIllegalScheduleNo,* *errIllegalSchedulerState*

See Also

GetSchedule (⧉ see page 90)

SetSchedule (⧉ see page 88)

StartSchedule (⧉ see page 93)

SuspendSchedule (⧉ see page 94)

ResumeSchedule (⧉ see page 95)

SetScheduleBreakPoint (⧉ see page 92)

PLinApi (⧉ see page 51)

Other languages: LIN_DeleteSchedule (⧉ see page 136)

4.3.1.1.1.30 SetScheduleBreakPoint

Sets a 'breakpoint' on a slot from a Schedule in a given Hardware.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_SetScheduleBreakPoint")]
public static extern TLINError SetScheduleBreakPoint(
    HLINCLIENT hClient,
    HLINHW hHw,
    int iBreakPointNumber,
    uint dwHandle
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_SetScheduleBreakPoint")> _
Public Shared Function SetScheduleBreakPoint( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal iBreakPointNumber As Integer, _
    ByVal dwHandle As Integer) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⧉ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⧉ see page 58)).
iBreakPointNumber	Breakpoint Number (0 or 1).
dwHandle	Slot Handle. See **Remarks**.

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue*, *errIllegalClient*, *errIllegalHardware*

Remarks

In order to 'delete' the breakpoint of a Schedule, call this method with the parameter '*dwHandle*' set to 0.

See Also

SetSchedule (⊡ see page 88)

GetSchedule (⊡ see page 90)

DeleteSchedule (⊡ see page 91)

StartSchedule (⊡ see page 93)

SuspendSchedule (⊡ see page 94)

ResumeSchedule (⊡ see page 95)

PLinApi (⊡ see page 51)

Other languages: LIN_SetScheduleBreakPoint (⊡ see page 137)

4.3.1.1.1.31 **StartSchedule**

Activates a Schedule in a given Hardware.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_StartSchedule")]
public static extern TLINError StartSchedule(
    HLINCLIENT hClient,
    HLINHW hHw,
    int iScheduleNumber
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_StartSchedule")> _
Public Shared Function StartSchedule( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal iScheduleNumber As Integer) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⊡ see page 58)).
iScheduleNumber	Schedule number. See LIN_MIN_SCHEDULE_NUMBER (⊡ see page 106) and LIN_MAX_SCHEDULE_NUMBER (⊡ see page 106)

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes

are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue,* *errIllegalClient,* *errIllegalHardware,* *errIllegalScheduleNo, errIllegalHardwareMode, errIllegalSchedule*

See Also

SetSchedule (🔳 see page 88)

GetSchedule (🔳 see page 90)

DeleteSchedule (🔳 see page 91)

SuspendSchedule (🔳 see page 94)

SetScheduleBreakPoint (🔳 see page 92)

ResumeSchedule (🔳 see page 95)

PLinApi (🔳 see page 51)

Other languages: LIN_StartSchedule (🔳 see page 138)

4.3.1.1.1.32 SuspendSchedule

Suspends an active Schedule in a given Hardware.

Namespace: Peak.Lin (🔳 see page 50)

Syntax

C#
```
[DllImport("plinapi.dll", EntryPoint = "LIN_SuspendSchedule")]
public static extern TLINError SuspendSchedule(
    HLINCLIENT hClient,
    HLINHW hHw
)
```

Visual Basic
```
<DllImport("plinapi.dll", EntryPoint:="LIN_SuspendSchedule")> _
Public Shared Function SuspendSchedule( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (🔳 see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (🔳 see page 58)).

Returns

The return value is an TLINError (🔳 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

SetSchedule (▣ see page 88)

GetSchedule (▣ see page 90)

DeleteSchedule (▣ see page 91)

StartSchedule (▣ see page 93)

ResumeSchedule (▣ see page 95)

SetScheduleBreakPoint (▣ see page 92)

PLinApi (▣ see page 51)

Other languages: LIN_SuspendSchedule (▣ see page 139)

4.3.1.1.1.33 ResumeSchedule

Restarts a configured Schedule in a given Hardware.

Namespace: Peak.Lin (▣ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_ResumeSchedule")]
public static extern TLINError ResumeSchedule(
    HLINCLIENT hClient,
    HLINHW hHw
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_ResumeSchedule")> _
Public Shared Function ResumeSchedule( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (▣ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (▣ see page 58)).

Returns

The return value is an TLINError (▣ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalSchedule, errIllegalHardwareMode, errIllegalSchedulerState*

See Also

SetSchedule (▣ see page 88)

GetSchedule (▣ see page 90)

DeleteSchedule (⧉ see page 91)

StartSchedule (⧉ see page 93)

SuspendSchedule (⧉ see page 94)

SetScheduleBreakPoint (⧉ see page 92)

PLinApi (⧉ see page 51)

Other languages: LIN_ResumeSchedule (⧉ see page 139)

4.3.1.1.1.34 XmtWakeUp

Sends a wake-up message impulse (single data byte F0h) to the given hardware.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_XmtWakeUp")]
public static extern TLINError XmtWakeUp(
    HLINCLIENT hClient,
    HLINHW hHw
)
```

Visual Basic

```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_XmtWakeUp")> _
Public Shared Function XmtWakeUp( _
          ByVal hClient As HLINCLIENT, _
          ByVal hHw As HLINHW) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⧉ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⧉ see page 58)).

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

Remarks

To know if a Hardware is in sleep-mode, use GetStatus (⧉ see page 97) and check the member '*State (⧉ see page 44)*' of the structure (⧉ see page 31) returned.

See Also

GetStatus (⧉ see page 97)

PLinApi (⧉ see page 51)

Other languages: LIN_XmtWakeUp (⧉ see page 140)

4.3.1.1.1.35 StartAutoBaud

Starts a process to detect the Baud rate of the LIN bus that is connected to the indicated Hardware.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_StartAutoBaud")]
public static extern TLINError StartAutoBaud(
    HLINCLIENT hClient,
    HLINHW hHw,
    ushort wTimeOut
)
```

Visual Basic

```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_StartAutoBaud")> _
Public Shared Function StartAutoBaud( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    ByVal wTimeOut As UShort) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (⊡ see page 53).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (⊡ see page 58)).
wTimeOut	Auto-baudrate timeout in milliseconds.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalHardwareState, errIllegalPeriod*

Remarks

The LIN Hardware must be not initialized in order to do an Auto-Baudrate procedure. To check if the hardware is initialized, use GetStatus (⊡ see page 97) and check the member '*State (⊡ see page 44)*' of the structure (⊡ see page 31) returned.

See Also

GetStatus (⊡ see page 97)

TLINHardwareStatus (⊡ see page 31)

GetClientParam (⊡ see page 58)

PLinApi (⊡ see page 51)

Other languages: LIN_StartAutoBaud (⊡ see page 141)

4.3.1.1.1.36 GetStatus

Retrieves current status information from the given Hardware.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetStatus")]
public static extern TLINError GetStatus(
    HLINHW hHw,
    out TLINHardwareStatus pStatusBuff
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetStatus")> _
Public Shared Function GetStatus( _
    ByVal hHw As HLINHW, _
    ByRef pStatusBuff As TLINHardwareStatus) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (⊡ see page 67)).
pStatusBuff	Buffer for a TLINHardwareStatus (⊡ see page 31) structure.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalHardware*

See Also

TLINHardwareStatus (⊡ see page 31)

TLINHardwareMode (⊡ see page 44)

TLINHardwareState (⊡ see page 44)

PLinApi (⊡ see page 51)

Other languages: LIN_GetStatus (⊡ see page 142)

4.3.1.1.1.37 CalculateChecksum

Calculates the checksum of a LIN Message.

Namespace: Peak.Lin (⊡ see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_CalculateChecksum")]
public static extern TLINError CalculateChecksum(
    ref TLINMsg pMsg
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_CalculateChecksum")> _
```

```
Public Shared Function CalculateChecksum( _
    ByRef pMsg As TLINMsg) As TLINError
End Function
```

Parameters

Parameters	Description
pMsg	Buffer for a TLINMsg (🔲 see page 26) structure.

Returns

The return value is an TLINError (🔲 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalLength*

Remarks

The member '*Checksum*' of the TLINMsg (🔲 see page 26) structure contained in the given '*pMsg*' buffer (message buffer), will be updated with the calculated checksum, when this function successfully completes.

See Also

TLINMsg (🔲 see page 26)

PLinApi (🔲 see page 51)

Other languages: LIN_CalculateChecksum (🔲 see page 143)

4.3.1.1.1.38 GetVersion

Returns a TLINVersion (🔲 see page 25) sttructure containing the PLIN-API DLL version.

Namespace: Peak.Lin (🔲 see page 50)

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetVersion")]
public static extern TLINError GetVersion(
    ref TLINVersion pVerBuffer
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint := "LIN_GetVersion")> _
Public Shared Function GetVersion( _
        ByRef pVerBuff As TLINVersion) As TLINError
End Function
```

Parameters

Parameters	Description
pVerBuffer	Buffer for a TLINVersion (🔲 see page 25) structure.

Returns

The return value is an TLINError (🔲 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue*

See Also

TLINVersion (⧉ see page 25)

PLinApi (⧉ see page 51)

Other languages: LIN_GetVersion (⧉ see page 144)

4.3.1.1.1.39 GetVersionInfo

Returns a string containing Copyright information.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#
```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetVersionInfo")]
public static extern TLINError GetVersionInfo(
    [MarshalAs(UnmanagedType.LPStr, SizeParamIndex = 1)]
    string pTextBuff,
    int wBuffSize
)
```

Visual Basic
```
<DllImport("plinapi.dll", EntryPoint := "LIN_GetVersionInfo")> _
Public Shared Function GetVersionInfo( _
    <MarshalAs(UnmanagedType.LPStr, SizeParamIndex:=1)> _
    ByVal pTextBuff As String, _
    ByVal wBuffSize As Integer) As TLINError
End Function
```

Parameters

Parameters	Description
pTextBuff	Buffer for a string value.
wBuffSize	Size in bytes of the *pBuff* buffer.

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

| DLL Interaction: | *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess* |
| API Return: | *errWrongParameterValue, errBufferInsufficient* |

See Also

PLinApi (⧉ see page 51)

Other languages: LIN_GetVersionInfo (⧉ see page 144)

4.3.1.1.1.40 GetErrorText

Returns a descriptive text of a given TLINError (⧉ see page 46) error code.

Namespace: Peak.Lin (⧉ see page 50)

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_GetErrorText")]
public static extern TLINError GetErrorText(
    TLINError dwError,
    byte bLanguage,
    [MarshalAs(UnmanagedType.LPStr, SizeParamIndex = 2)]
    string strTextBuff,
    int wBuffSize
)
```

Visual Basic

```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_GetErrorText")> _
Public Shared Function GetErrorText( _
    ByVal dwError As TLINError, _
    ByVal bLanguage As Byte, _
    <MarshalAs(UnmanagedType.LPStr, SizeParamIndex:=2)> _
    ByVal strTextBuff As String, _
    ByVal wBuffSize As Integer) As TLINError
End Function
```

Parameters

Parameters	Description
dwError	A TLINError (🗷 see page 46) Code.
bLanguage	Indicates a "Primary language ID". See **Remarks** for more information.
strTextBuff	Buffer for a string value.
wBuffSize	Size in bytes of the *strTextBuff* buffer.

Returns

The return value is an TLINError (🗷 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

| DLL Interaction: | *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess* |
| API Return: | *errWrongParameterValue, errBufferInsufficient* |

Remarks

The "Primary language IDs" are codes used by Windows OS from Microsoft, to identify a human language. The PLIN system currently support the following languages:

Language	Primary Language ID
Neutral (English)	00h (0)
English	09 (9)
German	07h (7)
Italian	10h (16)
Spanish	0Ah (10)

This method will fail if a code, not listed above, is used.

See Also

TLINError (🗷 see page 46)

PLinApi (🗷 see page 51)

Primary language ID

Other languages: LIN_GetErrorText ()

4.3.1.1.1.41 **GetPID**

Calculates the '*Parity Frame ID'* (PID) from a Frame ID.

Namespace: Peak.Lin ()

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_GetPID")]
public static extern TLINError GetPID(
    ref byte pFrameId
)
```

Visual Basic

```vb
<DllImport("plinapi.dll", EntryPoint:="LIN_GetPID")> _
Public Shared Function GetPID( _
    ByRef pFrameId As Byte) As TLINError
End Function
```

Parameters

Parameters	Description
pFrameId	Frame ID. A value between 0 and LIN_MAX_FRAME_ID (see page 106).

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalFrameID*

Remarks

The given '*pFrameId'* parameter will be updated with the calculated PID, when this function successfully completes.

See Also

Constants ()

PLinApi ()

4

Other languages: LIN_GetPID ()

4.3.1.1.1.42 **GetTargetTime**

Gets the time used by the LIN hardware adapter.

Namespace: Peak.Lin ()

Syntax

C#

```csharp
[DllImport("plinapi.dll", EntryPoint = "LIN_GetTargetTime")]
public static extern TLINError GetTargetTime(
    HLINHW hHw,
    out UInt64 pTargetTime
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetTargetTime")> _
Public Shared Function GetTargetTime( _
    ByVal hHw As HLINHW, _
    ByRef pTargetTime As ULong) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (☐ see page 67)).
pTargetTime	Buffer for a unsigned 64 bits integer value (Target time buffer).

Returns

The return value is an TLINError (☐ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalHardware*

See Also

GetSystemTime (☐ see page 105)

GetAvailableHardware (☐ see page 67)

PLinApi (☐ see page 51)

Other languages: LIN_GetTargetTime (☐ see page 147)

4.3.1.1.1.43 SetResponseRemap

Sets the publisher response remap. This will override the complete remap table of the hardware.

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_SetResponseRemap")]
public static extern TLINError SetResponseRemap(
    HLINCLIENT hClient,
    HLINHW hHw,
    [MarshalAs(UnmanagedType.LPArray, SizeConst = 64)]
    byte[] pRemapTab
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_SetResponseRemap")> _
Public Shared Function SetResponseRemap( _
    ByVal hClient As HLINCLIENT, _
    ByVal hHw As HLINHW, _
    <MarshalAs(UnmanagedType.LPArray, SizeConst:=64)> _
    ByVal pRemapTab As Byte()) As TLINError
End Function
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function RegisterClient (☐ see page 53).

hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See GetClientParam (▣ see page 58)).
pRemapTab	Buffer for an array of 64 bytes (IDs).

Returns

The return value is an TLINError (▣ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue*, *errIllegalClient*, *errIllegalHardware*, *errIllegalFrameID*

Remarks

- The response remap mechanism only affects the remapping of the publisher response data in slave mode.

- ID x is remapped to ID y: when ID x is received, the frame settings where taken from ID x, but the data itself is taken from ID y.

- Setting the remap will override the old mapping. All pending responses for single shot frames will be killed and therefore be lost.

The mode of use of this feature is showed in the following example:

C#:

```
byte[] RemapTab;
RemapTab = new byte[64];

...
// Remaps each ID to itself ( default)
//
for ( int i = 0; i <= LIN_MAX_FRAME_ID; i++)
  RemapTab[i] = i;
...

...
// Remaps publisher response from ID 11 to ID 9
//
RemapTab[11] = 9;
...

...
// The client hClient Sets the Remap Table on the hardware hHw
//
SetResponseRemap(hClient, hHw, RemapTab);
...
```

See Also

Processing Event Frames (▣ see page 21)

GetResponseRemap (▣ see page 104)

UpdateByteArray (▣ see page 85)

Other languages: LIN_SetResponseRemap (▣ see page 147)

4.3.1.1.1.44 GetResponseRemap

Gets the publisher remap table from a given hardware.

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetResponseRemap")]
public static extern TLINError GetResponseRemap (
    HLINHW hHw,
    [MarshalAs(UnmanagedType.LPArray, SizeConst = 64)]
    byte[] pRemapTab);
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetResponseRemap")> _
Public Shared Function GetResponseRemap( _
    ByVal hHw As HLINHW, _
    <MarshalAs(UnmanagedType.LPArray, SizeConst:=64)> _
    ByVal pRemapTab As Byte()) As TLINError
End Function
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function GetAvailableHardware (⧉ see page 67)).
pRemapTab	Buffer for an array of 64 bytes (IDs).

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalHardware*

See Also

Processing Event Frames (⧉ see page 21)

SetResponseRemap (⧉ see page 103)

UpdateByteArray (⧉ see page 85)

Other languages: LIN_GetResponseRemap (⧉ see page 149)

4.3.1.1.1.45 GetSystemTime

Gets the current system time. This time is returned by Windows as the elapsed number of microseconds since system start.

Syntax

C#

```
[DllImport("plinapi.dll", EntryPoint = "LIN_GetSystemTime")]
public static extern TLINError GetSystemTime(
    out UInt64 pTargetTime
)
```

Visual Basic

```
<DllImport("plinapi.dll", EntryPoint:="LIN_GetSystemTime")> _
Public Shared Function GetSystemTime( _
    ByRef pSystemTime As ULong) As TLINError
End Function
```

Parameters

Parameters	Description
pSystemTime	Buffer for a unsigned 64 bits integer value (System time buffer).

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errMemoryAccess*
API Return:	*errWrongParameterValue*

See Also

GetTargetTime (see page 102)

GetAvailableHardware (see page 67)

PLinApi (see page 51)

Other languages: LIN_GetSystemTime (see page 150)

4.3.1.1.2 Constants

The following constant values are defined within the class PLinApi (see page 51):

	Type	Constant	Value	Description
	Byte	INVALID_LIN_HANDLE	0	Invalid value for all LIN handles (Client, Hardware).
	Byte	LIN_HW_TYPE_USB	1	Hardware Type: LIN USB.
	Byte	LIN_MAX_FRAME_ID	63	Maximum allowed Frame ID.
	Int32	LIN_MAX_SCHEDULES	8	Maximum allowed Schedules per Hardware.
	Int32	LIN_MIN_SCHEDULE_NUMBER	0	Minimum Schedule number.
	Int32	LIN_MAX_SCHEDULE_NUMBER	7	Maximum Schedule number.
	Int32	LIN_MAX_SCHEDULE_SLOTS	256	Maximum allowed Schedule slots per Hardware.
	UInt16	LIN_MIN_BAUDRATE	1000	Minimum LIN Baudrate.
	UInt16	LIN_MAX_BAUDRATE	20000	Maximum LIN Baudrate.
	UInt16	LIN_MAX_NAME_LENGTH	48	Maximum number of bytes for Name / ID of a Hardware or Client.
	Int32	FRAME_FLAG_RESPONSE_ENABLE	1 (0x0001)	Slave Enable Publisher Response.
	Int32	FRAME_FLAG_SINGLE_SHOT	2 (0x0002)	Slave Publisher Single shot.
	Int32	FRAME_FLAG_IGNORE_INIT_DATA	4 (0x0004)	Ignores InitialData on set frame entry.
	Int32	LOG_FLAG_DEFAULT	0 (0x0000)	Logs system exceptions / errors
	Int32	LOG_FLAG_ENTRY	1 (0x0001)	Logs the entries to the PLIN-API functions
	Int32	LOG_FLAG_PARAMETERS	2 (0x0002)	Logs the parameters passed to the PLIN-API functions
	Int32	LOG_FLAG_LEAVE	4 (0x0004)	Logs the exits from the PLIN-API functions
	Int32	LOG_FLAG_WRITE	8 (0x0008)	Logs the LIN messages passed to the LIN_Write (see page 118) function

4

	Int32	LOG_FLAG_READ	16 (0x0010)	Logs the LIN messages received within the LIN_Read (☐ see page 117) function
	Int32	LOG_FLAG_ALL	65535 (0xFFFF)	Logs all possible information within the PLIN-API functions
	Int32	LIN_MAX_USER_DATA	24	Maximum number of bytes that a user can read/write on a Hardware.
	Int32	LIN_MIN_BREAK_LENGTH	13	Minimum number of bits that can be used as break field in a LIN frame.
	Int32	LIN_MAX_BREAK_LENGTH	32	Maximum number of bits that can be used as break field in a LIN frame.
	Int32	LIN_MAX_RCV_QUEUE_COUNT	65535	Maximum number of LIN frames that can be stored in the reception queue of a client

Remarks

The version for other languages (Pascal, C/C++) for this values are described in Definitions (☐ see page 150).

4.4 **Functions**

The functions of the PLIN API are divided in 5 groups of functionality:

Information

	Function	Description
◈	LIN_GetClientParam (☐ see page 113)	Gets a Client parameter.
◈	LIN_GetClientFilter (☐ see page 116)	Gets the filter corresponding to a given Client-Hardware pair.
◈	LIN_GetAvailableHardware (☐ see page 120)	Gets an array containing the handles of the current Hardware available in the system.
◈	LIN_GetHardwareParam (☐ see page 123)	Gets a Hardware parameter.
◈	LIN_IdentifyHardware (☐ see page 127)	Physically identifies a LIN Hardware.
◈	LIN_GetFrameEntry (☐ see page 129)	Gets the configuration of a LIN Frame from a given Hardware.
◈	LIN_GetSchedule (☐ see page 135)	Gets the slots of a Schedule from a Hardware.
◈	LIN_GetStatus (☐ see page 142)	Retrieves current status information from a Hardware.
◈	LIN_GetVersion (☐ see page 144)	Returns the PLIN-API DLL version.
◈	LIN_GetVersionInfo (☐ see page 144)	Returns a string containing Copyright information.
◈	LIN_GetErrorText (☐ see page 145)	Gets a description text for a LIN-Error code.
◈	LIN_GetPID (☐ see page 146)	Calculates a Frame ID with Parity.
◈	LIN_GetTargetTime (☐ see page 147)	Gets the time used by the LIN-USB adapter.
◈	LIN_CalculateChecksum (☐ see page 143)	Calculates the checksum of a LIN Message.
◈	LIN_GetResponseRemap (☐ see page 149)	Gets the publisher remap table from a given hardware.
◈	LIN_GetSystemTime (☐ see page 150)	Gets the current system time.

Client

	Function	Description
⇒♦	LIN_RegisterClient (⬈ see page 109)	Registers a Client at the LIN Manager.
⇒♦	LIN_RemoveClient (⬈ see page 110)	Removes a Client from the Client list of the LIN Manager.
⇒♦	LIN_ConnectClient (⬈ see page 110)	Connects a Client to a Hardware.
⇒♦	LIN_DisconnectClient (⬈ see page 111)	Disconnects a Client from a Hardware.
⇒♦	LIN_ResetClient (⬈ see page 112)	Flushes the Receive Queue of the Client and resets its counters.
⇒♦	LIN_SetClientParam (⬈ see page 112)	Sets a Client parameter to a given value.
⇒♦	LIN_SetClientFilter (⬈ see page 115)	Sets the filter of a Client.
⇒♦	LIN_RegisterFrameId (⬈ see page 127)	Modifies the filter of a Client.

Configuration

	Function	Description
⇒♦	LIN_InitializeHardware (⬈ see page 119)	Initializes a Hardware with a given Mode and Baud rate.
⇒♦	LIN_ResetHardware (⬈ see page 125)	Flushes the queues of the Hardware and resets its counters.
⇒♦	LIN_ResetHardwareConfig (⬈ see page 126)	Deletes the current configuration of the Hardware and sets its defaults.
⇒♦	LIN_SetHardwareParam (⬈ see page 121)	Sets a Hardware parameter to a given value.
⇒♦	LIN_SetFrameEntry (⬈ see page 128)	Configures a LIN Frame in a given Hardware.
⇒♦	LIN_SetSchedule (⬈ see page 133)	Configures the slots of a Schedule in a Hardware.
⇒♦	LIN_SetScheduleBreakPoint (⬈ see page 137)	Sets a 'breakpoint' on a slot from a Schedule in a Hardware.
⇒♦	LIN_DeleteSchedule (⬈ see page 136)	Removes all slots contained by a Schedule of a Hardware.
⇒♦	LIN_UpdateByteArray (⬈ see page 130)	Updates the data of a LIN Frame for a given Hardware.
⇒♦	LIN_SetResponseRemap (⬈ see page 147)	Sets the publisher response remap.

Control

	Function	Description
⇒♦	LIN_StartKeepAlive (⬈ see page 131)	Sets a Frame as Keep-Alive frame for a Hardware and starts to send.
⇒♦	LIN_SuspendKeepAlive (⬈ see page 132)	Suspends the sending of a Keep-Alive frame in a Hardware.
⇒♦	LIN_ResumeKeepAlive (⬈ see page 132)	Resumes the sending of a Keep-Alive frame in a Hardware.
⇒♦	LIN_StartSchedule (⬈ see page 138)	Activates a Schedule in a Hardware.
⇒♦	LIN_SuspendSchedule (⬈ see page 139)	Suspends an active Schedule in a Hardware.
⇒♦	LIN_ResumeSchedule (⬈ see page 139)	Restarts a configured Schedule in a Hardware.
⇒♦	LIN_StartAutoBaud (⬈ see page 141)	Starts a process to detect the Baud rate of the LIN bus on a Hardware.

Communication

4

Function	Description
⇒◆ LIN_Read (🗔 see page 117)	Reads the next message/status information from a Client's Receive Queue.
⇒◆ LIN_ReadMulti (🗔 see page 117)	Reads several received messages.
⇒◆ LIN_Write (🗔 see page 118)	Transmits a message to a LIN Hardware.
⇒◆ LIN_XmtWakeUp (🗔 see page 140)	Sends a Wake-Up message impulse.

4.4.1 LIN_RegisterClient

Registers a Client at the PLIN Manager. Creates a Client handle and allocates the Receive Queue (only one per Client). The hWnd parameter can be zero for DOS Box Clients. The Client does not receive any messages until LIN_RegisterFrameId (🗔 see page 127) or LIN_SetClientFilter (🗔 see page 115) is called. The handle has to be saved for further PLIN-API function calls.

Syntax

Pascal

```
function LIN_RegisterClient(
    const strName: PAnsiChar;
    hWnd: Longword;
    var hClient:
    HLINCLIENT
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_RegisterClient (
    LPSTR strName,
    DWORD hWnd,
    HLINCLIENT *hClient
);
```

Parameters

Parameters	Description
strName	Name of the Client. See LIN_MAX_NAME_LENGTH (🗔 see page 150)
hWnd	Window handle of the Client (only for information purposes).
hClient	Client handle buffer. Returns the client handle that has been assigned to the client.

Returns

The return value is an TLINError (🗔 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction: *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*

API Return: *errWrongParameterValue, errOutOfResource*

See Also

LIN_RemoveClient (🗔 see page 110)

LIN_ConnectClient (🗔 see page 110)

.NET Version: RegisterClient (🗔 see page 53)

4.4.2 **LIN_RemoveClient**

Removes a Client from the Client list of the LIN Manager. Frees all resources (receive queues, message counters, etc.). If the Client was a Boss-Client for one or more Hardware, the Boss-Client property for those Hardware will be set to INVALID_LIN_HANDLE (☐ see page 150).

Syntax

Pascal

```
function LIN_RemoveClient(
    hClient: HLINCLIENT
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_RemoveClient (
    HLINCLIENT hClient
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (☐ see page 109).

Returns

The return value is an TLINError (☐ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient*

See Also

LIN_RegisterClient (☐ see page 109)

.NET Version: RemoveClient (☐ see page 54)

4.4.3 **LIN_ConnectClient**

Connects a Client to a Hardware. The Hardware is assigned by its Handle.

Syntax

Pascal

```
function LIN_ConnectClient(
    hClient: HLINCLIENT;
    hHw: HLINHW
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_ConnectClient (
    HLINCLIENT hClient,
    HLINHW hHw
```

```
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊡ see page 109).
hHw	Handle of the Hardware to be connected. (One handle from the list of available hardware, returned by the function LIN_GetAvailableHardware (⊡ see page 120)).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

LIN_RegisterClient (⊡ see page 109)

LIN_DisconnectClient (⊡ see page 111)

LIN_GetAvailableHardware (⊡ see page 120)

.NET Version: ConnectClient (⊡ see page 55)

4.4.4 LIN_DisconnectClient

Disconnects a Client from a Hardware. This means: no more messages will be received by this Client from this Hardware.

Syntax

Pascal

```
function LIN_DisconnectClient(
    hClient: HLINCLIENT;
    hHw: HLINHW
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_DisconnectClient (
    HLINCLIENT hClient,
    HLINHW hHw
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊡ see page 109).
hHw	Handle of the Hardware to be disconnected. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊡ see page 113)).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

| **DLL Interaction:** | *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess* |
| **API Return:** | *errWrongParameterValue, errIllegalClient, errIllegalHardware* |

See Also

LIN_ConnectClient ()

LIN_RemoveClient ()

LIN_GetClientParam ()

.NET Version: DisconnectClient ()

4.4.5 LIN_ResetClient

Flushes the Receive Queue of the Client and resets its counters.

Syntax

Pascal

```
function LIN_ResetClient(
    hClient: HLINCLIENT
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_ResetClient (
    HLINCLIENT hClient
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (see page 109).

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

| **DLL Interaction:** | *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess* |
| **API Return:** | *errWrongParameterValue, errIllegalClient* |

See Also

LIN_RegisterClient ()

.NET Version: ResetClient ()

4.4.6 LIN_SetClientParam

Sets a Client parameter to a given value.

Syntax

Pascal

```
function LIN_SetClientParam(
    hClient: HLINCLIENT;
    wParam: TLINClientParam;
    dwValue: Integer
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_SetClientParam (
    HLINCLIENT hClient,
    TLINClientParam wParam,
    DWORD dwValue
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊠ see page 109).
wParam	The TLINClientParam (⊠ see page 36) parameter to be set. Allowed are: • clpReceiveStatusFrames • clpLogStatus • clpLogConfiguration
dwValue	Value to be set. 0 to deactivate it, otherwise to active it.

Returns

The return value is an TLINError (⊠ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient*

Remarks

The value of the parameter **hClient** is not important when configuring the Log functionality (*clpLogStatus*, *clpLogConfiguration*). Logging doesn't depend on a connected client since this is an API-wide operation. The value of **hClient** for this case should be set to **0** for clarity, though passing a valid handle also works. The handle is just ignored.

See Also

TLINClientParam (⊠ see page 36)

LIN_GetClientParam (⊠ see page 113)

Log File Generation (⊠ see page 152)

.NET Version: SetClientParam (⊠ see page 57)

4.4.7 **LIN_GetClientParam**

Gets a Client parameter.

Syntax

Pascal

```
function LIN_GetClientParam(
    hClient: HLINCLIENT;
    wParam: TLINClientParam;
    pBuff: Pointer;
    wBuffSize: Word
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetClientParam (
    HLINCLIENT hClient,
    TLINClientParam wParam,
    void *pBuff,
    WORD wBuffSize
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊡ see page 109).
wParam	The TLINClientParam (⊡ see page 36) parameter to be retrieved. Allowed are: • clpName • clpMessagesOnQueue • clpWindowHandle • clpConnectedHardware • clpTransmittedMessages • clpReceivedMessages • clpReceiveStatusFrames • clpLogStatus • clpLogConfiguration
pBuff	Buffer for the parameter value.
wBuffSize	Buffer size in bytes.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient, errBufferInsufficient*

Remarks

According with the type of the desired client parameter, apply the following rules:

• For an integer value (clpMessagesOnQueue, clpWindowHandle, clpTransmittedMessages, clpReceivedMessages, clpReceiveStatusFrames, clpLogStatus, clpLogConfiguration)

  • *pBuff* must be a pointer to an integer value.

  • *wBuffSize* should be set to 0 (default value, 4, is used) or to a value same or greater than 4 (size in bytes of an 32 bits integer).

• For a string value (clpName)

  • *pBuff* must be a pointer to a char array. The returned value is a null terminated string.

- *wBuffSize* should be set to the capacity of bytes supported by *pBuff*. The returned string can have a length of until LIN_MAX_NAME_LENGTH (see page 150)
- For an array of Hardware handles (clpConnectedHardware)
  - *pBuff* must be a pointer to an array of HLINHW (see page 34).
  - *wBuffSize* should be set to the capacity of bytes supported by *pBuff*. The array buffer must at least have a length of **"Total handles + 1" multiplied by the size of HLINHW (see page 34)**, where "Total handles" is the actual count of connected hardware. Example (3 hardware connected): the *pBuff* buffer must have a size of 8 bytes (*(3+1)*sizeof(HLINHW (see page 34))*).
  - The returned data in *pBuff* is to be interpreted as follow: The first position of the array (position 0) contains the amount of hardware (*n*) handles returned in the buffer. The connected hardware handles are returned beginning at the position 1 to *n*. Example (3 hardware connected): the *pBuff*[0] contain the value 3 and the positions [1], [2] and [3] the returned handles.

The value of the parameter **hClient** is not important when retrieving Log information (*clpLogStatus*, *clpLogConfiguration*). Logging doesn't depend on a connected client since this is an API-wide operation. The value of **hClient** for this case should be set to **0** for clarity, though passing a valid handle also works. The handle is just ignored.

See Also

TLINClientParam (see page 36)

LIN_SetClientParam (see page 112)

.NET Version: GetClientParam (see page 58)

4.4.8 **LIN_SetClientFilter**

Sets the filter of a Client and modifies the filter of the connected Hardware.

Syntax

Pascal

```
function LIN_SetClientFilter(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    iRcvMask: UInt64
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_SetClientFilter (
    HLINCLIENT hClient,
    HLINHW hHw,
    unsigned __int64 iRcvMask
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (see page 113)).
iRcvMask	Filter. Each bit corresponds to a Frame ID (0..63).

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes

are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue*, *errIllegalClient*, *errIllegalHardware*

Remarks

This function replaces any configured filter for the given client within the given hardware instead of expands it. To expand a Message Filter, use the function LIN_RegisterFrameId (🔲 see page 127).

See Also

LIN_GetClientFilter (🔲 see page 116)

LIN_GetClientParam (🔲 see page 113)

.NET Version: SetClientFilter (🔲 see page 62)

4.4.9 **LIN_GetClientFilter**

Gets the filter corresponding to a given Client-Hardware pair.

Syntax

Pascal

```
function LIN_GetClientFilter(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    var pRcvMask: UInt64
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetClientFilter (
    HLINCLIENT hClient,
    HLINHW hHw,
    unsigned __int64 *pRcvMask
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (🔲 see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (🔲 see page 113)).
pRcvMask	Filter buffer. Each bit corresponds to a Frame ID (0..63).

Returns

The return value is an TLINError (🔲 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue*, *errIllegalClient*, *errIllegalHardware*

See Also

LIN_SetClientFilter (🔲 see page 115)

LIN_GetClientParam (see page 113)

.NET Version: GetClientFilter (see page 62)

4.4.10 **LIN_Read**

Reads the next message/status information from a Client's Receive Queue. The message will be written to the given buffer.

Syntax

Pascal

```
function LIN_Read(
    hClient: HLINCLIENT;
    var pMsg: TLINRcvMsg
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_Read (
    HLINCLIENT hClient,
    TLINRcvMsg *pMsg
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (see page 109).
pMsg	A TLINRcvMsg (see page 27) buffer.

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errRcvQueueEmpty*

See Also

TLINRcvMsg (see page 27)

LIN_ReadMulti (see page 117)

.NET Version: Read (see page 63)

4.4.11 **LIN_ReadMulti**

Reads several received messages. This function is the equivalent to a multiple call of the LIN_Read (see page 117) function.

Syntax

Pascal

```
function LIN_ReadMulti(
    hClient: HLINCLIENT;
    pMsgBuff: Pointer;
    iMaxCount: Integer;
    var pCount: Integer
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_ReadMulti (
    HLINCLIENT hClient,
    TLINRcvMsg *pMsgBuff,
    int iMaxCount,
    int *pCount
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⧉ see page 109).
pMsgBuff	Buffer for an array of TLINRcvMsg (⧉ see page 27) messages. The size of this buffer must be at least iMaxCount * sizeof(TLINRcvMsg (⧉ see page 27)) bytes.
iMaxCount	Desired amount of messages to be read.
pCount	The actual count of messages read.

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errRcvQueueEmpty*

Remarks

The read buffer must be an array of '*iMaxCount*' entries (must have at least a size of *iMaxCount* * sizeof(TLINRcvMsg (⧉ see page 27)) bytes). The size '*iMaxCount*' of the array is the maximum amount of messages that can be received. The real number of read messages will be returned in '*pCount*'.

LIN_ReadMulti is like a multiple call of the LIN_Read (⧉ see page 117) function. The return value of LIN_ReadMulti corresponds to the return value of the last call of LIN_Read (⧉ see page 117). This it means, if all messages were read from the queue but the count of read messages is less than '*iMaxCount*', the error errRcvQueueEmpty is returned. If the count of read messages is just the same amount of desired messages, the return value is errOK.

See Also

TLINRcvMsg (⧉ see page 27)

LIN_Read (⧉ see page 117)

.NET Version: ReadMulti (⧉ see page 64)

4.4.12 **LIN_Write**

Transmits a message to a LIN Hardware.

Syntax

Pascal

```
function LIN_Write(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    var pMsg: TLINMsg
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_Write (
    HLINCLIENT hClient,
    HLINHW hHw,
    TLINMsg *pMsg
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (◪ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (◪ see page 113)).
pMsg	A write buffer.

Returns

The return value is an TLINError (◪ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalDirection, errIllegalLength, errXmtQueueFull*

Remarks

The Client 'hClient' transmits a message 'pMsg' to the Hardware 'hHw'. The message is written into the Transmit Queue of the Hardware.

See Also

TLINMsg (◪ see page 26)

LIN_GetClientParam (◪ see page 113)

.NET Version: Write (◪ see page 65)

4.4.13 **LIN_InitializeHardware**

Initializes a Hardware with a given mode and baud rate.

Syntax

Pascal

```
function LIN_InitializeHardware(
    hClient: HLINCLIENT;
```

```
    hHw: HLINHW;
    bMode: TLINHardwareMode;
    wBaudrate: Word
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_InitializeHardware (
    HLINCLIENT hClient,
    HLINHW hHw,
    TLINHardwareMode bMode,
    WORD wBaudrate
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊞ see page 109).
hHw	Handle of the Hardware to initialize. (One handle from the list of available hardware, returned by the function LIN_GetAvailableHardware (⊞ see page 120)).
byMode	Hardware mode. See TLINHardwareMode (⊞ see page 44).
wBaudrate	The baud rate to configure the speed of the hardware. See LIN_MIN_BAUDRATE (⊞ see page 150) and LIN_MAX_BAUDRATE (⊞ see page 150).

Returns

The return value is an TLINError (⊞ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalBaudrate*

Remarks

If the Hardware was initialized by another Client, the function will re-initialize the Hardware. All connected clients will be affected. It is the job of the user to manage the setting and/or configuration of Hardware, e.g. by using the Boss-Client (⊞ see page 70) parameter of the Hardware.

See Also

TLINHardwareMode (⊞ see page 44)

LIN_GetAvailableHardware (⊞ see page 120)

Definitions (⊞ see page 150)

.NET Version: InitializeHardware (⊞ see page 66)

4.4.14 **LIN_GetAvailableHardware**

Gets an array containing the handles of the current Hardware available in the system.

Syntax

Pascal

```
function LIN_GetAvailableHardware(
    pBuff: PHLINHW;
    wBuffSize: Word;
```

```
    var pCount: Integer
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetAvailableHardware(
    HLINHW *pBuff,
    WORD wBuffSize,
    int *pCount
);
```

Parameters

Parameters	Description
pBuff	Buffer for an array of HLINHW (◪ see page 34) handles.
wBuffSize	Size of the buffer in bytes (*pBuff*-capacitiy * sizeof(HLINHW (◪ see page 34))).
pCount	The count of available Hardware handles returned in the array 'pBuff'.

Returns

The return value is an TLINError (◪ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errBufferInsufficient*

Remarks

To only get the amount of hardware present in a system, call this function with an empty array ('pBuff' set to *NULL or nil*) and wBuffSize equal to **zero**. Than, the function returns errOK and the count of available hardware will be written in 'pCount'.

See Also

HLINHW (◪ see page 34)

.NET Version: GetAvailableHardware (◪ see page 67)

4.4.15 **LIN_SetHardwareParam**

Sets a Hardware parameter to a given value.

Syntax

Pascal

```
function LIN_SetHardwareParam(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    wParam: TLINHardwareParam;
    pBuff: Pointer;
    wBuffSize: Word
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_SetHardwareParam (
    HLINCLIENT hClient,
    HLINHW hHw,
    TLINHardwareParam wParam,
    void *pBuff,
    WORD wBuffSize
```

```
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊠ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊠ see page 113)).
wParam	The TLINHardwareParam (⊠ see page 37) parameter to be set. Allowed are: • hwpMessageFilter • hwpBossClient • hwpIdNumber • hwpUserData
pBuff	Buffer for the parameter value.
wBuffSize	Buffer size in bytes.

Returns

The return value is an TLINError (⊠ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterType, errWrongParameterValue, errIllegalClient, errIllegalHardware*

Remarks

According with the type of the desired client parameter, apply the following rules:

• For a 64 bits integer value (hwpMessageFilter)

  • *pBuff* must be a pointer to an unsigned 64 bits integer.

  • *wBuffSize* must be set to 8 or greater.

  • The data in *pBuff* must be formatted as follow: Each bit corresponds to a Frame ID (0..63).

• For a Client handle (hwpBossClient)

  • *pBuff* must be a pointer to a HLINCLIENT (⊠ see page 33) handle.

  • *wBuffSize* must be set to sizeof(HLINCLIENT (⊠ see page 33)) or greater.

• For a string value (hwpIdNumber)

  • *pBuff* must be a pointer to a unsigned 32 bits integer.

  • *wBuffSize* must be set to 8 or greater.

• *For a char/byte array value (hwpUserData):*

  • *pBuff must be a pointer to a char/byte array.*

  • *wBuffSize* must be set to the length of the data contained by *pBuff*. This value can not be greater than LIN_MAX_USER_DATA (⊠ see page 150).

About the Boss-Client:

The Boss-Client parameter is actually an user-controlled Access Permission. Setting this value **does not causes changes** in the operation of the LIN hardware or restricts other clients of doing a configuration, but gives the user the possibility to implement an "Access Protection" to avoid undesired configuration processes.

The principle is very simple: An application checks this parameter using the function LIN_GetHardwareParam (⊠ see page 123). The returned value is the handle of the Client who is registered as the Boss for that hardware in the PLIN system, i.e. this Client owns the permission to modify the hardware. If the return value is null (Nothing in VisualBasic), nobody has

control over the configuration of the hardware. By default, the Boss Client is configured to be the Client that initializes the hardware for the first time. To clear a configured Boss Client, a value of 0 within the buffer (pBuff) must be passed to the function LIN_SetHardwareParam.

It stays free to the user to decide how to react when checking this parameter.

Take in count that making changes in the configuration of the hardware with a client which is not the Boss-Client, will still work. The PLIN system **does not check** for Client permissions.

See Also

TLINHardwareParam (⚏ see page 37)

LIN_GetHardwareParam (⚏ see page 123)

Definitions (⚏ see page 150)

.NET Version: SetHardwareParam (⚏ see page 68)

4.4.16 **LIN_GetHardwareParam**

Gets a Hardware parameter.

Syntax

Pascal

```
function LIN_GetHardwareParam(
    hHw: HLINHW;
    wParam: TLINHardwareParam;
    pBuff: Pointer;
    wBuffSize: Word
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetHardwareParam (
    HLINHW hHw,
    TLINHardwareParam wParam,
    void *pBuff,
    WORD wBuffSize
);
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function LIN_GetAvailableHardware (⚏ see page 120)).

wParam	The TLINHardwareParam (⊡ see page 37) parameter to be retrieved. Allowed are:
	• hwpName
	• hwpDeviceNumber
	• hwpChannelNumber
	• hwpConnectedClients
	• hwpMessageFilter
	• hwpBaudrate
	• hwpMode
	• hwpFirmwareVersion
	• hwpBufferOverrunCount
	• hwpBossClient
	• hwpSerialNumber
	• hwpVersion
	• hwpType
	• hwpQueueOverrunCount
	• hwpIdNumber
	• hwpUserData
	• hwpScheduleActive
	• hwpScheduleState
	• hwpScheduleSuspendedSlot
pBuff	Buffer for the parameter value.
wBuffSize	Buffer size in bytes.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction: *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*

API Return: *errWrongParameterType, errWrongParameterValue, errIllegalHardware, errBufferInsufficient*

Remarks

According with the type of the desired client parameter, apply the following rules:

- For an integer value (hwpDeviceNumber, hwpChannelNumber, hwpBaudrate, hwpMode, hwpBufferOverrunCount, hwpSerialNumber, hwpVersion, hwpType, hwpQueueOverrunCount, hwpIdNumber, hwpScheduleActive, hwpScheduleState, hwpScheduleSuspendedSlot)

  - *pBuff* must be a pointer to an integer value.

  - *wBuffSize* should be set to 0 (default value, 4, is used) or to a value same or greater than 4 (size in bytes of an 32 bits integer).

- For a 64 bits integer value (hwpMessageFilter)

  - *pBuff* must be a pointer to an unsigned 64 bits integer.

  - The returned data in *pBuff* is to be interpreted as follow: Each bit corresponds to a Frame ID (0..63).

- For a string or byte array value (hwpName, hwpUserData)

  - *pBuff* must be a pointer to a char array. In case of returning a string, it will be null terminated.

  - *wBuffSize* should be set to the capacity of bytes supported by *pBuff*. The returned string can have a length of until LIN_MAX_NAME_LENGTH (⊡ see page 150). The returned User-Data has always a length of

   LIN_MAX_USER_DATA (⬜ see page 150).

- For a TLINVersion (⬜ see page 25) structure (hwpFirmwareVersion)

  - *pBuff* must be a pointer to a TLINVersion (⬜ see page 25) structure.

  - *wBuffSize* should be set to the capacity of bytes supported by *pBuff*. This value must be at least sizeof(TLINVersion (⬜ see page 25)).

- For a Client handle (hwpBossClient)

  - *pBuff* must be a pointer to a HLINCLIENT (⬜ see page 33) handle.

  - *wBuffSize* must be set to sizeof(HLINCLIENT (⬜ see page 33)) or greater.

- For an array of Client handles (hwpConnectedClients)

  - *pBuff* must be a pointer to an array of HLINCLIENT (⬜ see page 33).

  - *wBuffSize* should be set to the capacity of bytes supported by *pBuff*. The array buffer must at least have a length of **"Total handles + 1" multiplied by the size of HLINCLIENT (⬜ see page 33)**, where "Total handles" is the actual count of connected clients. Example (3 clients connected): the *pBuff* buffer must have a size of 4 bytes (*(3+1)*sizeof(HLINCLIENT (⬜ see page 33))*).

  - The returned data in *pBuff* is to be interpreted as follow: The first position of the array (position 0) contains the amount of client (*n*) handles returned in the buffer. The connected client handles are returned beginning at the position 1 to *n*. Example (3 clients connected): the *pBuff*[0] contain the value 3 and the positions [1], [2] and [3] the returned handles.

See Also

TLINHardwareParam (⬜ see page 37)

LIN_SetHardwareParam (⬜ see page 121)

Definitions (⬜ see page 150)

.NET Version: GetHardwareParam (⬜ see page 73)

4.4.17 **LIN_ResetHardware**

Flushes the queues of the Hardware and resets its counters.

Syntax

Pascal
```
function LIN_ResetHardware(
    hClient: HLINCLIENT;
    hHw: HLINHW
): TLINError; stdcall;
```

C++
```
TLINError __stdcall LIN_ResetHardware (
    HLINCLIENT hClient,
    HLINHW hHw
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⬜ see page 109).
hHw	Handle of the Hardware to reset. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⬜ see page 113)).

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

LIN_InitializeHardware (see page 119)

LIN_GetClientParam (see page 113)

.NET Version: ResetHardware (see page 80)

4.4.18 **LIN_ResetHardwareConfig**

Deletes the current configuration of the Hardware and sets its defaults.

Syntax

Pascal

```
function LIN_ResetHardwareConfig(
    hClient: HLINCLIENT;
    hHw: HLINHW
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_ResetHardwareConfig (
    HLINCLIENT hClient,
    HLINHW hHw
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (see page 109).
hHw	Handle of the Hardware to reset. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (see page 113)).

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

LIN_ResetHardware (see page 125)

LIN_GetClientParam (see page 113)

4.4.19 **LIN_IdentifyHardware**

Phisically identifies a LIN Hardware (a channel on a LIN Device) by blinking its associated LED.

Syntax

Pascal

```
function LIN_IdentifyHardware(
    hHw: HLINHW
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_IdentifyHardware (
    HLINHW hHw
);
```

Parameters

Parameters	Description
hHw	Handle of the Hardware to identify. (One handle from the list of available hardware, returned by the function LIN_GetAvailableHardware (⊡ see page 120)).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalHardware*

See Also

LIN_GetAvailableHardware (⊡ see page 120)

4

4.4.20 **LIN_RegisterFrameId**

Modifies the filter of a Client and, eventually, the filter of the connected Hardware.

Syntax

Pascal

```
function LIN_RegisterFrameId(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    bFromFrameId: Byte;
    bToFrameId: Byte
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_RegisterFrameId (
    HLINCLIENT hClient,
    HLINHW hHw,
    BYTE bFromFrameId,
    BYTE bToFrameId
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊡ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊡ see page 113)).
bFromFrameId	First ID of the frame range.
bToFrameId	Last ID of the frame range.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalFrameID*

Remarks

The messages with FrameID *'bFromFrameId'* to '*bToFrameId'* (including both limits) will be received. By example, calling this function with the values '*bFromFrameId*' = 1 and '*bToFrameId*' = 4, register the Frames 1,2,3 and 4.

This function expands the filter instead of replaces it. Continuing with the example above, If a client with an already configured filter (to receive the IDs 1 to 4), calls this function again with 'bFromFrameId' = 5 and 'bToFrameId' = 7, then its filter let messages from ID 1 to ID 7 to go through.

See Also

LIN_SetClientFilter (⊡ see page 115)

.NET Version: RegisterFrameId (⊡ see page 82)

4.4.21 **LIN_SetFrameEntry**

Configures a LIN Frame in a given Hardware.

Syntax

Pascal

```
function LIN_SetFrameEntry(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    var pFrameEntry: TLINFrameEntry
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_SetFrameEntry (
    HLINCLIENT hClient,
```

```
    HLINHW hHw,
    TLINFrameEntry *pFrameEntry
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊡ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊡ see page 113)).
pFrameEntry	Buffer for a TLINFrameEntry (⊡ see page 29) structure.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction: *errManagerNotLoaded, errManagerNotResponse, errMemoryAccess*

API Return: *errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalFrameID, errIllegalLength*

See Also

TLINFrameEntry (⊡ see page 29)

LIN_GetClientFilter (⊡ see page 116)

.NET Version: SetFrameEntry (⊡ see page 83)

4.4.22 **LIN_GetFrameEntry**

Gets the configuration of a LIN Frame from a given Hardware.

Syntax

Pascal

```
function LIN_GetFrameEntry(
    hHw: HLINHW;
    var pFrameEntry: TLINFrameEntry
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetFrameEntry (
    HLINHW hHw,
    TLINFrameEntry *pFrameEntry
);
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function LIN_GetAvailableHardware (⊡ see page 120)).
pFrameEntry	Buffer for a TLINFrameEntry (⊡ see page 29) structure. See **Remarks**.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes

are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue*, *errIllegalHardware*, *errIllegalFrameID*

Remarks

The member *'FrameId'* of the structure TLINFrameEntry (⊟ see page 29) given within the parameter *'pFrameEntry'* must be set to the ID of the frame, whose configuration should be returned.

See Also

TLINFrameEntry (⊟ see page 29)

LIN_SetFrameEntry (⊟ see page 128)

LIN_GetAvailableHardware (⊟ see page 120)

.NET Version: GetFrameEntry (⊟ see page 84)

4.4.23 **LIN_UpdateByteArray**

Updates the data of a LIN Frame for a given Hardware.

Syntax

Pascal

```
function LIN_UpdateByteArray(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    bFrameId: Byte;
    bIndex: Byte;
    bLen: Byte;
    pData: Pointer
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_UpdateByteArray (
    HLINCLIENT hClient,
    HLINHW hHw,
    BYTE bFrameId,
    BYTE bIndex,
    BYTE bLen,
    BYTE *pData
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊟ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊟ see page 113)).
bFrameId	ID of the Frame to be updated.
bIndex	Index where the update data starts (0..7).
bLen	Count of Data bytes to be updated.
pData	Data buffer with a size of at least '*bLen*'.

Returns

The return value is an TLINError (⊞ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalFrameID, errIllegalLength, errIllegalIndex, errIllegalRange*

See Also

LIN_SetFrameEntry (⊞ see page 128)

LIN_GetFrameEntry (⊞ see page 129)

.NET Version: UpdateByteArray (⊞ see page 85)

4.4.24 **LIN_StartKeepAlive**

Sets the Frame 'bFrameId' as Keep-Alive frame for the given Hardware and starts to send it periodically.

Syntax

Pascal

```
function LIN_StartKeepAlive(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    bFrameId: Byte;
    wPeriod: Word
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_StartKeepAlive (
    HLINCLIENT hClient,
    HLINHW hHw,
    BYTE bFrameId,
    WORD wPeriod
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊞ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊞ see page 113)).
bFrameId	ID of the Keep-Alive Frame.
wPeriod	Keep-Alive interval in milliseconds (at least 4 ms).

Returns

The return value is an TLINError (⊞ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalFrameID, errIllegalPeriod, errIllegalSchedulerState, errIllegalFrameConfiguration*

See Also

LIN_SuspendKeepAlive (☑ see page 132)

LIN_ResumeKeepAlive (☑ see page 132)

.NET Version: StartKeepAlive (☑ see page 86)

4.4.25 LIN_SuspendKeepAlive

Suspends the sending of a Keep-Alive frame in the given Hardware.

Syntax

Pascal

```
function LIN_SuspendKeepAlive(
    hClient: HLINCLIENT;
    hHw: HLINHW
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_SuspendKeepAlive (
    HLINCLIENT hClient,
    HLINHW hHw
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (☑ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (☑ see page 113)).

Returns

The return value is an TLINError (☑ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

| **DLL Interaction:** | *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess* |
| **API Return:** | *errWrongParameterValue, errIllegalClient, errIllegalHardware* |

See Also

LIN_StartKeepAlive (☑ see page 131)

LIN_ResumeKeepAlive (☑ see page 132)

.NET Version: SuspendKeepAlive (☑ see page 87)

4.4.26 LIN_ResumeKeepAlive

Resumes the sending of a Keep-Alive frame in the given Hardware.

Syntax

Pascal

```
function LIN_ResumeKeepAlive(
    hClient: HLINCLIENT;
    hHw: HLINHW
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_ResumeKeepAlive (
    HLINCLIENT hClient,
    HLINHW hHw
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊡ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊡ see page 113)).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue,* *errIllegalClient,* *errIllegalHardware,* *errIllegalSchedulerState,* *errIllegalFrameConfiguration*

See Also

LIN_StartKeepAlive (⊡ see page 131)

LIN_SuspendKeepAlive (⊡ see page 132)

.NET Version: ResumeKeepAlive (⊡ see page 88)

4.4.27 **LIN_SetSchedule**

Configures the slots of a Schedule in a given Hardware.

Syntax

Pascal

```
function LIN_SetSchedule(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    iScheduleNumber: Integer;
    pSchedule: Pointer;
    iSlotCount: Integer
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_SetSchedule (
    HLINCLIENT hClient,
```

```
    HLINHW hHw,
    int iScheduleNumber,
    TLINScheduleSlot *pSchedule,
    int iSlotCount
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊞ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊞ see page 113)).
iScheduleNumber	Schedule number. See LIN_MIN_SCHEDULE_NUMBER (⊞ see page 150) and LIN_MAX_SCHEDULE_NUMBER (⊞ see page 150)
pSchedule	Buffer for an array of TLINScheduleSlot (⊞ see page 30) structures. See **Remarks**.
iSlotCount	Count of Slots contained in the '*pSchedule*' buffer.

Returns

The return value is an TLINError (⊞ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue*, *errIllegalClient*, *errIllegalHardware*, *errIllegalScheduleNo*, *errIllegalSlotCount, errScheduleSlotPoolFull*

Remarks

The member '*Handle*' from each Schedule Slot will be updated by the hardware when this function successfully completes. This handle can be used to set up a "break point" on a schedule, using the function LIN_SetScheduleBreakPoint (⊞ see page 137).

Example:

The "main" schedule includes an event frame with a corresponding collision-resolve-schedule. Now the user can set a breakpoint e.g. on the first slot of this collision-resolving-schedule. During normal scheduling without a collision the breakpoint will never be reached. When a collision occurs within the event frame, the scheduler will branch to the resolving-schedule but will suspend itself before the first slot of the resolving-schedule is processed.

Now the user can analyse the received data and retrieve the reason for the collision.

See Also

LIN_GetSchedule (⊞ see page 135)

LIN_DeleteSchedule (⊞ see page 136)

LIN_StartSchedule (⊞ see page 138)

LIN_SuspendSchedule (⊞ see page 139)

LIN_ResumeSchedule (⊞ see page 139)

LIN_SetScheduleBreakPoint (⊞ see page 137)

.NET Version: SetSchedule (⊞ see page 88)

4.4.28 **LIN_GetSchedule**

Gets the slots of a Schedule from a given Hardware.

Syntax

Pascal

```pascal
function LIN_GetSchedule(
    hHw: HLINHW;
    iScheduleNumber: Integer;
    pScheduleBuff: Pointer;
    iMaxSlotCount: Integer;
    var pSlotCount: Integer
): TLINError; stdcall;
```

C++

```cpp
TLINError __stdcall LIN_GetSchedule (
    HLINHW  hHw,
    int iScheduleNumber,
    TLINScheduleSlot *pScheduleBuff,
    int iMaxSlotCount,
    int *pSlotCount
);
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function LIN_GetAvailableHardware (▣ see page 120)).
iScheduleNumber	Schedule number. See LIN_MIN_SCHEDULE_NUMBER (▣ see page 150) and LIN_MAX_SCHEDULE_NUMBER (▣ see page 150)
pScheduleBuff	Buffer for an array of TLINScheduleSlot (▣ see page 30) structures.
iMaxSlotCount	Maximum number of slots to be read.
pSlotCount	Real number of slots read returned in the array 'pScheduleBuff'.

Returns

The return value is an TLINError (▣ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalHardware, errIllegalScheduleNo, errIllegalSlotCount*

See Also

LIN_SetSchedule (▣ see page 133)

LIN_DeleteSchedule (▣ see page 136)

LIN_StartSchedule (▣ see page 138)

LIN_SuspendSchedule (▣ see page 139)

LIN_ResumeSchedule (▣ see page 139)

LIN_SetScheduleBreakPoint (▣ see page 137)

4.4.29 **LIN_DeleteSchedule**

Removes all slots contained by a Schedule of a given Hardware.

Syntax

Pascal

```
function LIN_DeleteSchedule(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    iScheduleNumber: Integer
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_DeleteSchedule (
    HLINCLIENT hClient,
    HLINHW hHw,
    int iScheduleNumber
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (see page 113)).
iScheduleNumber	Schedule number. See LIN_MIN_SCHEDULE_NUMBER (see page 150) and LIN_MAX_SCHEDULE_NUMBER (see page 150)

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalScheduleNo, errIllegalSchedulerState*

See Also

LIN_GetSchedule (see page 135)

LIN_SetSchedule (see page 133)

LIN_StartSchedule (see page 138)

LIN_SuspendSchedule (see page 139)

LIN_ResumeSchedule (see page 139)

LIN_SetScheduleBreakPoint (see page 137)

4.4.30 **LIN_SetScheduleBreakPoint**

Sets a 'breakpoint' on a slot from a Schedule in a given Hardware.

Syntax

Pascal

```
function LIN_SetScheduleBreakPoint(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    iBreakPointNumber: Integer;
    dwHandle: Integer
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_SetScheduleBreakPoint (
    HLINCLIENT hClient,
    HLINHW hHw,
    int iBreakPointNumber,
    DWORD dwHandle
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (🔲 see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (🔲 see page 113)).
iBreakPointNumber	Breakpoint Number (0 or 1).
dwHandle	Slot Handle. See **Remarks**.

Returns

The return value is an TLINError (🔲 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

Remarks

In order to 'delete' the breakpoint of a Schedule, call this function with the parameter '*dwHandle*' set to 0.

See Also

LIN_SetSchedule (🔲 see page 133)

LIN_GetSchedule (🔲 see page 135)

LIN_DeleteSchedule (🔲 see page 136)

LIN_StartSchedule (🔲 see page 138)

LIN_SuspendSchedule (🔲 see page 139)

LIN_ResumeSchedule (🔲 see page 139)

.NET Version: SetScheduleBreakPoint (🔲 see page 92)

4.4.31 **LIN_StartSchedule**

Activates a Schedule in a given Hardware.

Syntax

Pascal

```
function LIN_StartSchedule (
    hClient: HLINCLIENT;
    hHw: HLINHW;
    iScheduleNumber: Integer
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_StartSchedule (
    HLINCLIENT hClient,
    HLINHW hHw,
    int iScheduleNumber
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (🔲 see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (🔲 see page 113)).
iScheduleNumber	Schedule number. See LIN_MIN_SCHEDULE_NUMBER (🔲 see page 150) and LIN_MAX_SCHEDULE_NUMBER (🔲 see page 150)

Returns

The return value is an TLINError (🔲 see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalScheduleNo, errIllegalHardwareMode, errIllegalSchedule*

See Also

LIN_SetSchedule (🔲 see page 133)

LIN_GetSchedule (🔲 see page 135)

LIN_DeleteSchedule (🔲 see page 136)

LIN_SuspendSchedule (🔲 see page 139)

LIN_SetScheduleBreakPoint (🔲 see page 137)

LIN_ResumeSchedule (🔲 see page 139)

.NET Version: StartSchedule (🔲 see page 93)

4.4.32 **LIN_SuspendSchedule**

Suspends an active Schedule in a given Hardware.

Syntax

Pascal

```
function LIN_SuspendSchedule (
    hClient: HLINCLIENT;
    hHw: HLINHW
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_SuspendSchedule (
    HLINCLIENT hClient,
    HLINHW hHw
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊡ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊡ see page 113)).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware*

See Also

LIN_SetSchedule (⊡ see page 133)

LIN_GetSchedule (⊡ see page 135)

LIN_DeleteSchedule (⊡ see page 136)

LIN_StartSchedule (⊡ see page 138)

LIN_ResumeSchedule (⊡ see page 139)

LIN_SetScheduleBreakPoint (⊡ see page 137)

.NET Version: SuspendSchedule (⊡ see page 94)

4.4.33 **LIN_ResumeSchedule**

Restarts a configured Schedule in a given Hardware.

Syntax

Pascal

```
function LIN_ResumeSchedule(
    hClient: HLINCLIENT;
    hHw: HLINHW
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_ResumeSchedule (
    HLINCLIENT hClient,
    HLINHW hHw
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (◪ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (◪ see page 113)).

Returns

The return value is an TLINError (◪ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*

API Return:	*errWrongParameterValue,* *errIllegalClient,* *errIllegalHardware,* *errIllegalSchedule,* *errIllegalHardwareMode, errIllegalSchedulerState*

See Also

LIN_SetSchedule (◪ see page 133)

LIN_GetSchedule (◪ see page 135)

LIN_DeleteSchedule (◪ see page 136)

LIN_StartSchedule (◪ see page 138)

LIN_SuspendSchedule (◪ see page 139)

LIN_SetScheduleBreakPoint (◪ see page 137)

.NET Version: ResumeSchedule (◪ see page 95)

4.4.34 **LIN_XmtWakeUp**

Sends a wake-up message impulse (single data byte F0h) to the given hardware.

Syntax

Pascal

```
function LIN_XmtWakeUp(
    hClient: HLINCLIENT;
    hHw: HLINHW
```

```
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_XmtWakeUp (
    HLINCLIENT hClient,
    HLINHW hHw
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊠ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊠ see page 113)).

Returns

The return value is an TLINError (⊠ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue*, *errIllegalClient*, *errIllegalHardware*

Remarks

To know if a Hardware is in sleep-mode, use the function LIN_GetStatus (⊠ see page 142) and check the member '*State (⊠ see page 44)*' of the structure (⊠ see page 31) returned.

See Also

LIN_GetStatus (⊠ see page 142)

.NET Version: XmtWakeUp (⊠ see page 96)

4.4.35 **LIN_StartAutoBaud**

Starts a process to detect the Baud rate of the LIN bus that is connected to the indicated Hardware.

Syntax

Pascal

```
function LIN_StartAutoBaud(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    wTimeOut: Word
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_StartAutoBaud (
    HLINCLIENT hClient,
    HLINHW hHw,
    WORD wTimeOut
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (⊡ see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (⊡ see page 113)).
wTimeOut	Auto-baudrate timeout in milliseconds.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalHardwareState, errIllegalPeriod*

Remarks

The LIN Hardware must be not initialized in order to do an Auto-Baudrate procedure. To check if the hardware is initialized, use the function LIN_GetStatus (⊡ see page 142) and check the member '*State (⊡ see page 44)*' of the structure (⊡ see page 31) returned.

See Also

LIN_GetStatus (⊡ see page 142)

LIN_GetClientParam (⊡ see page 113)

TLINHardwareStatus (⊡ see page 31)

.NET Version: StartAutoBaud (⊡ see page 97)

4.4.36 **LIN_GetStatus**

Retrieves current status information from the given Hardware.

Syntax

Pascal

```
function LIN_GetStatus(
    hHw: HLINHW;
    var pStatusBuff: TLINHardwareStatus
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetStatus (
    HLINHW hHw,
    TLINHardwareStatus *pStatusBuff
);
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function LIN_GetAvailableHardware (⊡ see page 120)).

| pStatusBuff | Buffer for a TLINHardwareStatus (⊡ see page 31) structure. |

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalHardware*

See Also

TLINHardwareStatus (⊡ see page 31)

TLINHardwareMode (⊡ see page 44)

TLINHardwareState (⊡ see page 44)

.NET Version: GetStatus (⊡ see page 97)

4.4.37 **LIN_CalculateChecksum**

Calculates the checksum of a LIN Message.

Syntax

Pascal
```
function LIN_CalculateChecksum(
    var pMsg: TLINMsg
): TLINError; stdcall;
```

C++
```
TLINError __stdcall LIN_CalculateChecksum (
    TLINMsg *pMsg
);
```

Parameters

Parameters	Description
pMsg	Buffer for a TLINMsg (⊡ see page 26) structure.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalLength*

Remarks

The member '*Checksum*' of the TLINMsg (⊡ see page 26) structure contained in the given '*pMsg*' buffer (message buffer), will be updated with the calculated checksum, when this function successfully completes.

See Also

TLINMsg (⊡ see page 26)

.NET Version: CalculateChecksum (⊡ see page 98)

4.4.38 LIN_GetVersion

Returns a TLINVersion (⊡ see page 25) sttructure containing the PLIN-API DLL version.

Syntax

Pascal

```
function LIN_GetVersion(
    var pVerBuff: TLINVersion
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetVersion (
    TLINVersion *pVerBuff
);
```

Parameters

Parameters	Description
pVerBuffer	Buffer for a TLINVersion (⊡ see page 25) structure.

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue*

See Also

TLINVersion (⊡ see page 25)

.NET Version: GetVersion (⊡ see page 99)

4.4.39 LIN_GetVersionInfo

Returns a string containing Copyright information.

Syntax

Pascal

```
function LIN_GetVersionInfo(
    strTextBuff: PAnsiChar;
    wBuffSize: Word
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetVersionInfo (
    LPSTR  strTextBuff,
    WORD wBuffSize
);
```

Parameters

Parameters	Description
pTextBuff	Buffer for a null terminated char array.
wBuffSize	Size in bytes of the *pBuff* buffer.

Returns

The return value is an TLINError (⊿ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction: *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*

API Return: *errWrongParameterValue, errBufferInsufficient*

See Also

.NET Version: GetVersionInfo (⊿ see page 100)

4.4.40 **LIN_GetErrorText**

Returns a descriptive text of a given TLINError (⊿ see page 46) error code.

Syntax

Pascal

```
function LIN_GetErrorText (
    dwError: TLINError;
    bLanguage: Byte;
    strTextBuff: PAnsiChar;
    wBuffSize: Word
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetErrorText (
    TLINError dwError,
    BYTE bLanguage,
    LPSTR strTextBuff,
    WORD wBuffSize
);
```

Parameters

Parameters	Description
dwError	A TLINError (⊿ see page 46) Code.
bLanguage	Indicates a "Primary language ID". See **Remarks** for more information.
strTextBuff	Buffer for a null terminated char array.
wBuffSize	Size in bytes of the *strTextBuff* buffer.

Returns

The return value is an TLINError (⊿ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction: *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*

API Return: *errWrongParameterValue, errBufferInsufficient*

Remarks

The "Primary language IDs" are codes used by Windows OS from Microsoft, to identify a human language. The PLIN system currently support the following languages:

Language	Primary Language ID
Neutral (English)	00h (0)
English	09 (9)
German	07h (7)
Italian	10h (16)
Spanish	0Ah (10)

This function will fail if a code, not listed above, is used.

See Also

TLINError (⧉ see page 46)

Primary language ID

.NET Version: GetErrorText (⧉ see page 100)

4.4.41 **LIN_GetPID**

Calculates the '*Parity Frame ID'* (PID) from a Frame ID.

Syntax

Pascal

```
function LIN_GetPID(
    var pFrameId: Byte
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetPID(
    BYTE  *pFrameId
);
```

Parameters

Parameters	Description
pFrameId	Frame ID. A value between 0 and LIN_MAX_FRAME_ID (⧉ see page 150).

Returns

The return value is an TLINError (⧉ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

| **DLL Interaction:** | *errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess* |
| **API Return:** | *errWrongParameterValue, errIllegalFrameID* |

Remarks

The given '*pFrameId*' parameter will be updated with the calculated PID, when this function successfully completes.

See Also

Definitions (see page 150)

.NET Version: GetPID (see page 102)

4.4.42 LIN_GetTargetTime

Gets the time used by the LIN hardware adapter.

Syntax

Pascal

```
function LIN_GetTargetTime(
    hHw: HLINHW;
    var pTargetTime: UInt64
): TLINError; stdcall;
```

C++

```
TLINError __stdcall  LIN_GetTargetTime (
    HLINHW hHw,
    unsigned __int64  *pTargetTime
);
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function LIN_GetAvailableHardware (see page 120)).
pTargetTime	Buffer for a unsigned 64 bits integer value (Target time buffer).

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalHardware*

See Also

LIN_GetAvailableHardware (see page 120)

.NET Version: GetTargetTime (see page 102)

4.4.43 LIN_SetResponseRemap

Sets the publisher response remap. This will override the complete remap table of the hardware.

Syntax

Pascal

```
function LIN_SetResponseRemap(
    hClient: HLINCLIENT;
    hHw: HLINHW;
    pRemapTab: PBYTE
): TLINError; stdcall;
```

C++

```
TLINError __stdcall  LIN_SetResponseRemap (
    HLINCLIENT hClient,
    HLINHW hHw,
    BYTE  *pRemapTab
);
```

Parameters

Parameters	Description
hClient	Handle of the Client. This handle is returned by the function LIN_RegisterClient (see page 109).
hHw	Handle of the Hardware. (One handle from the list of connected hardware of the client above. See LIN_GetClientParam (see page 113)).
pRemapTab	Buffer for an array of 64 bytes (IDs).

Returns

The return value is an TLINError (see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalClient, errIllegalHardware, errIllegalFrameID*

Remarks

- The response remap mechanism only affects the remapping of the publisher response data in slave mode.

- ID x is remapped to ID y: when ID x is received, the frame settings where taken from ID x, but the data itself is taken from ID y.

- Setting the remap will override the old mapping. All pending responses for single shot frames will be killed and therefore be lost.

The mode of use of this feature is showed in the following example:

C/C++:

```
BYTE  RemapTab[64];

...
// Remaps each ID to itself ( default)
//
for ( int i = 0; i <= LIN_MAX_FRAME_ID; i++)
  RemapTab[i] = i;
...

...
// Remaps publisher response from ID 11 to ID 9
//
RemapTab[11] = 9;
...

...
// The client hClient Sets the Remap Table on the hardware hHw
//
LIN_SetResponseRemap(hClient, hHw, RemapTab);
...
```

See Also

Processing Event Frames (⊡ see page 21)

LIN_GetResponseRemap (⊡ see page 149)

LIN_UpdateByteArray (⊡ see page 130)

.NET Version: SetResponseRemap (⊡ see page 103)

4.4.44 LIN_GetResponseRemap

Gets the publisher remap table from a given hardware.

Syntax

Pascal

```
function LIN_GetResponseRemap(
    hHw: HLINHW;
    pRemapTab: PBYTE
): TLINError; stdcall;
```

C++

```
TLINError __stdcall  LIN_GetResponseRemap (
    HLINHW hHw,
    BYTE  *pRemapTab
);
```

Parameters

Parameters	Description
hHw	Handle of the Hardware. (One handle from the list of available hardware, returned by the function LIN_GetAvailableHardware (⊡ see page 120)).
pRemapTab	Buffer for an array of 64 bytes (IDs).

Returns

The return value is an TLINError (⊡ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errManagerNotLoaded*, *errManagerNotResponse*, *errMemoryAccess*
API Return:	*errWrongParameterValue, errIllegalHardware*

See Also

Processing Event Frames (⊡ see page 21)

LIN_SetResponseRemap (⊡ see page 147)

LIN_UpdateByteArray (⊡ see page 130)

.NET Version: GetResponseRemap (⊡ see page 104)

4.4.45 **LIN_GetSystemTime**

Gets the current system time. This time is returned by Windows as the elapsed number of microseconds since system start.

Syntax

Pascal

```
function LIN_GetTargetTime(
    var pTargetTime: UInt64
): TLINError; stdcall;
```

C++

```
TLINError __stdcall LIN_GetSystemTime(
    unsigned __int64  *pSystemTime
);
```

Parameters

Parameters	Description
pTargetTime	Buffer for a unsigned 64 bits integer value (System time buffer).

Returns

The return value is an TLINError (▨ see page 46) code. *errOK* is returned on success, otherwise one of the following codes are returned:

DLL Interaction:	*errMemoryAccess*
API Return:	*errWrongParameterValue*

See Also

4.5 **Definitions**

The PLIN API defines the following values:

Define	Value	Description
INVALID_LIN_HANDLE	0	Invalid value for all LIN handles (Client, Hardware).
LIN_HW_TYPE_USB	1	Hardware Type: LIN USB [**deprecated**].
LIN_HW_TYPE_USB_PRO	1	Hardware Type: PCAN-USB Pro LIN.
LIN_HW_TYPE_USB_PRO_FD	2	Hardware Type: PCAN-USB Pro FD LIN.
LIN_HW_TYPE_PLIN_USB	3	Hardware Type: PLIN-USB.
LIN_MAX_FRAME_ID	63	Maximum allowed Frame ID.

LIN_MAX_SCHEDULES	8	Maximum allowed Schedules per Hardware.
LIN_MIN_SCHEDULE_NUMBER	0	Minimum Schedule number.
LIN_MAX_SCHEDULE_NUMBER	7	Maximum Schedule number.
LIN_MAX_SCHEDULE_SLOTS	256	Maximum allowed Schedule slots per Hardware.
LIN_MIN_BAUDRATE	1000	Minimum LIN Baudrate.
LIN_MAX_BAUDRATE	20000	Maximum LIN Baudrate.
LIN_MAX_NAME_LENGTH	48	Maximum number of bytes for Name / ID of a Hardware or Client.
FRAME_FLAG_RESPONSE_ENABLE	1 (0x0001)	Slave Enable Publisher Response.
FRAME_FLAG_SINGLE_SHOT	2 (0x0002)	Slave Publisher Single shot.
FRAME_FLAG_IGNORE_INIT_DATA	4 (0x0004)	Ignore InitialData on set frame entry.
LOG_FLAG_DEFAULT	0 (0x0000)	Logs system exceptions / errors
LOG_FLAG_ENTRY	1 (0x0001)	Logs the entries to the PLIN-API functions
LOG_FLAG_PARAMETERS	2 (0x0002)	Logs the parameters passed to the PLIN-API functions
LOG_FLAG_LEAVE	4 (0x0004)	Logs the exits from the PLIN-API functions
LOG_FLAG_WRITE	8 (0x0008)	Logs the LIN messages passed to the LIN_Write (⊿ see page 118) function
LOG_FLAG_READ	16 (0x0010)	Logs the LIN messages received within the LIN_Read (⊿ see page 117) function
LOG_FLAG_ALL	65535 (0xFFFF)	Logs all possible information within the PLIN-API functions
LIN_MAX_USER_DATA	24	Maximum number of bytes that a user can read/write on a Hardware.
LIN_MIN_BREAK_LENGTH	13	Minimum number of bits that can be used as break field in a LIN frame.
LIN_MAX_BREAK_LENGTH	32	Maximum number of bits that can be used as break field in a LIN frame.
LIN_MAX_RCV_QUEUE_COUNT	65535	Maximum number of LIN frames that can be stored in the reception queue of a client

Remarks

The version for Microsoft .NET for this values are described in Constants (⊿ see page 106).

4

5 Additional Information

In the following topics you will find additional information regarding the PLIN environment and the API.

In this Chapter

Topics	Description
Log File Generation (⤢ see page 152)	This section contains information about logging debug data within PLIN-API.

5.1 Log File Generation

In order to support debugging of problems, that can arise during LIN communication, PLIN-API can generate a log file, containing a protocol of all API function calls. There are two different ways to configure and activate this logging functionality:

- Using the .
- Using the .

Log configuration using API

When the logging functionality is managed using the API, it is possible to configure and activate the logging process. The file is created near to the caller, this is, in the same folder where the application using the PLIN-API is located.

In order to configure the Log functionality, PLIN-API provides 2 parameters that can be configured with the function LIN_SetClientParam (⤢ see page 112) (**class-method**: SetClientParam (⤢ see page 57)). These parameters are:

- clpLogStatus, to enable/disable logging.
- clpLogConfiguration, to configure the content of the log file.

Example:

Within the following example, the PLIN-API is configured to log all possible information (LOG_FLAG_ALL = 0xFFFF).

C++:

```cpp
TLINError result;
DWORD iBuffer;
char strBuffer[MAX_PATH];

// Configures the data in the log file.
//
iBuffer = LOG_FLAG_ALL;
result = LIN_SetClientParam(0, clpLogConfiguration, iBuffer);
if(result != errOK)
{
    LIN_GetErrorText(result, 0x09, strBuffer, MAX_PATH-1);
    MessageBox(NULL, strBuffer, "Error configuring Log:", MB_OK);
}
else
{
    // Activates the log generation
    //
    iBuffer = 1; // zero denotes deactivation, non-zero activation
    result = LIN_SetClientParam(0, clpLogStatus, iBuffer);
    if(result != errOK)
    {
```

```
        LIN_GetErrorText(result, 0x09, strBuffer, MAX_PATH-1);
        MessageBox(NULL, strBuffer, "Error activating Log:", MB_OK);
    }
}
```

Log configuration using Windows Registry

When the logging functionality is managed over the registry, it is possible to configure and activate logging, and additionally, to set the place where the file should be stored.

In order to enable the log file generation, the following registry key must be created:

HKEY_CURRENT_USER\SOFTWARE\PEAK-System\PlinApi\Log

The existence of this key is analogous to use the function LIN_SetClientParam (🗗 see page 112) (**class-method**: SetClientParam (🗗 see page 57)) to set the parameter **clpLogStatus** to "on" (nonzero). If this key is not present, then no log file is generated.

Configuration:

If no further configuration is made, then the default values for **clpLogConfiguration** is used, which is logging only exceptions, and the file is created in the same location of the application caller. In order to configure the location and content of the log file, two registry values are used:

Flags: This is a DWORD value, that represents a logical **OR** operation between the values LOG_FLAG_*** (see Definitions (🗗 see page 150) / Constants (🗗 see page 106)) that are wanted to be included within the logging data. The value LOG_FLAG_ALL causes logging all possible information.

Path: This is a String value, that represents the path to a folder in the computer, where the log file will be created.

Example:

Within the following example, the PLIN-API is configured to log function entries (LOG_FLAG_ENTRY = 1), function parameters (LOG_FLAG_PARAMETERS = 2), and function outs (LOG_FLAG_LEAVE = 4), as well as to store the log file on the desktop of an user called "admin".

```
[HKEY_CURRENT_USER\SOFTWARE\PEAK-System\PlinAPi\Log]
"Flags"=dword:00000007
"Path"="C:\\Users\\admin\\desktop"
```

Remarks:

The registry key should be deleted (or renamed) after a debug session is done. If the key is leaved, all PLIN-API applications running under the same user account will remain writing data to their log files, generating in this way huge text files that consume hard-disk space unnecessarily.

5

Index