Ixxat®
BY HMS NETWORKS

CAN@net NT

C-API ixcan

USER MANUAL

4.02.0332.20003 1.1 en-US ENGLISH

HMS Connecting Devices™

Important User Information

Disclaimer

The information in this document is for informational purposes only. Please inform HMS Networks of any inaccuracies or omissions found in this document. HMS Networks disclaims any responsibility or liability for any errors that may appear in this document.

HMS Networks reserves the right to modify its products in line with its policy of continuous product development. The information in this document shall therefore not be construed as a commitment on the part of HMS Networks and is subject to change without notice. HMS Networks makes no commitment to update or keep current the information in this document.

The data, examples and illustrations found in this document are included for illustrative purposes and are only intended to help improve understanding of the functionality and handling of the product. In view of the wide range of possible applications of the product, and because of the many variables and requirements associated with any particular implementation, HMS Networks cannot assume responsibility or liability for actual use based on the data, examples or illustrations included in this document nor for any damages incurred during installation of the product. Those responsible for the use of the product must acquire sufficient knowledge in order to ensure that the product is used correctly in their specific application and that the application meets all performance and safety requirements including any applicable laws, regulations, codes and standards. Further, HMS Networks will under no circumstances assume liability or responsibility for any problems that may arise as a result from the use of undocumented features or functional side effects found outside the documented scope of the product. The effects caused by any direct or indirect use of such aspects of the product are undefined and may include e.g. compatibility issues and stability issues.

Table of Contents

Page

1 User Guide

Please read the manual carefully. Make sure you fully understand the manual before using the product.

1.1 Target Audience

This manual addresses trained personnel who are familiar with ASCII, CAN, CAN FD, the applicable national standards and programming in C. This manual answers common questions concerning the development of applications according to the ASCII Gateway Mode of the CAN@net NT.

1.2 Related Documents

Document	Author
Software Design Guide *CAN@net NT 100/200/420 Generic Protocol for Gateway Mode*	HMS
User Manual *CAN-Gateway Configurator*	HMS
User Manual *CAN@net NT 100/200/420*	HMS

1.3 Document History

Version	Date	Description
1.0	April 2020	First version
1.1	June 2021	Added UDP, command ixcan_send_multi

1.4 Trademark Information

Ixxat® is a registered trademark of HMS Industrial Networks. All other trademarks mentioned in this document are the property of their respective holders.

1.5 Conventions

Instructions and results are structured as follows:

- ► instruction 1
- ► instruction 2
  - → result 1
  - → result 2

Lists are structured as follows:

- item 1
- item 2

Bold typeface indicates interactive parts such as connectors and switches on the hardware, or menus and buttons in a graphical user interface.

```
This font is used to indicate program code and other
kinds of data input/output such as configuration scripts.
```

This is a cross-reference within this document: *Conventions, p. 4*

This is an external link (URL): www.hms-networks.com

This is additional information which may facilitate installation and/or operation.

This instruction must be followed to avoid a risk of reduced functionality and/or damage to the equipment, or to avoid a network security risk.

2 Description of the ixcan API for C

HMS recommends to read the User Manual CAN@net NT 100/200/420 Generic Protocol 2.1 for Gateway Mode V6, because the ixcan API for C is using the Generic ASCII protocol.

The C-API ixcan converts the API calls into corresponding ASCII commands according to the ASCII Gateway Mode of the CAN@net NT and the received ASCII commands to corresponding API calls. The assembled commands are sent via the TCP/UDP socket to the device.

2.1 Processing of Messages

The communication from and to the CAN@net NT is handled as TCP stream or as UDP packets.

2.1.1 TCP Stream Handling

When reading data from the TCP socket the host might receive or read only parts of a CAN message or a command response. Since only complete messages can be processed, the host must wait for the remaining data and then process the complete CAN message or command response. Wether a message is complete or not, the host identifies based on the end of line termination (dependent on the operating system and the respective settings in the CAN-Gateway Configurator with `\r\n`, `\r`, or `\n`).

To guarantee that messages are completely transferred (until end of line termination) to the CAN@net NT, the unit *ixcom.c* stores the bytes of a message that are not sent and retries to send these remaining bytes within the next call of the function `ixcom_send()`, before transmitting new data. In this case the function returns `IXERR_E_SOCKET_SEND_BUFFER_FULL`. Since there is no additional CAN transmit queue implemented on top of the OS socket queue, the handling is required in case the OS socket only stores parts of the message due to a full transmit queue. A full transmit queue is typically indicated by the OS socket function `send` returning less transmitted bytes than requested to be sent. In case the buffered bytes still cannot be sent completely, the new data is rejected and the remaining bytes of the message are tried to be transmitted with the next call of `ixcom_send()` again. The transmitting of the data is retried until the complete message is transmitted successfully.

2.1.2 UDP Packet Handling

When reading data from the UDP socket the host receives one or more complete messages. Several messages are separated by line termination (dependent on the operating system and the respective settings in the CAN-Gateway Configurator with `\r\n`, `\r`, or `\n`).

The unit *ixcom.c* uses a buffer of 1460 bytes to send one UDP packet per function call to function `ixcom_send()` or `ixcom_send_multi()`. If the application shall send more than one CAN message at once, it is preferred to use the function `ixcom_send_multi()` to get more performance.

2.1.3 Command/Response Handling

As the ASCII interface only handles one command after the other, the command execution is blocked until the response is received. In case of multi threading to the API, the command execution is protected by critical sections and mutexes.

The ixcan API is prepared to be thread safe by using macros that are defined in *ixos.h*. To enabled or disable the macros see `IXOS_cfg_SUPPORT_MULTITHREADING` in *ixos.h*.

2.1.4 Receive and Transmit Handling

On reception of the character stream via TCP/UDP socket the data is copied to a socket receive queue inside *ixcom.c*. From within this receive queue the stream is parsed for a complete command by checking for the end of line termination. The command is then parsed and dispatched. In case of received CAN messages, for example, these messages are converted into binary CAN messages and stored in the CAN receive queue. The CAN receive queue can be read by the application with the function `ixcan_receive()`. On reception of a command response (success or failure) the message is also parsed and handled based on the command/ response.

The transmit and receive handling is done inside `read_comchannel_and_dispatch()` inside *ixcan.c*.

`read_comchannel_and_dispatch()` is triggered inside `ixscheduler()`. `ixscheduler()` is triggered inside the ixcan API functions `ixcan_can_status()`, `ixcan_receive()`, `ixcan_send()`, and functions that are sending a command and waiting for a command response from the CAN@net NT.

2.1.5 CAN Status

The CAN status is polled cyclically (none blocking) from within `ixscheduler()`. The requests for the CAN status of all available CAN ports are assembled within one request. The responses are sent one by one by the CAN@net NT. The responses are parsed and stored inside the CAN status buffers on ixcan side. Calling `ixcan_can_status()` returns the buffered status.

To keep track of the correct number of free entries in the transmit CAN queue of the CAN@net NT, the number of triggered CAN messages since the last request of the CAN status is counted and used as correction value when receiving the CAN status response. Note that in remote access, when the Bridge and the ASCII interface are sharing the CAN transmit queue, that the free counter might be changed meanwhile due to the running bridge.

2.2 Limitations of the ixcan API

Auto baud rate detection and cyclic transmission are not supported by the ixcan API. To use auto baud rate detection and cyclic transmission use the generic protocol in ASCII Gateway Mode (see User Manual *CAN@net NT 100/200/420 Generic Protocol for Gateway Mode V6*).

2.3 Sources of the ixcan API and Ports to Other Targets

With the installation of the CAN-Gateway Configurator the sources of the ixcan API are also installed (check within support area on www.ixxat.com).

The files are located in OS specific folders. By default Windows 10 installs the sources and examples in *C:\Users\Public\Documents\HMS\Ixxat CAN-Gateway Configurator\Examples\C-API*.

The core sources are written in plain C.

The operating, compiler and IDE specifics are encapsulated in the following files:

- *ixos.h*
- *ixos.c* (only required for certain targets)
- *ixsocket.c* (socket abstraction unit, abstracts the OS specific socket API calls)

Only the required macros for the ixcan API for C are implemented to keep the files small and easy to port. For example no abstraction to create and destroy tasks is implemented here.

3 Using the C API ixcan with the CAN@net NT

HMS recommends to read the User Manual CAN@net NT 100/200/420 Generic Protocol 2.1 for Gateway Mode V6, because the ixcan API for C is using the Generic ASCII protocol.

The use of the C API ixcan is only possible in ASCII Gateway and Bridge operational modes.

The C API ixcan can access the CAN@net NT in two ways:

- exclusive access in ASCII Gateway operational mode
- remote access in Bridge operational modes (shared)

3.1 Exclusive Access

For exclusive access the operational mode ASCII Gateway must be selected. In the ASCII Gateway mode the CAN control and the settings of CAN filters is the responsibility of the application that uses the ixcan API for C.

3.1.1 Configuring the ASCII Gateway Mode

- Connect the device to the host computer and to power supply (for more information see User Manual of the device in use).
- Make sure that the latest CAN-Gateway Configurator is installed (check within support area on www.ixxat.com).
- Start the CAN-Gateway Configurator and connect the device in use (for more information see User Manual *CAN-Gateway Configurator*).
- Select operational mode **ASCII Gateway mode**.
- In the configuration tree select **Interface**.
- If checkbox **Only for specified device** is enabled, enter the serial number of the device to which the configuration can be written.
- Configure the protocol line ending (**4**).
- Define the IP port (**5**).
- In the toolbar open menu **Target** and select **Write configuration to target** to write the device configuration to the connected CAN NT device.
- Create the application with ixcan API for C.

3.1.2 Typical Sequence

- Configuration of ASCII Gateway Mode in the CAN-Gateway Configurator
- `ixcan_open()`
- `ixcan_identify()`
- `ixcan_stop_can()` (CAN controller must be stopped before initializing)
- `ixcan_initialize()`
- `ixcan_add_filter()` (all filters are closed by default)
- `ixcan_start_can()`
- `ixcan_can_status()` (cyclic call)
- `ixcan_receive()` (cyclic call)
- `ixcan_send()`
- `ixcan_additional_error_info()`
- `ixcan_close()`

3.1.3 Configuring Message Filters

With the initialization the CAN controller loses its filter settings and all messages are rejected. Configure the filter after initialization.

If a message passes several filters, this message is received several times.

All filters are closed by default and therefore no CAN messages are received. Defining message filters with the ixcan API is only possible with exclusive access when the ixcan API has CAN control. Received messages that match the mask/value filter are passed through. Other messages are discarded.

- ► Add a filter with function `ixcan_add_filter()`.
- ► In parameter *mode* define the format (standard or extended).
- ► Define the messages that pass the filter with parameters *value* and *mask*.

The mask value specifies the bit-position of the identifier, which must be checked (1 means “to be checked”).

Binary representation of mask:

- binary positions with value 1 are relevant for the filter
- binary positions with value 0 are not relevant for the filter

Binary representation of identifier:

- Defines the values for the positions that are marked as relevant (1) in mask.
- Values in positions that are marked as not relevant (0) in mask are ignored.

See *Examples for Mask/Value Filters, p. 9*.

Deleting Message Filters

It is possible to delete all filter entries of a CAN port. Deleting individual entries is not possible.

- Make sure, that the CAN controller is in state *stopped*.
- Call `ixcan_clr_filter` to delete all entries for 11 bit and 29 bit identifiers.
- In parameter *can_port* define the CAN port number.

Examples for Mask/Value Filters

Example 11 Bit Identifier

	hex	bin
Value	0x100	0001:0000:0000
Mask	0x700	0111:0000:0000
Result	0x1XX	0001:XXXX:XXXX
	Any identifier between 0x100 and 0x1FF passes the filter, as only the first 3 bits of the mask are marked as relevant.	

Example 29 Bit Identifier

	hex	bin
Value	0x10003344	0001:0000:0000:0000:0011:0011:0100:0100
Mask	0x1F00FFFF	0001:1111:0000:0000:1111:1111:1111:1111
Result	0x10003344	0001:0000:XXXX:XXXX:0011:0011:0100:0100
	All identifiers with 0x10xx3344 (positions xx can be any number) pass the filter.	

3.1.4 Connection Monitoring

> **!** Connection monitoring with PING REQUEST is only possible with exclusive access in ASCII Gateway Mode. If the ixcan API only has remote access the ixcan API does not request the connection monitoring, even when enabled, to prevent the CAN@net NT from performing a reset of the Bridge when the connection to the host is lost or closed.

The CAN@net NT supports connection monitoring to check if the connection between the host and CAN@net NT is up and running. The ixcan API cyclically transmits the command PING REQUEST that is triggered by `ixscheduler()`. The CAN@net NT answers to a PING REQUEST with a PING RESPONSE. The first PING REQUEST activates the connection monitoring. If no further PING REQUEST is received in the defined time (timeout), the CAN@net NT is disconnected and CAN and TCP/UDP server are reset.

- To enable and disable the connection monitoring see IXCAN_PING_REQ_ENABLE in *ixcan_cfg.h*.
  - 0 = disabled (e.g. during debugging to prevent the CAN@net from performing a reset on timeout)
  - 1 = enabled
  - → If enabled, the first reception of the command PING REQUEST starts the monitoring mechanism on the CAN@net NT.
  - → When a timeout occurs the CAN@net NT is disconnected and CAN and TCP/UDP server are reset.
  - → Defined value for timeout is 3 seconds.

3.2 Remote Access

If Remote access is **enabled**, a device that is used in Bridge mode can be accessed in ASCII Gateway mode simultaneously. The CAN controller must be configured and started by the Bridge mode configuration in the CAN-Gateway Configurator.

The CAN controller is controlled via the CAN@net NT Bridge configuration and all ASCII commands related to the control are blocked, this means the CAN controller cannot be stopped or modified via the C-API ixcan. Cyclic messages cannot be defined via ASCII commands in remote access. CAN messages can be sent and received via the ASCII protocol. To receive CAN messages on the host side via ASCII commands, the messages must be added in the Mapping table of the Bridge configuration. The ASCII device commands can also be used in Remote access.

3.2.1 Configuring Remote Access

- Connect the device to the host computer and to power supply (for more information see User Manual of the device in use).
- Make sure that the latest CAN-Gateway Configurator is installed (check within support area on www.ixxat.com).
- Start the CAN-Gateway Configurator and connect the device in use (for more information see User Manual *CAN-Gateway Configurator*).
- Select the desired Bridge operational mode.
- In the configuration tree select **General** and enable the Remote access in the drop-down list **Remote access**.
- In the configuration tree select **Remote Access** and configure the protocol line ending and the IP port if needed.

To access several devices via remote access in a CAN-Ethernet-CAN bridge:

  - Enable and configure the Remote Access for the Master and each Slave individually.
  - Define the IP port to establish the remote connections and the protocol line ending for each device individually.

- In the configuration tree select **Mapping Table** and add the messages to be received via ASCII interface/C-API (for more information see User Manual *CAN-Gateway Configurator*).

> **!** To receive CAN messages on the host side via ASCII, the messages must be added in the Mapping table of the Bridge configuration (CAN filter).

- In the toolbar open menu **Target** and select **Write configuration to target** to write the device configuration to the connected CAN NT device.
- In CAN-Ethernet-CAN bridges write the configuration to each device.

3.2.2 Restrictions in Remote Access

In Remote access the CAN transmit queue of the CAN NT device is used by the Bridge and by the ASCII interface. Therefore the reported free entries in the CAN transmit queue may have changed since the last status was read from the CAN device.

> **!** The CAN receive queue inside the API is not cleared when starting or stopping CAN. To flush the CAN receive queue, the queue must be read with function `ixcan_receive()`.

3.2.3 Typical Sequence

- Configurations in the CAN-Gateway Configurator:
  - configuring the Bridge operational mode
  - enabling Remote access
  - configuring bus off recovery mode (e.g. using Action Rules)
  - adding messages to be received via ASCII interface to the Mapping table (CAN filter)
- `ixcan_open()`
- `ixcan_identify()`
- `ixcan_can_status()` (cyclic call)
- `ixcan_receive()` (to keep the internal scheduler running either `ixcan_can_status()` or `ixcan_receive()` must be called cyclically)
- `ixcan_send()`
- `ixcan_additional_error_info()`
- `ixcan_close()`

4 API Functions

4.1 ixcan API for C

4.1.1 ixcan_get_api_version

Returns the version number of the ixcan API.

```
void ixcan_get_api_version(ixcan_api_version_t *p_api_version);
```

Parameter

Parameter	Dir.	Description
p_api_version	[out]	Pointer to the structure to store the version information

Return Value

No return values.

4.1.2 ixcan_open

Opens the communication interface and initializes the channel instance data.

```
void* ixcan_open(uint16_t protocol,
                 const char *p_ip_address,
                 uint16_t ip_port,
                 int *p_error);
```

Parameter

Parameter	Dir.	Description
protocol	[in]	Select the protocol to be used to connect to the CAN device, possible values: `IXCOM_PROTOCOL_ASCII_TCP_IP`: ASCII interface via TCP `IXCOM_PROTOCOL_ASCII_UDP_IP`: ASCII interface via UDP
p_ip_address	[in]	Pointer to string that contains the IP address of the CAN@net NT (format similar to “192.168.0.5”)
ip_port	[in]	IP port (e.g. 19228)
p_error	[out]	Pointer to store error return value (only valid when return value of function is `NULL`). For more information about error value see *List of Error Codes, p. 24*.

Return Value

Return value	Description
`!=NULL`	Pointer to channel instance data. This pointer is then used by other function calls to access the CAN@net NT.
`NULL`	Error occurred (see parameter *p_error* for details)

4.1.3 ixcan_close

Closes the communication interface.

```
int ixcan_close(void *p_channel);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data

Return Value

Return value	Description
`IXERR_S_OK`	No error occurred
`<IXERR_S_OK`	Error occurred (see*List of Error Codes, p. 24* for more information)

Remark

The function only closes the communication interface but does not change the state of the CAN controller. The application has to call for example `ixcan_stop_can()` in advance (`ixcan_stop_can()` is only allowed when ixcan API has CAN control).

In case monitoring with PING request is enabled the CAN device detects a monitoring timeout and performs a reset automatically (only when ixcan API has CAN control).

4.1.4 ixcan_identify

Gets information about the connected Ixxat CAN device.

```
int ixcan_identify(void *p_channel,
                   ixcan_identification_t *p_identification);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
p_identification	[out]	Pointer to the identification structure

Return Value

Return value	Description
`IXERR_S_OK`	No error occurred
`<IXERR_S_OK`	Error occurred (see*List of Error Codes, p. 24* for more information)

4.1.5 ixcan_initialize_can

Initializes the CAN controller. Calling the function is only allowed when ixcan API has CAN control.

```
int ixcan_initialize_can(void *p_channel,
                         uint16_t can_port,
                         uint8_t mode,
                         uint16_t bitrateA,
                         uint16_t bitrateD);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
can_port	[in]	CAN port number (1 to `IXCAN_MAX_FIELD_BUS_INTERFACES`)
mode	[in]	CAN bus mode, bit encoded, possible values: `IXCAN_CANINIT_MODE_STANDARD`: Tx active on CAN bus `IXCAN_CANINIT_MODE_LISTEN`: Listen only on CAN bus `IXCAN_CANINIT_MODE_CAN`: Classic CAN mode `IXCAN_CANINIT_MODE_CANFD`: CAN FD mode (only CAN FD) `IXCAN_CANINIT_MODE_ISO`: ISO mode (only CAN FD) `IXCAN_CANINIT_MODE_NONISO`: non ISO mode (only CAN FD)
bitrateA	[in]	Classic CAN: Bit rate value in KBaud, valid values: 5, 10, 20, 50, 100, 125, 250, 500, 800, 1000 CAN FD: Bit rate value in KBaud for arbitration phase, valid values: 5, 10, 20, 50, 100, 125, 250, 500, 800, 1000
bitrateD	[in]	Only CAN FD: Bit rate value in KBaud for data phase, valid values: 500, 1000, 2000, 4000, 5000, 6667, 8000, 10000

Return Value

Return value	Description
`IXERR_S_OK`	No error occurred
`<IXERR_S_OK`	Error occurred (see*List of Error Codes, p. 24* for more information)

4.1.6 ixcan_initialize_can_custom

Initializes the CAN controller with user defined register values. Calling the function is only allowed when ixcan API has CAN control.

```
int ixcan_initialize_can_custom
              (void *p_channel,
               uint16_t can_port,
               uint8_t mode,
               const ixcan_can_timing_register_t *p_can_timing_regA,
               const ixcan_can_timing_register_t *p_can_timing_regD);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
can_port	[in]	CAN port number (1 to IXCAN_MAX_FIELD_BUS_INTERFACES)
mode	[in]	CAN bus mode, bit encoded, possible values: IXCAN_CANINIT_MODE_STANDARD: Tx active on CAN bus IXCAN_CANINIT_MODE_LISTEN: Listen only on CAN bus IXCAN_CANINIT_MODE_CAN: Classic CAN mode IXCAN_CANINIT_MODE_CANFD: CAN FD mode (only CAN FD) IXCAN_CANINIT_MODE_ISO: ISO mode (only CAN FD) IXCAN_CANINIT_MODE_NONISO: non ISO mode (only CAN FD) IXCAN_CANINIT_MODE_CALC_TDO: TDO is calculated by CAN device (only CAN FD) IXCAN_CANINIT_MODE_SET_TDO: TDO is set in parameter p_can_timing_regD in *tdo*(only CAN FD)
p_can_timing_regA	[in]	Classic CAN: Bit rate value *register encoded* CAN FD: Bit rate value *register encoded* for arbitration phase *brp*: baud rate prescaler *sjw*: synchronization jump width *tseg1*: time segment 1 *tseg2*: time segment 2 *tdo*: not used for arbitration phase
p_can_timing_regD	[in]	Only CAN FD: Bit rate value *register encoded* for data phase *brp*: baud rate prescaler *sjw*: synchronization jump width *tseg1*: time segment 1 *tseg2*: time segment 2 *tdo*: transceiver delay offset. Value is only used if IXCAN_CANINIT_MODE_SET_TDO is set in parameter *mode*. If not the tdo value is calculated by the CAN device (tdo = (tseg1+1)*brp).

Return Value

Return value	Description
IXERR_S_OK	No error occurred
<IXERR_S_OK	Error occurred (see*List of Error Codes, p. 24* for more information)

Remark

Check in the CAN-Gateway configurator if the values result in a usable baud rate. With the integrated calculator all necessary register values for a desired baud rate can be calculated.

4.1.7 ixcan_add_filter

Adds an 11 bit or 29 bit message filter. By default all filters are closed. To add a filter, the CAN controller has to be in state *stopped*. Calling the function is only allowed when ixcan API has CAN control.

```
int ixcan_add_filter(void *p_channel,
                     uint16_t can_port,
                     uint8_t mode,
                     uint32_t value,
                     uint32_t mask);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
can_port	[in]	CAN port number (1 to IXCAN_MAX_FIELD_BUS_INTERFACES)
mode	[in]	Define the CAN format: IXCAN_CAN_FORMAT_STD_FRAME: Standard CAN frame format IXCAN_CAN_FORMAT_EXT_FRAME: Extended CAN frame format
value	[in]	Value for identifier to match
mask	[in]	Value of mask

Return Value

Return value	Description
IXERR_S_OK	No error occurred
<IXERR_S_OK	Error occurred (see*List of Error Codes, p. 24* for more information)

Remark

For examples and more information see*Configuring Message Filters, p. 8*.

4.1.8 ixcan_clr_filter

Deletes all filter entries for 11 bit and 29 bit identifiers. CAN controller has to be in state *stopped*. Calling the function is only allowed when ixcan API has CAN control.

```
int ixcan_clr_filter(void *p_channel,
                     uint16_t can_port);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
can_port	[in]	CAN port number (1 to IXCAN_MAX_FIELD_BUS_INTERFACES)

Return Value

Return value	Description
IXERR_S_OK	No error occurred
<IXERR_S_OK	Error occurred (see*List of Error Codes, p. 24* for more information)

4.1.9 ixcan_start_can

Start the CAN controller. Calling the function is only allowed when ixcan API has CAN control.

```
int ixcan_start_can(void *p_channel,
                    uint16_t can_port);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
can_port	[in]	CAN port number (1 to IXCAN_MAX_FIELD_BUS_INTERFACES)

Return Value

Return value	Description
IXERR_S_OK	No error occurred
<IXERR_S_OK	Error occurred (see*List of Error Codes, p. 24* for more information)

4.1.10 ixcan_stop_can

Stops the CAN controller. Calling the function is only allowed when ixcan API has CAN control.

```
int ixcan_stop_can(void *p_channel,
                   uint16_t can_port);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
can_port	[in]	CAN port number (1 to IXCAN_MAX_FIELD_BUS_INTERFACES)

Return Value

Return value	Description
IXERR_S_OK	No error occurred
<IXERR_S_OK	Error occurred (see*List of Error Codes, p. 24* for more information)

Remark

Only stop the CAN controller when the IXCAN_CAN_STATUS_TXPENDING flag is not set, to guarantee that all outstanding transmit requests are sent.

4.1.11 ixcan_can_status

Gets CAN status information.

```
int ixcan_can_status(void *p_channel,
                     uint16_t can_port,
                     uint16_t *p_can_sts);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
can_port	[in]	CAN port number (1 to IXCAN_MAX_FIELD_BUS_INTERFACES)
p_can_sts	[out]	Pointer to stored CAN status information. The bit coded status flags can be combined. IXCAN_CAN_STATUS_INIT: State *init* (*stopped*) when set, otherwise state *running* IXCAN_CAN_STATUS_TXPENDING: transmit pending IXCAN_CAN_STATUS_OVERRUN: data overrun (data loss between CAN controller and API) IXCAN_CAN_STATUS_ERRWARNING: error warning IXCAN_CAN_STATUS_BUSOFF: state *bus off* IXCAN_CAN_STATUS_INVALID: state and free entries of queue are invalid (status data from device not yet available).

Return Value

Return value	Description
IXERR_S_OK	No error occurred
<IXERR_S_OK	Error occurred (see*List of Error Codes, p. 24* for more information)

Remark

When device is working in shared mode and the CAN control is on the CAN device side, a bus-off might not be reported when the CAN device has already restarted the CAN controller before the status is read by the application with `ixcan_can_status()`.

In case of an overrun of the CAN message receive queue inside the API, the overrun bit inside `ixcan_can_sts_t->sts (IXCAN_CAN_STATUS_RXOVERRUN)` is set, as done when the CAN device is reporting a data overrun. Data overrun is cleared when read.

4.1.12 ixcan_receive

Receives a single CAN message.

```
int ixcan_receive(void *p_channel,
                  uint16_t can_port,
                  ixcan_can_msg_t *can_msg);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
can_port	[in]	CAN port number (1 to IXCAN_MAX_FIELD_BUS_INTERFACES)
can_msg	[out]	Pointer to the received CAN message

Return Value

Return value	Description
IXERR_S_OK	No message available in the receive queue.
IXERR_S_MSG_RECEIVED	Message received
<IXERR_S_OK	Error occurred (see*List of Error Codes, p. 24* for more information)

Remark

To make sure, that all received messages are read, repeat the function call as long as `IXERR_S_MSG_RECEIVED` is returned.

4.1.13 ixcan_send

Requests to transmit a single CAN message.

```
int ixcan_send(void *p_channel,
               uint16_t can_port,
               const ixcan_can_msg_t *p_can_msg);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
can_port	[in]	CAN port number (1 to `IXCAN_MAX_FIELD_BUS_INTERFACES`)
p_can_msg	[in]	Pointer to a CAN message to be transmitted

Return Value

Return value	Description
`IXERR_S_OK`	No error occurred
`IXCAN_E_CAN_TX_FIFO_FULL`	No space left in transmit queue
`<IXERR_S_OK`	Error occurred (see*List of Error Codes, p. 24* for more information)

Remark

For CAN FD message `p_can_msg->num_bytes` has to match the granularity of CAN FD Data Length Code (DLC) to avoid potentially uninitialized payload data. Valid ranges/values are: <=8, 12, 16, 20, 24, 32, 48, 64.

4.1.14 ixcan_send_multi

Requests to send multiple CAN messages. Either all provided messages or no messages are sent.

The maximum number of classic CAN messages to be sent is 30. For CAN FD messages the maximum number of messages is dependent on the CAN message payload and has to be checked before sending.

- maximum TCP/UDP packet size is 1460 characters
- CAN message size in characters is calculated as follows: 20 + <num data bytes> * 3 (e.g. a CAN FD message with 64 data bytes has a size of 212 characters)

`ixcan_send_multi` can be used to increase data throughput for UDP communication.

```
int ixcan_send_multi(void *p_channel,
                     uint16_t can_port,
                     const ixcan_can_msg_t *p_can_msg[]
                     uint16_t num_msg);
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
can_port	[in]	CAN port number (1 to `IXCAN_MAX_FIELD_BUS_INTERFACES`)
p_can_msg	[in]	Pointer to an array of CAN messages to be transmitted
num_msg	[in]	Number of provided CAN messages to be transmitted

Remark

For CAN FD message `p_can_msg->num_bytes` has to match the granularity of CAN FD Data Length Code (DLC) to avoid potentially uninitialized payload data. Valid ranges/values are: <=8, 12, 16, 20, 24, 32, 48, 64.

Return Value

Return value	Description
`IXERR_S_OK`	No error occurred
`IXCAN_E_CAN_TX_FIFO_FULL`	No space left in transmit queue
`<IXERR_S_OK`	Error occurred (see*List of Error Codes, p. 24* for more information)

4.1.15 ixcan_additional_error_info

Provides additional error information in case an error is issued by the ASCII interface. Only the first occurred error since the last call of this function is returned. Further errors are discarded.

```
void ixcan_additional_error_info(void *p_channel,
                                 uint8_t *p_error_code,
                                 char *p_error_string,
                                 uint8_t buffer_size)
```

Parameter

Parameter	Dir.	Description
p_channel	[in]	Pointer to channel instance data
p_error_code	[out]	Pointer to the stored error code received via ASCII interface. Code 255 indicates that no additional error information is available. For available additional error information in case a of code !=255 see User Manual *CAN@net NT 100/200/420 Generic Protocol 2.1 for Gateway Mode V6* for more information.
p_error_string	[out]	Pointer to a buffer of 100 bytes size to store an error string received via ASCII interface.
buffer_size	[in]	Size of error string buffer

Remark

Can be used for debugging purposes during development.

4.2 Socket Abstraction Unit

The socket abstraction unit abstracts the OS specific socket API calls and is included in file *ixsocket.h*.

4.2.1 ixsocket_open

Opens a socket communication channel.

```
int ixsocket_open(uint8_t protocol,
                  const char *p_ip_address,
                  uint16_t ip_port,
                  ixos_sockethdl_t *p_sockethdl);
```

Parameter

Parameter	Dir.	Description
protocol	[in]	Select the protocol to be used, possible values: `IXCOM_PROTOCOL_ASCII_TCP_IP`: ASCII interface via TCP `IXCOM_PROTOCOL_ASCII_UDP_IP`: ASCII interface via UDP
p_ip_address	[in]	Communication IP address (e.g. "192.168.100.101")
ip_port	[in]	Port (e.g. 19228)
p_sockethdl	[out]	Pointer to store the socket handle

Return Value

Return value	Description
`IXERR_S_OK`	No error occurred
`<IXERR_S_OK`	Error occurred (see*List of Error Codes, p. 24* for more information)

4.2.2 ixsocket_close

Closes a socket communication channel.

```
int ixsocket_close(ixos_sockethdl_t *p_sockethdl);
```

Parameter

Parameter	Dir.	Description
p_sockethdl	[in]	Pointer to the socket handle (see *ixsocket_open*)

Return Value

Return value	Description
`IXERR_S_OK`	No error occurred
`<IXERR_S_OK`	Error occurred (see*List of Error Codes, p. 24* for more information)

4.2.3 ixsocket_send

Sends a data buffer to the socket.

```
int ixsocket_send(ixos_sockethdl_t *p_sockethdl,
                  uint8_t protocol,
                  char *ip_dest,
                  char *p_data,
                  uint32_t *p_size);
```

Parameter

Parameter	Dir.	Description
p_sockethdl	[in]	Pointer to the socket handle
protocol	[in]	Select the protocol to be used, possible values: `IXCOM_PROTOCOL_ASCII_TCP_IP`: ASCII interface via TCP `IXCOM_PROTOCOL_ASCII_UDP_IP`: ASCII interface via UDP
ip_dest	[in]	Destination IP address
p_data	[in]	Pointer to data to be sent
p_size	[in/out]	In: number of bytes in the data to be sent Out: number of sent bytes (only valid when return value is `IXERR_S_OK`)

Return Value

Return value	Description
`IXERR_S_OK`	No error occurred
`<IXERR_S_OK`	Error occurred (see*List of Error Codes, p. 24* for more information)

Remark

When multi threading is used, the code in this function must be thread safe to allow simultaneous calls to different socket handles. If not protected by the OS socket library a protection mechanism must be added.

If the socket is configured as non-blocking and returns an error code of the kind *Would block* `ixsocket_send()` returns `IXERR_k_OK`, but sets the number of sent bytes to zero.

4.2.4 ixsocket_receive

Reads data from the socket.

```
int ixsocket_receive(ixos_sockethdl_t *p_sockethdl,
                     unit8_t protocol,
                     char *p_data,
                     uint32_t buffer_size);
```

Parameter

Parameter	Dir.	Description
p_sockethdl	[in]	Pointer to the socket handle
protocol	[in]	Select the protocol to be used, possible values: `IXCOM_PROTOCOL_ASCII_TCP_IP`: ASCII interface via TCP `IXCOM_PROTOCOL_ASCII_UDP_IP`: ASCII interface via UDP
p_data	[out]	Pointer to store read data
buffer_size	[in]	Maximum size of data buffer

Return Value

Return value	Description
`IXERR_S_OK (0)`	No message received
`<IXERR_S_OK`	Error occurred (see*List of Error Codes, p. 24* for more information)
`>IXERR_S_OK`	Number of received bytes

5 Data Types

5.1 ixcan_api_version_t

Describes the API version.

```
typedef struct _ixcan_api_version
{
 uint16_t api_version_major;
 uint16_t api_version_minor;
 uint16_t api_version_build;
} ixcan_api_version_t
```

Memeber	Description
api_version_major	Major API version number
api_version_minor	Minor API version number
api_version_build	Build API version number

5.2 ixcan_identification_t

Describes the device identification.

```
typedef struct _ixcan_identification
{
 char     dev_type_string[IXCAN_MAX_SIZE_DEV_TYPESTRING];
 uint16_t dev_type;
 uint8_t  dev_version_major;
 uint8_t  dev_version_minor;
 uint8_t  dev_version_revision;
 uint8_t  dev_protocol_major;
 uint8_t  dev_protocol_minor;
 uint8_t  dev_shared_mode;
 uint8_t  dev_interfaces[IXCAN_MAX_FIELD_BUS_INTERFACES];
 uint8_t  dev_identification_completed;
} ixcan_identification_t
```

Memeber	Description
dev_type_string	Device type string
dev_type	Device type identification `IXCAN_DEVTYPE_CANATNET_NT100`: CAN@net NT 100 `IXCAN_DEVTYPE_CANATNET_NT200`: CAN@net NT 200 `IXCAN_DEVTYPE_CANATNET_NT420`: CAN@net NT 420 `IXCAN_DEVTYPE_UNKNKOWN`: unknown by ixcan API
dev_version_major	Major firmware version number
dev_version_minor	Minor firmware version number
dev_version_revision	Revision firmware version number
dev_protocol_major	Major ASCII protocol version number
dev_protocol_minor	Minor ASCII protocol version number
dev_shared_mode	Flag that indicates if ASCII interface has exclusive access to the CAN controller or shared access with for example a Local Bridge. `TRUE`: shared access, CAN device has CAN control `FALSE`: exclusive access, ixcan API has CAN control
dev_interfaces	Type of field bus interfaces `IXCAN_FIELDBUS_NONE`: no fieldbus `IXCAN_FIELDBUS_CAN`: CAN controller `IXCAN_FIELDBUS_CANFD`: CAN FD controller `IXCAN_FIELDBUS_UNKNOWN`: Unknown by ixcan API
dev_identification_completed	Device identification completed `TRUE`: all identification data is read. `FALSE`: acquiring identification data

5.3 ixcan_can_msg_t

Describes the structure of a CAN messages.

```
typedef struct _ixcan_can_msg
{
 uint32_t identifier;
 uint8_t  num_bytes;
 uint8_t  format;
 uint8_t  reserved;
 uint8_t  payload[IXCAN_MAX_NUM_CAN_PAYLOAD];
} ixcan_can_msg_t
```

Memeber	Description
identifier	CAN identifier
num_bytes	Number of payload bytes (Classic CAN: <=8, for CAN FD: <=8, 12, 16, 20, 24, 32, 48, 64)
format	CAN format flags, bit encoded, can be combined. One bit encodes two states (e.g. Bit 0 encodes if Standard CAN frame (bit cleared) or Extended bit frame (bit set), possible values: `IXCAN_CAN_FORMAT_STD_FRAME 0x00`: Bit 0 Standard frame `IXCAN_CAN_FORMAT_EXT_FRAME 0x01`: Bit 0 Extended frame `IXCAN_CAN_FORMAT_DATA_FRAME 0x00`: Bit 1 Data frame `IXCAN_CAN_FORMAT_RTR_FRAME 0x02`: Bit 1 Remote frame `IXCAN_CAN_FORMAT_CAN 0x00`: Bit 4 Classic CAN frame `IXCAN_CAN_FORMAT_CANFD 0x10`: Bit 4 CAN FD frame `IXCAN_CAN_FORMAT_MSK_STD_EXT IXCAN_CAN_FORMAT_EXT_FRAME`: Mask Standard/Extended `IXCAN_CAN_FORMAT_MSK_DATA_RTR IXCAN_CAN_FORMAT_RTR_FRAME`: Mask Data/Remote `IXCAN_CAN_FORMAT_MSK_CAN_CAN_FD IXCAN_CAN_FORMAT_CANFD`: Mask CAN/CAN-FD See file *ixcan_api.h* for more information.
reserved	Reserved for alignment
payload	CAN payload

5.4 ixcan_can_timing_register_t

Describes the CAN register settings.

```
typedef struct _ixcan_can_timing_register
{
 uint16_t brp;
 uint8_t  sjw;
 uint8_t  tseg1;
 uint8_t  tseg2;
 uint8_t  tdo;
} ixcan_can_timing_register_t
```

Memeber	Description
brp	Baud rate prescaler
sjw	Synchronization jump width
tseg1	Time segment 1
tseg2	Time segment 2
tdo	Transceiver delay compensation offset (only for CAN FD data phase)

6 List of Error Codes

The list of error codes is included in file *ixerr.h*.

The return values of the ixcan API functions are coded as follows:

- Successful functions return value code 0.
- Errors return a negative value code.
- Additional information returns a positive value code.

Error code	Error	Description
0	IXERR_S_OK	No error occurred
1	IXERR_S_MSG_RECEIVED	Message received
2	IXERR_S_WAITING_FOR_CMD_RESP	Waiting for command response from device
-1	IXERR_E_ERROR	Generic error
-2	IXERR_E_BUFFER_TO_SMALL	Buffer too small to copy all data
-3	IXERR_E_SOCKET_ERROR	Generic socket error
-4	IXERR_E_SOCKET_OPEN	Unable to open the socket
-5	IXERR_E_SOCKET_CLOSING	Error closing the socket
-6	IXERR_E_SOCKET_SEND_FAILURE_GEN	Data could not be sent (generic)
-7	IXERR_E_SOCKET_SEND_BUFFER_FULL	Data could not be sent (x buffer full)
-8	IXERR_E_SOCKET_CONNECTION_ABORTED	Connection aborted
-9	IXCAN_E_CMD_REQ_UNKNOWN	Command request unknown
-10	IXCAN_E_CMD_RESP_TIMEOUT	Command response timeout reached
-11	IXCAN_E_CMD_RESP_ERROR	Command response error
-12	IXCAN_E_CMD_ASSEMBLY	Error generating ASCII command
-13	IXCAN_E_INVALID_PARAMETER	Invalid parameter
-14	IXCAN_E_INVALID_PROTOCOL_PARAMETER	Invalid value received by ASCII response
-15	IXCAN_E_DATA_NO_VALID	Invalid data
-16	IXCAN_E_CAN_TX_FIFO_FULL	Transmit queue full
-17	IXCAN_E_NO_MULTITHREADING	API seems to be called by different threads without enabled feature in ixcan API

This page intentionally left blank

Box 4126
300 04 Halmstad, Sweden

info@hms.se
4.02.0332.20003 1.1 en-US / 2021-06-28 / 22896