libicsneo Documentation

Release 0.2.0

Intrepid Control Systems, Inc.

Aug 16, 2023

Documentation

1 API Usage **1**
1.1 API Concepts . . . 1
1.1.1 Overview . . . 1
1.1.2 Events . . . 1
1.1.3 Event Callbacks . . . 2
1.1.4 Errors . . . 2
1.2 Device Concepts . . . 2
1.2.1 Open/Close Status . . . 2
1.2.2 Online/Offline Status . . . 2
1.2.3 Message Callbacks and Polling . . . 2
1.2.4 Write Blocking Status . . . 3
1.2.5 A2B Wave Output . . . 3

2 C++ API (icsneocpp) **5**
2.1 Reference . . . 5

3 C API (icsneoc) **7**
3.1 Reference . . . 7
3.1.1 Typedefs . . . 7
3.1.2 Structures . . . 7
3.1.3 Functions . . . 8

Index **25**

CHAPTER 1

API Usage

1.1 API Concepts

1.1.1 Overview

Events convey information about the API's inner workings to the user. There are 3 severity levels: `EventInfo`, `EventWarning`, and `Error`. **However, the API treats events of severities `EventInfo` and `EventWarning` differently than those of severity `Error` .** From here on out, when we (and the API functions) refer to "events", we refer exclusively to those of severities `EventInfo` and `EventWarning`, which use the *events* system. Those of severity `Error` are referred to as "errors", which use a separate *errors* system.

Events should periodically be read out in order to avoid overflowing, and the last error should be read out immediately after an API function fails.

Additionally, *event callbacks* can be registered, which may remove the need to periodically read events in some cases.

1.1.2 Events

The API stores events in a single buffer that can has a default size of 10,000 events. This limit includes 1 reserved slot at the end of the buffer for a potential Event of type `TooManyEvents` and severity `EventWarning`, which is added when there are too many events for the buffer to hold. This could occur if the events aren't read out by the user often enough, or if the user sets the size of the buffer to a value smaller than the number of existing events. There will only ever be one of these `TooManyEvents` events, and it will always be located at the very end of the buffer if it exists. Because of this reserved slot, the buffer by default is able to hold 9,999 events. If capacity is exceeded, the oldest events in the buffer are automatically discarded until the buffer is exactly at capacity again. When events are read out by the user, they are removed from the buffer. If an event filter is used, only the filtered events will be removed from the buffer.

In a multithreaded environment, all threads will log their events to the same buffer. In this case, the order of events will largely be meaningless, although the behavior of `TooManyEvents` is still guaranteed to be as described above.

1.1.3 Event Callbacks

Users may register event callbacks, which are automatically called whenever a matching event is logged. Message callbacks consist of a user-defined `std::function< void( std::shared_ptr<APIEvent> ) >` and optional EventFilter used for matching. If no EventFilter is provided, the default-constructed one will be used, which matches any event. Registering a callback returns an `int` representing the id of the callback, which should be stored by the user and later used to remove the callback when desired. Note that this functionality is only available in C and C++. C does not currently support filters.

Event callbacks are run after the event has been added to the buffer of events. The buffer of events may be safely modified within the callback, such as getting (flushing) the type and severity of the triggering event. Using event callbacks in this manner means that periodically reading events is unnecessary.

1.1.4 Errors

The error system is threadsafe and separate from the *events* system. Each thread keeps track of the last error logged on it, and getting the last error will return the last error from the calling thread, removing it in the process. Trying to get the last error when there is none will return an event of type `NoErrorFound` and severity `EventInfo`.

The API also contains some threads for internal use which may potentially log errors of their own and are inaccessible to the user. These threads have been marked to downgrade any errors that occur on them to severity `EventWarning` and will log the corresponding event in the *events* system described above.

1.2 Device Concepts

1.2.1 Open/Close Status

In order to access device functionality, the device must first be opened, which begins communication between the API and the device. The exception to this is setting the message polling status of the device. Trying to open/close the device when the device is already open/closed will result in an error being logged on the calling thread.

1.2.2 Online/Offline Status

Going online begins communication between the device and the rest of the network. In order to be online, the device must also be open. Trying to go online/offline when the device is already online/offline will result in an error being logged on the calling thread.

It is possible to have a device be both open and offline. In this situation, device settings such as the baudrate may still be read and changed. This is useful for setting up your device properly before going online and joining the network.

1.2.3 Message Callbacks and Polling

In order to handle messages, users may register message callbacks, which are automatically called whenever a matching message is received. Message callbacks consist of a user-defined `std::function< void( std::shared_ptr<Message> ) >` and optional message filter used for matching. If no message filter is provided, the default-constructed one will be used, which matches any message. Registering a callback returns an `int` representing the id of the callback, which should be stored by the user and later used to remove the callback when desired. Note that this functionality is only available in C and C++. C does not currently support filters.

The default method of handling messages is to enable message polling, which is built upon message callbacks. Enabling message polling will register a callback that stores each received message in a buffer for later retrieval. The default limit of this buffer is 20,000 messages. If the limit is exceeded, the oldest messages will be dropped until the buffer is at capacity, and an error will be logged on the calling thread. To avoid exceeding the limit, try to get messages periodically, which will flush the buffer upon each call. Attempting to read messages without first enabling message polling will result in an error being logged on the calling thread.

It is recommended to either enable message polling or manually register callbacks to handle messages, but not both.

1.2.4 Write Blocking Status

The write blocking status of the device determines the behavior of attempting to transmit to the device (likely via sending messages) with a large backlog of messages. If write blocking is enabled, then the transmitting thread will wait for the entire buffer to be transmitted. If write blocking is disabled, then the attempt to transmit will simply fail and an error will be logged on the calling thread.

1.2.5 A2B Wave Output

Users may add a `icsneo::A2BWAVOutput` message callback to their device in order to write A2B PCM data to a WAVE file. The message callback listens for `icsneo::A2BMessage` messages and writes both downstream and upstream channels to a single wave file. If downstream and upstream each have `32` channels, the wave file will contain `2*32 = 64` total channels. Channels are indexed at 0 and interleaved such that downstream are on even number channels and upstream on odd number channels. If we introduce a variable `IS_UPSTREAM` which is `0` when downstream and `1` when upstream and desired a channel `CHANNEL_NUM` the corresponding channel in the wave file would be `2*CHANNEL_NUM + IS_UPSTREAM`.

Wave files may be split by channel using programs such as `FFmpeg`. Consider a file `out.wav` which was generated using a `icsneo::A2BWAVOutput` object and contains `32` channels per stream. The `icsneo::A2BWavoutput` object injested PCM data with a sample rate of `44.1 kHz` and bit depth of `24`. The corresponding channel of upstream channel `8` in `out.wav` would be `2*CHANNEL_NUM + IS_UPSTREAM = 2*8 + 1 = 17`. The following `FFmpeg` command may be ran in a linux environment to create a new wave file `out_upstream_ch8.wav` which contains only PCM samples off of upstream channel `8`.

```
ffmpeg -i out.wav -ar 44100 -acodec pcm_s24le -map_channel 0.0.17
out_upstream_ch8.wav
```

C++ API (icsneocpp)

2.1 Reference

`namespace icsneo`

Warning: doxygenclass: Cannot find class "icsneo::Device" in doxygen xml output for project "libicsneo" from directory: build/doxygen/xml

CHAPTER 3

C API (icsneoc)

3.1 Reference

3.1.1 Typedefs

typedef void ***devicehandle_t**

typedef int32_t **neodevice_handle_t**

typedef uint32_t **devicetype_t**

3.1.2 Structures

struct neoversion_t

Public Members

uint16_t **major**

uint16_t **minor**

uint16_t **patch**

const char ***metadata**

const char ***buildBranch**

const char ***buildTag**

char **reserved**[32]

struct neodevice_t

Public Members

devicehandle_t **`device`**

neodevice_handle_t **`handle`**

devicetype_t **`type`**

char **`serial`**[7]

`struct neomessage_t`

Public Members

uint8_t **`_reserved1`**[16]

uint64_t **`timestamp`**

uint64_t **`_reservedTimestamp`**

uint8_t **`_reserved2`**[sizeof(size_t) * 2 + 7 + sizeof(neonetid_t) + sizeof(neonettype_t)]

neomessagetype_t **`messageType`**

uint8_t **`_reserved3`**[12]

`struct neomessage_can_t`

Public Members

neomessage_statusbitfield_t **`status`**

uint64_t **`timestamp`**

uint64_t **`_reservedTimestamp`**

`const` uint8_t ***`data`**

size_t **`length`**

uint32_t **`arbid`**

neonetid_t **`netid`**

neonettype_t **`type`**

uint8_t **`dlcOnWire`**

uint16_t **`description`**

neomessagetype_t **`messageType`**

uint8_t **`_reserved1`**[12]

3.1.3 Functions

Functions

void **`icsneo_findAllDevices`**(*neodevice_t* **devices*, size_t **count*)

Find Intrepid hardware connected via USB and Ethernet.

For each found device, a *neodevice_t* structure will be written into the memory you provide.

Parameters

- `devices`: Pointer to memory where devices should be written. If NULL, the current number of detected devices is written to count.
- `count`: Pointer to a size_t, which should initially contain the number of devices the buffer can hold, and will afterwards contain the number of devices found.

The *neodevice_t* can later be passed by reference into the API to perform actions relating to the device. The *neodevice_t* contains a handle to the internal memory for the icsneo::Device object. The memory for the internal icsneo::Device object is managed by the API.

Any *neodevice_t* objects which have not been opened will become invalid when *icsneo_findAllDevices()* is called again. To invoke this behavior without finding devices again, call *icsneo_freeUnconnectedDevices()*.

If the size provided is not large enough, the output will be truncated. An icsneo::APIEvent::OutputTruncatedError will be available in *icsneo_getLastError()* in this case.

void **`icsneo_freeUnconnectedDevices`**()

Invalidate *neodevice_t* objects which have not been opened.

See *icsneo_findAllDevices()* for information regarding the *neodevice_t* validity contract.

bool **`icsneo_serialNumToString`**(uint32_t *num*, char **str*, size_t **count*)

Convert a serial number in numerical format to its string representation.

On older devices, the serial number is one like 138635, the numerical representation is the same as the string representation.

Return True if str contains the string representation of the given serial number.

Parameters

- `num`: A numerical serial number.
- `str`: A pointer to a buffer where the string representation will be written. NULL can be passed, which will write a character count to `count`.
- `count`: A pointer to a size_t which, prior to the call, holds the maximum number of characters to be written (so str must be of size count + 1 to account for the NULL terminator), and after the call holds the number of characters written.

On newer devices, the serial number is one like RS2259, and this function can convert the numerical value back into the string seen on the back of the device.

A query for length (`str == NULL`) will return false. *icsneo_getLastError()* should be checked to verify that the *neodevice_t* provided was valid.

The client application should provide a buffer of size 7, as serial numbers are always 6 characters or fewer.

If the size provided is not large enough, the output will be **NOT** be truncated. Nothing will be written to the output. Instead, an icsneo::APIEvent::BufferInsufficient will be available in *icsneo_getLastError()*. False will be returned, and `count` will now contain the number of *bytes* necessary to store the full string.

uint32_t **`icsneo_serialStringToNum`**(**const** char **str*)

Convert a serial number in string format to its numerical representation.

On older devices, the serial number is one like 138635, and this string will simply be returned as a number.

Return The numerical representation of the serial number, or 0 if the conversion was unsuccessful.

Parameters

- `str`: A NULL terminated string containing the string representation of an Intrepid serial number.

On newer devices, the serial number is one like RS2259, and this function can convert that string to a number.

bool **`icsneo_isValidNeoDevice`**(**`const`** *neodevice_t *device*)

Verify that a *neodevice_t* is valid.

This check is automatically performed at the beginning of any API function that operates on a device. If there is a failure, an icsneo::APIEvent::InvalidNeoDevice will be available in *icsneo_getLastError()*.

Return True if the *neodevice_t* is valid.

Parameters

- `device`: A pointer to the *neodevice_t* structure to operate on.

See *icsneo_findAllDevices()* for information regarding the *neodevice_t* validity contract.

bool **`icsneo_openDevice`**(**`const`** *neodevice_t *device*)

Connect to the specified hardware.

The device **MUST** be opened before any other functions which operate on the device will be valid.

Return True if the connection could be opened.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to open.

See *icsneo_goOnline()* for information about enabling network communication once the device is open.

If the open did not succeed, *icsneo_getLastError()* should provide more information about why.

If the device was already open, an icsneo::APIEvent::DeviceCurrentlyOpen will be available in *icsneo_getLastError()*.

bool **`icsneo_closeDevice`**(**`const`** *neodevice_t *device*)

Close an open connection to the specified hardware.

After this function succeeds, the *neodevice_t* will be invalid. To connect again, you must call *icsneo_findAllDevices()* or similar to re-find the device.

Return True if the connection could be closed.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to close.

bool **`icsneo_isOpen`**(**`const`** *neodevice_t *device*)

Verify network connection for the specified hardware.

This function does not modify the working state of the device at all.

Return True if the device is connected.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

See *icsneo_openDevice()* for an explanation about the concept of being "open".

bool **`icsneo_goOnline`**(**`const`** *neodevice_t *device*)

Enable network communication for the specified hardware.

The device is not "online" when it is first opened. It is not possible to receive or transmit while the device is "offline". Network controllers are disabled. (i.e. In the case of CAN, the hardware will not send ACKs on the client application's behalf)

Return True if communication could be enabled.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

This allows filtering or handlers to be set up before allowing traffic to flow.

This also allows device settings to be set (i.e. baudrates) before enabling the controllers, which prevents momentarily causing loss of communication if the baud rates are not correct.

bool **`icsneo_goOffline`**(**const** *neodevice_t *device*)

Disable network communication for the specified hardware.

See *icsneo_goOnline()* for an explanation about the concept of being "online".

Return True if communication could be disabled.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

bool **`icsneo_isOnline`**(**const** *neodevice_t *device*)

Verify network communication for the specified hardware.

This function does not modify the working state of the device at all.

Return True if communication is enabled.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

See *icsneo_goOnline()* for an explanation about the concept of being "online".

bool **`icsneo_enableMessagePolling`**(**const** *neodevice_t *device*)

Enable buffering of messages for the specified hardware.

By default, traffic the device receives will not reach the client application. The client application must register traffic handlers, enable message polling, or both. This function addresses message polling.

Return True if polling could be enabled.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

With polling enabled, all traffic that the device receives will be stored in a buffer managed by the API. The client application should then call *icsneo_getMessages()* periodically to take ownership of the messages in that buffer.

The API managed buffer will only grow to a specified size, 20k messages by default. See *icsneo_getPollingMessageLimit()* and *icsneo_setPollingMessageLimit()* for more information.

In high traffic situations, the default 20k message limit can be reached very quickly. The client application will have to call *icsneo_getMessages()* very often to avoid losing messages, or change the limit.

If the message limit is exceeded before a call to *icsneo_getMessages()* takes ownership of the messages, the oldest message will be dropped (**LOST**) and an icsneo::APIEvent::PollingMessageOverflow will be flagged for the device.

This function will succeed even if the device is not open.

bool **`icsneo_disableMessagePolling`**(**const** *neodevice_t *device*)

Disable buffering of messages for the specified hardware.

See *icsneo_enableMessagePolling()* for more information about the message polling system.

Return True if polling could be disabled.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

Any messages left in the API managed buffer will be lost upon disabling polling.

bool **icsneo_isMessagePollingEnabled**(**const** *neodevice_t *device*)

Verify message polling status for the specified hardware.

This function does not modify the working state of the device at all.

Return True if polling is enabled.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

See *icsneo_enableMessagePolling()* for an explanation about how polling works.

bool **icsneo_getMessages**(**const** *neodevice_t *device*, *neomessage_t *messages*, size_t **items*, uint64_t *timeout*)

Read out messages which have been recieved.

Messages are available using this function if *icsneo_goOnline()* and *icsneo_enableMessagePolling()* have been called. See those functions for more information.

Return True if the messages were read out successfully (even if there were no messages to read) or if the count was read successfully.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `messages`: A pointer to a buffer which *neomessage_t* structures will be written to. NULL can be passed, which will write the current message count to size.
- `items`: A pointer to a size_t which, prior to the call, holds the maximum number of messages to be written, and after the call holds the number of messages written.
- `timeout`: The number of milliseconds to wait for a message to arrive. A value of 0 indicates a non-blocking call. Querying for the current message count is always asynchronous and ignores this value.

Messages are read out of the API managed buffer in order of oldest to newest. As they are read out, they are removed from the API managed buffer.

If size is too small to contain all messages, as many messages as will fit will be read out. Subsequent calls to *icsneo_getMessages()* can retrieve any messages which were not read out.

The memory for the data pointer within the *neomessage_t* is managed by the API. Do *not* attempt to free the data pointer. The memory will become invalid the next time *icsneo_getMessages()* is called for this device.

```
size_t messageCount;
bool result = icsneo_getMessages(device, NULL, &messageCount, 0); // Reading the␣
↪message count
// Handle errors here
neomessage_t* messages = malloc(messageCount * sizeof(neomessage_t)); // It is␣
↪also possible and encouraged to use a static buffer
result = icsneo_getMessages(device, messages, &messageCount, 0); // Non-blocking
// Handle errors here
for(size_t i = 0; i < messageCount; i++) {
    switch(messages[i].type) {
        case ICSNEO_NETWORK_TYPE_CAN: {
            // All messages of type CAN can be accessed using neomessage_can_t
            neomessage_can_t* canMessage = (neomessage_can_t*)&messages[i];
            printf("ArbID: 0x%x\n", canMessage->arbid);
            printf("DLC: %u\n", canMessage->length);
```

(continues on next page)

(continued from previous page)

```
                printf("Data: ");
                for(size_t i = 0; i < canMessage->length; i++) {
                    printf("%02x ", canMessage->data[i]);
                }
                printf("\nTimestamp: %lu\n", canMessage->timestamp);
            }
        }
}
free(messages);
```

Warning Do not call icsneo_close() while another thread is waiting on *icsneo_getMessages()*. Always allow the other thread to timeout first!

int **icsneo_getPollingMessageLimit** (**const** *neodevice_t* **device*)
Get the maximum number of messages which will be held in the API managed buffer for the specified hardware.

See *icsneo_enableMessagePolling()* for more information about the message polling system.

Return Number of messages, or -1 if device is invalid.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

bool **icsneo_setPollingMessageLimit** (**const** *neodevice_t* **device*, size_t *newLimit*)
Set the maximum number of messages which will be held in the API managed buffer for the specified hardware.

See *icsneo_enableMessagePolling()* for more information about the message polling system.

Return True if the limit was set successfully.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `newLimit`: The new limit to be enforced.

Setting the maximum lower than the current number of stored messages will cause the oldest messages to be dropped (**LOST**) and an icsneo::APIEvent::PollingMessageOverflow to be flagged for the device.

int **icsneo_addMessageCallback** (**const** *neodevice_t* **device*, void (**callback*)) *neomessage_t*
, void *Adds a message callback to the specified device to be called when a new message is received.

Return The id of the callback added, or -1 if the operation failed.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `callback`: A function pointer with void return type and a single *neomessage_t* parameter.
- `filter`: Unused for now. Exists as a placeholder here for future backwards-compatibility.

bool **icsneo_removeMessageCallback** (**const** *neodevice_t* **device*, int *id*)
Removes a message callback from the specified device.

Return True if the callback was successfully removed.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

- `id`: The id of the callback to remove.

neonetid_t **`icsneo_getNetworkByNumber`**(**`const`** *neodevice_t* **device*, neonettype_t *type*, unsigned int *number*)

Get the network ID for the nth network of a specified type on this device.

This function is designed so that networks can be enumerated without knowledge of the specific device. For instance, on a ValueCAN 4-2, the second CAN network is ICSNEO_NETID_HSCAN2, while on a neoVI FIRE the second CAN network is ICSNEO_NETID_MSCAN.

Return The netid if the call succeeds, ICSNEO_NETID_INVALID otherwise

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `type`: An ICSNEO_NETWORK_TYPE_* constant denoting the network type
- `number`: The number of this network starting from 1

bool **`icsneo_getProductName`**(**`const`** *neodevice_t* **device*, char **str*, size_t **maxLength*)

Get the friendly product name for a specified device.

In the case of a neoVI FIRE 2, this function will write a string "neoVI FIRE 2" with a NULL terminator into str.

Return True if str was written to

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `str`: A pointer to a buffer where the string will be written. NULL can be passed, which will write a character count to maxLength.
- `maxLength`: A pointer to a size_t which, prior to the call, holds the maximum number of characters to be written (so str must be of size maxLength + 1 to account for the NULL terminator), and after the call holds the number of characters written.

The constant ICSNEO_DEVICETYPE_LONGEST_NAME is defined for the client application to create static buffers of the correct length.

See also *icsneo_describeDevice()*.

A query for length (`str == NULL`) will return false. *icsneo_getLastError()* should be checked to verify that the *neodevice_t* provided was valid.

If the size provided is not large enough, the output will be truncated. An icsneo::APIEvent::OutputTruncatedError will be available in *icsneo_getLastError()* in this case. True will still be returned.

bool **`icsneo_getProductNameForType`**(*devicetype_t* *type*, char **str*, size_t **maxLength*)

Get the friendly product name for a specified devicetype.

In the case of a neoVI FIRE 2, this function will write a string "neoVI FIRE 2" with a NULL terminator into str.

Return True if str was written to

Parameters

- `type`: A *neodevice_t* structure specifying the device to operate on.
- `str`: A pointer to a buffer where the string will be written. NULL can be passed, which will write a character count to maxLength.

- `maxLength`: A pointer to a size_t which, prior to the call, holds the maximum number of characters to be written (so str must be of size maxLength + 1 to account for the NULL terminator), and after the call holds the number of characters written.

Note that icsneo_getProductName should *always* be preferred where available, as the product name may change based on device-specific factors, such as the serial number.

The constant ICSNEO_DEVICETYPE_LONGEST_NAME is defined for the client application to create static buffers of the correct length.

See also *icsneo_describeDevice()*.

A query for length (`str == NULL`) will return false. *icsneo_getLastError()* should be checked to verify that the *neodevice_t* provided was valid.

If the size provided is not large enough, the output will be truncated. An icsneo::APIEvent::OutputTruncatedError will be available in *icsneo_getLastError()* in this case. True will still be returned.

bool **`icsneo_settingsRefresh`**(**`const`** *neodevice_t *device*)

Trigger a refresh of the settings structure for a specified device.

Return True if the refresh succeeded.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

bool **`icsneo_settingsApply`**(**`const`** *neodevice_t *device*)

Commit the settings structure for a specified device to non-volatile storage.

When modifications are made to the device settings, this function (or *icsneo_settingsApplyTemporary()*) must be called to send the changes to the device and make them active.

Return True if the settings were applied.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

This function sets the settings such that they will survive device power cycles.

If the function fails, the settings will be refreshed so that the structure in the API matches the one held by the device.

bool **`icsneo_settingsApplyTemporary`**(**`const`** *neodevice_t *device*)

Apply the settings structure for a specified device temporarily.

See *icsneo_settingsApply()* for further information about applying settings.

Return True if the settings were applied.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

This function sets the settings such that they will revert to the values saved in non-volatile storage when the device loses power.

bool **`icsneo_settingsApplyDefaults`**(**`const`** *neodevice_t *device*)

Apply the default settings structure for a specified device.

See *icsneo_settingsApply()* for further information about applying settings.

Return True if the default settings were applied.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

This function sets the default settings such that they will survive device power cycles.

bool **`icsneo_settingsApplyDefaultsTemporary`**(**`const`** *neodevice_t *device*)

Apply the default settings structure for a specified device temporarily.

See *icsneo_settingsApply()* for further information about applying settings. See *icsneo_settingsApplyDefaults()* for further information about applying default settings.

Return True if the default settings were applied.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

This function sets the default settings such that they will revert to the values saved in non-volatile storage when the device loses power.

int **`icsneo_settingsReadStructure`**(**`const`** *neodevice_t *device*, void **structure*, size_t *structureSize*)

Apply the default settings structure for a specified device temporarily.

See *icsneo_settingsApply()* for further information about applying settings. See *icsneo_settingsApplyDefaults()* for further information about applying default settings.

Return Number of bytes written to structure, or -1 if the operation failed.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `structure`: A pointer to a device settings structure for the current device.
- `structureSize`: The size of the current device settings structure in bytes.

This function sets the default settings such that they will revert to the values saved in non-volatile storage when the device loses power.

If possible, use functions specific to the operation you want to acomplish (such as *icsneo_setBaudrate()*) instead of modifying the structure directly. This allows the client application to work with other hardware.

bool **`icsneo_settingsApplyStructure`**(**`const`** *neodevice_t *device*, **`const`** void **structure*, size_t *structureSize*)

Apply a provided settings structure for a specified device.

This function immediately applies the provided settings. See *icsneo_settingsApplyTemporary()* for further information about applying settings.

Return True if the settings were applied.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `structure`: A pointer to a device settings structure for the current device.
- `structureSize`: The size of the current device settings structure in bytes.

If possible, use functions specific to the operation you want to acomplish (such as *icsneo_setBaudrate()*) instead of modifying the structure directly. This allows the client application to work with other hardware.

bool **`icsneo_settingsApplyStructureTemporary`**(**`const`** *neodevice_t *device*, **`const`** void **structure*, size_t *structureSize*)

Apply a provided settings structure for a specified device without saving to non-volatile EEPROM.

This function immediately applies the provided settings. See *icsneo_settingsApply()* for further information about applying settings.

Return True if the settings were applied.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `structure`: A pointer to a device settings structure for the current device.
- `structureSize`: The size of the current device settings structure in bytes.

This function sets the default settings such that they will revert to the values saved in non-volatile storage when the device loses power.

If possible, use functions specific to the operation you want to acomplish (such as *icsneo_setBaudrate()*) instead of modifying the structure directly. This allows the client application to work with other hardware.

int64_t **icsneo_getBaudrate**(**const** *neodevice_t* **device*, neonetid_t *netid*)

Get the network baudrate for a specified device.

In the case of CAN, this function gets the standard CAN baudrate. See *icsneo_getFDBaudrate()* to get the baudrate for (the baudrate-switched portion of) CAN FD.

Return The value in baud with no multipliers. (i.e. 500k becomes 500000) A negative value is returned if an error occurs.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `netid`: The network for which the baudrate should be retrieved.

bool **icsneo_setBaudrate**(**const** *neodevice_t* **device*, neonetid_t *netid*, int64_t *newBaudrate*)

Set the network baudrate for a specified device.

In the case of CAN, this function sets the standard CAN baudrate. See *icsneo_setFDBaudrate()* to set the baudrate for (the baudrate-switched portion of) CAN FD.

Return True if the baudrate could be set.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `netid`: The network to which the new baudrate should apply.
- `newBaudrate`: The requested baudrate, with no multipliers. (i.e. 500K CAN should be represented as 500000)

Call *icsneo_settingsApply()* or similar to make the changes active on the device.

int64_t **icsneo_getFDBaudrate**(**const** *neodevice_t* **device*, neonetid_t *netid*)

Get the CAN FD baudrate for a specified device.

See *icsneo_getBaudrate()* to get the baudrate for the non baudrate-switched portion of CAN FD, classical CAN 2.0, and other network types.

Return The value in baud with no multipliers. (i.e. 500k becomes 500000) A negative value is returned if an error occurs.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `netid`: The network for which the baudrate should be retrieved.

bool **icsneo_setFDBaudrate**(**const** *neodevice_t* **device*, neonetid_t *netid*, int64_t *newBaudrate*)

Set the CAN FD baudrate for a specified device.

See *icsneo_setBaudrate()* to set the baudrate for the non baudrate-switched portion of CAN FD, classical CAN 2.0, and other network types.

Return True if the baudrate could be set.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `netid`: The network to which the new baudrate should apply.
- `newBaudrate`: The requested baudrate, with no multipliers. (i.e. 2Mbaud CAN FD should be represented as 2000000)

Call *icsneo_settingsApply()* or similar to make the changes active on the device.

bool **icsneo_transmit**(**const** *neodevice_t* **device*, **const** *neomessage_t* **message*)

Transmit a single message.

To transmit a message, you must set the `data`, `length`, and `netid` attributes of the *neomessage_t*.

Return True if the message was verified transmittable and enqueued for transmit.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to transmit on.
- `message`: A pointer to the *neomessage_t* structure defining the message.

The `data` attribute must be set to a pointer to a buffer of at least `length` which holds the payload bytes. This buffer only needs to be valid for the duration of this call, and can safely be deallocated or reused after the return.

You may also have to set network dependent variables. For CAN, you must set the `arbid` attribute defined in *neomessage_can_t*.

Other attributes of the *neomessage_t* such as `timestamp`, `type` and `reserved` which are not used should be set to 0. Unused status bits should also be set to 0.

Any types defined `neomessage_*_t` are designed to be binary compatible with *neomessage_t*.

For instance, for CAN, it is recommended to use *neomessage_can_t* as it exposes the `arbid` field.

```
neomessage_can_t mySendMessage = {}; // Zero all before use
uint8_t myData[3] = { 0xAA, 0xBB, 0xCC }; // Either heap or stack allocated is␣
↪okay
mySendMessage.netid = ICSNEO_NETID_HSCAN;
mySendMessage.arbid = 0x1c5001c5;
mySendMessage.length = 3;
mySendMessage.data = myData;
mySendMessage.status.canfdFDF = true; // CAN FD
mySendMessage.status.extendedFrame = true; // Extended (29-bit) arbitration IDs
mySendMessage.status.canfdBRS = true; // CAN FD Baudrate Switch
bool result = icsneo_transmit(device, (neomessage_t*)&mySendMessage);

myData[1] = 0x55; // The message and buffer can be safely reused for the next␣
↪message
result = icsneo_transmit(device, (neomessage_t*)&mySendMessage);
```

bool **icsneo_transmitMessages**(**const** *neodevice_t* **device*, **const** *neomessage_t* **messages*, size_t *count*)

Transmit a multiple messages.

See *icsneo_transmit()* for information regarding transmitting messages.

Return True if the messages were verified transmittable and enqueued for transmit.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to transmit on.
- `messages`: A pointer to the *neomessage_t* structures defining the messages.
- `count`: The number of messages to transmit.

On a per-network basis, messages will be transmitted in the order that they were enqueued.

In this case, messages will be enqueued in order of increasing index.

void **`icsneo_setWriteBlocks`**(**`const`** *neodevice_t *device*, bool *blocks*)

Set the behavior of whether writing is a blocking action or not.

By default, writing is a blocking action.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to transmit on.
- `blocks`: Whether or not writing is a blocking action.

bool **`icsneo_describeDevice`**(**`const`** *neodevice_t *device*, char **str*, size_t **maxLength*)

Get the friendly description for a specified device.

In the case of a neoVI FIRE 2 with serial number CY2285, this function will write a string "neoVI FIRE 2 CY2285" with a NULL terminator into str.

Return True if str was written to

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `str`: A pointer to a buffer where the string will be written. NULL can be passed, which will write a character count to maxLength.
- `maxLength`: A pointer to a size_t which, prior to the call, holds the maximum number of characters to be written (so str must be of size maxLength + 1 to account for the NULL terminator), and after the call holds the number of characters written.

The constant ICSNEO_DEVICETYPE_LONGEST_DESCRIPTION is defined for the client application to create static buffers of the correct length.

See also *icsneo_getProductName()*.

A query for length (`str == NULL`) will return false. *icsneo_getLastError()* should be checked to verify that the *neodevice_t* provided was valid.

If the size provided is not large enough, the output will be truncated. An icsneo::APIEvent::OutputTruncatedError will be available in *icsneo_getLastError()* in this case. True will still be returned.

neoversion_t **`icsneo_getVersion`**(void)

Get the version of libicsneo in use.

Return A *neoversion_t* structure containing the version.

int **`icsneo_addEventCallback`**(void (**callback*)) neoevent_t

, void *Adds an event callback to be called when a new event is added.

Do not attempt to add or remove callbacks inside of a callback, as the stored callbacks are locked during calls.

Return The id of the callback added. Does not error.

Parameters

- `callback`: A function pointer with void return type and a single neoevent_t parameter.
- `filter`: Unused for now. Exists as a placeholder here for future backwards-compatibility.

bool **`icsneo_removeEventCallback`**(int *id*)

Removes an event callback.

Return True if the callback was successfully removed.

Parameters

- `id`: The id of the callback to remove.

bool **`icsneo_getEvents`**(neoevent_t **events*, size_t **size*)

Read out events which have occurred in API operation.

Events contain INFO and WARNINGS, and may potentially contain one TooManyEvents WARNING at the end. No ERRORS are found in Events, see *icsneo_getLastError()* instead.

Return True if the events were read out successfully (even if there were no events to report).

Parameters

- `events`: A pointer to a buffer which neoevent_t structures will be written to. NULL can be passed, which will write the current event count to size.
- `size`: A pointer to a size_t which, prior to the call, holds the maximum number of events to be written, and after the call holds the number of events written.

Events can be caused by API usage, such as providing too small of a buffer or disconnecting from a device.

Events can also occur asynchronously to the client application threads, in the case of a device communication event or similar.

Events are read out of the API managed buffer in order of oldest to newest. As they are read out, they are removed from the API managed buffer.

If size is too small to contain all events, as many events as will fit will be read out. Subsequent calls to *icsneo_getEvents()* can retrieve any events which were not read out.

bool **`icsneo_getDeviceEvents`**(**const** *neodevice_t* **device*, neoevent_t **events*, size_t **size*)

Read out events which have occurred in API operation for a specific device.

See *icsneo_getEvents()* for more information about the event system.

Return True if the events were read out successfully (even if there were no events to report).

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to read out events for. NULL can be passed, which indicates that **ONLY** events *not* associated with a device are desired (API events).
- `events`: A pointer to a buffer which neoevent_t structures will be written to. NULL can be passed, which will write the current event count to size.
- `size`: A pointer to a size_t which, prior to the call, holds the maximum number of events to be written, and after the call holds the number of events written.

bool **icsneo_getLastError**(neoevent_t **error*)

Read out the last error which occurred in API operation on this thread.

All errors are stored on a per-thread basis, meaning that calling *icsneo_getLastError()* will return the last error that occured on the calling thread. Any errors can only be retrieved through this function, and NOT *icsneo_getEvents()* or similar! Only INFO and WARNING level events are accessible through those. Only the last error is stored, so the intention is for this function to be called immediately following another failed API call.

Return True if an error was read out.

Parameters

- `error`: A pointer to a buffer which a neoevent_t structure will be written to.

The API error system is thread-safe. Only an API error which occurred on the current thread will be returned.

See *icsneo_getEvents()* for more information about the event system.

This operation removes the returned error from the buffer, so subsequent calls to error functions will not include the error.

void **icsneo_discardAllEvents**(void)

Discard all events which have occurred in API operation. Does NOT discard any errors.

void **icsneo_discardDeviceEvents**(**const** *neodevice_t* **device*)

Discard all events which have occurred in API operation.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to discard events for. NULL can be passed, which indicates that **ONLY** events *not* associated with a device are desired (API events). Does NOT discard any errors (device or otherwise).

void **icsneo_setEventLimit**(size_t *newLimit*)

Set the number of events which will be held in the API managed buffer before icsneo::APIEvent::TooManyEvents.

If the event limit is reached, an icsneo::APIEvent::TooManyEvents will be flagged.

Parameters

- `newLimit`: The new limit. Must be >10. 1 event slot is always reserved for a potential icsneo::APIEvent::TooManyEvents, so (newLimit - 1) other events can be stored.

If the `newLimit` is smaller than the current event count, events will be removed in order of decreasing age. This will also flag an icsneo::APIEvent::TooManyEvents.

size_t **icsneo_getEventLimit**(void)

Get the number of events which can be held in the API managed buffer.

Return The current limit.

bool **icsneo_getSupportedDevices**(*devicetype_t* **devices*, size_t **count*)

Get the devices supported by the current version of the API.

See *icsneo_getProductNameForType()* to get textual descriptions of each device.

Return True if devices was written to

Parameters

- `devices`: A pointer to a buffer of devicetype_t structures which will be written to. NULL can be passed, which will write the current supported device count to count.

- `count`: A pointer to a size_t which, prior to the call, holds the maximum number of devicetype_t structures to be written, and after the call holds the number of devicetype_t structures written.

A query for length (`devices == NULL`) will return false.

If the count provided is not large enough, the output will be truncated. An icsneo::APIEvent::OutputTruncatedError will be available in *icsneo_getLastError()* in this case. True will still be returned.

bool **`icsneo_getTimestampResolution`**(**`const`** *neodevice_t* **device*, uint16_t **resolution*)
Get the timestamp resolution for the given device.

Return True if the resolution was written

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to read out the timestamp for.
- `resolution`: A pointer to a uint16_t where the resolution will be stored. This value is in nanoseconds.

bool **`icsneo_getDigitalIO`**(**`const`** *neodevice_t* **device*, neoio_t *type*, uint32_t *number*, bool **value*)
Get the value of a digital IO for the given device.

These values are often not populated if the device is not "online".

Return True if the value is read successfully

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `type`: The IO type
- `number`: The index within the IO type, starting from 1
- `value`: A pointer to the uint8_t which will store the value of the IO port, if successful

bool **`icsneo_setDigitalIO`**(**`const`** *neodevice_t* **device*, neoio_t *type*, uint32_t *number*, bool *value*)
Get the value of a digital IO for the given device.

Note that this function is not synchronous with the device confirming the change.

Return True if the parameters and connection state are correct to submit the request to the device

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `type`: The IO type
- `number`: The index within the IO type, starting from 1
- `value`: The value which will be written to the IO

bool **`icsneo_isTerminationSupportedFor`**(**`const`** *neodevice_t* **device*, neonetid_t *netid*)
Check whether software switchable termination is supported for a given network on this device.

This does not check whether another network in the termination group has termination enabled, check canTerminationBeEnabledFor for that.

Return True if software switchable termination is supported

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.

- `netid`: The network ID to check

bool **`icsneo_canTerminationBeEnabledFor`**(**`const`** *neodevice_t *device*, neonetid_t *netid*)

Check whether software switchable termination can currently be enabled for a given network.

If another network in the group is already enabled, or if termination is not supported on this network, false is returned and an error will have been reported in *icsneo_getLastError()*.

Return True if software switchable termination can currently be enabled

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `netid`: The network ID to check

bool **`icsneo_isTerminationEnabledFor`**(**`const`** *neodevice_t *device*, neonetid_t *netid*)

Check whether software switchable termination is currently enabled for a given network in the currently active device settings.

Note that if the termination status is set, but not yet applied to the device, the current device status will be reflected here rather than the pending status.

Return True if software switchable termination is currently enabled

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `netid`: The network ID to check

False will be returned and an error will be set in icsneo_getLastError if the setting is unreadable.

bool **`icsneo_setTerminationFor`**(**`const`** *neodevice_t *device*, neonetid_t *netid*, bool *enabled*)

Enable or disable software switchable termination for a given network.

All other networks in the termination group must be disabled prior to the call, but the change does not need to be applied to the device before enqueing the enable.

Return True if if the call was successful, otherwise an error will have been reported in *icsneo_getLastError()*.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `netid`: The network ID to affect
- `enabled`: Whether to enable or disable switchable termination

int **`icsneo_getDeviceStatus`**(**`const`** *neodevice_t *device*, void **status*, size_t **size*)

Return the device status structures for a specified device.

This function populates the device status structures and sub members.

Return True if the device status was successfully read.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to operate on.
- `status`: A pointer to a device status structure for the current device.
- `size`: The size of the current device status structure in bytes.

bool **`icsneo_getRTC`**(**`const`** *neodevice_t *device*, uint64_t **output*)

Get the real-time clock for the given device.

Return True if the RTC was successfully retrieved.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to read the RTC from.
- `output`: A pointer to a uint64_t where the RTC will be stored. This value is in seconds.

bool **`icsneo_setRTC`**(**`const`** *neodevice_t* **device*, uint64_t *input*)

Set the real-time clock for the given device.

Return True if the RTC was successfully set.

Parameters

- `device`: A pointer to the *neodevice_t* structure specifying the device to write the RTC to.
- `input`: A uint64_t object holding the RTC value. This value is in seconds.

Index

D

devicehandle_t (*C++ type*), 7
devicetype_t (*C++ type*), 7

I

icsneo (*C++ type*), 5
icsneo_addEventCallback (*C++ function*), 19
icsneo_addMessageCallback (*C++ function*), 13
icsneo_canTerminationBeEnabledFor (*C++ function*), 23
icsneo_closeDevice (*C++ function*), 10
icsneo_describeDevice (*C++ function*), 19
icsneo_disableMessagePolling (*C++ function*), 11
icsneo_discardAllEvents (*C++ function*), 21
icsneo_discardDeviceEvents (*C++ function*), 21
icsneo_enableMessagePolling (*C++ function*), 11
icsneo_findAllDevices (*C++ function*), 8
icsneo_freeUnconnectedDevices (*C++ function*), 9
icsneo_getBaudrate (*C++ function*), 17
icsneo_getDeviceEvents (*C++ function*), 20
icsneo_getDeviceStatus (*C++ function*), 23
icsneo_getDigitalIO (*C++ function*), 22
icsneo_getEventLimit (*C++ function*), 21
icsneo_getEvents (*C++ function*), 20
icsneo_getFDBaudrate (*C++ function*), 17
icsneo_getLastError (*C++ function*), 20
icsneo_getMessages (*C++ function*), 12
icsneo_getNetworkByNumber (*C++ function*), 14
icsneo_getPollingMessageLimit (*C++ function*), 13
icsneo_getProductName (*C++ function*), 14
icsneo_getProductNameForType (*C++ function*), 14
icsneo_getRTC (*C++ function*), 23
icsneo_getSupportedDevices (*C++ function*), 21
icsneo_getTimestampResolution (*C++ function*), 22
icsneo_getVersion (*C++ function*), 19
icsneo_goOffline (*C++ function*), 11
icsneo_goOnline (*C++ function*), 10
icsneo_isMessagePollingEnabled (*C++ function*), 12
icsneo_isOnline (*C++ function*), 11
icsneo_isOpen (*C++ function*), 10
icsneo_isTerminationEnabledFor (*C++ function*), 23
icsneo_isTerminationSupportedFor (*C++ function*), 22
icsneo_isValidNeoDevice (*C++ function*), 9
icsneo_openDevice (*C++ function*), 10
icsneo_removeEventCallback (*C++ function*), 20
icsneo_removeMessageCallback (*C++ function*), 13
icsneo_serialNumToString (*C++ function*), 9
icsneo_serialStringToNum (*C++ function*), 9
icsneo_setBaudrate (*C++ function*), 17
icsneo_setDigitalIO (*C++ function*), 22
icsneo_setEventLimit (*C++ function*), 21
icsneo_setFDBaudrate (*C++ function*), 17
icsneo_setPollingMessageLimit (*C++ function*), 13
icsneo_setRTC (*C++ function*), 24
icsneo_setTerminationFor (*C++ function*), 23
icsneo_settingsApply (*C++ function*), 15
icsneo_settingsApplyDefaults (*C++ function*), 15
icsneo_settingsApplyDefaultsTemporary (*C++ function*), 16
icsneo_settingsApplyStructure (*C++ function*), 16
icsneo_settingsApplyStructureTemporary (*C++ function*), 16

icsneo_settingsApplyTemporary (*C++ function*), 15
icsneo_settingsReadStructure (*C++ function*), 16
icsneo_settingsRefresh (*C++ function*), 15
icsneo_setWriteBlocks (*C++ function*), 19
icsneo_transmit (*C++ function*), 18
icsneo_transmitMessages (*C++ function*), 18

N

neodevice_handle_t (*C++ type*), 7
neodevice_t (*C++ class*), 7
neodevice_t::device (*C++ member*), 8
neodevice_t::handle (*C++ member*), 8
neodevice_t::serial (*C++ member*), 8
neodevice_t::type (*C++ member*), 8
neomessage_can_t (*C++ class*), 8
neomessage_can_t::_reserved1 (*C++ member*), 8
neomessage_can_t::_reservedTimestamp (*C++ member*), 8
neomessage_can_t::arbid (*C++ member*), 8
neomessage_can_t::data (*C++ member*), 8
neomessage_can_t::description (*C++ member*), 8
neomessage_can_t::dlcOnWire (*C++ member*), 8
neomessage_can_t::length (*C++ member*), 8
neomessage_can_t::messageType (*C++ member*), 8
neomessage_can_t::netid (*C++ member*), 8
neomessage_can_t::status (*C++ member*), 8
neomessage_can_t::timestamp (*C++ member*), 8
neomessage_can_t::type (*C++ member*), 8
neomessage_t (*C++ class*), 8
neomessage_t::_reserved1 (*C++ member*), 8
neomessage_t::_reserved2 (*C++ member*), 8
neomessage_t::_reserved3 (*C++ member*), 8
neomessage_t::_reservedTimestamp (*C++ member*), 8
neomessage_t::messageType (*C++ member*), 8
neomessage_t::timestamp (*C++ member*), 8
neoversion_t (*C++ class*), 7
neoversion_t::buildBranch (*C++ member*), 7
neoversion_t::buildTag (*C++ member*), 7
neoversion_t::major (*C++ member*), 7
neoversion_t::metadata (*C++ member*), 7
neoversion_t::minor (*C++ member*), 7
neoversion_t::patch (*C++ member*), 7
neoversion_t::reserved (*C++ member*), 7