NI-XNET Python API Documentation

Release 0.3.1

National Instruments

Apr 26, 2021

Table of Contents:

Info	Communicate over CAN or LIN via NI-XNET hardware with Python.
Author	National Instruments

```
>>> import nixnet
>>> with nixnet.FrameInStreamSession('CAN1') as input_session:
>>>     input_session.intf.can_term = constants.CanTerm.ON
>>>     input_session.intf.baud_rate = 125000

>>>     frames = input_session.frames.read(count)
>>>     for frame in frames:
>>>         print('Received frame:')
>>>         print(frame)
```

CHAPTER 1

Quick Start

Running **nixnet** requires NI-XNET or NI-XNET Runtime. Visit the ni.com/downloads to download the latest version of NI-XNET.

nixnet can be installed with pip:

```
$ python -m pip install nixnet~=0.3.1
```

Now you should be able to move onto the Examples.

CHAPTER 2

Resources

- Documentation.
- Source.

2.1 Product Support

The **nixnet** package and NI-XNET are supported by NI. For support, open a request through the NI support portal at ni.com.

2.2 Bugs / Feature Requests

We welcome all kinds of contributions. If you have a bug to report or a feature request for **nixnet**, feel free to open an issue on Github or contribute the change yourself.

CHAPTER 3

Status

nixnet package is created and maintained by National Instruments.

- **The following support is included:**
  - CAN and LIN protocol
  - Frames, Signals, and frame/signal conversion
  - Database APIs
  - For a complete list of supported features and functions, see the documentation.
- See the enhancement issues for potential future work.
- Breaking API changes will be kept to a minimum. If a breaking change is made, it will be planned through breaking-change isssues and communicated via semver and the release notes.
- Known issues.

nixnet currently supports

- Windows operating system.
- CPython 2.7.0+, 3.4+, PyPy2, and PyPy3.
- NI-XNET 15.5+

CHAPTER 4

License

nixnet is licensed under an MIT-style license (see LICENSE). Other incorporated projects may be licensed under different licenses. All licenses allow for non-commercial and commercial use.

4.1 Installation

Running **nixnet** requires NI-XNET or NI-XNET Runtime. Visit the ni.com/downloads to download the latest version of NI-XNET.

nixnet can be installed with pip:

```
$ python -m pip install nixnet
```

Or **easy_install** from setuptools:

```
$ python -m easy_install nixnet
```

You also can download the project source and run:

```
$ python setup.py install
```

4.2 API Reference

4.2.1 nixnet.session

class nixnet.session.**FrameInStreamSession**(*interface_name*, *database_name=':memory:'*, *cluster_name=''*)

Bases: `nixnet._session.base.SessionBase`

Frame Input Stream session.

This session reads all frames received from the network using a single stream.

The input data is returned as a list of frames. Because all frames are returned, your application must evaluate identification in each frame (such as a CAN identifier or FlexRay slot/cycle/channel) to interpret the frame payload data.

Previously, you could use only one Frame Input Stream session for a given interface. Now, multiple Frame Input Stream sessions can be open at the same time on CAN and LIN interfaces.

While using one or more Frame Input Stream sessions, you can use other sessions with different input modes. Received frames are copied to Frame Input Stream sessions in addition to any other applicable input session. For example, if you create a Frame Input Single-Point session for frame_a, then create a Frame Input Stream session, when frame_a is received, its data is returned from the call to read function of both sessions. This duplication of incoming frames enables you to analyze overall traffic while running a higher level application that uses specific frame or signal data.

When used with a FlexRay interface, frames from both channels are returned. For example, if a frame is received in a static slot on both channel A and channel B, two frames are returned from the read function.

Note: Typical use case: Analyzing and/or logging all frame traffic in the network.

`application_protocol`

This property returns the application protocol that the session uses.

The database used with the session determines the application protocol.

Type *`nixnet._enums.AppProtocol`*

`auto_start`

Automatically starts the output session on the first call to the appropriate write function.

For input sessions, start always is performed within the first call to the appropriate read function (if not already started using *`nixnet._session.base.SessionBase.start`*). This is done because there is no known use case for reading a stopped input session.

For output sessions, as long as the first call to the appropriate write function contains valid data, you can leave this property at its default value of true. If you need to call the appropriate write function multiple times prior to starting the session, or if you are starting multiple sessions simultaneously, you can set this property to false. After calling the appropriate write function as desired, you can call *`nixnet._session.base.SessionBase.start`* to start the session(s).

When automatic start is performed, it is equivalent to *`nixnet._session.base.SessionBase.start`* with scope set to Normal. This starts the session itself, and if the interface is not already started, it starts the interface also.

Type bool

`can_comm`

CAN Communication state

Type *`nixnet.types.CanComm`*

`change_lin_diagnostic_schedule`(*schedule*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the diagnostic schedule.

Parameters **`schedule`** (*`nixnet._enums.LinDiagnosticSchedule`*) – Diagnostic schedule that the LIN master executes.

`change_lin_schedule`(*sched_index*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the running schedule.

According to the LIN protocol, only the master executes schedules, not slaves. If the `nixnet._session.intf.Interface.lin_master` property is false (slave), this write function implicitly sets that property to true (master). If the interface currently is running as a slave, this write returns an error, because it cannot change to master while running.

Parameters **sched_index** (*int*) – Index to the schedule table that the LIN master executes.

The schedule tables are sorted the way they are returned from the database with the *nixnet.database._cluster.Cluster.lin_schedules* property.

check_fault()

Check for an asynchronous fault.

A fault is an error that occurs asynchronously to the NI-XNET application calls. The fault cause may be related to network communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, nxReadState provides a detection method distinct from the status of NI-XNET function calls, yet easy to use alongside the common practice of checking the communication state.

close()

Close (clear) the XNET session.

This function stops communication for the session and releases all resources the session uses. It internally calls `nixnet._session.base.SessionBase.stop` with normal scope, so if this is the last session using the interface, communication stops.

You typically use 'close' when you need to close the existing session to create a new session that uses the same objects. For example, if you create a session for a frame named frame_a using Frame Output Single-Point mode, then you create a second session for frame_a using Frame Output Queued mode, the second call to the session constructor returns an error, because frame_a can be accessed using only one output mode. If you call 'close' before the second constructor call, you can close the previous use of frame_a to create the new session.

cluster_name

This property returns the cluster (network) name used with the session.

Type str

connect_terminals(*source*, *destination*)

Connect terminals on the XNET interface.

This function connects a source terminal to a destination terminal on the interface hardware. The XNET terminal represents an external or internal hardware connection point on a National Instruments XNET hardware product. External terminals include PXI Trigger lines for a PXI card, RTSI terminals for a PCI card, or the single external terminal for a C Series module. Internal terminals include timebases (clocks) and logical entities such as a start trigger.

The terminal inputs use the Terminal I/O names. Typically, one of the pair is an internal and the other an external.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

database_name

This property returns the database name used with the session.

Type str

disconnect_terminals(*source*, *destination*)
Disconnect terminals on the XNET interface.

This function disconnects a specific pair of source/destination terminals previously connected with *nixnet._session.base.SessionBase.connect_terminals*.

When the final session for a given interface is cleared, NI-XNET automatically disconnects all terminal connections for that interface. Therefore, 'disconnect_terminals' is not required for most applications.

This function typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using *nixnet._session.base.SessionBase.stop* with the Interface Only scope. Then you can call 'disconnect_terminals' and *nixnet._session.base.SessionBase.connect_terminals* to adjust terminal connections. Finally, you can call *nixnet._session.base.SessionBase.start* with the Interface Only scope to restart the interface.

You can disconnect only a terminal that has been previously connected. Attempting to disconnect a non-connected terminal results in an error.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

flush()
Flushes (empties) all XNET session queues.

With the exception of single-point modes, all sessions use queues to store frames. For input modes, the queues store frame values (or corresponding signal values) that have been received, but not obtained by calling the read function. For output sessions, the queues store frame values provided to write function, but not transmitted successfully.

nixnet._session.base.SessionBase.start and *nixnet._session.base.SessionBase.stop* have no effect on these queues. Use 'flush' to discard all values in the session's queues.

For example, if you call a write function to write three frames, then immediately call *nixnet._session.base.SessionBase.stop*, then call *nixnet._session.base.SessionBase.start* a few seconds later, the three frames transmit. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, no frames transmit.

As another example, if you receive three frames, then call *nixnet._session.base.SessionBase.stop*, the three frames remains in the queue. If you call *nixnet._session.base.SessionBase.start* a few seconds later, then call a read function, you obtain the three frames received earlier, potentially followed by other frames received after calling *nixnet._session.base.SessionBase.start*. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, read function returns only frames received after the calling *nixnet._session.base.SessionBase.start*.

frames
Operate on session's frames

Type *nixnet._session.frames.InFrames*

intf
Returns the Interface configuration object for the session.

Type *nixnet._session.intf.Interface*

j1939

Returns the J1939 configuration object for the session.

Type *nixnet._session.j1939.J1939*

lin_comm

LIN Communication state

Type *nixnet.types.LinComm*

mode

This property returns the mode associated with the session.

For more information, refer to *nixnet._enums.CreateSessionMode*.

Type *nixnet._enums.CreateSessionMode*

num_pend

This property returns the number of values (frames or signals) pending for the session.

For input sessions, this is the number of frame/signal values available to the appropriate read function. If you call the appropriate read function with number to read of this number and timeout of 0.0, the appropriate read function should return this number of values successfully.

For output sessions, this is the number of frames/signal values provided to the appropriate write function but not yet transmitted onto the network.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

num_unused

This property returns the number of values (frames or signals) unused for the session.

If you get this property prior to starting the session, it provides the size of the underlying queue(s). Contrary to the Queue Size property, this value is in number of frames for Frame I/O, not number of bytes; for Signal I/O, it is the number of signal values in both cases. After start, this property returns the queue size minus the *Number of Values Pending* property.

For input sessions, this is the number of frame/signal values unused in the underlying queue(s).

For output sessions, this is the number of frame/signal values you can provide to a subsequent call to the appropriate write function. If you call the appropriate write function with this number of values and timeout of 0.0, it should return success.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

`protocol`
This property returns the protocol that the interface in the session uses.

Type *`nixnet._enums.Protocol`*

`queue_size`
Get or set queue size.

For output sessions, queues store data passed to the appropriate write function and not yet transmitted onto the network. For input sessions, queues store data received from the network and not yet obtained using the appropriate read function.

For most applications, the default queue sizes are sufficient. You can write to this property to override the default. When you write (set) this property, you must do so prior to the first session start. You cannot set this property again after calling *`nixnet._session.base.SessionBase.stop`*.

For signal I/O sessions, this property is the number of signal values stored. This is analogous to the number of values you use with the appropriate read or write function.

For frame I/O sessions, this property is the number of bytes of frame data stored.

For standard CAN or LIN frame I/O sessions, each frame uses exactly 24 bytes. You can use this number to convert the Queue Size (in bytes) to/from the number of frame values.

For CAN FD and FlexRay frame I/O sessions, each frame value size can vary depending on the payload length. For more information, refer to Raw Frame Format.

For Signal I/O XY sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. According to the formulas below, the default queue sizes can be different for each frame. If you read the default Queue Size property for a Signal Input XY session, the largest queue size is returned, so that a call to the appropriate read function of that size can empty all queues. If you read the default Queue Size property for a Signal Output XY session, the smallest queue size is returned, so that a call to the appropriate write function of that size can succeed when all queues are empty. If you write the Queue Size property for a Signal I/O XY session, that size is used for all frames, so you must ensure that it is sufficient for the frame with the fastest transmit time.

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

For Single-Point sessions (signal or frame), this property is ignored. Single-Point sessions always use a value of 1 as the effective queue size.

Type int

`start`(*scope=<StartStopScope.NORMAL: 0>*)
Start communication for the XNET session.

Because the session is started automatically by default, this function is optional. This function is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the *nixnet._session.base.SessionBase.auto_start* property.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can start each logical unit separately. When a session is started, all contained frames or signals are placed in a state where they are ready to communicate. When the interface is started, it takes data from all started sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

If an output session starts before you write data, or you read an input session before it receives a frame, default data is used. For more information, refer to the XNET Frame Default Payload and XNET Signal Default Value properties.

Parameters **scope** (*nixnet._enums.StartStopScope*) – Describes the impact of this operation on the underlying state models for the session and its interface.

state

Session running state.

Type *nixnet._enums.SessionInfoState*

stop(*scope=<StartStopScope.NORMAL: 0>*)

Stop communication for the XNET session.

Because the session is stopped automatically when closed (cleared), this function is optional.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can stop each logical unit separately. When a session is stopped, all contained frames or signals are placed in a state where they are no longer ready to communicate. When the interface is stopped, it no longer takes data from sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

Parameters **scope** (*nixnet._enums.StartStopScope*) – Describes the impact of this operation on the underlying state models for the session and its interface.

time_communicating

Time the interface started communicating.

The time is usually later than `time_start` because the interface must undergo a communication startup procedure.

Type int

time_current

Current interface time.

Type int

time_start

Time the interface was started.

Type int

`wait_for_intf_communicating`(*timeout=10*)

Wait for the interface to begin communication on the network.

If a start trigger is configured for the interface, this first waits for the trigger. Once the interface is started, this waits for the protocol's communication state to transition to a value that indicates communication with remote nodes.

After this wait succeeds, calls to 'read_state' will return:

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_ACTIVE'

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_PASSIVE'

'constants.ReadState.TIME_COMMUNICATING': Valid time for communication (invalid time of 0 prior)

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_intf_remote_wakeup`(*timeout=10*)

Wait for interface remote wakeup.

Wait for the interface to wakeup due to activity by a remote node on the network. This wait is used for CAN, when you set the 'can_tcvr_state' property to 'constants.CanTcvrState.SLEEP'. Although the interface itself is ready to communicate, this places the transceiver into a sleep state. When a remote CAN node transmits a frame, the transceiver wakes up, and communication is restored. This wait detects that remote wakeup.

This wait is used for LIN when you set 'lin_sleep' property to 'constants.LinSleep.REMOTE_SLEEP' or 'constants.LinSleep.LOCAL_SLEEP'. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_transmit_complete`(*timeout=10*)

Wait for transmition to complete.

All frames written for the session have been transmitted on the bus. This condition applies to CAN, LIN, and FlexRay. This condition is state based, and the state is Boolean (true/false).

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`class` `nixnet.session.`**`FrameOutStreamSession`**(*interface_name*, *database_name=':memory:'*, *cluster_name=''*)

Bases: *`nixnet._session.base.SessionBase`*

Frame Output Stream session.

This session transmits an arbitrary sequence of frame values using a single stream. The values are not limited to a single frame in the database, but can transmit any frame.

The data passed to the write frame function is a list of frame values, each of which transmits as soon as possible. Frames transmit sequentially (one after another).

Like Frame Input Stream sessions, you can create more than one Frame Output Stream session for a given interface.

For CAN, frame values transmit on the network based entirely on the time when you call the write frame function. The timing of each frame as specified in the database is ignored. For example, if you provide four frame values to the the write frame function, the first frame value transmits immediately, followed by the next three values transmitted back to back. For this session, the CAN frame payload length in the database is ignored, and the write frame function is always used.

Similarly for LIN, frame values transmit on the network based entirely on the time when you call the write frame function. The timing of each frame as specified in the database is ignored. The LIN frame payload length in the database is ignored, and the write frame function is always used. For LIN, this session/mode is allowed only on the interface as master. If the payload for a frame is empty, only the header part of the frame is transmitted. For a nonempty payload, the header + response for the frame is transmitted. If a frame for transmit is defined in the database (in-memory or otherwise), it is transmitted using its database checksum type. If the frame for transmit is not defined in the database, it is transmitted using enhanced checksum.

This session is not supported for FlexRay.

The frame values for this session are stored in a queue, such that every value provided is transmitted.

`application_protocol`

This property returns the application protocol that the session uses.

The database used with the session determines the application protocol.

Type *`nixnet._enums.AppProtocol`*

`auto_start`

Automatically starts the output session on the first call to the appropriate write function.

For input sessions, start always is performed within the first call to the appropriate read function (if not already started using *`nixnet._session.base.SessionBase.start`*). This is done because there is no known use case for reading a stopped input session.

For output sessions, as long as the first call to the appropriate write function contains valid data, you can leave this property at its default value of true. If you need to call the appropriate write function multiple times prior to starting the session, or if you are starting multiple sessions simultaneously, you can set this property to false. After calling the appropriate write function as desired, you can call *`nixnet._session.base.SessionBase.start`* to start the session(s).

When automatic start is performed, it is equivalent to *`nixnet._session.base.SessionBase.start`* with scope set to Normal. This starts the session itself, and if the interface is not already started, it starts the interface also.

Type bool

`can_comm`

CAN Communication state

Type *`nixnet.types.CanComm`*

`change_lin_diagnostic_schedule`(*schedule*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the diagnostic schedule.

Parameters **`schedule`** (*`nixnet._enums.LinDiagnosticSchedule`*) – Diagnostic schedule that the LIN master executes.

`change_lin_schedule`(*sched_index*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the running schedule.

According to the LIN protocol, only the master executes schedules, not slaves. If the *`nixnet._session.intf.Interface.lin_master`* property is false (slave), this write function implicitly sets that property to true (master). If the interface currently is running as a slave, this write returns an error, because it cannot change to master while running.

Parameters **`sched_index`** (*`int`*) – Index to the schedule table that the LIN master executes.

The schedule tables are sorted the way they are returned from the database with the *nixnet.database._cluster.Cluster.lin_schedules* property.

`check_fault`()

Check for an asynchronous fault.

A fault is an error that occurs asynchronously to the NI-XNET application calls. The fault cause may be related to network communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, nxReadState provides a detection method distinct from the status of NI-XNET function calls, yet easy to use alongside the common practice of checking the communication state.

`close`()

Close (clear) the XNET session.

This function stops communication for the session and releases all resources the session uses. It internally calls *`nixnet._session.base.SessionBase.stop`* with normal scope, so if this is the last session using the interface, communication stops.

You typically use 'close' when you need to close the existing session to create a new session that uses the same objects. For example, if you create a session for a frame named frame_a using Frame Output Single-Point mode, then you create a second session for frame_a using Frame Output Queued mode, the second call to the session constructor returns an error, because frame_a can be accessed using only one output mode. If you call 'close' before the second constructor call, you can close the previous use of frame_a to create the new session.

`cluster_name`

This property returns the cluster (network) name used with the session.

Type str

`connect_terminals`(*source*, *destination*)

Connect terminals on the XNET interface.

This function connects a source terminal to a destination terminal on the interface hardware. The XNET terminal represents an external or internal hardware connection point on a National Instruments XNET hardware product. External terminals include PXI Trigger lines for a PXI card, RTSI terminals for a PCI card, or the single external terminal for a C Series module. Internal terminals include timebases (clocks) and logical entities such as a start trigger.

The terminal inputs use the Terminal I/O names. Typically, one of the pair is an internal and the other an external.

Parameters

- **`source`** (*`str`*) – Connection source name.
- **`destination`** (*`str`*) – Connection destination name.

`database_name`

This property returns the database name used with the session.

Type str

`disconnect_terminals`(*source*, *destination*)

Disconnect terminals on the XNET interface.

This function disconnects a specific pair of source/destination terminals previously connected with *`nixnet._session.base.SessionBase.connect_terminals`*.

When the final session for a given interface is cleared, NI-XNET automatically disconnects all terminal connections for that interface. Therefore, 'disconnect_terminals' is not required for most applications.

This function typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using `nixnet._session.base.SessionBase.stop` with the Interface Only scope. Then you can call 'disconnect_terminals' and `nixnet._session.base.SessionBase.connect_terminals` to adjust terminal connections. Finally, you can call `nixnet._session.base.SessionBase.start` with the Interface Only scope to restart the interface.

You can disconnect only a terminal that has been previously connected. Attempting to disconnect a non-connected terminal results in an error.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

flush()

Flushes (empties) all XNET session queues.

With the exception of single-point modes, all sessions use queues to store frames. For input modes, the queues store frame values (or corresponding signal values) that have been received, but not obtained by calling the read function. For output sessions, the queues store frame values provided to write function, but not transmitted successfully.

`nixnet._session.base.SessionBase.start` and `nixnet._session.base.SessionBase.stop` have no effect on these queues. Use 'flush' to discard all values in the session's queues.

For example, if you call a write function to write three frames, then immediately call `nixnet._session.base.SessionBase.stop`, then call `nixnet._session.base.SessionBase.start` a few seconds later, the three frames transmit. If you call 'flush' between `nixnet._session.base.SessionBase.stop` and `nixnet._session.base.SessionBase.start`, no frames transmit.

As another example, if you receive three frames, then call `nixnet._session.base.SessionBase.stop`, the three frames remains in the queue. If you call `nixnet._session.base.SessionBase.start` a few seconds later, then call a read function, you obtain the three frames received earlier, potentially followed by other frames received after calling `nixnet._session.base.SessionBase.start`. If you call 'flush' between `nixnet._session.base.SessionBase.stop` and `nixnet._session.base.SessionBase.start`, read function returns only frames received after the calling `nixnet._session.base.SessionBase.start`.

frames

Operate on session's frames

Type `nixnet._session.frames.InFrames`

intf

Returns the Interface configuration object for the session.

Type `nixnet._session.intf.Interface`

j1939

Returns the J1939 configuration object for the session.

Type `nixnet._session.j1939.J1939`

lin_comm

LIN Communication state

Type *nixnet.types.LinComm*

mode

This property returns the mode associated with the session.

For more information, refer to *nixnet._enums.CreateSessionMode*.

Type *nixnet._enums.CreateSessionMode*

num_pend

This property returns the number of values (frames or signals) pending for the session.

For input sessions, this is the number of frame/signal values available to the appropriate read function. If you call the appropriate read function with number to read of this number and timeout of 0.0, the appropriate read function should return this number of values successfully.

For output sessions, this is the number of frames/signal values provided to the appropriate write function but not yet transmitted onto the network.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

num_unused

This property returns the number of values (frames or signals) unused for the session.

If you get this property prior to starting the session, it provides the size of the underlying queue(s). Contrary to the Queue Size property, this value is in number of frames for Frame I/O, not number of bytes; for Signal I/O, it is the number of signal values in both cases. After start, this property returns the queue size minus the *Number of Values Pending* property.

For input sessions, this is the number of frame/signal values unused in the underlying queue(s).

For output sessions, this is the number of frame/signal values you can provide to a subsequent call to the appropriate write function. If you call the appropriate write function with this number of values and timeout of 0.0, it should return success.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

protocol
This property returns the protocol that the interface in the session uses.

Type *nixnet._enums.Protocol*

queue_size
Get or set queue size.

For output sessions, queues store data passed to the appropriate write function and not yet transmitted onto the network. For input sessions, queues store data received from the network and not yet obtained using the appropriate read function.

For most applications, the default queue sizes are sufficient. You can write to this property to override the default. When you write (set) this property, you must do so prior to the first session start. You cannot set this property again after calling *nixnet._session.base.SessionBase.stop*.

For signal I/O sessions, this property is the number of signal values stored. This is analogous to the number of values you use with the appropriate read or write function.

For frame I/O sessions, this property is the number of bytes of frame data stored.

For standard CAN or LIN frame I/O sessions, each frame uses exactly 24 bytes. You can use this number to convert the Queue Size (in bytes) to/from the number of frame values.

For CAN FD and FlexRay frame I/O sessions, each frame value size can vary depending on the payload length. For more information, refer to Raw Frame Format.

For Signal I/O XY sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. According to the formulas below, the default queue sizes can be different for each frame. If you read the default Queue Size property for a Signal Input XY session, the largest queue size is returned, so that a call to the appropriate read function of that size can empty all queues. If you read the default Queue Size property for a Signal Output XY session, the smallest queue size is returned, so that a call to the appropriate write function of that size can succeed when all queues are empty. If you write the Queue Size property for a Signal I/O XY session, that size is used for all frames, so you must ensure that it is sufficient for the frame with the fastest transmit time.

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

For Single-Point sessions (signal or frame), this property is ignored. Single-Point sessions always use a value of 1 as the effective queue size.

Type int

start(*scope=<StartStopScope.NORMAL: 0>*)
Start communication for the XNET session.

Because the session is started automatically by default, this function is optional. This function is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the *nixnet._session.base.SessionBase.auto_start* property.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can start each logical unit separately. When a session is started, all contained frames or signals are placed in a state where they are ready to communicate. When the interface is started, it takes data from all started sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

If an output session starts before you write data, or you read an input session before it receives a frame, default data is used. For more information, refer to the XNET Frame Default Payload and XNET Signal Default Value properties.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`state`

Session running state.

Type *`nixnet._enums.SessionInfoState`*

`stop`(*scope=<StartStopScope.NORMAL: 0>*)

Stop communication for the XNET session.

Because the session is stopped automatically when closed (cleared), this function is optional.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can stop each logical unit separately. When a session is stopped, all contained frames or signals are placed in a state where they are no longer ready to communicate. When the interface is stopped, it no longer takes data from sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`time_communicating`

Time the interface started communicating.

The time is usually later than `time_start` because the interface must undergo a communication startup procedure.

Type int

`time_current`

Current interface time.

Type int

`time_start`

Time the interface was started.

Type int

`wait_for_intf_communicating`(*timeout=10*)

Wait for the interface to begin communication on the network.

If a start trigger is configured for the interface, this first waits for the trigger. Once the interface is started, this waits for the protocol's communication state to transition to a value that indicates communication with remote nodes.

After this wait succeeds, calls to 'read_state' will return:

nixnet._enums.CanCommState: 'constants.CAN_COMM.ERROR_ACTIVE'

nixnet._enums.CanCommState: 'constants.CAN_COMM.ERROR_PASSIVE'

'constants.ReadState.TIME_COMMUNICATING': Valid time for communication (invalid time of 0 prior)

Parameters **timeout** (*float*) – The maximum amount of time to wait in seconds.

wait_for_intf_remote_wakeup(*timeout=10*)

Wait for interface remote wakeup.

Wait for the interface to wakeup due to activity by a remote node on the network. This wait is used for CAN, when you set the 'can_tcvr_state' property to 'constants.CanTcvrState.SLEEP'. Although the interface itself is ready to communicate, this places the transceiver into a sleep state. When a remote CAN node transmits a frame, the transceiver wakes up, and communication is restored. This wait detects that remote wakeup.

This wait is used for LIN when you set 'lin_sleep' property to 'constants.LinSleep.REMOTE_SLEEP' or 'constants.LinSleep.LOCAL_SLEEP'. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

Parameters **timeout** (*float*) – The maximum amount of time to wait in seconds.

wait_for_transmit_complete(*timeout=10*)

Wait for transmition to complete.

All frames written for the session have been transmitted on the bus. This condition applies to CAN, LIN, and FlexRay. This condition is state based, and the state is Boolean (true/false).

Parameters **timeout** (*float*) – The maximum amount of time to wait in seconds.

class nixnet.session.**FrameInQueuedSession**(*interface_name*, *database_name*, *cluster_name*, *frame*)

Bases: *nixnet._session.base.SessionBase*

Frame Input Queued session.

This session reads data from a dedicated queue per frame. It enables your application to read a sequence of data specific to a frame (for example, a CAN identifier).

You specify only one frame for the session, and the read frame function returns values for that frame only. If you need sequential data for multiple frames, create multiple sessions, one per frame.

The input data is returned as a list of frame values. These values represent all values received for the frame since the previous call to the read frame function.

application_protocol

This property returns the application protocol that the session uses.

The database used with the session determines the application protocol.

Type *nixnet._enums.AppProtocol*

auto_start

Automatically starts the output session on the first call to the appropriate write function.

For input sessions, start always is performed within the first call to the appropriate read function (if not already started using `nixnet._session.base.SessionBase.start`). This is done because there is no known use case for reading a stopped input session.

For output sessions, as long as the first call to the appropriate write function contains valid data, you can leave this property at its default value of true. If you need to call the appropriate write function multiple times prior to starting the session, or if you are starting multiple sessions simultaneously, you can set this property to false. After calling the appropriate write function as desired, you can call `nixnet._session.base.SessionBase.start` to start the session(s).

When automatic start is performed, it is equivalent to `nixnet._session.base.SessionBase.start` with scope set to Normal. This starts the session itself, and if the interface is not already started, it starts the interface also.

Type bool

`can_comm`

CAN Communication state

Type `nixnet.types.CanComm`

`change_lin_diagnostic_schedule`(*schedule*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the diagnostic schedule.

Parameters **`schedule`** (`nixnet._enums.LinDiagnosticSchedule`) – Diagnostic schedule that the LIN master executes.

`change_lin_schedule`(*sched_index*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the running schedule.

According to the LIN protocol, only the master executes schedules, not slaves. If the `nixnet._session.intf.Interface.lin_master` property is false (slave), this write function implicitly sets that property to true (master). If the interface currently is running as a slave, this write returns an error, because it cannot change to master while running.

Parameters **`sched_index`** (`int`) – Index to the schedule table that the LIN master executes.

The schedule tables are sorted the way they are returned from the database with the *nixnet.database._cluster.Cluster.lin_schedules* property.

`check_fault`()

Check for an asynchronous fault.

A fault is an error that occurs asynchronously to the NI-XNET application calls. The fault cause may be related to network communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, nxReadState provides a detection method distinct from the status of NI-XNET function calls, yet easy to use alongside the common practice of checking the communication state.

`close`()

Close (clear) the XNET session.

This function stops communication for the session and releases all resources the session uses. It internally calls `nixnet._session.base.SessionBase.stop` with normal scope, so if this is the last session using the interface, communication stops.

You typically use 'close' when you need to close the existing session to create a new session that uses the same objects. For example, if you create a session for a frame named frame_a using Frame Output Single-Point mode, then you create a second session for frame_a using Frame Output Queued mode, the second call to the session constructor returns an error, because frame_a can be accessed using only one output mode. If you call 'close' before the second constructor call, you can close the previous use of frame_a to create the new session.

`cluster_name`

This property returns the cluster (network) name used with the session.

Type str

`connect_terminals`(*source*, *destination*)

Connect terminals on the XNET interface.

This function connects a source terminal to a destination terminal on the interface hardware. The XNET terminal represents an external or internal hardware connection point on a National Instruments XNET hardware product. External terminals include PXI Trigger lines for a PXI card, RTSI terminals for a PCI card, or the single external terminal for a C Series module. Internal terminals include timebases (clocks) and logical entities such as a start trigger.

The terminal inputs use the Terminal I/O names. Typically, one of the pair is an internal and the other an external.

Parameters

- **`source`** (`str`) – Connection source name.
- **`destination`** (`str`) – Connection destination name.

`database_name`

This property returns the database name used with the session.

Type str

`disconnect_terminals`(*source*, *destination*)

Disconnect terminals on the XNET interface.

This function disconnects a specific pair of source/destination terminals previously connected with `nixnet._session.base.SessionBase.connect_terminals`.

When the final session for a given interface is cleared, NI-XNET automatically disconnects all terminal connections for that interface. Therefore, 'disconnect_terminals' is not required for most applications.

This function typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using `nixnet._session.base.SessionBase.stop` with the Interface Only scope. Then you can call 'disconnect_terminals' and `nixnet._session.base.SessionBase.connect_terminals` to adjust terminal connections. Finally, you can call `nixnet._session.base.SessionBase.start` with the Interface Only scope to restart the interface.

You can disconnect only a terminal that has been previously connected. Attempting to disconnect a nonconnected terminal results in an error.

Parameters

- **`source`** (`str`) – Connection source name.
- **`destination`** (`str`) – Connection destination name.

`flush`()

Flushes (empties) all XNET session queues.

With the exception of single-point modes, all sessions use queues to store frames. For input modes, the queues store frame values (or corresponding signal values) that have been received, but not obtained by calling the read function. For output sessions, the queues store frame values provided to write function, but not transmitted successfully.

nixnet._session.base.SessionBase.start and *nixnet._session.base.SessionBase.stop* have no effect on these queues. Use 'flush' to discard all values in the session's queues.

For example, if you call a write function to write three frames, then immediately call *nixnet._session.base.SessionBase.stop*, then call *nixnet._session.base.SessionBase.start* a few seconds later, the three frames transmit. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, no frames transmit.

As another example, if you receive three frames, then call *nixnet._session.base.SessionBase.stop*, the three frames remains in the queue. If you call *nixnet._session.base.SessionBase.start* a few seconds later, then call a read function, you obtain the three frames received earlier, potentially followed by other frames received after calling *nixnet._session.base.SessionBase.start*. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, read function returns only frames received after the calling *nixnet._session.base.SessionBase.start*.

frames
Operate on session's frames

Type *nixnet._session.frames.InFrames*

intf
Returns the Interface configuration object for the session.

Type *nixnet._session.intf.Interface*

j1939
Returns the J1939 configuration object for the session.

Type *nixnet._session.j1939.J1939*

lin_comm
LIN Communication state

Type *nixnet.types.LinComm*

mode
This property returns the mode associated with the session.

For more information, refer to *nixnet._enums.CreateSessionMode*.

Type *nixnet._enums.CreateSessionMode*

num_pend
This property returns the number of values (frames or signals) pending for the session.

For input sessions, this is the number of frame/signal values available to the appropriate read function. If you call the appropriate read function with number to read of this number and timeout of 0.0, the appropriate read function should return this number of values successfully.

For output sessions, this is the number of frames/signal values provided to the appropriate write function but not yet transmitted onto the network.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

`num_unused`

This property returns the number of values (frames or signals) unused for the session.

If you get this property prior to starting the session, it provides the size of the underlying queue(s). Contrary to the Queue Size property, this value is in number of frames for Frame I/O, not number of bytes; for Signal I/O, it is the number of signal values in both cases. After start, this property returns the queue size minus the *`Number of Values Pending`* property.

For input sessions, this is the number of frame/signal values unused in the underlying queue(s).

For output sessions, this is the number of frame/signal values you can provide to a subsequent call to the appropriate write function. If you call the appropriate write function with this number of values and timeout of 0.0, it should return success.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

`protocol`

This property returns the protocol that the interface in the session uses.

Type *`nixnet._enums.Protocol`*

`queue_size`

Get or set queue size.

For output sessions, queues store data passed to the appropriate write function and not yet transmitted onto the network. For input sessions, queues store data received from the network and not yet obtained using the appropriate read function.

For most applications, the default queue sizes are sufficient. You can write to this property to override the default. When you write (set) this property, you must do so prior to the first session start. You cannot set this property again after calling *`nixnet._session.base.SessionBase.stop`*.

For signal I/O sessions, this property is the number of signal values stored. This is analogous to the number of values you use with the appropriate read or write function.

For frame I/O sessions, this property is the number of bytes of frame data stored.

For standard CAN or LIN frame I/O sessions, each frame uses exactly 24 bytes. You can use this number to convert the Queue Size (in bytes) to/from the number of frame values.

For CAN FD and FlexRay frame I/O sessions, each frame value size can vary depending on the payload length. For more information, refer to Raw Frame Format.

For Signal I/O XY sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. According to the formulas below, the default queue sizes can be different for each frame. If you read the default Queue Size property for a Signal Input XY session, the largest queue size is returned, so that a call to the appropriate read function of that size can empty all queues. If you read the default Queue Size property for a Signal Output XY session, the smallest queue size is returned, so that a call to the appropriate write function of that size can succeed when all queues are empty. If you write the Queue Size property for a Signal I/O XY session, that size is used for all frames, so you must ensure that it is sufficient for the frame with the fastest transmit time.

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

For Single-Point sessions (signal or frame), this property is ignored. Single-Point sessions always use a value of 1 as the effective queue size.

Type int

`start`(*scope=<StartStopScope.NORMAL: 0>*)
Start communication for the XNET session.

Because the session is started automatically by default, this function is optional. This function is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the `nixnet._session.base.SessionBase.auto_start` property.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can start each logical unit separately. When a session is started, all contained frames or signals are placed in a state where they are ready to communicate. When the interface is started, it takes data from all started sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

If an output session starts before you write data, or you read an input session before it receives a frame, default data is used. For more information, refer to the XNET Frame Default Payload and XNET Signal Default Value properties.

Parameters **`scope`** (`nixnet._enums.StartStopScope`) – Describes the impact of this operation on the underlying state models for the session and its interface.

`state`
Session running state.

Type `nixnet._enums.SessionInfoState`

`stop`(*scope=<StartStopScope.NORMAL: 0>*)
Stop communication for the XNET session.

Because the session is stopped automatically when closed (cleared), this function is optional.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can stop each logical unit separately. When a session is stopped, all contained frames or signals are placed in a state where they are no longer ready to communicate. When the interface is stopped, it no longer takes data from sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`time_communicating`
Time the interface started communicating.

The time is usually later than `time_start` because the interface must undergo a communication startup procedure.

Type int

`time_current`
Current interface time.

Type int

`time_start`
Time the interface was started.

Type int

`wait_for_intf_communicating`(*timeout=10*)
Wait for the interface to begin communication on the network.

If a start trigger is configured for the interface, this first waits for the trigger. Once the interface is started, this waits for the protocol's communication state to transition to a value that indicates communication with remote nodes.

After this wait succeeds, calls to 'read_state' will return:

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_ACTIVE'

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_PASSIVE'

'constants.ReadState.TIME_COMMUNICATING': Valid time for communication (invalid time of 0 prior)

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_intf_remote_wakeup`(*timeout=10*)
Wait for interface remote wakeup.

Wait for the interface to wakeup due to activity by a remote node on the network. This wait is used for CAN, when you set the 'can_tcvr_state' property to 'constants.CanTcvrState.SLEEP'. Although the interface itself is ready to communicate, this places the transceiver into a sleep state. When a remote CAN node transmits a frame, the transceiver wakes up, and communication is restored. This wait detects that remote wakeup.

This wait is used for LIN when you set 'lin_sleep' property to 'constants.LinSleep.REMOTE_SLEEP' or 'constants.LinSleep.LOCAL_SLEEP'. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

Parameters **timeout** (*float*) – The maximum amount of time to wait in seconds.

wait_for_transmit_complete(*timeout=10*)

Wait for transmition to complete.

All frames written for the session have been transmitted on the bus. This condition applies to CAN, LIN, and FlexRay. This condition is state based, and the state is Boolean (true/false).

Parameters **timeout** (*float*) – The maximum amount of time to wait in seconds.

class nixnet.session.**FrameOutQueuedSession**(*interface_name*, *database_name*, *cluster_name*, *frame*)

Bases: *nixnet._session.base.SessionBase*

Frame Output Queued session.

This session provides a sequence of values for a single frame, for transmit using that frame's timing as specified in the database.

The output data is provided as a list of frame values, to be transmitted sequentially for the frame specified in the session.

You can only specify one frame for this session. To transmit sequential values for multiple frames, use a different Frame Output Queued session for each frame or use the Frame Output Stream session.

The frame values for this session are stored in a queue, such that every value provided is transmitted.

For this session, NI-XNET transmits each frame according to its properties in the database. Therefore, when you call the write frame function, the number of payload bytes in each frame value must match that frame's Payload Length property. The other frame value elements are ignored, so you can leave them uninitialized. For CAN interfaces, if the number of payload bytes you write is smaller than the Payload Length configured in the database, the requested number of bytes transmits. If the number of payload bytes is larger than the Payload Length configured in the database, the queue is flushed and no frames transmit. For other interfaces, transmitting a number of payload bytes different than the frame's payload may cause unexpected results on the bus.

application_protocol

This property returns the application protocol that the session uses.

The database used with the session determines the application protocol.

Type *nixnet._enums.AppProtocol*

auto_start

Automatically starts the output session on the first call to the appropriate write function.

For input sessions, start always is performed within the first call to the appropriate read function (if not already started using *nixnet._session.base.SessionBase.start*). This is done because there is no known use case for reading a stopped input session.

For output sessions, as long as the first call to the appropriate write function contains valid data, you can leave this property at its default value of true. If you need to call the appropriate write function multiple times prior to starting the session, or if you are starting multiple sessions simultaneously, you can set this property to false. After calling the appropriate write function as desired, you can call *nixnet._session.base.SessionBase.start* to start the session(s).

When automatic start is performed, it is equivalent to *nixnet._session.base.SessionBase.start* with scope set to Normal. This starts the session itself, and if the interface is not already started, it starts the interface also.

Type bool

can_comm

CAN Communication state

Type `nixnet.types.CanComm`

`change_lin_diagnostic_schedule`(*schedule*)
Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the diagnostic schedule.

Parameters **`schedule`** (`nixnet._enums.LinDiagnosticSchedule`) – Diagnostic schedule that the LIN master executes.

`change_lin_schedule`(*sched_index*)
Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the running schedule.

According to the LIN protocol, only the master executes schedules, not slaves. If the `nixnet._session.intf.Interface.lin_master` property is false (slave), this write function implicitly sets that property to true (master). If the interface currently is running as a slave, this write returns an error, because it cannot change to master while running.

Parameters **`sched_index`** (`int`) – Index to the schedule table that the LIN master executes.

The schedule tables are sorted the way they are returned from the database with the *nixnet.database._cluster.Cluster.lin_schedules* property.

`check_fault`()
Check for an asynchronous fault.

A fault is an error that occurs asynchronously to the NI-XNET application calls. The fault cause may be related to network communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, nxReadState provides a detection method distinct from the status of NI-XNET function calls, yet easy to use alongside the common practice of checking the communication state.

`close`()
Close (clear) the XNET session.

This function stops communication for the session and releases all resources the session uses. It internally calls `nixnet._session.base.SessionBase.stop` with normal scope, so if this is the last session using the interface, communication stops.

You typically use 'close' when you need to close the existing session to create a new session that uses the same objects. For example, if you create a session for a frame named frame_a using Frame Output Single-Point mode, then you create a second session for frame_a using Frame Output Queued mode, the second call to the session constructor returns an error, because frame_a can be accessed using only one output mode. If you call 'close' before the second constructor call, you can close the previous use of frame_a to create the new session.

`cluster_name`
This property returns the cluster (network) name used with the session.

Type str

`connect_terminals`(*source*, *destination*)
Connect terminals on the XNET interface.

This function connects a source terminal to a destination terminal on the interface hardware. The XNET terminal represents an external or internal hardware connection point on a National Instruments XNET hardware product. External terminals include PXI Trigger lines for a PXI card, RTSI terminals for a PCI card, or the single external terminal for a C Series module. Internal terminals include timebases (clocks) and logical entities such as a start trigger.

The terminal inputs use the Terminal I/O names. Typically, one of the pair is an internal and the other an external.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

database_name

This property returns the database name used with the session.

Type str

disconnect_terminals(*source*, *destination*)

Disconnect terminals on the XNET interface.

This function disconnects a specific pair of source/destination terminals previously connected with *nixnet._session.base.SessionBase.connect_terminals*.

When the final session for a given interface is cleared, NI-XNET automatically disconnects all terminal connections for that interface. Therefore, 'disconnect_terminals' is not required for most applications.

This function typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using *nixnet._session.base.SessionBase.stop* with the Interface Only scope. Then you can call 'disconnect_terminals' and *nixnet._session.base.SessionBase.connect_terminals* to adjust terminal connections. Finally, you can call *nixnet._session.base.SessionBase.start* with the Interface Only scope to restart the interface.

You can disconnect only a terminal that has been previously connected. Attempting to disconnect a non-connected terminal results in an error.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

flush()

Flushes (empties) all XNET session queues.

With the exception of single-point modes, all sessions use queues to store frames. For input modes, the queues store frame values (or corresponding signal values) that have been received, but not obtained by calling the read function. For output sessions, the queues store frame values provided to write function, but not transmitted successfully.

nixnet._session.base.SessionBase.start and *nixnet._session.base.SessionBase.stop* have no effect on these queues. Use 'flush' to discard all values in the session's queues.

For example, if you call a write function to write three frames, then immediately call *nixnet._session.base.SessionBase.stop*, then call *nixnet._session.base.SessionBase.start* a few seconds later, the three frames transmit. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, no frames transmit.

As another example, if you receive three frames, then call *nixnet._session.base.SessionBase.stop*, the three frames remains in the queue. If you call *nixnet._session.base.SessionBase.start* a few seconds later, then call a read function, you obtain the three frames received earlier, potentially followed by other frames received after calling *nixnet._session.base.SessionBase.start*. If you call 'flush' between *nixnet._session.base.SessionBase.*

stop and *nixnet._session.base.SessionBase.start*, read function returns only frames received after the calling *nixnet._session.base.SessionBase.start*.

frames
Operate on session's frames

Type *nixnet._session.frames.OutFrames*

intf
Returns the Interface configuration object for the session.

Type *nixnet._session.intf.Interface*

j1939
Returns the J1939 configuration object for the session.

Type *nixnet._session.j1939.J1939*

lin_comm
LIN Communication state

Type *nixnet.types.LinComm*

mode
This property returns the mode associated with the session.

For more information, refer to *nixnet._enums.CreateSessionMode*.

Type *nixnet._enums.CreateSessionMode*

num_pend
This property returns the number of values (frames or signals) pending for the session.

For input sessions, this is the number of frame/signal values available to the appropriate read function. If you call the appropriate read function with number to read of this number and timeout of 0.0, the appropriate read function should return this number of values successfully.

For output sessions, this is the number of frames/signal values provided to the appropriate write function but not yet transmitted onto the network.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

num_unused
This property returns the number of values (frames or signals) unused for the session.

If you get this property prior to starting the session, it provides the size of the underlying queue(s). Contrary to the Queue Size property, this value is in number of frames for Frame I/O, not number of bytes; for Signal I/O, it is the number of signal values in both cases. After start, this property returns the queue size minus the *Number of Values Pending* property.

For input sessions, this is the number of frame/signal values unused in the underlying queue(s).

For output sessions, this is the number of frame/signal values you can provide to a subsequent call to the appropriate write function. If you call the appropriate write function with this number of values and timeout of 0.0, it should return success.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

`protocol`

This property returns the protocol that the interface in the session uses.

Type *`nixnet._enums.Protocol`*

`queue_size`

Get or set queue size.

For output sessions, queues store data passed to the appropriate write function and not yet transmitted onto the network. For input sessions, queues store data received from the network and not yet obtained using the appropriate read function.

For most applications, the default queue sizes are sufficient. You can write to this property to override the default. When you write (set) this property, you must do so prior to the first session start. You cannot set this property again after calling *`nixnet._session.base.SessionBase.stop`*.

For signal I/O sessions, this property is the number of signal values stored. This is analogous to the number of values you use with the appropriate read or write function.

For frame I/O sessions, this property is the number of bytes of frame data stored.

For standard CAN or LIN frame I/O sessions, each frame uses exactly 24 bytes. You can use this number to convert the Queue Size (in bytes) to/from the number of frame values.

For CAN FD and FlexRay frame I/O sessions, each frame value size can vary depending on the payload length. For more information, refer to Raw Frame Format.

For Signal I/O XY sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. According to the formulas below, the default queue sizes can be different for each frame. If you read the default Queue Size property for a Signal Input XY session, the largest queue size is returned, so that a call to the appropriate read function of that size can empty all queues. If you read the default Queue Size property for a Signal Output XY session, the smallest queue size is returned, so that a call to the appropriate write function of that size can succeed when all queues are empty. If you write the Queue Size property for a Signal I/O XY session, that size is used for all frames, so you must ensure that it is sufficient for the frame with the fastest transmit time.

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

For Single-Point sessions (signal or frame), this property is ignored. Single-Point sessions always use a value of 1 as the effective queue size.

Type int

`start`(*scope=<StartStopScope.NORMAL: 0>*)

Start communication for the XNET session.

Because the session is started automatically by default, this function is optional. This function is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the *`nixnet._session.base.SessionBase.auto_start`* property.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can start each logical unit separately. When a session is started, all contained frames or signals are placed in a state where they are ready to communicate. When the interface is started, it takes data from all started sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

If an output session starts before you write data, or you read an input session before it receives a frame, default data is used. For more information, refer to the XNET Frame Default Payload and XNET Signal Default Value properties.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`state`

Session running state.

Type *`nixnet._enums.SessionInfoState`*

`stop`(*scope=<StartStopScope.NORMAL: 0>*)

Stop communication for the XNET session.

Because the session is stopped automatically when closed (cleared), this function is optional.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can stop each logical unit separately. When a session is stopped, all contained frames or signals are placed in a state where they are no longer ready to communicate. When the interface is stopped, it no longer takes data from sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`time_communicating`

Time the interface started communicating.

The time is usually later than `time_start` because the interface must undergo a communication startup procedure.

Type int

`time_current`
Current interface time.

Type int

`time_start`
Time the interface was started.

Type int

`wait_for_intf_communicating`(*timeout=10*)
Wait for the interface to begin communication on the network.

If a start trigger is configured for the interface, this first waits for the trigger. Once the interface is started, this waits for the protocol's communication state to transition to a value that indicates communication with remote nodes.

After this wait succeeds, calls to 'read_state' will return:

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_ACTIVE'

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_PASSIVE'

'constants.ReadState.TIME_COMMUNICATING': Valid time for communication (invalid time of 0 prior)

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_intf_remote_wakeup`(*timeout=10*)
Wait for interface remote wakeup.

Wait for the interface to wakeup due to activity by a remote node on the network. This wait is used for CAN, when you set the 'can_tcvr_state' property to 'constants.CanTcvrState.SLEEP'. Although the interface itself is ready to communicate, this places the transceiver into a sleep state. When a remote CAN node transmits a frame, the transceiver wakes up, and communication is restored. This wait detects that remote wakeup.

This wait is used for LIN when you set 'lin_sleep' property to 'constants.LinSleep.REMOTE_SLEEP' or 'constants.LinSleep.LOCAL_SLEEP'. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_transmit_complete`(*timeout=10*)
Wait for transmition to complete.

All frames written for the session have been transmitted on the bus. This condition applies to CAN, LIN, and FlexRay. This condition is state based, and the state is Boolean (true/false).

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

class `nixnet.session.`**`FrameInSinglePointSession`**(*interface_name*, *database_name*, *cluster_name*, *frames*)
Bases: *`nixnet._session.base.SessionBase`*

Frame Input Single-Point session.

This session reads the most recent value received for each frame.

This session does not use queues to store each received frame. If the interface receives two frames prior to calling the read frame function, that read returns signals for the second frame.

The input data is returned as a list of frames, one for each frame specified for the session.

Note: Typical use case: Control or simulation applications that require lower level access to frames (not signals).

`application_protocol`

This property returns the application protocol that the session uses.

The database used with the session determines the application protocol.

Type *`nixnet._enums.AppProtocol`*

`auto_start`

Automatically starts the output session on the first call to the appropriate write function.

For input sessions, start always is performed within the first call to the appropriate read function (if not already started using *`nixnet._session.base.SessionBase.start`*). This is done because there is no known use case for reading a stopped input session.

For output sessions, as long as the first call to the appropriate write function contains valid data, you can leave this property at its default value of true. If you need to call the appropriate write function multiple times prior to starting the session, or if you are starting multiple sessions simultaneously, you can set this property to false. After calling the appropriate write function as desired, you can call *`nixnet._session.base.SessionBase.start`* to start the session(s).

When automatic start is performed, it is equivalent to *`nixnet._session.base.SessionBase.start`* with scope set to Normal. This starts the session itself, and if the interface is not already started, it starts the interface also.

Type bool

`can_comm`

CAN Communication state

Type *`nixnet.types.CanComm`*

`change_lin_diagnostic_schedule`(*schedule*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the diagnostic schedule.

Parameters **`schedule`** (*`nixnet._enums.LinDiagnosticSchedule`*) – Diagnostic schedule that the LIN master executes.

`change_lin_schedule`(*sched_index*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the running schedule.

According to the LIN protocol, only the master executes schedules, not slaves. If the *`nixnet._session.intf.Interface.lin_master`* property is false (slave), this write function implicitly sets that property to true (master). If the interface currently is running as a slave, this write returns an error, because it cannot change to master while running.

Parameters **`sched_index`** (*`int`*) – Index to the schedule table that the LIN master executes.

The schedule tables are sorted the way they are returned from the database with the *nixnet.database._cluster.Cluster.lin_schedules* property.

`check_fault`()

Check for an asynchronous fault.

A fault is an error that occurs asynchronously to the NI-XNET application calls. The fault cause may be related to network communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, nxReadState provides a detection method distinct from the status of NI-XNET function calls, yet easy to use alongside the common practice of checking the communication state.

`close`()

Close (clear) the XNET session.

This function stops communication for the session and releases all resources the session uses. It internally calls *`nixnet._session.base.SessionBase.stop`* with normal scope, so if this is the last session using the interface, communication stops.

You typically use 'close' when you need to close the existing session to create a new session that uses the same objects. For example, if you create a session for a frame named frame_a using Frame Output Single-Point mode, then you create a second session for frame_a using Frame Output Queued mode, the second call to the session constructor returns an error, because frame_a can be accessed using only one output mode. If you call 'close' before the second constructor call, you can close the previous use of frame_a to create the new session.

`cluster_name`

This property returns the cluster (network) name used with the session.

Type str

`connect_terminals`(*source*, *destination*)

Connect terminals on the XNET interface.

This function connects a source terminal to a destination terminal on the interface hardware. The XNET terminal represents an external or internal hardware connection point on a National Instruments XNET hardware product. External terminals include PXI Trigger lines for a PXI card, RTSI terminals for a PCI card, or the single external terminal for a C Series module. Internal terminals include timebases (clocks) and logical entities such as a start trigger.

The terminal inputs use the Terminal I/O names. Typically, one of the pair is an internal and the other an external.

Parameters

- **`source`** (*`str`*) – Connection source name.
- **`destination`** (*`str`*) – Connection destination name.

`database_name`

This property returns the database name used with the session.

Type str

`disconnect_terminals`(*source*, *destination*)

Disconnect terminals on the XNET interface.

This function disconnects a specific pair of source/destination terminals previously connected with *`nixnet._session.base.SessionBase.connect_terminals`*.

When the final session for a given interface is cleared, NI-XNET automatically disconnects all terminal connections for that interface. Therefore, 'disconnect_terminals' is not required for most applications.

This function typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using *`nixnet._session.base.SessionBase.stop`* with the Interface Only scope. Then you can call 'disconnect_terminals' and *nixnet._session.base.SessionBase.connect_terminals* to adjust terminal connections. Finally, you can call *nixnet._session.base.SessionBase.start* with the Interface Only scope to restart the interface.

You can disconnect only a terminal that has been previously connected. Attempting to disconnect a nonconnected terminal results in an error.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

flush()

Flushes (empties) all XNET session queues.

With the exception of single-point modes, all sessions use queues to store frames. For input modes, the queues store frame values (or corresponding signal values) that have been received, but not obtained by calling the read function. For output sessions, the queues store frame values provided to write function, but not transmitted successfully.

nixnet._session.base.SessionBase.start and *nixnet._session.base.SessionBase.stop* have no effect on these queues. Use 'flush' to discard all values in the session's queues.

For example, if you call a write function to write three frames, then immediately call *nixnet._session.base.SessionBase.stop*, then call *nixnet._session.base.SessionBase.start* a few seconds later, the three frames transmit. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, no frames transmit.

As another example, if you receive three frames, then call *nixnet._session.base.SessionBase.stop*, the three frames remains in the queue. If you call *nixnet._session.base.SessionBase.start* a few seconds later, then call a read function, you obtain the three frames received earlier, potentially followed by other frames received after calling *nixnet._session.base.SessionBase.start*. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, read function returns only frames received after the calling *nixnet._session.base.SessionBase.start*.

frames

Operate on session's frames

Type *nixnet._session.frames.InFrames*

intf

Returns the Interface configuration object for the session.

Type *nixnet._session.intf.Interface*

j1939

Returns the J1939 configuration object for the session.

Type *nixnet._session.j1939.J1939*

lin_comm

LIN Communication state

Type *nixnet.types.LinComm*

mode

This property returns the mode associated with the session.

For more information, refer to *nixnet._enums.CreateSessionMode*.

Type *nixnet._enums.CreateSessionMode*

num_pend

This property returns the number of values (frames or signals) pending for the session.

For input sessions, this is the number of frame/signal values available to the appropriate read function. If you call the appropriate read function with number to read of this number and timeout of 0.0, the appropriate read function should return this number of values successfully.

For output sessions, this is the number of frames/signal values provided to the appropriate write function but not yet transmitted onto the network.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

num_unused

This property returns the number of values (frames or signals) unused for the session.

If you get this property prior to starting the session, it provides the size of the underlying queue(s). Contrary to the Queue Size property, this value is in number of frames for Frame I/O, not number of bytes; for Signal I/O, it is the number of signal values in both cases. After start, this property returns the queue size minus the *Number of Values Pending* property.

For input sessions, this is the number of frame/signal values unused in the underlying queue(s).

For output sessions, this is the number of frame/signal values you can provide to a subsequent call to the appropriate write function. If you call the appropriate write function with this number of values and timeout of 0.0, it should return success.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

protocol

This property returns the protocol that the interface in the session uses.

Type *nixnet._enums.Protocol*

queue_size

Get or set queue size.

For output sessions, queues store data passed to the appropriate write function and not yet transmitted onto the network. For input sessions, queues store data received from the network and not yet obtained using the appropriate read function.

For most applications, the default queue sizes are sufficient. You can write to this property to override the default. When you write (set) this property, you must do so prior to the first session start. You cannot set this property again after calling `nixnet._session.base.SessionBase.stop`.

For signal I/O sessions, this property is the number of signal values stored. This is analogous to the number of values you use with the appropriate read or write function.

For frame I/O sessions, this property is the number of bytes of frame data stored.

For standard CAN or LIN frame I/O sessions, each frame uses exactly 24 bytes. You can use this number to convert the Queue Size (in bytes) to/from the number of frame values.

For CAN FD and FlexRay frame I/O sessions, each frame value size can vary depending on the payload length. For more information, refer to Raw Frame Format.

For Signal I/O XY sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. According to the formulas below, the default queue sizes can be different for each frame. If you read the default Queue Size property for a Signal Input XY session, the largest queue size is returned, so that a call to the appropriate read function of that size can empty all queues. If you read the default Queue Size property for a Signal Output XY session, the smallest queue size is returned, so that a call to the appropriate write function of that size can succeed when all queues are empty. If you write the Queue Size property for a Signal I/O XY session, that size is used for all frames, so you must ensure that it is sufficient for the frame with the fastest transmit time.

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

For Single-Point sessions (signal or frame), this property is ignored. Single-Point sessions always use a value of 1 as the effective queue size.

Type int

`start`(*scope=<StartStopScope.NORMAL: 0>*)

Start communication for the XNET session.

Because the session is started automatically by default, this function is optional. This function is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the `nixnet._session.base.SessionBase.auto_start` property.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can start each logical unit separately. When a session is started, all contained frames or signals are placed in a state where they are ready to communicate. When the interface is started, it takes data from all started sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

If an output session starts before you write data, or you read an input session before it receives a frame, default data is used. For more information, refer to the XNET Frame Default Payload and XNET Signal Default Value properties.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`state`

Session running state.

Type *`nixnet._enums.SessionInfoState`*

`stop`(*scope=<StartStopScope.NORMAL: 0>*)

Stop communication for the XNET session.

Because the session is stopped automatically when closed (cleared), this function is optional.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can stop each logical unit separately. When a session is stopped, all contained frames or signals are placed in a state where they are no longer ready to communicate. When the interface is stopped, it no longer takes data from sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`time_communicating`

Time the interface started communicating.

The time is usually later than `time_start` because the interface must undergo a communication startup procedure.

Type int

`time_current`

Current interface time.

Type int

`time_start`

Time the interface was started.

Type int

`wait_for_intf_communicating`(*timeout=10*)

Wait for the interface to begin communication on the network.

If a start trigger is configured for the interface, this first waits for the trigger. Once the interface is started, this waits for the protocol's communication state to transition to a value that indicates communication with remote nodes.

After this wait succeeds, calls to 'read_state' will return:

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_ACTIVE'

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_PASSIVE'

'constants.ReadState.TIME_COMMUNICATING': Valid time for communication (invalid time of 0 prior)

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_intf_remote_wakeup`(*timeout=10*)

Wait for interface remote wakeup.

Wait for the interface to wakeup due to activity by a remote node on the network. This wait is used for CAN, when you set the 'can_tcvr_state' property to 'constants.CanTcvrState.SLEEP'. Although the interface itself is ready to communicate, this places the transceiver into a sleep state. When a remote CAN node transmits a frame, the transceiver wakes up, and communication is restored. This wait detects that remote wakeup.

This wait is used for LIN when you set 'lin_sleep' property to 'constants.LinSleep.REMOTE_SLEEP' or 'constants.LinSleep.LOCAL_SLEEP'. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_transmit_complete`(*timeout=10*)

Wait for transmition to complete.

All frames written for the session have been transmitted on the bus. This condition applies to CAN, LIN, and FlexRay. This condition is state based, and the state is Boolean (true/false).

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

class `nixnet.session.`**`FrameOutSinglePointSession`**(*interface_name*, *database_name*, *cluster_name*, *frames*)

Bases: *`nixnet._session.base.SessionBase`*

Frame Output Single-Point session.

This session writes frame values for the next transmit.

This session does not use queues to store frame values. If the write frame function is called twice before the next transmit, the transmitted frame uses the value from the second call to the write frame function.

The output data is provided as a list of frames, one for each frame specified for the session.

For this session, NI-XNET transmits each frame according to its properties in the database. Therefore, when you call the write frame function, the number of payload bytes in each frame value must match that frame's Payload Length property. The other frame value elements are ignored, so you can leave them uninitialized. For CAN interfaces, if the number of payload bytes you write is smaller than the Payload Length configured in the database, the requested number of bytes transmit. If the number of payload bytes is larger than the Payload Length configured in the database, the queue is flushed and no frames transmit. For other interfaces, transmitting a number of payload bytes different than the frame payload may cause unexpected results on the bus.

Note: Typical use case: Control or simulation applications that require lower level access to frames (not signals).

`application_protocol`

This property returns the application protocol that the session uses.

The database used with the session determines the application protocol.

Type *`nixnet._enums.AppProtocol`*

auto_start

Automatically starts the output session on the first call to the appropriate write function.

For input sessions, start always is performed within the first call to the appropriate read function (if not already started using *nixnet._session.base.SessionBase.start*). This is done because there is no known use case for reading a stopped input session.

For output sessions, as long as the first call to the appropriate write function contains valid data, you can leave this property at its default value of true. If you need to call the appropriate write function multiple times prior to starting the session, or if you are starting multiple sessions simultaneously, you can set this property to false. After calling the appropriate write function as desired, you can call *nixnet._session.base.SessionBase.start* to start the session(s).

When automatic start is performed, it is equivalent to *nixnet._session.base.SessionBase.start* with scope set to Normal. This starts the session itself, and if the interface is not already started, it starts the interface also.

Type bool

can_comm

CAN Communication state

Type *nixnet.types.CanComm*

change_lin_diagnostic_schedule(*schedule*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the diagnostic schedule.

Parameters **schedule** (*nixnet._enums.LinDiagnosticSchedule*) – Diagnostic schedule that the LIN master executes.

change_lin_schedule(*sched_index*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the running schedule.

According to the LIN protocol, only the master executes schedules, not slaves. If the *nixnet._session.intf.Interface.lin_master* property is false (slave), this write function implicitly sets that property to true (master). If the interface currently is running as a slave, this write returns an error, because it cannot change to master while running.

Parameters **sched_index** (*int*) – Index to the schedule table that the LIN master executes.

The schedule tables are sorted the way they are returned from the database with the *nixnet.database._cluster.Cluster.lin_schedules* property.

check_fault()

Check for an asynchronous fault.

A fault is an error that occurs asynchronously to the NI-XNET application calls. The fault cause may be related to network communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, nxReadState provides a detection method distinct from the status of NI-XNET function calls, yet easy to use alongside the common practice of checking the communication state.

close()

Close (clear) the XNET session.

This function stops communication for the session and releases all resources the session uses. It internally calls *nixnet._session.base.SessionBase.stop* with normal scope, so if this is the last session using the interface, communication stops.

You typically use 'close' when you need to close the existing session to create a new session that uses the same objects. For example, if you create a session for a frame named frame_a using Frame Output Single-Point mode, then you create a second session for frame_a using Frame Output Queued mode, the second call to the session constructor returns an error, because frame_a can be accessed using only one output mode. If you call 'close' before the second constructor call, you can close the previous use of frame_a to create the new session.

`cluster_name`

This property returns the cluster (network) name used with the session.

Type str

`connect_terminals`(*source*, *destination*)

Connect terminals on the XNET interface.

This function connects a source terminal to a destination terminal on the interface hardware. The XNET terminal represents an external or internal hardware connection point on a National Instruments XNET hardware product. External terminals include PXI Trigger lines for a PXI card, RTSI terminals for a PCI card, or the single external terminal for a C Series module. Internal terminals include timebases (clocks) and logical entities such as a start trigger.

The terminal inputs use the Terminal I/O names. Typically, one of the pair is an internal and the other an external.

Parameters

- **`source`** (*`str`*) – Connection source name.
- **`destination`** (*`str`*) – Connection destination name.

`database_name`

This property returns the database name used with the session.

Type str

`disconnect_terminals`(*source*, *destination*)

Disconnect terminals on the XNET interface.

This function disconnects a specific pair of source/destination terminals previously connected with *`nixnet._session.base.SessionBase.connect_terminals`*.

When the final session for a given interface is cleared, NI-XNET automatically disconnects all terminal connections for that interface. Therefore, 'disconnect_terminals' is not required for most applications.

This function typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using *`nixnet._session.base.SessionBase.stop`* with the Interface Only scope. Then you can call 'disconnect_terminals' and *`nixnet._session.base.SessionBase.connect_terminals`* to adjust terminal connections. Finally, you can call *`nixnet._session.base.SessionBase.start`* with the Interface Only scope to restart the interface.

You can disconnect only a terminal that has been previously connected. Attempting to disconnect a non-connected terminal results in an error.

Parameters

- **`source`** (*`str`*) – Connection source name.
- **`destination`** (*`str`*) – Connection destination name.

`flush`()

Flushes (empties) all XNET session queues.

With the exception of single-point modes, all sessions use queues to store frames. For input modes, the queues store frame values (or corresponding signal values) that have been received, but not obtained by calling the read function. For output sessions, the queues store frame values provided to write function, but not transmitted successfully.

nixnet._session.base.SessionBase.start and *nixnet._session.base.SessionBase.stop* have no effect on these queues. Use 'flush' to discard all values in the session's queues.

For example, if you call a write function to write three frames, then immediately call *nixnet._session.base.SessionBase.stop*, then call *nixnet._session.base.SessionBase.start* a few seconds later, the three frames transmit. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, no frames transmit.

As another example, if you receive three frames, then call *nixnet._session.base.SessionBase.stop*, the three frames remains in the queue. If you call *nixnet._session.base.SessionBase.start* a few seconds later, then call a read function, you obtain the three frames received earlier, potentially followed by other frames received after calling *nixnet._session.base.SessionBase.start*. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, read function returns only frames received after the calling *nixnet._session.base.SessionBase.start*.

frames
Operate on session's frames

Type *nixnet._session.frames.InFrames*

intf
Returns the Interface configuration object for the session.

Type *nixnet._session.intf.Interface*

j1939
Returns the J1939 configuration object for the session.

Type *nixnet._session.j1939.J1939*

lin_comm
LIN Communication state

Type *nixnet.types.LinComm*

mode
This property returns the mode associated with the session.

For more information, refer to *nixnet._enums.CreateSessionMode*.

Type *nixnet._enums.CreateSessionMode*

num_pend
This property returns the number of values (frames or signals) pending for the session.

For input sessions, this is the number of frame/signal values available to the appropriate read function. If you call the appropriate read function with number to read of this number and timeout of 0.0, the appropriate read function should return this number of values successfully.

For output sessions, this is the number of frames/signal values provided to the appropriate write function but not yet transmitted onto the network.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

`num_unused`

This property returns the number of values (frames or signals) unused for the session.

If you get this property prior to starting the session, it provides the size of the underlying queue(s). Contrary to the Queue Size property, this value is in number of frames for Frame I/O, not number of bytes; for Signal I/O, it is the number of signal values in both cases. After start, this property returns the queue size minus the *`Number of Values Pending`* property.

For input sessions, this is the number of frame/signal values unused in the underlying queue(s).

For output sessions, this is the number of frame/signal values you can provide to a subsequent call to the appropriate write function. If you call the appropriate write function with this number of values and timeout of 0.0, it should return success.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

`protocol`

This property returns the protocol that the interface in the session uses.

Type *`nixnet._enums.Protocol`*

`queue_size`

Get or set queue size.

For output sessions, queues store data passed to the appropriate write function and not yet transmitted onto the network. For input sessions, queues store data received from the network and not yet obtained using the appropriate read function.

For most applications, the default queue sizes are sufficient. You can write to this property to override the default. When you write (set) this property, you must do so prior to the first session start. You cannot set this property again after calling *`nixnet._session.base.SessionBase.stop`*.

For signal I/O sessions, this property is the number of signal values stored. This is analogous to the number of values you use with the appropriate read or write function.

For frame I/O sessions, this property is the number of bytes of frame data stored.

For standard CAN or LIN frame I/O sessions, each frame uses exactly 24 bytes. You can use this number to convert the Queue Size (in bytes) to/from the number of frame values.

For CAN FD and FlexRay frame I/O sessions, each frame value size can vary depending on the payload length. For more information, refer to Raw Frame Format.

For Signal I/O XY sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. According to the formulas below, the default queue sizes can be different for each frame. If you read the default Queue Size property for a Signal Input XY session, the largest queue size is returned, so that a call to the appropriate read function of that size can empty all queues. If you read the default Queue Size property for a Signal Output XY session, the smallest queue size is returned, so that a call to the appropriate write function of that size can succeed when all queues are empty. If you write the Queue Size property for a Signal I/O XY session, that size is used for all frames, so you must ensure that it is sufficient for the frame with the fastest transmit time.

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

For Single-Point sessions (signal or frame), this property is ignored. Single-Point sessions always use a value of 1 as the effective queue size.

Type int

`start`(*scope=<StartStopScope.NORMAL: 0>*)

Start communication for the XNET session.

Because the session is started automatically by default, this function is optional. This function is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the `nixnet._session.base.SessionBase.auto_start` property.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can start each logical unit separately. When a session is started, all contained frames or signals are placed in a state where they are ready to communicate. When the interface is started, it takes data from all started sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

If an output session starts before you write data, or you read an input session before it receives a frame, default data is used. For more information, refer to the XNET Frame Default Payload and XNET Signal Default Value properties.

Parameters **`scope`** (`nixnet._enums.StartStopScope`) – Describes the impact of this operation on the underlying state models for the session and its interface.

`state`

Session running state.

Type `nixnet._enums.SessionInfoState`

`stop`(*scope=<StartStopScope.NORMAL: 0>*)

Stop communication for the XNET session.

Because the session is stopped automatically when closed (cleared), this function is optional.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can stop each logical unit separately. When a session is stopped, all contained frames or signals are placed in a state where they are no longer ready to communicate. When the interface is stopped, it no longer takes data from sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`time_communicating`

Time the interface started communicating.

The time is usually later than `time_start` because the interface must undergo a communication startup procedure.

Type int

`time_current`

Current interface time.

Type int

`time_start`

Time the interface was started.

Type int

`wait_for_intf_communicating`(*timeout=10*)

Wait for the interface to begin communication on the network.

If a start trigger is configured for the interface, this first waits for the trigger. Once the interface is started, this waits for the protocol's communication state to transition to a value that indicates communication with remote nodes.

After this wait succeeds, calls to 'read_state' will return:

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_ACTIVE'

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_PASSIVE'

'constants.ReadState.TIME_COMMUNICATING': Valid time for communication (invalid time of 0 prior)

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_intf_remote_wakeup`(*timeout=10*)

Wait for interface remote wakeup.

Wait for the interface to wakeup due to activity by a remote node on the network. This wait is used for CAN, when you set the 'can_tcvr_state' property to 'constants.CanTcvrState.SLEEP'. Although the interface itself is ready to communicate, this places the transceiver into a sleep state. When a remote CAN node transmits a frame, the transceiver wakes up, and communication is restored. This wait detects that remote wakeup.

This wait is used for LIN when you set 'lin_sleep' property to 'constants.LinSleep.REMOTE_SLEEP' or 'constants.LinSleep.LOCAL_SLEEP'. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_transmit_complete`(*timeout=10*)

Wait for transmition to complete.

All frames written for the session have been transmitted on the bus. This condition applies to CAN, LIN, and FlexRay. This condition is state based, and the state is Boolean (true/false).

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

class `nixnet.session.`**`SignalInSinglePointSession`**(*interface_name*, *database_name*, *cluster_name*, *signals*)

Bases: *`nixnet._session.base.SessionBase`*

Signal Input Single-Point session.

This session reads the most recent value received for each signal.

This session does not use queues to store each received frame. If the interface receives two frames prior to calling *`nixnet._session.signals.SinglePointInSignals.read`*, that call to *`nixnet._session.signals.SinglePointInSignals.read`* returns signals for the second frame.

Use *`nixnet._session.signals.SinglePointInSignals.read`* for this session.

You also can specify a trigger signal for a frame. This signal name is :trigger:.<frame name>, and once it is specified in the __init__ `signals` list, it returns a value of 0.0 if the frame did not arrive since the last Read (or Start), and 1.0 if at least one frame of this ID arrived. You can specify multiple trigger signals for different frames in the same session. For multiplexed signals, a signal may or may not be contained in a received frame. To define a trigger signal for a multiplexed signal, use the signal name :trigger:.<frame name>.<signal name>. This signal returns 1.0 only if a frame with appropriate set multiplexer bit has been received since the last Read or Start.

Note: Typical use case: Control or simulation applications, such as Hardware In the Loop (HIL).

`application_protocol`

This property returns the application protocol that the session uses.

The database used with the session determines the application protocol.

Type *`nixnet._enums.AppProtocol`*

`auto_start`

Automatically starts the output session on the first call to the appropriate write function.

For input sessions, start always is performed within the first call to the appropriate read function (if not already started using *`nixnet._session.base.SessionBase.start`*). This is done because there is no known use case for reading a stopped input session.

For output sessions, as long as the first call to the appropriate write function contains valid data, you can leave this property at its default value of true. If you need to call the appropriate write function multiple times prior to starting the session, or if you are starting multiple sessions simultaneously, you can set this property to false. After calling the appropriate write function as desired, you can call *`nixnet._session.base.SessionBase.start`* to start the session(s).

When automatic start is performed, it is equivalent to *`nixnet._session.base.SessionBase.start`* with scope set to Normal. This starts the session itself, and if the interface is not already started, it starts the interface also.

Type bool

can_comm
CAN Communication state

Type *nixnet.types.CanComm*

change_lin_diagnostic_schedule(*schedule*)
Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the diagnostic schedule.

Parameters **schedule** (*nixnet._enums.LinDiagnosticSchedule*) – Diagnostic schedule that the LIN master executes.

change_lin_schedule(*sched_index*)
Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the running schedule.

According to the LIN protocol, only the master executes schedules, not slaves. If the *nixnet._session.intf.Interface.lin_master* property is false (slave), this write function implicitly sets that property to true (master). If the interface currently is running as a slave, this write returns an error, because it cannot change to master while running.

Parameters **sched_index** (*int*) – Index to the schedule table that the LIN master executes.

The schedule tables are sorted the way they are returned from the database with the *nixnet.database._cluster.Cluster.lin_schedules* property.

check_fault()
Check for an asynchronous fault.

A fault is an error that occurs asynchronously to the NI-XNET application calls. The fault cause may be related to network communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, nxReadState provides a detection method distinct from the status of NI-XNET function calls, yet easy to use alongside the common practice of checking the communication state.

close()
Close (clear) the XNET session.

This function stops communication for the session and releases all resources the session uses. It internally calls *nixnet._session.base.SessionBase.stop* with normal scope, so if this is the last session using the interface, communication stops.

You typically use 'close' when you need to close the existing session to create a new session that uses the same objects. For example, if you create a session for a frame named frame_a using Frame Output Single-Point mode, then you create a second session for frame_a using Frame Output Queued mode, the second call to the session constructor returns an error, because frame_a can be accessed using only one output mode. If you call 'close' before the second constructor call, you can close the previous use of frame_a to create the new session.

cluster_name
This property returns the cluster (network) name used with the session.

Type str

connect_terminals(*source*, *destination*)
Connect terminals on the XNET interface.

This function connects a source terminal to a destination terminal on the interface hardware. The XNET terminal represents an external or internal hardware connection point on a National Instruments XNET hardware product. External terminals include PXI Trigger lines for a PXI card, RTSI terminals for a PCI card, or the single external terminal for a C Series module. Internal terminals include timebases (clocks) and logical entities such as a start trigger.

The terminal inputs use the Terminal I/O names. Typically, one of the pair is an internal and the other an external.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

database_name

This property returns the database name used with the session.

Type str

disconnect_terminals(*source*, *destination*)

Disconnect terminals on the XNET interface.

This function disconnects a specific pair of source/destination terminals previously connected with *nixnet._session.base.SessionBase.connect_terminals*.

When the final session for a given interface is cleared, NI-XNET automatically disconnects all terminal connections for that interface. Therefore, 'disconnect_terminals' is not required for most applications.

This function typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using *nixnet._session.base.SessionBase.stop* with the Interface Only scope. Then you can call 'disconnect_terminals' and *nixnet._session.base.SessionBase.connect_terminals* to adjust terminal connections. Finally, you can call *nixnet._session.base.SessionBase.start* with the Interface Only scope to restart the interface.

You can disconnect only a terminal that has been previously connected. Attempting to disconnect a non-connected terminal results in an error.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

flush()

Flushes (empties) all XNET session queues.

With the exception of single-point modes, all sessions use queues to store frames. For input modes, the queues store frame values (or corresponding signal values) that have been received, but not obtained by calling the read function. For output sessions, the queues store frame values provided to write function, but not transmitted successfully.

nixnet._session.base.SessionBase.start and *nixnet._session.base.SessionBase.stop* have no effect on these queues. Use 'flush' to discard all values in the session's queues.

For example, if you call a write function to write three frames, then immediately call *nixnet._session.base.SessionBase.stop*, then call *nixnet._session.base.SessionBase.start* a few seconds later, the three frames transmit. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, no frames transmit.

As another example, if you receive three frames, then call *nixnet._session.base.SessionBase.stop*, the three frames remains in the queue. If you call *nixnet._session.base.SessionBase.start* a few seconds later, then call a read function, you obtain the three frames received earlier, potentially followed by other frames received after calling `nixnet._session.base.SessionBase.start`. If you call 'flush' between `nixnet._session.base.SessionBase.stop` and `nixnet._session.base.SessionBase.start`, read function returns only frames received after the calling `nixnet._session.base.SessionBase.start`.

intf

Returns the Interface configuration object for the session.

Type `nixnet._session.intf.Interface`

j1939

Returns the J1939 configuration object for the session.

Type `nixnet._session.j1939.J1939`

lin_comm

LIN Communication state

Type `nixnet.types.LinComm`

mode

This property returns the mode associated with the session.

For more information, refer to `nixnet._enums.CreateSessionMode`.

Type `nixnet._enums.CreateSessionMode`

num_pend

This property returns the number of values (frames or signals) pending for the session.

For input sessions, this is the number of frame/signal values available to the appropriate read function. If you call the appropriate read function with number to read of this number and timeout of 0.0, the appropriate read function should return this number of values successfully.

For output sessions, this is the number of frames/signal values provided to the appropriate write function but not yet transmitted onto the network.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

num_unused

This property returns the number of values (frames or signals) unused for the session.

If you get this property prior to starting the session, it provides the size of the underlying queue(s). Contrary to the Queue Size property, this value is in number of frames for Frame I/O, not number of bytes; for Signal I/O, it is the number of signal values in both cases. After start, this property returns the queue size minus the `Number of Values Pending` property.

For input sessions, this is the number of frame/signal values unused in the underlying queue(s).

For output sessions, this is the number of frame/signal values you can provide to a subsequent call to the appropriate write function. If you call the appropriate write function with this number of values and timeout of 0.0, it should return success.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

`protocol`

This property returns the protocol that the interface in the session uses.

Type *`nixnet._enums.Protocol`*

`queue_size`

Get or set queue size.

For output sessions, queues store data passed to the appropriate write function and not yet transmitted onto the network. For input sessions, queues store data received from the network and not yet obtained using the appropriate read function.

For most applications, the default queue sizes are sufficient. You can write to this property to override the default. When you write (set) this property, you must do so prior to the first session start. You cannot set this property again after calling *`nixnet._session.base.SessionBase.stop`*.

For signal I/O sessions, this property is the number of signal values stored. This is analogous to the number of values you use with the appropriate read or write function.

For frame I/O sessions, this property is the number of bytes of frame data stored.

For standard CAN or LIN frame I/O sessions, each frame uses exactly 24 bytes. You can use this number to convert the Queue Size (in bytes) to/from the number of frame values.

For CAN FD and FlexRay frame I/O sessions, each frame value size can vary depending on the payload length. For more information, refer to Raw Frame Format.

For Signal I/O XY sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. According to the formulas below, the default queue sizes can be different for each frame. If you read the default Queue Size property for a Signal Input XY session, the largest queue size is returned, so that a call to the appropriate read function of that size can empty all queues. If you read the default Queue Size property for a Signal Output XY session, the smallest queue size is returned, so that a call to the appropriate write function of that size can succeed when all queues are empty. If you write the Queue Size property for a Signal I/O XY session, that size is used for all frames, so you must ensure that it is sufficient for the frame with the fastest transmit time.

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

For Single-Point sessions (signal or frame), this property is ignored. Single-Point sessions always use a value of 1 as the effective queue size.

Type int

`signals`
Operate on session's signals

Type *`nixnet._session.signals.SinglePointInSignals`*

`start`(*scope=<StartStopScope.NORMAL: 0>*)
Start communication for the XNET session.

Because the session is started automatically by default, this function is optional. This function is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the *`nixnet._session.base.SessionBase.auto_start`* property.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can start each logical unit separately. When a session is started, all contained frames or signals are placed in a state where they are ready to communicate. When the interface is started, it takes data from all started sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

If an output session starts before you write data, or you read an input session before it receives a frame, default data is used. For more information, refer to the XNET Frame Default Payload and XNET Signal Default Value properties.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`state`
Session running state.

Type *`nixnet._enums.SessionInfoState`*

`stop`(*scope=<StartStopScope.NORMAL: 0>*)
Stop communication for the XNET session.

Because the session is stopped automatically when closed (cleared), this function is optional.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can stop each logical unit separately. When a session is stopped, all contained frames or signals are placed in a state where they are no longer ready to communicate. When the interface is stopped, it no longer takes data from sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`time_communicating`
Time the interface started communicating.

The time is usually later than `time_start` because the interface must undergo a communication startup procedure.

Type int

`time_current`
Current interface time.

Type int

`time_start`
Time the interface was started.

Type int

`wait_for_intf_communicating`(*timeout=10*)
Wait for the interface to begin communication on the network.

If a start trigger is configured for the interface, this first waits for the trigger. Once the interface is started, this waits for the protocol's communication state to transition to a value that indicates communication with remote nodes.

After this wait succeeds, calls to 'read_state' will return:

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_ACTIVE'

`nixnet._enums.CanCommState`: 'constants.CAN_COMM.ERROR_PASSIVE'

'constants.ReadState.TIME_COMMUNICATING': Valid time for communication (invalid time of 0 prior)

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_intf_remote_wakeup`(*timeout=10*)
Wait for interface remote wakeup.

Wait for the interface to wakeup due to activity by a remote node on the network. This wait is used for CAN, when you set the 'can_tcvr_state' property to 'constants.CanTcvrState.SLEEP'. Although the interface itself is ready to communicate, this places the transceiver into a sleep state. When a remote CAN node transmits a frame, the transceiver wakes up, and communication is restored. This wait detects that remote wakeup.

This wait is used for LIN when you set 'lin_sleep' property to 'constants.LinSleep.REMOTE_SLEEP' or 'constants.LinSleep.LOCAL_SLEEP'. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_transmit_complete`(*timeout=10*)
Wait for transmition to complete.

All frames written for the session have been transmitted on the bus. This condition applies to CAN, LIN, and FlexRay. This condition is state based, and the state is Boolean (true/false).

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`class` `nixnet.session.`**`SignalOutSinglePointSession`**(*interface_name*, *database_name*, *cluster_name*, *signals*)
Bases: *`nixnet._session.base.SessionBase`*

Signal Out Single-Point session.

This session writes signal values for the next frame transmit.

This session does not use queues to store signal values. If *nixnet._session.signals.SinglePointOutSignals.write* is called twice before the next transmit, the transmitted frame uses signal values from the second call to *nixnet._session.signals.SinglePointOutSignals.write*.

Use *nixnet._session.signals.SinglePointOutSignals.write* for this session.

You also can specify a trigger signal for a frame. This signal name is :trigger:.<frame name>, and once it is specified in the __init__ `signals` list, you can write a value of 0.0 to suppress writing of that frame, or any value not equal to 0.0 to write the frame. You can specify multiple trigger signals for different frames in the same session.

Note: Typical use case: Control or simulation applications, such as Hardware In the Loop (HIL).

application_protocol
This property returns the application protocol that the session uses.

The database used with the session determines the application protocol.

Type *nixnet._enums.AppProtocol*

auto_start
Automatically starts the output session on the first call to the appropriate write function.

For input sessions, start always is performed within the first call to the appropriate read function (if not already started using *nixnet._session.base.SessionBase.start*). This is done because there is no known use case for reading a stopped input session.

For output sessions, as long as the first call to the appropriate write function contains valid data, you can leave this property at its default value of true. If you need to call the appropriate write function multiple times prior to starting the session, or if you are starting multiple sessions simultaneously, you can set this property to false. After calling the appropriate write function as desired, you can call *nixnet._session.base.SessionBase.start* to start the session(s).

When automatic start is performed, it is equivalent to *nixnet._session.base.SessionBase.start* with scope set to Normal. This starts the session itself, and if the interface is not already started, it starts the interface also.

Type bool

can_comm
CAN Communication state

Type *nixnet.types.CanComm*

change_lin_diagnostic_schedule(*schedule*)
Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the diagnostic schedule.

Parameters **schedule** (*nixnet._enums.LinDiagnosticSchedule*) – Diagnostic schedule that the LIN master executes.

change_lin_schedule(*sched_index*)
Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the running schedule.

According to the LIN protocol, only the master executes schedules, not slaves. If the `nixnet._session.intf.Interface.lin_master` property is false (slave), this write function implicitly sets that property to true (master). If the interface currently is running as a slave, this write returns an error, because it cannot change to master while running.

Parameters **`sched_index`** (*`int`*) – Index to the schedule table that the LIN master executes.

The schedule tables are sorted the way they are returned from the database with the *nixnet.database._cluster.Cluster.lin_schedules* property.

`check_fault`()

Check for an asynchronous fault.

A fault is an error that occurs asynchronously to the NI-XNET application calls. The fault cause may be related to network communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, nxReadState provides a detection method distinct from the status of NI-XNET function calls, yet easy to use alongside the common practice of checking the communication state.

`close`()

Close (clear) the XNET session.

This function stops communication for the session and releases all resources the session uses. It internally calls `nixnet._session.base.SessionBase.stop` with normal scope, so if this is the last session using the interface, communication stops.

You typically use 'close' when you need to close the existing session to create a new session that uses the same objects. For example, if you create a session for a frame named frame_a using Frame Output Single-Point mode, then you create a second session for frame_a using Frame Output Queued mode, the second call to the session constructor returns an error, because frame_a can be accessed using only one output mode. If you call 'close' before the second constructor call, you can close the previous use of frame_a to create the new session.

`cluster_name`

This property returns the cluster (network) name used with the session.

Type str

`connect_terminals`(*source*, *destination*)

Connect terminals on the XNET interface.

This function connects a source terminal to a destination terminal on the interface hardware. The XNET terminal represents an external or internal hardware connection point on a National Instruments XNET hardware product. External terminals include PXI Trigger lines for a PXI card, RTSI terminals for a PCI card, or the single external terminal for a C Series module. Internal terminals include timebases (clocks) and logical entities such as a start trigger.

The terminal inputs use the Terminal I/O names. Typically, one of the pair is an internal and the other an external.

Parameters

- **`source`** (*`str`*) – Connection source name.
- **`destination`** (*`str`*) – Connection destination name.

`database_name`

This property returns the database name used with the session.

Type str

`disconnect_terminals`(*source*, *destination*)

Disconnect terminals on the XNET interface.

This function disconnects a specific pair of source/destination terminals previously connected with *nixnet._session.base.SessionBase.connect_terminals*.

When the final session for a given interface is cleared, NI-XNET automatically disconnects all terminal connections for that interface. Therefore, 'disconnect_terminals' is not required for most applications.

This function typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using *nixnet._session.base.SessionBase.stop* with the Interface Only scope. Then you can call 'disconnect_terminals' and *nixnet._session.base.SessionBase.connect_terminals* to adjust terminal connections. Finally, you can call *nixnet._session.base.SessionBase.start* with the Interface Only scope to restart the interface.

You can disconnect only a terminal that has been previously connected. Attempting to disconnect a non-connected terminal results in an error.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

flush()

Flushes (empties) all XNET session queues.

With the exception of single-point modes, all sessions use queues to store frames. For input modes, the queues store frame values (or corresponding signal values) that have been received, but not obtained by calling the read function. For output sessions, the queues store frame values provided to write function, but not transmitted successfully.

nixnet._session.base.SessionBase.start and *nixnet._session.base.SessionBase.stop* have no effect on these queues. Use 'flush' to discard all values in the session's queues.

For example, if you call a write function to write three frames, then immediately call *nixnet._session.base.SessionBase.stop*, then call *nixnet._session.base.SessionBase.start* a few seconds later, the three frames transmit. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, no frames transmit.

As another example, if you receive three frames, then call *nixnet._session.base.SessionBase.stop*, the three frames remains in the queue. If you call *nixnet._session.base.SessionBase.start* a few seconds later, then call a read function, you obtain the three frames received earlier, potentially followed by other frames received after calling *nixnet._session.base.SessionBase.start*. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, read function returns only frames received after the calling *nixnet._session.base.SessionBase.start*.

intf

Returns the Interface configuration object for the session.

Type *nixnet._session.intf.Interface*

j1939

Returns the J1939 configuration object for the session.

Type *nixnet._session.j1939.J1939*

lin_comm

LIN Communication state

Type *nixnet.types.LinComm*

mode

This property returns the mode associated with the session.

For more information, refer to *nixnet._enums.CreateSessionMode*.

Type *nixnet._enums.CreateSessionMode*

num_pend

This property returns the number of values (frames or signals) pending for the session.

For input sessions, this is the number of frame/signal values available to the appropriate read function. If you call the appropriate read function with number to read of this number and timeout of 0.0, the appropriate read function should return this number of values successfully.

For output sessions, this is the number of frames/signal values provided to the appropriate write function but not yet transmitted onto the network.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

num_unused

This property returns the number of values (frames or signals) unused for the session.

If you get this property prior to starting the session, it provides the size of the underlying queue(s). Contrary to the Queue Size property, this value is in number of frames for Frame I/O, not number of bytes; for Signal I/O, it is the number of signal values in both cases. After start, this property returns the queue size minus the *Number of Values Pending* property.

For input sessions, this is the number of frame/signal values unused in the underlying queue(s).

For output sessions, this is the number of frame/signal values you can provide to a subsequent call to the appropriate write function. If you call the appropriate write function with this number of values and timeout of 0.0, it should return success.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

protocol

This property returns the protocol that the interface in the session uses.

Type *nixnet._enums.Protocol*

queue_size
Get or set queue size.

For output sessions, queues store data passed to the appropriate write function and not yet transmitted onto the network. For input sessions, queues store data received from the network and not yet obtained using the appropriate read function.

For most applications, the default queue sizes are sufficient. You can write to this property to override the default. When you write (set) this property, you must do so prior to the first session start. You cannot set this property again after calling *nixnet._session.base.SessionBase.stop*.

For signal I/O sessions, this property is the number of signal values stored. This is analogous to the number of values you use with the appropriate read or write function.

For frame I/O sessions, this property is the number of bytes of frame data stored.

For standard CAN or LIN frame I/O sessions, each frame uses exactly 24 bytes. You can use this number to convert the Queue Size (in bytes) to/from the number of frame values.

For CAN FD and FlexRay frame I/O sessions, each frame value size can vary depending on the payload length. For more information, refer to Raw Frame Format.

For Signal I/O XY sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. According to the formulas below, the default queue sizes can be different for each frame. If you read the default Queue Size property for a Signal Input XY session, the largest queue size is returned, so that a call to the appropriate read function of that size can empty all queues. If you read the default Queue Size property for a Signal Output XY session, the smallest queue size is returned, so that a call to the appropriate write function of that size can succeed when all queues are empty. If you write the Queue Size property for a Signal I/O XY session, that size is used for all frames, so you must ensure that it is sufficient for the frame with the fastest transmit time.

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

For Single-Point sessions (signal or frame), this property is ignored. Single-Point sessions always use a value of 1 as the effective queue size.

Type int

signals
Operate on session's signals

Type *nixnet._session.signals.SinglePointInSignals*

start(*scope=<StartStopScope.NORMAL: 0>*)
Start communication for the XNET session.

Because the session is started automatically by default, this function is optional. This function is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the *nixnet._session.base.SessionBase.auto_start* property.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can start each logical unit separately. When a session is started, all contained frames or signals are placed in a state where they are ready to communicate. When the interface is started, it takes data from all started sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

If an output session starts before you write data, or you read an input session before it receives a frame, default data is used. For more information, refer to the XNET Frame Default Payload and XNET Signal Default Value properties.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`state`

Session running state.

Type *`nixnet._enums.SessionInfoState`*

`stop`(*scope=<StartStopScope.NORMAL: 0>*)

Stop communication for the XNET session.

Because the session is stopped automatically when closed (cleared), this function is optional.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can stop each logical unit separately. When a session is stopped, all contained frames or signals are placed in a state where they are no longer ready to communicate. When the interface is stopped, it no longer takes data from sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`time_communicating`

Time the interface started communicating.

The time is usually later than `time_start` because the interface must undergo a communication startup procedure.

Type int

`time_current`

Current interface time.

Type int

`time_start`

Time the interface was started.

Type int

`wait_for_intf_communicating`(*timeout=10*)

Wait for the interface to begin communication on the network.

If a start trigger is configured for the interface, this first waits for the trigger. Once the interface is started, this waits for the protocol's communication state to transition to a value that indicates communication with remote nodes.

After this wait succeeds, calls to 'read_state' will return:

nixnet._enums.CanCommState: 'constants.CAN_COMM.ERROR_ACTIVE'

nixnet._enums.CanCommState: 'constants.CAN_COMM.ERROR_PASSIVE'

'constants.ReadState.TIME_COMMUNICATING': Valid time for communication (invalid time of 0 prior)

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_intf_remote_wakeup`(*timeout=10*)
Wait for interface remote wakeup.

Wait for the interface to wakeup due to activity by a remote node on the network. This wait is used for CAN, when you set the 'can_tcvr_state' property to 'constants.CanTcvrState.SLEEP'. Although the interface itself is ready to communicate, this places the transceiver into a sleep state. When a remote CAN node transmits a frame, the transceiver wakes up, and communication is restored. This wait detects that remote wakeup.

This wait is used for LIN when you set 'lin_sleep' property to 'constants.LinSleep.REMOTE_SLEEP' or 'constants.LinSleep.LOCAL_SLEEP'. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

`wait_for_transmit_complete`(*timeout=10*)
Wait for transmition to complete.

All frames written for the session have been transmitted on the bus. This condition applies to CAN, LIN, and FlexRay. This condition is state based, and the state is Boolean (true/false).

Parameters **`timeout`** (*`float`*) – The maximum amount of time to wait in seconds.

nixnet.session.frames

`class` `nixnet._session.frames.`**`Frame`**(*handle*, *index*, *name*)
Bases: `nixnet._session.collection.Item`

Frame configuration for a session.

`set_can_start_time_off`(*offset*)
Set CAN Start Time Offset.

Use this function to have more control over the schedule of frames on the bus, to offer more determinism by configuring cyclic frames to be spaced evenly.

If you do not call this function or you set it to a negative number, NI-XNET chooses this start time offset based on the arbitration identifier and periodic transmit time.

`offset` takes effect whenever a session is started. If you stop a session and restart it, the start time offset is re-evaluated.

Parameters **`offset`** (*`float`*) – The amount of time that must elapse between the session being started and the time that the first frame is transmitted across the bus. This is different than the cyclic rate, which determines the time between subsequent frame transmissions.

set_can_tx_time(*time*)
Set CAN Transmit Time.

If you call this function while a frame object is currently started, the frame object is stopped, the cyclic rate updated, and then the frame object is restarted. Because of the stopping and starting, the frame's start time offset is re-evaluated.

The first time a queued frame object is started, the XNET frame's transmit time determines the object's default queue size. Changing this rate has no impact on the queue size. Depending on how you change the rate, the queue may not be sufficient to store data for an extended period of time. You can mitigate this by setting the session Queue Size property to provide sufficient storage for all rates you use. If you are using a single-point session, this is not relevant.

Parameters **time** (*float*) – Frame's transmit time while the session is running. The transmit time is the amount of time that must elapse between subsequent transmissions of a cyclic frame. The default value of this property comes from the database (the XNET Frame CAN Transmit Time property).

set_j1939_addr_filter(*address=''*)
Set J1939 Address Filter.

Define a filter for the source address of the PGN transmitting node. You can use it when multiple nodes with different addresses are transmitting the same PGN.

If the filter is active, the session accepts only frames transmitted by a node with the defined address. All other frames with the same PGN but transmitted by other nodes are ignored.

Note: You can use this function in input sessions only.

Parameters **address** (*str or int*) – Decimal value of the address. Leave blank to reset the filter.

set_lin_tx_n_corrupted_chksums(*n*)
Set LIN Transmit N Corrupted Checksums.

When set to a nonzero value, this function causes the next N number of checksums to be corrupted. The checksum is corrupted by negating the value calculated per the database; (EnhancedValue * -1) or (ClassicValue * -1).

If the frame is transmitted in an unconditional or sporadic schedule slot, N is always decremented for each frame transmission. If the frame is transmitted in an event-triggered slot and a collision occurs, N is not decremented. In that case, N is decremented only when the collision resolving schedule is executed and the frame is successfully transmitted. If the frame is the only one to transmit in the event-triggered slot (no collision), N is decremented at event-triggered slot time.

This function is useful for testing ECU behavior when a corrupted checksum is transmitted.

Note: This function is valid only for output sessions.

Parameters **n** (*int*) – Number of checksums to be corrupted.

set_skip_n_cyclic_frames(*n*)
Set Skip N Cyclic Frames

When the frame's transmission time arrives and the skip count is nonzero, a frame value is dequeued (if this is not a single-point session), and the skip count is decremented, but the frame actually is not transmitted across the bus. When the skip count decrements to zero, subsequent cyclic transmissions resume.

This function is useful for testing of ECU behavior when a cyclic frame is expected, but is missing for N cycles.

Note: Only CAN interfaces currently support this function.

Note: This property is valid only for output sessions and frames with cyclic timing (that is, not event-based frames).

Parameters **`n`** (*`int`*) – Skip the next N cyclic frames when nonzero.

`class` `nixnet._session.frames.`**`Frames`**(*handle*)

Bases: `nixnet._session.collection.Collection`

Frames in a session.

`count`(*value*) → integer – return number of occurrences of value

`get`(*index*, *default=None*)

Access an item, returning `default` on failure.

Parameters

- **`index`** (*`str or int`*) – Item name or index
- **`default`** – Value to return when lookup fails

`index`(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

`payld_len_max`

Returns the maximum payload length of all frames in this session, expressed as bytes (0-254).

For CAN Stream (Input and Output), this property depends on the XNET Cluster CAN I/O Mode property. If the I/O mode is *constants.CanIoMode.CAN*, this property is 8 bytes. If the I/O mode is 'constants.CanIoMode.CAN_FD' or 'constants.CanIoMode.CAN_FD_BRS', this property is 64 bytes.

For LIN Stream (Input and Output), this property always is 8 bytes.

For FlexRay Stream (Input and Output), this property is the same as the XNET Cluster FlexRay Payload Length Maximum property value.

For Queued and Single-Point (Input and Output), this is the maximum payload of all frames specified in the List property.

Type int

`class` `nixnet._session.frames.`**`InFrames`**(*handle*)

Bases: *`nixnet._session.frames.Frames`*

Frames in a session.

`count`(*value*) → integer – return number of occurrences of value

get (*index*, *default=None*)
Access an item, returning `default` on failure.

Parameters

- **index** (*str or int*) – Item name or index
- **default** – Value to return when lookup fails

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

payld_len_max
Returns the maximum payload length of all frames in this session, expressed as bytes (0-254).

For CAN Stream (Input and Output), this property depends on the XNET Cluster CAN I/O Mode property. If the I/O mode is *constants.CanIoMode.CAN*, this property is 8 bytes. If the I/O mode is 'constants.CanIoMode.CAN_FD' or 'constants.CanIoMode.CAN_FD_BRS', this property is 64 bytes.

For LIN Stream (Input and Output), this property always is 8 bytes.

For FlexRay Stream (Input and Output), this property is the same as the XNET Cluster FlexRay Payload Length Maximum property value.

For Queued and Single-Point (Input and Output), this is the maximum payload of all frames specified in the List property.

Type int

read (*num_frames*, *timeout=0*, *frame_type=<class 'nixnet.types.XnetFrame'>*)
Read frames.

Parameters

- **num_frames** (*int*) – Number of frames to read.
- **timeout** (*float*) – The time in seconds to wait for number to read frame bytes to become available.

  If 'timeout' is positive, this function waits for 'num_frames' frames to be received, then returns complete frames up to that number. If the frames do not arrive prior to the 'timeout', an error is returned.

  If 'timeout' is 'constants.TIMEOUT_INFINITE', this function waits indefinitely for 'num_frames' frames.

  If 'timeout' is 'constants.TIMEOUT_NONE', this function does not wait and immediately returns all available frames up to the limit 'num_frames' specifies.
- **frame_type** (*nixnet.types.FrameFactory*) – A factory for the desired frame formats.

Yields *nixnet.types.Frame*

read_bytes (*num_bytes*, *timeout=0*)
Read data as a list of raw bytes (frame data).

The raw bytes encode one or more frames using the Raw Frame Format.

Parameters

- **num_bytes** (*int*) – The number of bytes to read.

- **timeout** (*float*) – The time in seconds to wait for number to read frame bytes to become available.

  To avoid returning a partial frame, even when 'num_bytes' are available from the hardware, this read may return fewer bytes in buffer. For example, assume you pass 'num_bytes' 70 bytes and 'timeout' of 10 seconds. During the read, two frames are received, the first 24 bytes in size, and the second 56 bytes in size, for a total of 80 bytes. The read returns after the two frames are received, but only the first frame is copied to data. If the read copied 46 bytes of the second frame (up to the limit of 70), that frame would be incomplete and therefore difficult to interpret. To avoid this problem, the read always returns complete frames in buffer.

  If 'timeout' is positive, this function waits for 'num_bytes' frame bytes to be received, then returns complete frames up to that number. If the bytes do not arrive prior to the 'timeout', an error is returned.

  If 'timeout' is 'constants.TIMEOUT_INFINITE', this function waits indefinitely for 'num_bytes' frame bytes.

  If 'timeout' is 'constants.TIMEOUT_NONE', this function does not wait and immediately returns all available frame bytes up to the limit 'num_bytes' specifies.

Returns A list of raw bytes representing the data.

class nixnet._session.frames.**OutFrames**(*handle*)

Bases: *nixnet._session.frames.Frames*

Frames in a session.

count(*value*) → integer – return number of occurrences of value

get(*index*, *default=None*)

Access an item, returning `default` on failure.

Parameters

- **index** (*str or int*) – Item name or index
- **default** – Value to return when lookup fails

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

payld_len_max

Returns the maximum payload length of all frames in this session, expressed as bytes (0-254).

For CAN Stream (Input and Output), this property depends on the XNET Cluster CAN I/O Mode property. If the I/O mode is *constants.CanIoMode.CAN*, this property is 8 bytes. If the I/O mode is 'constants.CanIoMode.CAN_FD' or 'constants.CanIoMode.CAN_FD_BRS', this property is 64 bytes.

For LIN Stream (Input and Output), this property always is 8 bytes.

For FlexRay Stream (Input and Output), this property is the same as the XNET Cluster FlexRay Payload Length Maximum property value.

For Queued and Single-Point (Input and Output), this is the maximum payload of all frames specified in the List property.

Type int

write(*frames*, *timeout=10*)

Write frame data.

Parameters

- **`frames`** (*`list of float`*) – One or more `nixnet.types.Frame` objects to be written to the session.
- **`timeout`** (*`float`*) – The time in seconds to wait for number to read frame bytes to become available.

  If 'timeout' is positive, this function waits up to that 'timeout' for space to become available in queues. If the space is not available prior to the 'timeout', a 'timeout' error is returned.

  If 'timeout' is 'constants.TIMEOUT_INFINITE', this functions waits indefinitely for space to become available in queues.

  If 'timeout' is 'constants.TIMEOUT_NONE', this function does not wait and immediately returns with a 'timeout' error if all data cannot be queued. Regardless of the 'timeout' used, if a 'timeout' error occurs, none of the data is queued, so you can attempt to call this function again at a later time with the same data.

`write_bytes` (*frame_bytes*, *timeout=10*)
Write a list of raw bytes (frame data).

The raw bytes encode one or more frames using the Raw Frame Format.

Parameters

- **`frame_bytes`** (*`bytes`*) – Frames to transmit.
- **`timeout`** (*`float`*) – The time in seconds to wait for number to read frame bytes to become available.

  If 'timeout' is positive, this function waits up to that 'timeout' for space to become available in queues. If the space is not available prior to the 'timeout', a 'timeout' error is returned.

  If 'timeout' is 'constants.TIMEOUT_INFINITE', this functions waits indefinitely for space to become available in queues.

  If 'timeout' is 'constants.TIMEOUT_NONE', this function does not wait and immediately returns with a 'timeout' error if all data cannot be queued. Regardless of the 'timeout' used, if a 'timeout' error occurs, none of the data is queued, so you can attempt to call this function again at a later time with the same data.

class `nixnet._session.frames.`**`SinglePointInFrames`**(*handle*)
Bases: *`nixnet._session.frames.Frames`*

Frames in a session.

`count`(*value*) → integer – return number of occurrences of value

`get`(*index*, *default=None*)
Access an item, returning `default` on failure.

Parameters

- **`index`** (*`str or int`*) – Item name or index
- **`default`** – Value to return when lookup fails

`index`(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

payld_len_max

Returns the maximum payload length of all frames in this session, expressed as bytes (0-254).

For CAN Stream (Input and Output), this property depends on the XNET Cluster CAN I/O Mode property. If the I/O mode is *constants.CanIoMode.CAN*, this property is 8 bytes. If the I/O mode is 'constants.CanIoMode.CAN_FD' or 'constants.CanIoMode.CAN_FD_BRS', this property is 64 bytes.

For LIN Stream (Input and Output), this property always is 8 bytes.

For FlexRay Stream (Input and Output), this property is the same as the XNET Cluster FlexRay Payload Length Maximum property value.

For Queued and Single-Point (Input and Output), this is the maximum payload of all frames specified in the List property.

Type int

read(*frame_type=<class 'nixnet.types.XnetFrame'>*)

Read frames.

Parameters **frame_type** (`nixnet.types.FrameFactory`) – A factory for the desired frame formats.

Yields `nixnet.types.Frame`

read_bytes(*num_bytes*)

Read data as a list of raw bytes (frame data).

Parameters **num_bytes** (`int`) – Number of bytes to read.

Returns Raw bytes representing the data.

Return type bytes

class nixnet._session.frames.**SinglePointOutFrames**(*handle*)

Bases: `nixnet._session.frames.Frames`

Frames in a session.

count(*value*) → integer – return number of occurrences of value

get(*index*, *default=None*)

Access an item, returning `default` on failure.

Parameters

- **index** (`str or int`) – Item name or index
- **default** – Value to return when lookup fails

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

payld_len_max

Returns the maximum payload length of all frames in this session, expressed as bytes (0-254).

For CAN Stream (Input and Output), this property depends on the XNET Cluster CAN I/O Mode property. If the I/O mode is *constants.CanIoMode.CAN*, this property is 8 bytes. If the I/O mode is 'constants.CanIoMode.CAN_FD' or 'constants.CanIoMode.CAN_FD_BRS', this property is 64 bytes.

For LIN Stream (Input and Output), this property always is 8 bytes.

For FlexRay Stream (Input and Output), this property is the same as the XNET Cluster FlexRay Payload Length Maximum property value.

For Queued and Single-Point (Input and Output), this is the maximum payload of all frames specified in the List property.

Type int

`write` (*frames*)
Write frame data.

Parameters **`frames`** (*`list of float`*) – One or more *`nixnet.types.Frame`* objects to be written to the session.

`write_bytes` (*frame_bytes*)
Write a list of raw bytes (frame data).

The raw bytes encode one or more frames using the Raw Frame Format.

Parameters **`frame_bytes`** (*`bytes`*) – Frames to transmit.

nixnet.session.signals

class `nixnet._session.signals.`**`Signal`** (*handle*, *index*, *name*)
Bases: `nixnet._session.collection.Item`

Signal configuration for a session.

class `nixnet._session.signals.`**`Signals`** (*handle*)
Bases: `nixnet._session.collection.Collection`

Signals in a session.

`count` (*value*) → integer – return number of occurrences of value

`get` (*index*, *default=None*)
Access an item, returning `default` on failure.

Parameters

- **`index`** (*`str or int`*) – Item name or index
- **`default`** – Value to return when lookup fails

`index` (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

class `nixnet._session.signals.`**`SinglePointInSignals`** (*handle*)
Bases: *`nixnet._session.signals.Signals`*

Writeable signals in a session.

`count` (*value*) → integer – return number of occurrences of value

`get` (*index*, *default=None*)
Access an item, returning `default` on failure.

Parameters

- **`index`** (*`str or int`*) – Item name or index
- **`default`** – Value to return when lookup fails

`index` (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

read()

Read data from a Signal Input Single-Point session.

Yields *tuple of int and float* – Timestamp and signal

class nixnet._session.signals.**SinglePointOutSignals**(*handle*)

Bases: *nixnet._session.signals.Signals*

Writeable signals in a session.

count(*value*) → integer – return number of occurrences of value

get(*index*, *default=None*)

Access an item, returning `default` on failure.

Parameters

- **index** (*str or int*) – Item name or index
- **default** – Value to return when lookup fails

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

write(*signals*)

Write data to a Signal Output Single-Point session.

Parameters **signals** (*list of float*) – A list of signal values (float).

nixnet.session.intf

class nixnet._session.intf.**Interface**(*handle*)

Bases: `object`

Interface configuration for a session

baud_rate

CAN, FlexRay, or LIN interface baud rate.

The default value for this interface property is the same as the cluster's baud rate in the database. Your application can set this interface baud rate to override the value in the database, or when no database is used.

CAN

When the upper nibble (0xF0000000) is clear, this is a numeric baud rate (for example, 500000).

NI-XNET CAN hardware currently accepts the following numeric baud rates: 33333, 40000, 50000, 62500, 80000, 83333, 100000, 125000, 160000, 200000, 250000, 400000, 500000, 800000, and 1000000.

LIN

When the upper nibble (0xF0000000) is clear, you can set only baud rates within the LIN-specified range (2400 to 20000) for the interface.

Type int

bus_err_to_in_strm

Bus Error Frames to Input Stream?

Specifies whether the hardware should place a CAN or LIN bus error frame into the Stream Input queue after it is generated.

Type bool

can_disable_prot_exception_handling

CAN Disable Protocol Exception Handling.

A protocol exception occurs when the CAN hardware detects an invalid combination of bits on the CAN bus reserved for a future protocol expansion. NI-XNET allows you to define how the hardware should behave in case of a protocol exception:

False (default): the CAN hardware stops receiving frames and starts a bus integration.

True: the CAN hardware transmits an error frame when it detects a protocol exception condition.

Type bool

can_edge_filter

CAN Enable Edge Filter.

When this property is enabled, the CAN hardware requires two consecutive dominant tq for hard synchronization.

Type bool

can_fd_baud_rate

The fast data baud rate for *can_io_mode* of *nixnet._enums.CanIoMode* `CAN_FD_BRS`

The default value for this interface property is the same as the cluster's FD baud rate in the database. Your application can set this interface FD baud rate to override the value in the database.

When the upper nibble (0xF0000000) is clear, this is a numeric baud rate (for example, 500000).

NI-XNET CAN hardware currently accepts the following numeric baud rates: 200000, 250000, 400000, 500000, 800000, 1000000, 1250000, 1600000, 2000000, 2500000, 4000000, 5000000, and 8000000.

Note: Not all CAN transceivers are rated to transmit at the requested rate. If you attempt to use a rate that exceeds the transceiver's qualified rate, XNET Start returns a warning. NI-XNET Hardware Overview describes the CAN transceivers' limitations.

Type int

can_fd_iso_mode

CAN FS ISO Mode.

This property is valid only when the interface is in CAN FD(+BRS) mode. It specifies whether the interface is working in the ISO CAN FD standard (ISO standard 11898-1:2015) or non-ISO CAN FD standard (Bosch CAN FD 1.0 specification). Two ports using different standards (ISO CAN FD vs. non-ISO CAN FD) cannot communicate with each other.

When you use a CAN FD database (DBC or FIBEX file created with NI-XNET), you can specify the ISO CAN FD mode when creating an alias name for the database. An alias is created automatically when you open a new database in the NI-XNET Database Editor. The specified ISO CAN FD mode is used as default, which you can change in the session using this property.

Type *nixnet._enums.CanFdIsoMode*

can_io_mode

CAN IO Mode.

This property indicates the I/O Mode the interface is using.

The value is initialized from the database cluster when the session is created and cannot be changed later. However, you can transmit standard CAN frames on a CAN FD network.

Type *nixnet._enums.CanIoMode*

can_lstn_only

Listen Only? property configures whether the CAN interface transmits any information to the CAN bus.

When this property is false, the interface can transmit CAN frames and acknowledge received CAN frames.

When this property is true, the interface can neither transmit CAN frames nor acknowledge a received CAN frame. The true value enables passive monitoring of network traffic, which can be useful for debugging scenarios when you do not want to interfere with a communicating network cluster.

Type bool

can_pend_tx_order

Pending Transmit Order

The Pending Transmit Order property configures how the CAN interface manages the internal queue of frames. More than one frame may desire to transmit at the same time. NI-XNET stores the frames in an internal queue and transmits them onto the CAN bus when the bus is idle.

Note: You can modify this property only when the interface is stopped.

Note: Setting this property causes the internal queue to be flushed. If you start a session, queue frames, and then stop the session and change this mode, some frames may be lost. Set this property to the desired value once; do not constantly change modes.

Type *nixnet._enums.CanPendTxOrder*

can_sing_shot

Single Shot Transmit?

The Single Shot Transmit? property configures whether the CAN interface retries failed transmissions.

When this property is false, failed transmissions retry as specified by the CAN protocol (ISO 11898-1, 6.11 Automatic Retransmission). If a CAN frame is not transmitted successfully, the interface attempts to retransmit the frame as soon as the bus is idle again. This retransmit process continues until the frame is successfully transmitted.

When this property is true, failed transmissions do not retry. If a CAN frame is not transmitted successfully, no further transmissions are attempted.

Note: You can modify this property only when the interface is stopped.

Note: Setting this property causes the internal queue to be flushed. If you start a session, queue frames, and then stop the session and change this mode, some frames may be lost. Set this property to the desired value once; do not constantly change modes.

Type bool

can_tcvr_state
CAN Transceiver State.

The Transceiver State property configures the CAN transceiver and CAN controller modes. The transceiver state controls whether the transceiver is asleep or communicating, as well as configuring other special modes.

Type *nixnet._enums.CanTcvrState*

can_tcvr_type
CAN Transceiver Type.

For XNET hardware that provides a software-selectable transceiver, the Transceiver Type property allows you to set the transceiver type. Use the XNET Interface CAN.Tranceiver Capability property to determine whether your hardware supports a software-selectable transceiver.

The default value for this property depends on your type of hardware. If you have fixed-personality hardware, the default value is the hardware value. If you have hardware that supports software-selectable transceivers, the default is High-Speed.

Type *nixnet._enums.CanTcvrType*

can_term
CAN Termination.

The Termination property configures the onboard termination of the NI-XNET interface CAN connector (port). The enumeration is generic and supports two values: Off and On. However, different CAN hardware has different termination requirements, and the Off and On values have different meanings, see *nixnet._enums.CanTerm*.

Note: You can modify this property only when the interface is stopped.

Note: This property does not take effect until the interface is started.

Type *nixnet._enums.CanTerm*

can_transmit_pause
CAN Transmit Pause.

When this property is enabled, the CAN hardware waits for two bit times before transmitting the next frame. This allows other CAN nodes to transmit lower priority CAN messages while this CAN node is transmitting high-priority CAN messages with high speed.

Type bool

can_tx_io_mode
CAN Transmit IO Mode

This property specifies the I/O Mode the interface uses when transmitting a CAN frame. By default, it is the same as the XNET Cluster CAN:I/O Mode property. However, even if the interface is in CAN FD+BRS mode, you can force it to transmit frames in the standard CAN format. For this purpose, set this property to CAN.

The Transmit I/O mode may not exceed the mode set by the XNET Cluster CAN:I/O Mode property.

Note: This property is not supported in CAN FD+BRS ISO mode. If you are using ISO CAN FD mode, you define the transmit I/O mode in the database with the I/O Mode property of the frame. (When a database is not used (for example, in frame stream mode), define the transmit I/O mode with the frame type field of the frame data.) Note that ISO CAN FD mode is the default mode for CAN FD in NI-XNET.

Note: This property affects only the transmission of frames. Even if you set the transmit I/O mode to CAN, the interface still can receive frames in FD modes (if the XNET Cluster CAN:I/O Mode property is configured in an FD mode).

Type *nixnet._enums.CanIoMode*

echo_tx

Echo Transmit?

Determines whether Frame Input or Signal Input sessions contain frames that the interface transmits.

When this property is true, and a frame transmit is complete for an Output session, the frame is echoed to the Input session. Frame Input sessions can use the Flags field to differentiate frames received from the bus and frames the interface transmits. When reading frames with the *nixnet.types.RawFrame*, you can parse the Flags field manually by reviewing the Raw Frame Format section. Signal Input sessions cannot differentiate the origin of the incoming data.

Note: Echoed frames are placed into the input sessions only after the frame transmit is complete. If there are bus problems (for example, no listener) such that the frame did not transmit, the frame is not received.

Type bool

lin_alw_start_wo_bus_pwr

LIN Start Allowed without Bus Power?

Configures whether the LIN interface does not check for bus power present at interface start, or checks and reports an error if bus power is missing.

When this property is true, the LIN interface does not check for bus power present at start, so no error is reported if the interface is started without bus power.

When this property is false, the LIN interface checks for bus power present at start, and an error is reported if the interface is started without bus power.

Note: You can modify this property only when the interface is stopped.

Type bool

lin_break_length

LIN Break Length

The length of the serial break used at the start of a frame header (schedule entry). The value is specified in bit-times.

The valid range is 10-36 (inclusive). The default value is 13, which is the value the LIN standard specifies.

At baud rates below 9600, the upper limit may be lower than 36 to avoid violating hold times for the bus. For example, at 2400 baud, the valid range is 10-14.

Note: This property is applicable only when the interface is the master.

Type int

lin_checksum_to_in_strm
LIN Checksum to Input Stream?

Configure the hardware to place the received checksum for each LIN Data frame into the Event ID (Info) field. When `False`, the Event ID field contains `0` for all LIN Data stream input frames.

Type bool

lin_diag_p2min
LIN Diag P2min

This is the minimum time in seconds between reception of the last frame of the diagnostic request message and transmission of the response for the first frame in the diagnostic response message by the slave.

Note: This property applies only to the interface as slave.

Type float

lin_diag_stmin
LIN Diag STmin

master: The minimum time in seconds the interface places between the end of transmission of a frame in a diagnostic request message and the start of transmission of the next frame in the diagnostic request message.

slave: The minimum time in seconds the interface places between the end of transmission of a frame in a diagnostic response message and the start of transmission of the response for the next frame in the diagnostic response message.

Type float

lin_master
LIN Master?

Specifies the NI-XNET LIN interface role on the network: master (true) or slave (false).

In a LIN network (cluster), there always is a single ECU in the system called the master. The master transmits a schedule of frame headers. Each frame header is a remote request for a specific frame ID. For each header, typically a single ECU in the network (slave) responds by transmitting the requested ID payload. The master ECU can respond to a specific header as well, and thus the master can transmit payload data for the slave ECUs to receive.

The default value for this property is false (slave). This means that by default, the interface does not transmit frame headers onto the network. When you use input sessions, you read frames that other ECUs transmit. When you use output sessions, the NI-XNET interface waits for the remote master to send a header for a frame in the output sessions, then the interface responds with data for the requested frame.

If you call the *`nixnet._session.base.SessionBase.change_lin_schedule`* function to request execution of a schedule, that implicitly sets this property to true (master). You also can set this property to true using, but no schedule is active by default, so you still must call the *`nixnet._session.`*

base.SessionBase.change_lin_schedule function at some point to request a specific schedule.

Regardless of this property's value, you use can input and output sessions. This property specifies which hardware transmits the scheduled frame headers: NI-XNET (true) or a remote master ECU (false).

Type bool

lin_no_response_to_in_strm

LIN No Response Frames to Input Stream?

Configure the hardware to place a LIN no response frame into the Stream Input queue after it is generated. A no response frame is generated when the hardware detects a header with no response. For more information about the no response frame, see `nixnet.types.NoResponseFrame`.

Type bool

lin_ostr_slv_rsp_lst_by_nad

LIN Output Stream Slave Response List By NAD

A list of NADs for use with the replay feature (*nixnet._session.intf.Interface.out_strm_timng* set to Replay Exclusive or Replay Inclusive).

For LIN, the array of frames to replay might contain multiple slave response frames, each with the same slave response identifier, but each having been transmitted by a different slave (per the NAD value in the data payload). This means that processing slave response frames for replay requires two levels of filtering. First, you can include or exclude the slave response frame or ID for replay using Interface:Output Stream List or Interface:Output Stream List By ID. If you do not include the slave response frame or ID for replay, no slave responses are transmitted. If you do include the slave response frame or ID for replay, you can use the Output Stream Slave Response List by NAD property to filter which slave responses (per the NAD values in the array) are transmitted. This property is always inclusive, regardless of the replay mode (inclusive or exclusive). If the NAD is in the list and the response frame or ID has been enabled for replay, any slave response for that NAD is transmitted. If the NAD is not in the list, no slave response for that NAD is transmitted.

Type list of int

lin_sched_names

LIN Schedule Names

List of schedules for use when the NI-XNET LIN interface acts as a master (`lin_master` is true). When the interface is master, you can pass the index of one of these schedules to the *nixnet._session.base.SessionBase.change_lin_schedule* function to request a schedule change.

This list of schedules is the same as `Cluster.lin_schedules` used to configure the session.

Type list of str

lin_term

LIN Termination

The Termination property configures the NI-XNET interface LIN connector (port) onboard termination. The enumeration is generic and supports two values: Off (disabled) and On (enabled).

Per the LIN 2.1 standard, the Master ECU has a ~1 kOhm termination resistor between Vbat and Vbus. Therefore, use this property only if you are using your interface as the master and do not already have external termination.

Note: You can modify this property only when the interface is stopped.

Note: This property does not take effect until the interface is started.

Type *nixnet._enums.LinTerm*

out_strm_list

Output Stream List.

The Output Stream List property provides a list of frames for use with the replay feature (*out_strm_timng* property set to *OutStrmTimng* `REPLAY_EXCLUSIVE` or `REPLAY_INCLUSIVE`). In Replay Exclusive mode, the hardware transmits only frames that do not appear in the list. In Replay Inclusive mode, the hardware transmits only frames that appear in the list. For a LIN interface, the header of each frame written to stream output is transmitted, and the Exclusive or Inclusive mode controls the response transmission. Using these modes, you can either emulate an ECU (Replay Inclusive, where the list contains the frames the ECU transmits) or test an ECU (Replay Exclusive, where the list contains the frames the ECU transmits), or some other combination.

This property's data type is an array of database handles to frames. If you are not using a database file or prefer to specify the frames using CAN arbitration IDs or LIN unprotected IDs, you can use Interface:Output Stream List By ID instead of this property.

Note: Only CAN and LIN interfaces currently support this property.

out_strm_list_by_id

Output Stream List by Frame Identifier.

Provide a list of frames for use with the replay feature Interface:Output Stream Timing property.

This property serves the same purpose as Interface:Output Stream List, in that it provides a list of frames for replay filtering. This property provides an alternate format for you to specify the frames by their CAN arbitration ID or LIN unprotected ID. The property's data type is an array of integers. Each integer represents a CAN or LIN frame's identifier, using the same encoding as *nixnet.types.RawFrame*.

For CAN Frames, see *nixnet.types.CanIdentifier* for parsing and generating raw identifiers.

LIN frame ID values may be within the range of possible LIN IDs (0-63).

See also *Interface.out_strm_list*.

Type int

out_strm_timng

Output Stream Timing.

The Output Stream Timing property configures how the hardware transmits frames queued using a Frame Output Stream session.

See also *Interface.out_strm_list*.

Note: Only CAN and LIN interfaces currently support this property.

Type *nixnet._enums.OutStrmTimng*

set_can_ext_tcvr_config(*value*)

Configure XS series CAN hardware to communicate properly with your external transceiver.

Parameters **value** (*int*) – Bitfield

set_lin_sleep (*state*)
Set LIN Sleep State

Use the Sleep property to change the NI-XNET LIN interface sleep/awake state and optionally to change remote node (ECU) sleep/awake states.

Note: Setting a new value is effectively a request, and the function returns before the request is complete. To detect the current interface sleep/wake state, use *nixnet._session.base.SessionBase.lin_comm*.

Parameters **state** (*nixnet._enums.LinSleep*) – Desired state.

src_term_start_trigger
Source Terminal Start Trigger

Specifies the name of the internal terminal to use as the interface Start Trigger.

This property is supported for C Series modules in a CompactDAQ chassis. It is not supported for CompactRIO, PXI, or PCI (refer to *nixnet._session.base.SessionBase.connect_terminals* for those platforms).

The digital trigger signal at this terminal is for the Start Interface transition, to begin communication for all sessions that use the interface. This property routes the start trigger, but not the timebase (used for timestamp of received frames and cyclic transmit of frames). Routing the timebase is not required for CompactDAQ, because all modules in the chassis automatically use a shared timebase.

Use this property to connect the interface Start Trigger to triggers in other modules and/or interfaces. When you read this property, you specify the interface Start Trigger as the source of a connection. When you write this property, you specify the interface Start Trigger as the destination of a connection, and the value you write represents the source.

The connection this property creates is disconnected when you clear (close) all sessions that use the interface.

Type string

start_trig_to_in_strm
Start Trigger Frames to Input Stream?

Configures the hardware to place a start trigger frame into the Stream Input queue after it is generated. A Start Trigger frame is generated when the interface is started.

The start trigger frame is especially useful if you plan to log and replay CAN data.

Type bool

nixnet.session.j1939

class nixnet._session.j1939.**J1939** (*handle*)
Bases: object

J1939 configuration for a session

include_dest_addr_in_pgn
SAE J1939 Include Destination Address in PGN

Incoming J1939 frames are matched to an XNET database by the Parameter Group Number (PGN) of the frame. When receiving PDU1 frames, the destination address of the frame (J1939 PS field) is ignored when calculating the PGN, in accordance to the J1939 specification. This causes an XNET session to receive all frames that share the same PGN, making it difficult to distinguish destinations for traffic.

When set to `True`, this property instructs NI-XNET to include the destination address when extracting the PGN from the frame. This allows the same PGN sent to different destination addresses to be handled by separate input sessions.

This property may be set at any time. When set after session start, it will not affect frames already received.

This property is valid only for input sessions. It is not valid for stream sessions. This property affects all frames in a session.

Type bool

nixnet.session.base

class `nixnet._session.base.`**`SessionBase`**(*database_name*, *cluster_name*, *list*, *interface_name*, *mode*)

Bases: `object`

Session base object.

`application_protocol`

This property returns the application protocol that the session uses.

The database used with the session determines the application protocol.

Type *`nixnet._enums.AppProtocol`*

`auto_start`

Automatically starts the output session on the first call to the appropriate write function.

For input sessions, start always is performed within the first call to the appropriate read function (if not already started using *`nixnet._session.base.SessionBase.start`*). This is done because there is no known use case for reading a stopped input session.

For output sessions, as long as the first call to the appropriate write function contains valid data, you can leave this property at its default value of true. If you need to call the appropriate write function multiple times prior to starting the session, or if you are starting multiple sessions simultaneously, you can set this property to false. After calling the appropriate write function as desired, you can call *`nixnet._session.base.SessionBase.start`* to start the session(s).

When automatic start is performed, it is equivalent to *`nixnet._session.base.SessionBase.start`* with scope set to Normal. This starts the session itself, and if the interface is not already started, it starts the interface also.

Type bool

`can_comm`

CAN Communication state

Type *`nixnet.types.CanComm`*

`change_lin_diagnostic_schedule`(*schedule*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the diagnostic schedule.

Parameters **`schedule`** (*`nixnet._enums.LinDiagnosticSchedule`*) – Diagnostic schedule that the LIN master executes.

change_lin_schedule(*sched_index*)

Writes communication states of an XNET session.

This function writes a request for the LIN interface to change the running schedule.

According to the LIN protocol, only the master executes schedules, not slaves. If the `nixnet._session.intf.Interface.lin_master` property is false (slave), this write function implicitly sets that property to true (master). If the interface currently is running as a slave, this write returns an error, because it cannot change to master while running.

Parameters **sched_index** (*int*) – Index to the schedule table that the LIN master executes.

The schedule tables are sorted the way they are returned from the database with the *nixnet.database._cluster.Cluster.lin_schedules* property.

check_fault()

Check for an asynchronous fault.

A fault is an error that occurs asynchronously to the NI-XNET application calls. The fault cause may be related to network communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, nxReadState provides a detection method distinct from the status of NI-XNET function calls, yet easy to use alongside the common practice of checking the communication state.

close()

Close (clear) the XNET session.

This function stops communication for the session and releases all resources the session uses. It internally calls `nixnet._session.base.SessionBase.stop` with normal scope, so if this is the last session using the interface, communication stops.

You typically use 'close' when you need to close the existing session to create a new session that uses the same objects. For example, if you create a session for a frame named frame_a using Frame Output Single-Point mode, then you create a second session for frame_a using Frame Output Queued mode, the second call to the session constructor returns an error, because frame_a can be accessed using only one output mode. If you call 'close' before the second constructor call, you can close the previous use of frame_a to create the new session.

cluster_name

This property returns the cluster (network) name used with the session.

Type str

connect_terminals(*source*, *destination*)

Connect terminals on the XNET interface.

This function connects a source terminal to a destination terminal on the interface hardware. The XNET terminal represents an external or internal hardware connection point on a National Instruments XNET hardware product. External terminals include PXI Trigger lines for a PXI card, RTSI terminals for a PCI card, or the single external terminal for a C Series module. Internal terminals include timebases (clocks) and logical entities such as a start trigger.

The terminal inputs use the Terminal I/O names. Typically, one of the pair is an internal and the other an external.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

database_name

This property returns the database name used with the session.

Type str

disconnect_terminals(*source*, *destination*)

Disconnect terminals on the XNET interface.

This function disconnects a specific pair of source/destination terminals previously connected with *nixnet._session.base.SessionBase.connect_terminals*.

When the final session for a given interface is cleared, NI-XNET automatically disconnects all terminal connections for that interface. Therefore, 'disconnect_terminals' is not required for most applications.

This function typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using *nixnet._session.base.SessionBase.stop* with the Interface Only scope. Then you can call 'disconnect_terminals' and *nixnet._session.base.SessionBase.connect_terminals* to adjust terminal connections. Finally, you can call *nixnet._session.base.SessionBase.start* with the Interface Only scope to restart the interface.

You can disconnect only a terminal that has been previously connected. Attempting to disconnect a nonconnected terminal results in an error.

Parameters

- **source** (*str*) – Connection source name.
- **destination** (*str*) – Connection destination name.

flush()

Flushes (empties) all XNET session queues.

With the exception of single-point modes, all sessions use queues to store frames. For input modes, the queues store frame values (or corresponding signal values) that have been received, but not obtained by calling the read function. For output sessions, the queues store frame values provided to write function, but not transmitted successfully.

nixnet._session.base.SessionBase.start and *nixnet._session.base.SessionBase.stop* have no effect on these queues. Use 'flush' to discard all values in the session's queues.

For example, if you call a write function to write three frames, then immediately call *nixnet._session.base.SessionBase.stop*, then call *nixnet._session.base.SessionBase.start* a few seconds later, the three frames transmit. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, no frames transmit.

As another example, if you receive three frames, then call *nixnet._session.base.SessionBase.stop*, the three frames remains in the queue. If you call *nixnet._session.base.SessionBase.start* a few seconds later, then call a read function, you obtain the three frames received earlier, potentially followed by other frames received after calling *nixnet._session.base.SessionBase.start*. If you call 'flush' between *nixnet._session.base.SessionBase.stop* and *nixnet._session.base.SessionBase.start*, read function returns only frames received after the calling *nixnet._session.base.SessionBase.start*.

intf

Returns the Interface configuration object for the session.

Type *nixnet._session.intf.Interface*

j1939

Returns the J1939 configuration object for the session.

Type *nixnet._session.j1939.J1939*

lin_comm
LIN Communication state

Type *nixnet.types.LinComm*

mode
This property returns the mode associated with the session.

For more information, refer to *nixnet._enums.CreateSessionMode*.

Type *nixnet._enums.CreateSessionMode*

num_pend
This property returns the number of values (frames or signals) pending for the session.

For input sessions, this is the number of frame/signal values available to the appropriate read function. If you call the appropriate read function with number to read of this number and timeout of 0.0, the appropriate read function should return this number of values successfully.

For output sessions, this is the number of frames/signal values provided to the appropriate write function but not yet transmitted onto the network.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

num_unused
This property returns the number of values (frames or signals) unused for the session.

If you get this property prior to starting the session, it provides the size of the underlying queue(s). Contrary to the Queue Size property, this value is in number of frames for Frame I/O, not number of bytes; for Signal I/O, it is the number of signal values in both cases. After start, this property returns the queue size minus the *Number of Values Pending* property.

For input sessions, this is the number of frame/signal values unused in the underlying queue(s).

For output sessions, this is the number of frame/signal values you can provide to a subsequent call to the appropriate write function. If you call the appropriate write function with this number of values and timeout of 0.0, it should return success.

Stream frame sessions using FlexRay or CAN FD protocol may use a variable size of frames. In these cases, this property assumes the largest possible frame size. If you use smaller frames, the real number of pending values might be higher.

The largest possible frames sizes are:

CAN FD: 64 byte payload.

FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay Payload Length Maximum property provides this value.

Type int

protocol

This property returns the protocol that the interface in the session uses.

Type *nixnet._enums.Protocol*

queue_size

Get or set queue size.

For output sessions, queues store data passed to the appropriate write function and not yet transmitted onto the network. For input sessions, queues store data received from the network and not yet obtained using the appropriate read function.

For most applications, the default queue sizes are sufficient. You can write to this property to override the default. When you write (set) this property, you must do so prior to the first session start. You cannot set this property again after calling *nixnet._session.base.SessionBase.stop*.

For signal I/O sessions, this property is the number of signal values stored. This is analogous to the number of values you use with the appropriate read or write function.

For frame I/O sessions, this property is the number of bytes of frame data stored.

For standard CAN or LIN frame I/O sessions, each frame uses exactly 24 bytes. You can use this number to convert the Queue Size (in bytes) to/from the number of frame values.

For CAN FD and FlexRay frame I/O sessions, each frame value size can vary depending on the payload length. For more information, refer to Raw Frame Format.

For Signal I/O XY sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. According to the formulas below, the default queue sizes can be different for each frame. If you read the default Queue Size property for a Signal Input XY session, the largest queue size is returned, so that a call to the appropriate read function of that size can empty all queues. If you read the default Queue Size property for a Signal Output XY session, the smallest queue size is returned, so that a call to the appropriate write function of that size can succeed when all queues are empty. If you write the Queue Size property for a Signal I/O XY session, that size is used for all frames, so you must ensure that it is sufficient for the frame with the fastest transmit time.

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

For Single-Point sessions (signal or frame), this property is ignored. Single-Point sessions always use a value of 1 as the effective queue size.

Type int

start(*scope=<StartStopScope.NORMAL: 0>*)

Start communication for the XNET session.

Because the session is started automatically by default, this function is optional. This function is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the *nixnet._session.base.SessionBase.auto_start* property.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can start each logical unit separately. When a session is started, all contained frames or signals are placed in a state where they are ready to communicate. When the interface is started, it takes data from all started sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

If an output session starts before you write data, or you read an input session before it receives a frame, default data is used. For more information, refer to the XNET Frame Default Payload and XNET Signal Default Value properties.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`state`

Session running state.

Type *`nixnet._enums.SessionInfoState`*

`stop`(*scope=<StartStopScope.NORMAL: 0>*)

Stop communication for the XNET session.

Because the session is stopped automatically when closed (cleared), this function is optional.

For each physical interface, the NI-XNET hardware is divided into two logical units:

Sessions: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.

Interface: The interface physically connects to the bus and transmits (or receives) data for the sessions.

You can stop each logical unit separately. When a session is stopped, all contained frames or signals are placed in a state where they are no longer ready to communicate. When the interface is stopped, it no longer takes data from sessions to communicate with other nodes on the bus. For a specification of the state models for the session and interface, refer to State Models.

Parameters **`scope`** (*`nixnet._enums.StartStopScope`*) – Describes the impact of this operation on the underlying state models for the session and its interface.

`time_communicating`

Time the interface started communicating.

The time is usually later than `time_start` because the interface must undergo a communication startup procedure.

Type int

`time_current`

Current interface time.

Type int

`time_start`

Time the interface was started.

Type int

`wait_for_intf_communicating`(*timeout=10*)

Wait for the interface to begin communication on the network.

If a start trigger is configured for the interface, this first waits for the trigger. Once the interface is started, this waits for the protocol's communication state to transition to a value that indicates communication with remote nodes.

After this wait succeeds, calls to 'read_state' will return:

nixnet._enums.CanCommState: 'constants.CAN_COMM.ERROR_ACTIVE'

nixnet._enums.CanCommState: 'constants.CAN_COMM.ERROR_PASSIVE'

'constants.ReadState.TIME_COMMUNICATING': Valid time for communication (invalid time of 0 prior)

Parameters **timeout** (*float*) – The maximum amount of time to wait in seconds.

wait_for_intf_remote_wakeup(*timeout=10*)

Wait for interface remote wakeup.

Wait for the interface to wakeup due to activity by a remote node on the network. This wait is used for CAN, when you set the 'can_tcvr_state' property to 'constants.CanTcvrState.SLEEP'. Although the interface itself is ready to communicate, this places the transceiver into a sleep state. When a remote CAN node transmits a frame, the transceiver wakes up, and communication is restored. This wait detects that remote wakeup.

This wait is used for LIN when you set 'lin_sleep' property to 'constants.LinSleep.REMOTE_SLEEP' or 'constants.LinSleep.LOCAL_SLEEP'. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

Parameters **timeout** (*float*) – The maximum amount of time to wait in seconds.

wait_for_transmit_complete(*timeout=10*)

Wait for transmition to complete.

All frames written for the session have been transmitted on the bus. This condition applies to CAN, LIN, and FlexRay. This condition is state based, and the state is Boolean (true/false).

Parameters **timeout** (*float*) – The maximum amount of time to wait in seconds.

4.2.2 nixnet.convert

class nixnet.convert.**SignalConversionSinglePointSession**(*database_name*, *cluster_name*, *signals*)

Bases: object

Convert NI-XNET signal data to frame data or vice versa.

Conversion works similar to Single-Point mode. You specify a set of signals that can span multiple frames. Signal to frame conversion reads a set of values for the signals specified and writes them to the respective frame(s). Frame to signal conversion parses a set of frames and returns the latest signal value read from a corresponding frame.

application_protocol

This property returns the application protocol that the session uses.

The database used with the session determines the application protocol.

Type *nixnet._enums.AppProtocol*

close()

Close (clear) the XNET session.

cluster_name

This property returns the cluster (network) name used with the session.

Type str

convert_frames_to_signals (*frames*)

Convert Frames to signals.

The frames passed into the `frames` array are read one by one, and the signal values found are written to internal buffers for each signal. Frames are identified by their identifier (FlexRay: slot) field. After all frames in `frames` array are processed, the internal signal buffers' status is returned with the corresponding timestamps from the frames where a signal value was found. The signal internal buffers' status is being preserved over multiple calls to this function.

This way, for example, data returned from multiple calls of nxFrameRead for a Frame Input Stream Mode session (or any other Frame Input session) can be passed to this function directly.

Note: Frames unknown to the session are silently ignored.

convert_signals_to_frames (*signals*, *frame_type=<class 'nixnet.types.XnetFrame'>*)

Convert signals to frames.

The signal values written to the `signals` array are written to a raw frame buffer array. For each frame included in the session, one frame is generated in the array that contains the signal values. Signals not present in the session are written as their respective default values; empty space in the frames that signals do not occupy is written with the frame's default payload.

The frame header values are filled with appropriate values so that this function's output can be directly written to a Frame Output session.

Parameters

- **signals** (*list of float*) – Values corresponding to signals configured in this session.
- **frame_type** (*nixnet.types.FrameFactory*) – A factory for the desired frame formats.

Yields *nixnet.types.Frame*

database_name

This property returns the database name used with the session.

Type str

j1939

Returns the J1939 configuration object for the session.

Type *nixnet._session.j1939.J1939*

mode

This property returns the mode associated with the session.

For more information, refer to *nixnet._enums.CreateSessionMode*.

Type *nixnet._enums.CreateSessionMode*

protocol

This property returns the protocol that the interface in the session uses.

Type *nixnet._enums.Protocol*

signals
Operate on session's signals

Type *nixnet._session.signals.Signals*

4.2.3 nixnet.system

nixnet.system.system

class nixnet.system.system.**System**
Interact with the NI driver and interface hardware.

databases
Operate on systems's database's aliases

Type *nixnet.system._databases.AliasCollection*

dev_refs
Physical XNET devices in the system.

Type iter of *nixnet.system._device.Device*

intf_refs
Available interfaces on the system.

Type iter of *nixnet.system._interface.Interface*

intf_refs_all
Available interfaces on the system.

This Includes those not equipped with a Transceiver Cable.

Type iter of *nixnet.system._interface.Interface*

intf_refs_can
Available interfaces on the system (CAN Protocol).

Type iter of *nixnet.system._interface.Interface*

intf_refs_lin
Available interfaces on the system (LIN Protocol).

Type iter of *nixnet.system._interface.Interface*

ver
The driver version (larger numbers imply a newer version).

Use this for:

- Determining the driver functionality or release date
- Determining upgrade availability

Type *nixnet.types.DriverVersion*

nixnet.system.databases

class nixnet.system._databases.**Alias**(*database_alias*, *database_filepath*)
Bases: object

Alias alias.

filepath
Get the filepath associated with the Alias object

Type str

class nixnet.system._databases.**AliasCollection**(*handle*)
Bases: collections.abc.Mapping

Alias aliases.

add_alias(*database_alias*, *database_filepath*, *default_baud_rate=None*)
Add a new alias with baud rate size of up to 64 bits to a database file.

NI-XNET uses alias names for database files. The alias names provide a shorter name for display, allow for changes to the file system without changing the application.

This function is supported on Windows only.

Parameters

- **database_alias** (*str*) – Provides the desired alias name. Unlike the names of other XNET database objects, the alias name can use special characters such as space and dash. Commas are not allowed in the alias name. If the alias name already exists, this function changes the previous filepath to the specified filepath.
- **database_filepath** (*str*) – Provides the path to the CANdb, FIBEX, or LDF file. Commas are not allowed in the filepath name.
- **default_baud_rate** (*int*) – Provides the default baud rate, used when filepath refers to a CANdb database (.dbc) or an NI-CAN database (.ncd). These database formats are specific to CAN and do not specify a cluster baud rate. Use this default baud rate to specify a default CAN baud rate to use with this alias. If database_filepath refers to a FIBEX database (.xml) or LIN LDF file, the default_baud_rate parameter is ignored. The FIBEX and LDF database formats require a valid baud rate for every cluster, and NI-XNET uses that baud rate as the default.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items()
Return all aliases and database objects associated with those aliases in the system.

Yields An iterator to tuple pairs of alias and database objects in the system.

keys()
Return all keys (alias names) used in the AliasCollection object.

Yields An iterator to all the keys in the Alias object.

values()
Return all Alias objects in the system.

Yields An iterator to all the values in the AliasCollection object.

nixnet.system.device

class nixnet.system._device.**Device**(*handle*)
Bases: object

Physical XNET devices in the system.

form_fac
XNET board form factor.

Type *nixnet._enums.DevForm*

intf_refs
Interfaces associated with this device.

Type iter of *nixnet.system._interface.Interface*

intf_refs_all
Interfaces associated with this device.

This Includes those not equipped with a Transceiver Cable.

Type iter of *nixnet.system._interface.Interface*

num_ports
The number of physical port connectors on the XNET board.

Type int

num_ports_all
The number of physical port connectors on the XNET board.

This Includes those not equipped with a Transceiver Cable.

Type int

product_name
The XNET device product name.

Type str

product_num
The numeric portion of the XNET device product name.

Type int

ser_num
Serial number associated with the XNET device.

Type int

slot_num
Physical slot where the module is located within a chassis.

Type int

nixnet.system.interface

class nixnet.system._interface.**Interface**(*handle*)
Bases: object

Interfaces associated with a physical hardware device.

blink(*modifier*)
Blinks LEDs for the XNET interface to identify its physical port in the system.

Each XNET device contains one or two physical ports. Each port is labeled on the hardware as Port 1 or Port 2. The XNET device also provides two LEDs per port. For a two-port board, LEDs 1 and 2 are assigned to Port 1, and LEDs 3 and 4 are assigned to physical Port 2.

When your application uses multiple XNET devices, this function helps to identify each interface to associate its software behavior to its hardware connection (port). Prior to running your XNET sessions, you can call this function to blink the interface LEDs.

For example, if you have a system with three PCI CAN cards, each with two ports, you can use this function to blink the LEDs for interface CAN4, to identify it among the six CAN ports.

The LEDs of each port support two states:

Identification: Blink LEDs to identify the physical port assigned to the interface.

In Use: LED behavior that XNET sessions control.

Identification LED State

You can use the `blink` function only in the Identification state. If you call this function while one or more XNET sessions for the interface are open (created), it returns an error, because the port's LEDs are in the In Use state.

In Use LED State

When you create an XNET session for the interface, the LEDs for that physical port transition to the In Use state. If you called the `blink` function previously to enable blinking for identification, that LED behavior no longer applies. The In Use LED state remains until all XNET sessions are cleared. This typically occurs when the application terminates. The patterns that appear on the LEDs while In Use are documented in LEDs.

Parameters **`moodifier`** (*`nixnet._enums.BlinkMode`*) – Controls LED blinking

Both LEDs blink green (not red). The blinking rate is approximately three times per second.

`can_tcvr_cap`

Indicates the CAN bus physical transceiver support.

Type *`nixnet._enums.CanTcvrCap`*

`can_term_cap`

Indicates whether the XNET interface can terminate the CAN bus.

Signal reflections on the CAN bus can cause communication failure. To prevent reflections, termination can be present as external resistance or resistance the XNET board applies internally. This property determines whether the XNET board can add termination to the bus.

Type *`nixnet._enums.CanTermCap`*

`dongle_compatible_firmware_version`

The oldest driver version compatible with the connected Transceiver Cable's firmware.

The number is relative to the first driver version that supported the Transceiver Cable, starting with 1 for the original revision.

..note:: A Transceiver Cable running an updated firmware version may require a later XNET driver than the version it shipped with for operation.

Type int

`dongle_compatible_revision`

The oldest driver version compatible with the connected Transceiver Cable's hardware revision.

The number is relative to the first driver version that supported the particular Transceiver Cable model, starting with 1 for the original revision.

Note: A Transceiver Cable hardware revision might require a later XNET driver than the version that introduced support for this model for operation.

Type int

`dongle_firmware_version`
The connected Transceiver Cable's firmware revision number.

Type int

`dongle_id`
Indicates the connected Transceiver Cable's type.

Dongle-Less Design indicates this interface is not a Transceiver Cable but a regular XNET expansion card, cDAQ Module, and so on.

Type *`nixnet._enums.DongleId`*

`dongle_revision`
The connected Transceiver Cable's hardware revision number.

Type int

`dongle_state`
Indicates the connected Transceiver Cable's state.

Some Transceiver Cable types require external power from the network connector for operation. Refer to the hardware-specific manual for more information.

Type *`nixnet._enums.DongleState`*

`num`
The unique number associated with the XNET interface.

The XNET driver assigns each port connector in the system a unique number XNET driver. This number, plus its protocol name, is the interface name.

Type int

`port_num`
Physical port number printed near the connector on the XNET device.

The port numbers on an XNET board are physically identified with numbering. Use this property, along with the XNET Device Serial Number property, to associate an XNET interface with a physical (XNET board and port) combination.

Type int

`protocol`
Protocol supported by the interface.

Type *`nixnet._enums.Protocol`*

4.2.4 nixnet.database

nixnet.database.cluster

class `nixnet.database._cluster.`**`Cluster`**(***kwargs*)
Bases: `nixnet.database._database_object.DatabaseObject`

Database cluster

`application_protocol`
Get or set the application protocol.

Type *`AppProtocol`*

baud_rate

Get or set the buad rate all custer nodes use.

This baud rate represents the rate from the database, so it is read-only from the session. Use a session interface property (for example, *Interface.baud_rate*) to override the database baud rate with an application-specific baud rate.

CAN

For CAN, this rate can be 33333, 40000, 50000, 62500, 80000, 83333, 100000, 125000, 160000, 200000, 250000, 400000, 500000, 800000, or 1000000. Some transceivers may support only a subset of these values.

LIN

For LIN, this rate can be 2400-20000 inclusive.

If you need values other than these, use the custom settings as described in *Interface.baud_rate*.

Type int

can_fd_baud_rate

Get or set the fast data baud rate when *Cluster.can_io_mode* is `CanIoMode.CAN_FD_BRS`.

Refer to the *CanIoMode* for a description of `CanIoMode.CAN_FD_BRS`. Use a session interface property (for example, *Interface.can_fd_baud_rate*) to override the database fast baud rate with an application-specific fast baud rate.

NI-XNET CAN hardware currently accepts the following numeric baud rates: 200000, 250000, 400000, 500000, 800000, 1000000, 1250000, 1600000, 2000000, 2500000, 4000000, 5000000, and 8000000. Some transceivers may support only a subset of these values.

If you need values other than these, use the custom settings as described in *Interface.can_fd_baud_rate*.

Type int

can_fd_iso_mode

Returns the mode of a CAN FD cluster.

The default is `CanFdIsoMode.ISO`. You define the value in a dialog box that appears when you define an alias for the database.

Type *CanFdIsoMode*

can_io_mode

Get or set the CAN I/O Mode of the cluster.

Type *CanIoMode*

check_config_status()

Check this cluster's configuration status.

By default, incorrectly configured clusters in the database are not returned from *Database.clusters* because they cannot be used in the bus communication. You can change this behavior by setting *Database.show_invalid_from_open* to *True*. When a cluster configuration status becomes invalid after the database is opened, the cluster still is returned from *Database.clusters* even if *Database.show_invalid_from_open* is *False*.

Raises *XnetError* – The cluster is incorrectly configured.

comment

Get or set a comment describing the cluster object.

A comment is a string containing up to 65535 characters.

Type str

`dbc_attributes`
Access the cluster's DBC attributes.

Type *`DbcAttributeCollection`*

`ecus`
Returns a collection of *`Ecu`* objects in this cluster.

An ECU is assigned to a cluster when the ECU object is created. You cannot change this assignment afterwards.

Type *`DbCollection`*

`export`(*db_filepath*)
Exports this cluster to a CANdb++ or LIN database file format.

A CAN cluster is exported as a CANdb++ database file (.dbc). A LIN cluster is exported as a LIN database file (.ldf). If the target file exists, it is overwritten.

Exporting a cluster is not supported under Real-Time (RT).

Parameters **`db_filepath`** (*`str`*) – Contains the pathname to the database file.

`find`(*object_class*, *object_name*)
Finds an object in the database.

This function finds a database object relative to this parent object. This object may be a grandparent or great-grandparent.

If this object is a direct parent (for example, *`Frame`* for *`Signal`*), the `object_name` to search for can be short, and the search proceeds quickly.

If this object is not a direct parent (for example, *`Database`* for *`Signal`*), the `object_name` to search for must be qualified such that it is unique within the scope of this object.

For example, if the class of this object is *`Cluster`*, and `object_class` is *`Signal`*, you can specify `object_name` of `mySignal`, assuming that signal name is unique to the cluster. If not, you must include the *`Frame`* name as a prefix, such as `myFrameA.mySignal`.

NI-XNET supports the following subclasses of `DatabaseObject` as arguments for `object_class`:

- *`nixnet.database.Cluster`*
- *`nixnet.database.Frame`*
- *`nixnet.database.Pdu`*
- *`nixnet.database.Signal`*
- *`nixnet.database.SubFrame`*
- *`nixnet.database.Ecu`*
- *`nixnet.database.LinSched`*
- *`nixnet.database.LinSchedEntry`*

Parameters

- **`object_class`** (`DatabaseObject`) – The class of the object to find.
- **`object_name`** (*`str`*) – The name of the object to find.

Returns An instance of the found object.

Raises

- `ValueError` – Unsupported value provided for argument `object_class`.
- *`XnetError`* – The object is not found.

`frames`

Returns a collection of *`Frame`* objects in this cluster.

A frame is assigned to a cluster when the frame object is created. You cannot change this assignment afterwards.

Type *`DbCollection`*

`lin_schedules`

Returns a collection of *`LinSched`* defined in this cluster.

You assign a LIN schedule to a cluster when you create the LIN schedule object. You cannot change this assignment afterwards. The schedules in this collection are sorted alphabetically by schedule name.

Type *`DbCollection`*

`lin_tick`

Returns the relative time between LIN ticks (relative f64 in seconds).

The *`LinSchedEntry.delay`* property must be a multiple of this tick.

This tick is referred to as the "timebase" in the LIN specification.

The *`Ecu.lin_master`* property defines the Tick property in this cluster. You cannot use the Tick property when there is no LIN Master property defined in this cluster.

Type float

`merge`(*source_obj*, *copy_mode*, *prefix*, *wait_for_complete*)

Merges database objects and related subobjects from the source to this cluster.

The source can be any of the following objects:

- *`Frame`*
- *`Pdu`*
- *`Ecu`*
- *`LinSched`*
- *`Cluster`*

All listed objects must have unique names in the cluster. They are referenced here as objects, as opposed to child objects (for example, a signal is a child of a frame).

If the source object name is not used in the target cluster, this function copies the source objects with the child objects to the target. If an object with the same name exists in this cluster, you can avoid name collisions by specifying the prefix to be added to the name.

If an object with the same name exists in this cluster, the merge behavior depends on the `copy_mode` input.

Example

Target frame F1(v1) has signals S1 and S2(v1). Source frame F1(v2) has signals S2(v2) and S3.

(v1) and (v2) are two versions of one object with same name, but with different properties.

- Result when `copy_mode` is `COPY_USE_SOURCE`: F1(v2), S2(v2), S3.

- Result when `copy_mode` is `COPY_USE_TARGET`: F1(v1), S1, S2(v1).
- Result when `copy_mode` is `MERGE_USE_SOURCE`: F1(v2), S1, S2(v2), S3.
- Result when `copy_mode` is `MERGE_USE_TARGET`: F1(v1), S1, S2(v1), S3.

If the source object is a cluster, this function copies all contained PDUs, ECUs, and LIN schedules with their child objects to this cluster.

Depending on the number of contained objects in the source and destination clusters, the execution can take a longer time. If `wait_for_complete` is `True`, this function waits until the merging process gets completed. If the execution completes without errors, `perecent_complete` returns `100`. If `wait_for_complete` is `False`, the function returns quickly, and `perecent_complete` returns values less than `100`. You must call *`Cluster.merge`* repeatedly until `perecent_complete` returns `100`. You can use the time between calls to perform asynchronous tasks.

Parameters

- **`source_obj`** (*`object`*) – The object to be merged into this cluster.
- **`copy_mode`** (*`Merge`*) – Defines the merging behavior if this cluster already contains an object with the same name.
- **`prefix`** (*`str`*) – The prefix to be added to the source object name if an object with the same name and type exists in this cluster.
- **`wait_for_complete`** (*`bool`*) – Determines whether the function returns directly or waits until the entire transmission is completed.

Returns A value which indicates the merging progress as a percentage. `100` indicates completion.

Return type int

`name`

Get or set the name of the cluster object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

If you use a FIBEX file, the short name comes from the file. If you use a CANdb (.dbc), LDF (.ldf), or NI-CAN (.ncd) file, no cluster name is stored in the file, so NI-XNET uses the name Cluster. If you create the cluster yourself, the name that you provide is used.

A cluster name must be unique for all clusters in a database.

This short name does not include qualifiers to ensure that it is unique, such as the database name. It is for display purposes.

Type str

`pdus`

Returns a collection of *`Pdu`* objects in this cluster.

A PDU is assigned to a cluster when the PDU object is created. You cannot change this assignment afterwards.

Type *`DbCollection`*

`pdus_reqd`

Returns whether using *`PDUs`* in the database API is required for this cluster.

If this property returns `False`, it is safe to use signals as child objects of a frame without PDUs. This behavior is compatible with NI-XNET 1.1 or earlier. Clusters from .dbc, .ncd, or FIBEX 2 files always return `False` for this property, so using PDUs from those files is not required.

If this property returns `True`, the cluster contains PDU configuration, which requires reading the PDUs as frame child objects and then signals as PDU child objects, as shown in the following figure.

Internally, the database always uses PDUs, but shows the same signal objects also as children of a frame.

For this property to return `False`, the following conditions must be fulfilled for all frames in the cluster:

- Only one PDU is mapped to the frame.
- This PDU is not mapped to other frames.
- The PDU Start Bit in the frame is 0.
- The PDU Update Bit is not used.

If the conditions are not fulfilled for a given frame, signals from the frame are still returned, but reading the property returns a warning.

Type bool

`protocol`

Get or set the cluster protocol.

Type *`Protocol`*

`sigs`

Returns a list of all *`Signal`* objects in this cluster.

Type list of *`Signal`*

nixnet.database.database

class `nixnet.database.database.`**`Database`**(*database_name*)

Bases: `nixnet.database._database_object.DatabaseObject`

Opens a database file.

When an already open database is opened, this class grants access to the same database and increases an internal reference counter. A multiple referenced (open) database must be closed as many times as it has been opened. Until it is completely closed, the access to this database remains granted, and the database uses computer resources (memory and handles). For more information, refer to *`Database.close`*.

Parameters **`database_name`** (*`str`*) – The database alias or file pathname to open.

`close`(*close_all_refs=False*)

Closes the database.

For the case that different threads of an application are using the same database, *`Database`* and *`Database.close`* maintain a reference counter indicating how many times the database is open. Every thread can open the database, work with it, and close the database independently using `close_all_refs` set to `False`. Only the last call to *`Database.close`* actually closes access to the database.

Note: `Database.__exit__` calls *`Database.close`* with `close_all_refs` set to `False`. See examples of this in *CAN Dynamic Database Creation* and *LIN Dynamic Database Creation*.

Another option is that only one thread executes *`Database.close`* once, using `close_all_refs` set to `True`, which closes access for all other threads. This may be convenient when, for example, the main program needs to stop all running threads and be sure the database is closed properly, even if some threads could not execute *`Database.close`*.

Parameters **`close_all_refs`** (*`bool`*) – Indicates that a database open multiple times (refer to *`Database`*) should be closed completely (`close_all_refs` is `True`), or just the reference counter should be decremented (`close_all_refs` is `False`), and the database remains open. When the database is closed completely, all references to objects in this database become invalid.

`clusters`

Returns a collection of *`Cluster`* objects in this database.

A cluster is assigned to a database when the cluster object is created. You cannot change this assignment afterwards.

FIBEX and AUTOSAR files can contain any number of clusters, and each cluster uses a unique name.

For CANdb (.dbc), LDF (.ldf), or NI-CAN (.ncd) files, the file contains only one cluster, and no cluster name is stored in the file. For these database formats, NI-XNET uses the name Cluster for the single cluster.

Type *`DbCollection`*

`find`(*object_class*, *object_name*)

Finds an object in the database.

This function finds a database object relative to this parent object. This object may be a grandparent or great-grandparent.

If this object is a direct parent (for example, *`Frame`* for *`Signal`*), the `object_name` to search for can be short, and the search proceeds quickly.

If this object is not a direct parent (for example, *`Database`* for *`Signal`*), the `object_name` to search for must be qualified such that it is unique within the scope of this object.

For example, if the class of this object is *`Cluster`*, and `object_class` is *`Signal`*, you can specify `object_name` of `mySignal`, assuming that signal name is unique to the cluster. If not, you must include the *`Frame`* name as a prefix, such as `myFrameA.mySignal`.

NI-XNET supports the following subclasses of `DatabaseObject` as arguments for `object_class`:

- *`nixnet.database.Cluster`*
- *`nixnet.database.Frame`*
- *`nixnet.database.Pdu`*
- *`nixnet.database.Signal`*
- *`nixnet.database.SubFrame`*
- *`nixnet.database.Ecu`*
- *`nixnet.database.LinSched`*
- *`nixnet.database.LinSchedEntry`*

Parameters

- **object_class** (`DatabaseObject`) – The class of the object to find.
- **object_name** (*str*) – The name of the object to find.

Returns An instance of the found object.

Raises

- `ValueError` – Unsupported value provided for argument `object_class`.
- *XnetError* – The object is not found.

save(*db_filepath=''*)

Saves the open database to a FIBEX 3.1.0 file.

The file extension must be .xml. If the target file exists, it is overwritten.

XNET saves to the FIBEX file only features that XNET sessions use to communicate on the network. If the original file was created using non-XNET software, the target file may be missing details from the original file. For example, NI-XNET supports only linear scaling. If the original FIBEX file used a rational equation that cannot be expressed as a linear scaling, XNET converts this to a linear scaling with factor 1.0 and offset 0.0.

If `db_filepath` is empty, the file is saved to the same FIBEX file specified when opened. If opened as a file path, it uses that file path. If opened as an alias, it uses the file path registered for that alias.

Saving a database is not supported under Real-Time (RT), but you can deploy and use a database saved on Windows on a Real-Time (RT) target (refer to *Database.deploy*).

Parameters **db_filepath** (*str*) – Contains the pathname to the database file or is empty (saves to the original filepath).

show_invalid_from_open

Show or hide *Frame* and *Signal* objects that are invalid.

After opening a database, this property always is set to `False`, meaning that invalid *Cluster*, *Frame*, and *Signal* objects are not returned in properties that return a *DbCollection* for the database (for example, *Cluster.frames* and *Frame.mux_static_signals*). Invalid *Cluster*, *Frame*, and *Signal* objects are incorrectly defined and therefore cannot be used in the bus communication. The `False` setting is recommended when you use the database to create XNET sessions.

In case the database was opened to correct invalid configuration (for example, in a database editor), you must set the property to `True` prior to reading properties that return a *DbCollection* for the database (for example, *Cluster.frames* and *Frame.mux_static_signals*).

For invalid objects, the *Cluster.check_config_status*, *Frame.check_config_status*, and *Signal.check_config_status* methods raise an exception if there is a problem. For valid objects, no error is raised.

Cluster, *Frame*, and *Signal* objects that became invalid after the database is opened are still returned from the *Database.clusters*, *Cluster.frames*, and *Frame.mux_static_signals*, even if *Database.show_invalid_from_open* is `False` and Configuration Status returns an error code. For example, if you open a *Frame* with valid properties, then you set *Signal.start_bit* beyond the *Frame.payload_len*, *Frame.check_config_status* raises an exception, but the frame is returned from *Cluster.frames*.

Type bool

nixnet.database.ecu

class `nixnet.database._ecu.`**`Ecu`**(***kwargs*)

Bases: `nixnet.database._database_object.DatabaseObject`

Database ECU

`check_config_status`()

Check this ECU's configuration status.

By default, incorrectly configured ECUs in the database are not returned from *`Cluster.ecus`* because they cannot be used in the bus communication. You can change this behavior by setting *`Database.show_invalid_from_open`* to *True*. When an ECU configuration status becomes invalid after the database is opened, the ECU still is returned from *`Cluster.ecus`* even if *`Database.show_invalid_from_open`* is *False*.

Raises *`XnetError`* – The ECU is incorrectly configured.

`clst`

Returns the parent cluster to which the ECU is connected.

The parent cluster is determined when the ECU object is created. You cannot change it afterwards.

Type *`Cluster`*

`comment`

Get or set a comment describing the ECU object.

A comment is a string containing up to 65535 characters.

Type str

`dbc_attributes`

Access the ECU's DBC attributes.

Type *`DbcAttributeCollection`*

`j1939_node_name`

Get or set the preferred J1939 node address to be used when simulating this ECU.

If you assign this ECU to an XNET session (*j1939.set_ecu*), XNET will start address claiming for this address using this node name and *`Ecu.j1939_preferred_address`*.

Type int

`j1939_preferred_address`

Get or set the preferred J1939 node address to be used when simulating this ECU.

If you assign this ECU to an XNET session (*j1939.set_ecu*), XNET will start address claiming for this address using *`Ecu.j1939_node_name`* and use the address for the session when the address is granted.

Type int

`lin_config_nad`

Get or set the configured NAD of a LIN slave node.

NAD is the address of a slave node and is used in diagnostic services. Initial NAD is replaced by configured NAD with node configuration services.

Warning: This property is not saved in the FIBEX database. You can import it only from an LDF file.

Type int

lin_function_id
Get or set the function ID.

Function ID is a 16-bit value identifying the function of the LIN node (ECU).

> **Warning:** This property is not saved in the FIBEX database. You can import it only from an LDF file.

Type int

lin_initial_nad
Get or set the initial NAD of a LIN slave node.

NAD is the address of a slave node and is used in diagnostic services. Initial NAD is replaced by configured NAD with node configuration services.

> **Warning:** This property is not saved in the FIBEX database. You can import it only from an LDF file.

Type int

lin_master
Get or set whether the ECU is a LIN master (`True`) or LIN slave (`False`).

Type bool

lin_p2_min
Get or set the minimum time in seconds between frame reception and node response.

This is the minimum time between reception of the last frame of the diagnostic request and the response sent by the node.

> **Warning:** This property is not saved in the FIBEX database. You can import it only from an LDF file.

Type float

lin_protocol_ver
Get or set the version of the LIN standard this ECU uses.

Type *LinProtocolVer*

lin_st_min
Get or set the minimum time in seconds for node preparation.

This is the minimum time the node requires to prepare for the next frame of the diagnostic service.

> **Warning:** This property is not saved in the FIBEX database. You can import it only from an LDF file.

Type float

lin_supplier_id
Get or set the supplier ID.

Supplier ID is a 16-bit value identifying the supplier of the LIN node (ECU).

Warning: This property is not saved in the FIBEX database. You can import it only from an LDF file.

Type int

name

Get or set the name of the ECU object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

An ECU name must be unique for all ECUs in a cluster.

This short name does not include qualifiers to ensure that it is unique, such as the database and cluster name. It is for display purposes.

Type str

rx_frms

Get or set a list of frames the ECU receives.

This property defines all frames the ECU receives. All frames an ECU receives in a given cluster must be defined in the same cluster.

Type list of *Frame*

tx_frms

Get or set a list of frames the ECU transmits.

This property defines all frames the ECU transmits. All frames an ECU transmits in a given cluster must be defined in the same cluster.

Type list of *Frame*

nixnet.database.frame

class `nixnet.database._frame.`**`Frame`**`(`***kwargs*`)`

Bases: `nixnet.database._database_object.DatabaseObject`

Database frame

application_protocol

Get or set the frame's application protocol.

Type *AppProtocol*

can_ext_id

Get or set whether the *Frame.id* property in a CAN cluster is extended.

The frame identifier represents a standard 11-bit (`False`) or extended 29-bit (`True`) arbitration ID.

Type bool

can_io_mode

Get or set the frame's I/O mode.

This property is used in ISO CAN FD+BRS mode only. In this mode, you can specify every frame to be transmitted in CAN 2.0, CAN FD, or CAN FD+BRS mode. CAN FD+BRS frames require the interface to be in CAN FD+BRS mode; otherwise, it is transmitted in CAN FD mode.

When the interface is in Non-ISO CAN FD or Legacy ISO CAN FD mode, this property is disregarded. In Non-ISO CAN FD and Legacy ISO CAN FD mode, you must use *Interface.can_tx_io_mode* to switch the transmit mode.

When the assigned database does not define the property in ISO CAN FD mode, the frames are transmitted with *Interface.can_io_mode*.

Type *CanIoMode*

can_timing_type

Get or set the CAN frame timing.

Because this property specifies the behavior of the frame's transfer within the embedded system (for example, a vehicle), it describes the transfer between ECUs in the network. In the following description, transmitting ECU refers to the ECU that transmits the CAN data frame (and possibly receives the associated CAN remote frame). Receiving ECU refers to an ECU that receives the CAN data frame (and possibly transmits the associated CAN remote frame).

When you use the frame within an NI-XNET session, an output session acts as the transmitting ECU, and an input session acts as a receiving ECU. For a description of how these CAN timing types apply to the NI-XNET session mode, refer to *CAN Timing Type and Session Mode*.

If you are using a FIBEX or AUTOSAR database, this property is a required part of the XML schema for a frame, so the default (initial) value is obtained from the file.

If you are using a CANdb (.dbc) database, this property is an optional attribute in the file. If NI-XNET finds an attribute named GenMsgSendType, that attribute is the default value of this property. If the GenMsgSendType attribute begins with cyclic, this property's default value is `CYCLIC_DATA`; otherwise, it is `EVENT_DATA`. If the CANdb file does not use the GenMsgSendType attribute, this property uses a default value of `EVENT_DATA`, which you can change in your application.

If you are using an .ncd database or an in-memory database, this property uses a default value of `EVENT_DATA`. Within your application, change this property to the desired timing type.

Type *FrmCanTiming*

can_tx_time

Get or set the time between consecutive frames from the transmitting ECU.

The units are in seconds.

Although the fractional part of the float can provide resolution of picoseconds, the NI-XNET CAN transmit supports an accuracy of 500 microseconds. Therefore, when used within an NI-XNET output session, this property is rounded to the nearest 500 microsecond increment (0.0005).

For a *Frame.can_timing_type* of `CYCLIC_DATA` or `CYCLIC_REMOTE`, this property specifies the time between consecutive data/remote frames. A time of 0.0 is invalid.

For a *Frame.can_timing_type* of `EVENT_DATA` or `EVENT_REMOTE`, this property specifies the minimum time between consecutive data/remote frames when the event occurs quickly. This is also known as the debounce time or minimum interval. The time is measured from the end of previous frame (acknowledgment) to the start of the next frame. A time of 0.0 specifies no minimum (back to back frames allowed).

If you are using a FIBEX or AUTOSAR database, this property is a required part of the XML schema for a frame, so the default (initial) value is obtained from the file.

If you are using a CANdb (.dbc) database, this property is an optional attribute in the file. If NI-XNET finds an attribute named GenMsgCycleTime, that attribute is interpreted as a number of milliseconds and used as the default value of this property. If the CANdb file does not use the GenMsgCycleTime attribute, this property uses a default value of 0.1 (100 ms), which you can change in your application.

If you are using a .ncd database or an in-memory database, this property uses a default value of 0.1 (100 ms). Within your application, change this property to the desired time.

Type float

check_config_status()

Check this frame's configuration status.

By default, incorrectly configured frames in the database are not returned from *Cluster.frames* because they cannot be used in the bus communication. You can change this behavior by setting *Database.show_invalid_from_open* to *True*. When a frame configuration status becomes invalid after the database is opened, the frame still is returned from *Cluster.frames* even if *Database.show_invalid_from_open* is *False*.

Raises *XnetError* – The frame is incorrectly configured.

cluster

Get the parent cluster in which the frame has been created.

You cannot change the parent cluster after the frame object has been created.

Type *Cluster*

comment

Get or set a comment describing the frame object.

A comment is a string containing up to 65535 characters.

Type str

dbc_attributes

Access the frame's DBC attributes.

Type *DbcAttributeCollection*

default_payload

Get or set the frame default payload, specified as a list of ints.

Each int in the list represents a byte (U8). The number of bytes in the list must match the *Frame.payload_len* property.

This property's initial value is an list of all `0`, except the frame is located in a CAN cluster with J1939 application protocol, which uses `0xFF` by default. For the database formats NI-XNET supports, this property is not provided in the database file.

When you use this frame within an NI-XNET session, this property's use varies depending on the session mode. The following sections describe this property's behavior for each session mode.

Frame Output Single-Point and Frame Output Queued Modes: Use this property when a frame transmits prior to a call to write. This can occur when you set the *SessionBase.auto_start* property to `False` and start a session prior to writing. When *SessionBase.auto_start* is `True` (default), the first frame write also starts frame transmit, so this property is not used.

The following frame configurations potentially can transmit prior to a call to write:

- *Frame.can_timing_type* is `CYCLIC_DATA`.
- *Frame.can_timing_type* is `CYCLIC_REMOTE`. (for example, a remote frame received prior to a call to writing).
- *Frame.can_timing_type* is `EVENT_REMOTE`. (for example, a remote frame received prior to a call to writing).
- *Frame.can_timing_type* is `CYCLIC_EVENT`.

- LIN frame in a schedule entry where *`LinSchedEntry.type`* is `UNCONDITIONAL`.

The following frame configurations cannot transmit prior to writing, so this property is not used:

- *`Frame.can_timing_type`* is `EVENT_DATA`..
- LIN frame in a schedule entry where *`LinSchedEntry.type`* is `SPORADIC` or `EVENT_TRIGGERED`.

Frame Output Stream Mode: This property is not used. Transmit is limited to frames provided to write.

Signal Output Single-Point, Signal Output Waveform, and Signal Output XY Modes: Use this property when a frame transmits prior to a call to write. Refer to Frame Output Single-Point and Frame Output Queued Modes for a list of applicable frame configurations.

This property is used as the initial payload, then each XNET Signal Default Value is mapped into that payload, and the result is used for the frame transmit.

Frame Input Stream and Frame Input Queued Modes: This property is not used. These modes do not return data prior to receiving frames.

Frame Input Single-Point Mode: This property is used for frames read returns prior to receiving the first frame.

Signal Input Single-Point, Signal Input Waveform, and Signal Input XY Modes: This property is not used. Each *`Signal.default`* is used when reading from a session prior to receiving the first frame.

Type list of int

`find`(*object_class*, *object_name*)

Finds an object in the database.

This function finds a database object relative to this parent object. This object may be a grandparent or great-grandparent.

If this object is a direct parent (for example, *`Frame`* for *`Signal`*), the `object_name` to search for can be short, and the search proceeds quickly.

If this object is not a direct parent (for example, *`Database`* for *`Signal`*), the `object_name` to search for must be qualified such that it is unique within the scope of this object.

For example, if the class of this object is *`Cluster`*, and `object_class` is *`Signal`*, you can specify `object_name` of `mySignal`, assuming that signal name is unique to the cluster. If not, you must include the *`Frame`* name as a prefix, such as `myFrameA.mySignal`.

NI-XNET supports the following subclasses of `DatabaseObject` as arguments for `object_class`:

- *`nixnet.database.Cluster`*
- *`nixnet.database.Frame`*
- *`nixnet.database.Pdu`*
- *`nixnet.database.Signal`*
- *`nixnet.database.SubFrame`*
- *`nixnet.database.Ecu`*
- *`nixnet.database.LinSched`*
- *`nixnet.database.LinSchedEntry`*

Parameters

- **object_class** (`DatabaseObject`) – The class of the object to find.
- **object_name** (*str*) – The name of the object to find.

Returns An instance of the found object.

Raises

- `ValueError` – Unsupported value provided for argument `object_class`.
- *XnetError* – The object is not found.

id

Get or set the frame identifier.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value at runtime using this property.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

CAN: For CAN frames, this is the Arbitration ID.

When *Frame.can_ext_id* is set to `False`, this is the standard CAN identifier with a size of 11 bits, which results in allowed range of 0-2047. However, the CAN standard disallows identifiers in which the first 7 bits are all recessive, so the working range of identifiers is 0-2031.

When *Frame.can_ext_id* is set to `True`, this is the extended CAN identifier with a size of 29 bits, which results in allowed range of 0-536870911.

LIN: For LIN frames, this is the frame's ID (unprotected). The valid range for a LIN frame ID is 0-63 (inclusive)

Type int

lin_checksum

Returns whether the LIN frame transmitted checksum is classic or enhanced.

The enhanced checksum considers the protected identifier when it is generated.

The checksum is determined from the *Ecu.lin_protocol_ver* properties of the transmitting and receiving the frame. The lower version of both ECUs is significant. If the LIN version of both ECUs is 2.0 or higher, the checksum type is enhanced; otherwise, the checksum type is classic.

Diagnostic frames (with decimal identifier 60 or 61) always use classic checksum, even on LIN 2.x.

Type *FrmLinChecksum*

mux_data_mux_sig

Returns a data multiplexer signal object in the frame.

Use the *Frame.mux_is_muxed* property to determine whether the frame contains a multiplexer signal.

You can create a data multiplexer signal by creating a signal and then setting the *Signal.mux_is_data_mux* property to `True`.

A frame can contain only one data multiplexer signal.

Raises `XnetError` – The data multiplexer signal is not defined in the frame

Type `Signal`

`mux_is_muxed`

Returns whether this frame is data multiplexed.

This property returns `True` if the frame contains a multiplexer signal. Frames containing a multiplexer contain subframes that allow using bits of the frame payload for different information (signals) depending on the multiplexer value.

Type bool

`mux_static_signals`

Collection of static `Signal` objects in this frame.

Static signals are contained in every frame transmitted, as opposed to dynamic signals, which are transmitted depending on the multiplexer value.

If the frame is not multiplexed, this property returns the same objects as `Frame.sigs`.

Type `DbCollection`

`mux_subframes`

Collection of `SubFrame` objects in this frame.

A subframe defines a group of signals transmitted using the same multiplexer value. Only one subframe at a time is transmitted in the frame.

A subframe is defined by creating a subframe object as a child of a frame.

Type `DbCollection`

`name`

String identifying a frame object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A frame name must be unique for all frames in a cluster.

This short name does not include qualifiers to ensure that it is unique, such as the database and cluster name. It is for display purposes.

Type str

`payload_len`

Get or set the number of bytes of data in the payload.

For CAN and LIN, this is 0-8.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value at runtime using this property.

This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

Type int

pdu_properties

Get or set a list that maps existing PDUs to a frame.

A mapped PDU is transmitted inside the frame payload when the frame is transmitted. You can map one or more PDUs to a frame and one PDU to multiple frames.

Mapping PDUs to a frame requires setting pdu_properties with a list of PduProperties tuples. Each tuple contains the following properties:

- `PduProperties.pdu`: Defines the sequence of values for the other two properties.
- `PduProperties.start_bit`: Defines the start bit of the PDU inside the frame.
- `PduProperties.update_bit`: Defines the update bit for the PDU inside the frame. If the update bit is not used, set the value to `-1`.

Databases imported from FIBEX prior to version 3.0, from DBC, NCD, or LDF files have a strong one-to-one relationship between frames and PDUs. Every frame has exactly one PDU mapped, and every PDU is mapped to exactly one frame.

To unmap PDUs from a frame, set this property to an empty list. A frame without mapped PDUs contains no signals.

For CAN and LIN, NI-XNET supports only a one-to-one relationship between frames and PDUs. For those interfaces, advanced PDU configuration returns raises an exception when calling `Frame.check_config_status` and when creating a session. If you do not use advanced PDU configuration, you can avoid using PDUs in the database API and create signals and subframes directly on a frame.

Type list of `PduProperties`

sigs

Get a list of all `Signal` objects in the frame.

This property returns a list to all `Signal` objects in the frame, including static and dynamic signals and the multiplexer signal.

Type list of `Signal`

nixnet.database.lin_sched

class `nixnet.database._lin_sched.`**LinSched**(***kwargs*)

Bases: `nixnet.database._database_object.DatabaseObject`

Database LIN schedule

check_config_status()

Check this LIN schedule's configuration status.

By default, incorrectly configured schedules in the database are not returned from `Cluster.lin_schedules` because they cannot be used in the bus communication. You can change this behavior by setting `Database.show_invalid_from_open` to *True*. When a schedule configuration status becomes invalid after the database is opened, the schedule still is returned from `Cluster.lin_schedules` even if `Database.show_invalid_from_open` is *False*.

Raises `XnetError` – The LIN schedule is incorrectly configured.

clst

Get the parent cluster in which the you created the schedule.

You cannot change the parent cluster after creating the schedule object.

Type `Cluster`

comment

Get or set a comment describing the schedule object.

A comment is a string containing up to 65535 characters.

Type str

entries

Collection of `LinSchedEntry` for this LIN schedule.

The position of each entry in this collection specifies the position in the schedule. The database file and/or the order that you create entries at runtime determine the position.

Type `DbCollection`

find(*object_class*, *object_name*)

Finds an object in the database.

This function finds a database object relative to this parent object. This object may be a grandparent or great-grandparent.

If this object is a direct parent (for example, `Frame` for `Signal`), the `object_name` to search for can be short, and the search proceeds quickly.

If this object is not a direct parent (for example, `Database` for `Signal`), the `object_name` to search for must be qualified such that it is unique within the scope of this object.

For example, if the class of this object is `Cluster`, and `object_class` is `Signal`, you can specify `object_name` of `mySignal`, assuming that signal name is unique to the cluster. If not, you must include the `Frame` name as a prefix, such as `myFrameA.mySignal`.

NI-XNET supports the following subclasses of `DatabaseObject` as arguments for `object_class`:

- `nixnet.database.Cluster`
- `nixnet.database.Frame`
- `nixnet.database.Pdu`
- `nixnet.database.Signal`
- `nixnet.database.SubFrame`
- `nixnet.database.Ecu`
- `nixnet.database.LinSched`
- `nixnet.database.LinSchedEntry`

Parameters

- **object_class** (`DatabaseObject`) – The class of the object to find.
- **object_name** (`str`) – The name of the object to find.

Returns An instance of the found object.

Raises

- `ValueError` – Unsupported value provided for argument `object_class`.

- *XnetError* – The object is not found.

name
Get or set the name of the LIN schedule object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A schedule name must be unique for all schedules in a cluster.

Type str

priority
Get or set the priority of a run-once LIN schedule.

This priority applies when multiple run-once schedules are pending for execution.

The valid range for this property is 1-254. Lower values correspond to higher priority.

This property applies only when the *LinSched.run_mode* property is `ONCE`. Run-once schedule requests are queued for execution based on this property. When all run-once schedules have completed, the master returns to the previously running continuous schedule (or null).

Run-continuous schedule requests are not queued. Only the most recent run-continuous schedule is used, and it executes only if no run-once schedule is pending. Therefore, a run-continuous schedule has an effective priority of `255`, but this property is not used.

Null schedule requests take effect immediately and supercede any running run-once or run-continuous schedule. The queue of pending run-once schedule requests is flushed (emptied without running them). Therefore, a null schedule has an effective priority of `0`, but this property is not used.

This property is not read from the database, but is handled like a database property. After opening the database, the default value is returned, and you can change the property. But similar to database properties, you cannot change it after a session is created.

Type int

run_mode
Get or set how the master runs this schedule.

This property is not read from the database, but is handled like a database property. After opening the database, the default value is returned, and you can change the property. But similar to database properties, you cannot change it after a session is created.

Usually, the default value for the run mode is `CONTINUOUS`. If the schedule is configured to be a collision resolving table for an event-triggered entry, the default is `ONCE`.

Type *LinSchedRunMode*

nixnet.database.lin_sched_entry

class `nixnet.database._lin_sched_entry.`**LinSchedEntry**(***kwargs*)
Bases: `nixnet.database._database_object.DatabaseObject`

Database LIN schedule entry

collision_res_sched
Get or set a LIN schedule that resolves a collision for this event-triggered entry.

This property applies only when *LinSchedEntry.type* is EVENT_TRIGGERED. When a collision occurs for the event-triggered entry in this schedule, the master must switch to the collision resolving schedule to transfer the unconditional frames successfully.

Raises *XnetError* – The property requires that *LinSchedEntry.type* be set to EVENT_TRIGGERED.

Type *LinSched*

delay

Get or set the time from the start of this entry (slot) to the start of the next entry.

The property uses a float value in seconds, with the fractional part used for milliseconds or microseconds.

Type float

event_id

Get or set the event-triggered entry identifier.

This identifier is unprotected (NI-XNET handles the protection).

This property applies only when *LinSchedEntry.type* is EVENT_TRIGGERED. This identifier is for the event triggered entry itself, and the first payload byte is for the protected identifier of the contained unconditional frame.

Type int

frames

Get or set a list of frames for this LIN schedule entry.

If *LinSchedEntry.type* is UNCONDITIONAL, this list contains one frame, which is the single unconditional frame for this entry.

If *LinSchedEntry.type* is SPORADIC, this list contains one or more unconditional frames for this entry. When multiple frames are pending for this entry, the order in the list determines the priority to transmit.

If *LinSchedEntry.type* is EVENT_TRIGGERED, this list contains one or more unconditional frames for this entry. When multiple frames for this entry are pending to be sent by distinct slaves, this property uses the *LinSchedEntry.collision_res_sched* to process the frames.

Type list of *Frame*

name

Get or set the name of the LIN schedule entry object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A schedule entry name must be unique for all entries in the same schedule.

Type str

name_unique_to_cluster

Returns a LIN schedule entry name unique to the cluster that contains the object.

If the single name is not unique within the cluster, the name is <schedule-name>.<schedule-entry-name>.

You can pass the name to the *find* function to retrieve the reference to the object, while the single name is not guaranteed success in *find* because it may be not unique in the cluster.

Type str

nc_ff_data_bytes

Get or set a list of 8 ints containing raw data for LIN node configuration.

Node configuration defines a set of services used to configure slave nodes in the cluster. Every service has a specific set of parameters coded in this int list. In the LDF, file those parameters are stored, for example, in the node (ECU) or the frame object. NI-XNET LDF reader composes those parameters to the byte values like they are sent on the bus. The LIN specification document describes the node configuration services and the mapping of the parameters to the free format bytes.

The node configuration service is executed only if *`LinSchedEntry.type`* is set to `NODE_CONFIG_SERVICE`.

> **Warning:** This property is not saved to the FIBEX file. If you write this property, save the database, and reopen it, the node configuration services are not contained in the database. Writing this property is useful only in the NI-XNET session immediately following.

Type list of int

sched

Returns the LIN schedule that uses this entry.

This LIN schedule is considered this entry's parent. You define the parent schedule when you create the entry object. You cannot change it afterwards.

Type *`LinSched`*

type

Get or set the LIN schedule entry type.

All frames that contain a payload are `UNCONDITIONAL`. The LIN schedule entry type determines the mechanism for transferring frames in this entry (slot).

Type *`LinSchedEntryType`*

nixnet.database.pdu

class `nixnet.database._pdu.`**`Pdu`**(***kwargs*)

Bases: `nixnet.database._database_object.DatabaseObject`

Database PDU

check_config_status()

Check this PDU's configuration status.

By default, incorrectly configured PDUs in the database are not returned from *`Cluster.pdus`* because they cannot be used in the bus communication. You can change this behavior by setting *`Database.show_invalid_from_open`* to *True*. When a PDU configuration status becomes invalid after the database is opened, the PDU still is returned from *`Cluster.pdus`* even if *`Database.show_invalid_from_open`* is *False*.

Raises *`XnetError`* – The PDU is incorrectly configured.

cluster

Get the parent cluster in which the PDU has been created.

You cannot change the parent cluster after creating the PDU object.

Type *`Cluster`*

comment
Get or set a comment describing the PDU object.

A comment is a string containing up to 65535 characters.

Type str

find(*object_class*, *object_name*)
Finds an object in the database.

This function finds a database object relative to this parent object. This object may be a grandparent or great-grandparent.

If this object is a direct parent (for example, *Frame* for *Signal*), the `object_name` to search for can be short, and the search proceeds quickly.

If this object is not a direct parent (for example, *Database* for *Signal*), the `object_name` to search for must be qualified such that it is unique within the scope of this object.

For example, if the class of this object is *Cluster*, and `object_class` is *Signal*, you can specify `object_name` of `mySignal`, assuming that signal name is unique to the cluster. If not, you must include the *Frame* name as a prefix, such as `myFrameA.mySignal`.

NI-XNET supports the following subclasses of `DatabaseObject` as arguments for `object_class`:

- *nixnet.database.Cluster*
- *nixnet.database.Frame*
- *nixnet.database.Pdu*
- *nixnet.database.Signal*
- *nixnet.database.SubFrame*
- *nixnet.database.Ecu*
- *nixnet.database.LinSched*
- *nixnet.database.LinSchedEntry*

Parameters

- **object_class** (`DatabaseObject`) – The class of the object to find.
- **object_name** (*str*) – The name of the object to find.

Returns An instance of the found object.

Raises

- `ValueError` – Unsupported value provided for argument `object_class`.
- *XnetError* – The object is not found.

frms
Returns a list of all frames to which the PDU is mapped.

A PDU is transmitted within the frames to which it is mapped.

To map a PDU to a frame, use the *Frame.pdu_properties* property. You can map one PDU to multiple frames.

Type list of *Frame*

mux_data_mux_sig

Data multiplexer signal in the PDU.

This property returns the reference to the data multiplexer signal. If data multiplexer is not defined in the PDU, the property raises an *XnetError* exception. Use the *Pdu.mux_is_muxed* property to determine whether the PDU contains a multiplexer signal.

You can create a data multiplexer signal by creating a signal and then setting the *Signal.mux_is_data_mux* property to `True`.

A PDU can contain only one data multiplexer signal.

Raises *XnetError* – The data multiplexer is not defined in the PDU.

Type *Signal*

mux_is_muxed

Returns `True` if the PDU contains a multiplexer signal.

PDUs containing a multiplexer contain subframes that allow using bits of the payload for different information (signals), depending on the value of the *SubFrame.mux_value* property.

Type bool

mux_static_sigs

Returns a list of static signals in the PDU.

Returns an list of signal objects in the PDU that do not depend on value of the *SubFrame.mux_value* property. Static signals are contained in every PDU transmitted, as opposed to dynamic signals, which are transmitted depending on the value of the *SubFrame.mux_value* property.

You can create static signals by specifying the PDU as the parent object. You can create dynamic signals by specifying a subframe as the parent.

If the PDU is not multiplexed, this property returns the same list as the *Pdu.signals* property.

Type list of *Signal*

mux_subframes

Collection of *SubFrame* objects in this PDU.

A subframe defines a group of signals transmitted using the same value of the *SubFrame.mux_value*. Only one subframe is transmitted in the PDU at a time.

Type *DbCollection*

name

Get or set the name of the PDU object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A PDU name must be unique for all PDUs in a cluster.

Type str

payload_len

Get or set the size of the PDU data in bytes.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this PDU, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value at runtime using this property.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

  Type int

signals

Collection of all `Signal` objects in this PDU.

The collection includes all signals in the PDU, including static and dynamic signals and the multiplexer signal.

Type `DbCollection`

nixnet.database.signal

class `nixnet.database._signal.`**`Signal`**(***kwargs*)

Bases: `nixnet.database._database_object.DatabaseObject`

Database signal

byte_ordr

Signal byte order in the frame payload.

This property defines how signal bytes are ordered in the frame payload when the frame is loaded in memory.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the nxdbSetProperty function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

  Type `SigByteOrdr`

check_config_status()

Check this signal's configuration status.

By default, incorrectly configured signals in the database are not returned from `Frame.sigs` because they cannot be used in the bus communication. You can change this behavior by setting `Database.show_invalid_from_open` to *True*. When a signal configuration status becomes invalid after the database is opened, the signal still is returned from `Frame.sigs` even if `Database.show_invalid_from_open` is *False*.

Examples of invalid signal configuration:

- The signal is specified using bits outside the frame payload.

- The signal overlaps another signal in the frame. For example, two multiplexed signals with the same multiplexer value are using the same bit in the frame payload.
- The signal with integer data type (signed or unsigned) is specified with more than 52 bits. This is not allowed due to internal limitation of the double data type that NI-XNET uses for signal values.
- The frame containing the signal is invalid (for example, a CAN frame is defined with more than 8 payload bytes).

Raises `XnetError` – The signal is incorrectly configured.

`comment`

Get or set a comment describing the signal object.

A comment is a string containing up to 65535 characters.

Type str

`data_type`

Get or set the signal data type.

This property determines how the bits of a signal in a frame must be interpreted to build a value.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value at runtime using this property.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

Type `SigDataType`

`dbc_attributes`

Access the signal's DBC attributes.

Type `DbcAttributeCollection`

`dbc_signal_value_table`

Access the signal's DBC value table.

Type `DbcSignalValueTable`

`default`

Get or set the signal default value, specified as scaled floating-point units.

The initial value of this property comes from the database. If the database does not provide a value, this property uses a default value of 0.0.

For all three signal output sessions, this property is used when a frame transmits prior to writing to a session. The `Frame.default_payload` property is used as the initial payload, then the default value of each signal is mapped into that payload using this property, and the result is used for the frame transmit.

For all three signal input sessions, this property is returned for each signal when reading a session prior to receiving the first frame.

For more information about when this property is used, refer to the discussion of read and write for each session mode.

Type float

frame

Returns the signal parent frame object.

The parent frame is defined when the signal object is created. You cannot change it afterwards.

Type *Frame*

max

Get or set the scaled signal value maximum.

Session read and write methods do not limit the signal value to a maximum value. Use this database property to set the maximum value.

Type float

min

The scaled signal value minimum.

Session read and write methods do not limit the signal value to a minimum value. Use this database property to set the minimum value.

Type float

mux_is_data_mux

Get or set whether this signal is a multiplexer signal.

A frame containing a multiplexer value is called a multiplexed frame.

A multiplexer defines an area within the frame to contain different information (dynamic signals) depending on the multiplexer signal value. Dynamic signals with a different multiplexer value (defined in a different subframe) can share bits in the frame payload. The multiplexer signal value determines which dynamic signals are transmitted in the given frame.

To define dynamic signals in the frame transmitted with a given multiplexer value, you first must create a subframe in this frame and set the multiplexer value in the subframe. Then you must create dynamic signals using `SubFrame.dyn_signals` to create child signals of this subframe.

Multiplexer signals may not overlap other static or dynamic signals in the frame.

Dynamic signals may overlap other dynamic signals when they have a different multiplexer value.

A frame may contain only one multiplexer signal.

The multiplexer signal is not scaled. Scaling factor and offset do not apply.

In NI-CAN, the multiplexer signal was called mode channel.

Type bool

mux_is_dynamic

returns whether this signal is a dynamic signal.

Use this property to determine if a signal is static or dynamic. Dynamic signals are transmitted in the frame when the multiplexer signal in the frame has a given value specified in the subframe. Use the `Signal.mux_value` property to determine with which multiplexer value the dynamic signal is transmitted.

This property is read only. To create a dynamic signal, create the signal object as a child of a subframe instead of a frame. The dynamic signal cannot be changed to a static signal afterwards.

In NI-CAN, dynamic signals were called mode-dependent signals.

Type bool

`mux_subfrm`

Returns the subframe parent.

This property is valid only for dynamic signals that have a subframe parent. For static signals or the multiplexer signal, this property raises an *`XnetError`* exception.

Raises *`XnetError`* – The signal does not have a subframe parent.

Type *`SubFrame`*

`mux_value`

Returns the multiplexer value of a dynamic signal.

The multiplexer value applies to dynamic signals only (when *`Signal.mux_is_dynamic`* is `True`). This property defines which multiplexer value is transmitted in the multiplexer signal when this dynamic signal is transmitted in the frame.

The multiplexer value is determined in the subframe. All dynamic signals that are children of the same subframe object use the same multiplexer value.

Dynamic signals with the same multiplexer value may not overlap each other, the multiplexer signal, or static signals.

Type int

`name`

Get or set a string identifying a signal object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A signal name must be unique for all signals in a frame.

This short name does not include qualifiers to ensure that it is unique, such as the database, cluster, and frame name. It is for display purposes.

Type str

`name_unique_to_cluster`

Returns a signal name unique to the cluster that contains the signal.

If the single name is not unique within the cluster, the name is <frame-name>.<signal-name>.

You can pass the name to the *find* function to retrieve the reference to the object, while the single name is not guaranteed success in *find* because it may be not unique in the cluster.

Type str

`num_bits`

The number of bits the signal uses in the frame payload.

IEEE Float numbers are limited to 32 bit or 64 bit.

Integer (signed and unsigned) numbers are limited to 1-52 bits. NI-XNET converts all integers to doubles (64-bit IEEE Float). Integer numbers with more than 52 bits (the size of the mantissa in a 64-bit IEEE Float) cannot be converted exactly to double, and vice versa; therefore, NI-XNET does not support this.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value at runtime using this property.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

  Type int

`pdu`
Returns to the signal's parent PDU.

The parent PDU is defined when the signal object is created. You cannot change it afterwards.

Type `Pdu`

`scale_fac`
Get or set factor *a* for linear scaling *ax+b*.

Linear scaling is applied to all signals with the IEEE Float data type, unsigned and signed. For identical scaling 1.0x+0.0, NI-XNET optimized scaling routines do not perform the multiplication and addition

Type float

`scale_off`
Get or set offset *b* for linear scaling *ax+b*.

Linear scaling is applied to all signals with the IEEE Float data type, unsigned and signed. For identical scaling 1.0x+0.0, NI-XNET optimized scaling routines do not perform the multiplication and addition

Type float

`start_bit`
Get or set the least significant signal bit position in the frame payload.

This property determines the signal starting point in the frame. For the integer data type (signed and unsigned), it means the binary signal representation least significant bit position. For IEEE Float signals, it means the mantissa least significant bit.

The NI-XNET Database Editor shows a graphical overview of the frame. It enumerates the frame bytes on the left and the byte bits on top. The bit number in the frame is calculated as byte number x 8 + bit number. The maximum bit number in a CAN or LIN frame is 63 (7 x 8 + 7); the maximum bit number in a FlexRay frame is 2031 (253 x 8 + 7).

Frame Overview in the NI-XNET Database Editor with a Signal Starting in Bit 12

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value at runtime using this property.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

Type int

unit
Get or set the signal value unit.

NI-XNET does not use the unit internally for calculations. You can use the string to display the signal value along with the unit.

Type str

nixnet.database.subframe

class `nixnet.database._subframe.`**SubFrame**(***kwargs*)
Bases: `nixnet.database._database_object.DatabaseObject`

Database subframe

check_config_status()
Check this subframe's configuration status.

By default, incorrectly configured subframes in the database are not returned from *Frame.mux_subframes* because they cannot be used in the bus communication. You can change this behavior by setting *Database.show_invalid_from_open* to *True*. When a subframe configuration status becomes invalid after the database is opened, the subframe still is returned from *Frame.mux_subframes* even if *Database.show_invalid_from_open* is *False*.

Raises *XnetError* – The subframe is incorrectly configured.

dyn_signals
Returns a collection of dynamic *Signal* objects in the subframe.

Those signals are transmitted when the multiplexer signal in the frame has the multiplexer value defined in the subframe.

Type *DbCollection*

find(*object_class*, *object_name*)
Finds an object in the database.

This function finds a database object relative to this parent object. This object may be a grandparent or great-grandparent.

If this object is a direct parent (for example, *Frame* for *Signal*), the `object_name` to search for can be short, and the search proceeds quickly.

If this object is not a direct parent (for example, *Database* for *Signal*), the `object_name` to search for must be qualified such that it is unique within the scope of this object.

For example, if the class of this object is *Cluster*, and `object_class` is *Signal*, you can specify `object_name` of `mySignal`, assuming that signal name is unique to the cluster. If not, you must include the *Frame* name as a prefix, such as `myFrameA.mySignal`.

NI-XNET supports the following subclasses of `DatabaseObject` as arguments for `object_class`:

- *nixnet.database.Cluster*
- *nixnet.database.Frame*
- *nixnet.database.Pdu*
- *nixnet.database.Signal*
- *nixnet.database.SubFrame*

- *nixnet.database.Ecu*
- *nixnet.database.LinSched*
- *nixnet.database.LinSchedEntry*

Parameters

- **object_class** (DatabaseObject) – The class of the object to find.
- **object_name** (*str*) – The name of the object to find.

Returns An instance of the found object.

Raises

- ValueError – Unsupported value provided for argument object_class.
- *XnetError* – The object is not found.

frm

Returns the reference to the parent frame.

The parent frame is defined when the subframe is created, and you cannot change it afterwards.

Type *Frame*

mux_value

Get or set the multiplexer value for this subframe.

This property specifies the multiplexer signal value used when the dynamic signals in this subframe are transmitted in the frame. Only one subframe is transmitted at a time in the frame.

There also is a multiplexer value for a signal object as a read-only property. It reflects the value set on the parent subframe object.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this subframe, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value at runtime using this property.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

Type int

name

Get or set the name of the subframe object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A subframe name must be unique for all subframes in a frame.

This short name does not include qualifiers to ensure that it is unique, such as the database, cluster, and frame name. It is for display purposes.

Type str

name_unique_to_cluster
Returns a subframe name unique to the cluster that contains the subframe.

If the single name is not unique within the cluster, the name is <frame-name>.<subframe-name>.

You can pass the name to the *find* function to retrieve the reference to the object, while the single name is not guaranteed success in *find* because it may be not unique in the cluster.

Type str

pdu
Returns the subframe's parent PDU.

This property returns the reference to the subframe's parent PDU. The parent PDU is defined when the subframe object is created. You cannot change it afterwards.

Type `Pdu`

nixnet.database.collection

class `nixnet.database._collection.`**DbCollection**(*handle*, *db_type*, *prop_id*, *factory*)
Bases: `collections.abc.Mapping`

Collection of database objects.

add(*name*)
Add a new database object to the collection.

Parameters **name** (`str`) – Name of the new database object.

Returns An instance of the new database object.

Return type `DatabaseObject`

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items()
Return all database object names and objects in the collection.

Yields An iterator to tuple pairs of database object names and objects in the collection

keys()
Return database object names in the collection.

Yields An iterator to database object names in the collection.

values()
Return database objects in the collection.

Yields An iterator to database objects in the collection.

nixnet.database.dbc_attributes

class `nixnet.database._dbc_attributes.`**DbcAttributeCollection**(*handle*)
Bases: `collections.abc.Mapping`

Collection for accessing DBC attributes.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items()
Return all attribute names and values in the collection.

Yields An iterator to tuple pairs of attribute names and values in the collection.

keys()
Return all attribute names in the collection.

Yields An iterator to all attribute names in the collection.

values()
Return all attribute values in the collection.

Yields An iterator to all attribute values in the collection.

nixnet.database.dbc_signal_value_table

class `nixnet.database._dbc_signal_value_table.`**DbcSignalValueTable**(*handle*)
Bases: `collections.abc.Mapping`

Collection for accessing a DBC signal value table.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items()
Return all value descriptions and values in the collection.

Yields An iterator to tuple pairs of value descriptions and values in the collection.

keys()
Return all value descriptions in the collection.

Yields An iterator to all value descriptions in the collection.

values()
Return all values in the collection.

Yields An iterator to all values in the collection.

4.2.5 nixnet.constants

class `nixnet._enums.`**AppProtocol**
Bases: `enum.Enum`

Application Protocol.

Values:

NONE: The default application protocol.

J1939: Indicates J1939 clusters. The value enables the following features:

- Sending/receiving long frames as the SAE J1939 specification specifies, using the J1939 transport protocol.
- Using a special notation for J1939 identifiers.
- Using J1939 address claiming.

class `nixnet._enums.`**BlinkMode**
Bases: `enum.Enum`

Interface blink mode.

Values:

DISABLE: Disable blinking for identification. This option turns off both LEDs for the port.

ENABLE: Enable blinking for identification. Both LEDs of the interface's physical port turn on and off. The hardware blinks the LEDs automatically until you disable.

class `nixnet._enums.`**`CanCommState`**

Bases: `enum.Enum`

CAN Comm State.

Values:

ERROR_ACTIVE: This state reflects normal communication, with few errors detected. The CAN interface remains in this state as long as receive error counter and transmit error counter are both below 128.

ERROR_PASSIVE: If either the receive error counter or transmit error counter increment above 127, the CAN interface transitions into this state. Although communication proceeds, the CAN device generally is assumed to have problems with receiving frames.

When a CAN interface is in error passive state, acknowledgement errors do not increment the transmit error counter. Therefore, if the CAN interface transmits a frame with no other device (ECU) connected, it eventually enters error passive state due to retransmissions, but does not enter bus off state.

BUS_OFF: If the transmit error counter increments above 255, the CAN interface transitions into this state. Communication immediately stops under the assumption that the CAN interface must be isolated from other devices.

When a CAN interface transitions to the bus off state, communication stops for the interface. All NI-XNET sessions for the interface no longer receive or transmit frame values. To restart the CAN interface and all its sessions, call *`nixnet._session.base.SessionBase.start`*.

INIT: This is the CAN interface initial state on power-up. The interface is essentially off, in that it is not attempting to communicate with other nodes (ECUs).

When the start trigger occurs for the CAN interface, it transitions from the Init state to the Error Active state. When the interface stops due to a call to *`nixnet._session.base.SessionBase.stop`*., the CAN interface transitions from either Error Active or Error Passive to the Init state. When the interface stops due to the Bus Off state, it remains in that state until you restart.

class `nixnet._enums.`**`CanFdIsoMode`**

Bases: `enum.Enum`

CAN FD ISO MODE.

Values:

ISO: ISO CAN FD standard (ISO standard 11898-1:2015)

In ISO CAN FD mode, for every transmitted frame, you can specify in the database or frame header whether a frame must be sent in CAN 2.0, CAN FD, or CAN FD+BRS mode. In the frame type field of the frame header, received frames indicate whether they have been sent with CAN 2.0, CAN FD, or CAN FD+BRS. You cannot use the Interface:CAN:Transmit I/O Mode property in ISO CAN FD mode, as the frame defines the transmit mode.

NON_ISO: non-ISO CAN FD standard (Bosch CAN FD 1.0 specification)

In Non-ISO CAN FD mode, CAN data frames are received at CAN data typed frames, which is either CAN 2.0, CAN FD, or CAN FD+BRS, but you cannot distinguish the standard in which the frame has been transmitted.

ISO_LEGACY: You also can set the mode to Legacy ISO mode. In this mode, the behavior is the same as in Non-ISO CAN FD mode (Interface:CAN:Transmit I/O Mode is working, and received frames have the CAN data type). But the interface is working in ISO CAN FD mode, so you can communicate with other ISO CAN FD devices. Use this mode only for compatibility with existing applications.

class `nixnet._enums.`**`CanIoMode`**

Bases: `enum.Enum`

CAN I/O Mode.

Values:

CAN: This is the default CAN 2.0 A/B standard I/O mode as defined in ISO 11898-1:2003. A fixed baud rate is used for transfer, and the payload length is limited to 8 bytes.

CAN_FD: This is the CAN FD mode as specified in the CAN with *Flexible Data-Rate specification*, version 1.0. Payload lengths up to 64 are allowed, but they are transmitted at a single fixed baud rate (defined by *`Cluster.can_fd_baud_rate`* or *`Interface.can_fd_baud_rate`*).

CAN_FD_BRS: This is the CAN FD as specified in the *CAN with Flexible Data-Rate* specification, version 1.0, with the optional Baud Rate Switching enabled. The same payload lengths as CAN FD mode are allowed; additionally, the data portion of the CAN frame is transferred at a different (higher) baud rate (defined by *`Cluster.can_fd_baud_rate`* or *`Interface.can_fd_baud_rate`*).

class `nixnet._enums.`**`CanLastErr`**

Bases: `enum.Enum`

CAN Last Error

Values:

NONE: The last receive or transmit was successful.

STUFF: More than 5 equal bits have occurred in sequence, which the CAN specification does not allow.

FORM: A fixed format part of the received frame used the wrong format.

ACK: Another node (ECU) did not acknowledge the frame transmit.

If you call the appropriate `write` function and do not have a cable connected, or the cable is connected to a node that is not communicating, you see this error repeatedly. The CAN communication state eventually transitions to Error Passive, and the frame transmit retries indefinitely.

BIT1: During a frame transmit (with the exception of the arbitration ID field), the interface wanted to send a recessive bit (logical 1), but the monitored bus value was dominant (logical 0).

BIT0: During a frame transmit (with the exception of the arbitration ID field), the interface wanted to send a dominant bit (logical 0), but the monitored bus value was recessive (logical 1).

CRC: The CRC contained within a received frame does not match the CRC calculated for the incoming bits.

class `nixnet._enums.`**`CanPendTxOrder`**

Bases: `enum.Enum`

Can Pending Transmit Order.

Values:

AS_SUBMITTED: Frames are transmitted in the order that they were submitted into the queue. There is no reordering of any frames, and a higher priority frame may be delayed due to the transmission or retransmission of a previously submitted frame. However, this mode has the highest performance.

BY_IDENTIFIER: Frames with the highest priority identifier (lower CAN ID value) transmit first. The frames are stored in a priority queue sorted by ID. If a frame currently being transmitted requires retransmission (for example, it lost arbitration or failed with a bus error), and a higher priority frame is queued in the meantime, the lower priority frame is not immediately retried, but the higher priority frame is transmitted instead. In this mode, you can emulate multiple ECUs and still see a behavior similar to a real bus in that the highest priority message is transmitted on the bus. This mode may be slower in performance (possible delays between transmissions as the queue is re-evaluated), and lower priority messages may be delayed indefinitely due to frequent high-priority messages.

class `nixnet._enums.`**`CanTcvrCap`**

Bases: `enum.Enum`

CAN bus phusical transceivers support.

Values:

HS: High-Speed / Flexible Data-Rate (HS/FD).

LS: Low-Speed / Fault-Tolerant (LS//FT)

XS: XS (HS//FD, LS/FT, SW, or External)

XSHSLS: XS (HS//FD, LS/FT)

class `nixnet._enums.`**`CanTcvrState`**

Bases: `enum.Enum`

CAN Transceiver State.

Values:

NORMAL: This state sets the transceiver to normal communication mode. If the transceiver is in the Sleep mode, this performs a local wakeup of the transceiver and CAN controller chip.

SLEEP: This state sets the transceiver and CAN controller chip to Sleep (or standby) mode. You can set the interface to Sleep mode only while the interface is communicating. If the interface has not been started, setting the transceiver to Sleep mode returns an error.

Before going to sleep, all pending transmissions are transmitted onto the CAN bus. Once all pending frames have been transmitted, the interface and transceiver go into Sleep (or standby) mode. Once the interface enters Sleep mode, further communication is not possible until a wakeup occurs. The transceiver and CAN controller wake from Sleep mode when either a local wakeup or remote wakeup occurs.

A local wakeup occurs when the application sets the transceiver state to either Normal or Single Wire Wakeup.

A remote wakeup occurs when a remote node transmits a CAN frame (referred to as the wakeup frame). The wakeup frame wakes up the NI-XNET interface transceiver and CAN controller chip. The CAN controller chip does not receive or acknowledge the wakeup frame. After detecting the wakeup frame and idle bus, the CAN interface enters Normal mode.

When the local or remote wakeup occurs, frame transmissions resume from the point at which the original Sleep mode was set.

SW_WAKEUP: For a remote wakeup to occur for Single Wire transceivers, the node that transmits the wakeup frame first must place the network into the Single Wire Wakeup Transmission mode by asserting a higher voltage.

This state sets a Single Wire transceiver into the Single Wire Wakeup Transmission mode, which forces the Single Wire transceiver to drive a higher voltage level on the network to wake up all sleeping nodes. Other than this higher voltage, this mode is similar to Normal mode. CAN frames can be received and transmitted normally.

If you are not using a Single Wire transceiver, setting this state returns an error. If your current mode is Single Wire High-Speed, setting this mode returns an error because you are not allowed to wake up the bus in high-speed mode.

The application controls the timing of how long the wakeup voltage is driven. The application typically changes to Single Wire Wakeup mode, transmits a single wakeup frame, and then returns to Normal mode.

SW_HIGH_SPEED: This state sets a Single Wire transceiver into Single Wire High-Speed Communication mode. If you are not using a Single Wire transceiver, setting this state returns an error.

Single Wire High-Speed Communication mode disables the transceiver's internal waveshaping function, allowing the SAE J2411 High-Speed baud rate of 83.333 kbytes/s to be used. The disadvantage versus Single Wire Normal Communication mode, which allows only the SAE J2411 baud rate of 33.333 kbytes/s, is degraded EMC performance. Other than the disabled waveshaping, this mode is similar to Normal mode. CAN frames can be received and transmitted normally.

This mode has no relationship to High-Speed transceivers. It is merely a higher speed mode of the Single Wire transceiver, typically used to download data when the onboard network is attached to an offboard tester ECU.

The Single Wire transceiver does not support use of this mode in conjunction with Sleep mode. For example, a remote wakeup cannot transition from sleep to this Single Wire High-Speed mode. Therefore, setting the mode to Sleep from Single Wire High-Speed mode returns an error.

class `nixnet._enums.`**`CanTcvrType`**

Bases: `enum.Enum`

CAN Transceiver Type

Values:

High-Speed (HS): This configuration enables the High-Speed transceiver. This transceiver supports baud rates of 40 kbaud to 1 Mbaud. When using a High-Speed transceiver, you also can communicate with a CAN FD bus. Refer to NI-XNET Hardware Overview to determine which CAN FD baud rates are supported.

Low-Speed/Fault-Tolerant (LS): This configuration enables the Low-Speed/Fault-Tolerant transceiver. This transceiver supports baud rates of 40-125 kbaud.

Single Wire (SW): This configuration enables the Single Wire transceiver. This transceiver supports baud rates of 33.333 kbaud and 83.333 kbaud.

External (EXT): This configuration allows you to use an external transceiver to connect to your CAN bus. Refer to the XNET Session Interface:CAN:External Transceiver Config property for more information.

Disconnect (DISC): This configuration allows you to disconnect the CAN controller chip from the connector. You can use this value when you physically change the external transceiver.

class `nixnet._enums.`**`CanTerm`**

Bases: `enum.Enum`

CAN Termination.

Different CAN hardware has different termination requirements, and the OFF and ON values have different meanings.

High-Speed CAN

High-Speed CAN networks are typically terminated on the bus itself instead of within a node. However, NI-XNET allows you to configure termination within the node to simplify testing. If your bus already has the correct amount of termination, leave this property in the default state of Off. However, if you require termination, set this property to On.

Values:

OFF: Termination is disabled.

On: Termination (120 Ohms) is enabled.

Low-Speed/Fault-Tolerant CAN

Every node on a Low-Speed CAN network requires termination for each CAN data line (CAN_H and CAN_L). This configuration allows the Low-Speed/Fault-Tolerant CAN port to provide fault detection and recovery. Refer to Termination for more information about low-speed termination. In general, if the existing network has an overall network termination of 125 Ohms or less, turn on termination to enable the 4.99 kOhms option. Otherwise, you should select the default 1.11 kOhms option.

Values:

OFF: Termination is set to 1.11 kOhms.

ON: Termination is set to 4.99 kOhms.

Single-Wire CAN

The ISO standard requires Single-Wire transceivers to have a 9.09 kOhms resistor, and no additional configuration is supported.

class `nixnet._enums.`**`CanTermCap`**
Bases: `enum.Enum`

CAN Termination Capability.

Values: NO YES

class `nixnet._enums.`**`ClstFlexRaySampClkPer`**
Bases: `enum.Enum`

An enumeration.

class `nixnet._enums.`**`Condition`**
Bases: `enum.Enum`

An enumeration.

class `nixnet._enums.`**`CreateSessionMode`**
Bases: `enum.Enum`

Create Session Mode.

The session mode specifies the data type (signals or frames), direction (input or output), and how data is transferred between your application and the network.

Values:

SIGNAL_IN_SINGLE_POINT: Reads the most recent value received for each signal. This mode typically is used for control or simulation applications, such as Hardware In the Loop (HIL).

SIGNAL_IN_WAVEFORM: Using the time when the signal frame is received, resamples the signal data to a waveform with a fixed sample rate. This mode typically is used for synchronizing XNET data with DAQmx analog/digital input channels.

SIGNAL_IN_XY: For each frame received, provides its signals as a value/timestamp pair. This is the recommended mode for reading a sequence of all signal values.

SIGNAL_OUT_SINGLE_POINT: Writes signal values for the next frame transmit. This mode typically is used for control or simulation applications, such as Hardware In the Loop (HIL).

SIGNAL_OUT_WAVEFORM: Using the time when the signal frame is transmitted according to the database, resamples the signal data from a waveform with a fixed sample rate. This mode typically is used for synchronizing XNET data with DAQmx analog/digital output channels.

SIGNAL_OUT_XY: Provides a sequence of signal values for transmit using each frame's timing as the database specifies. This is the recommended mode for writing a sequence of all signal values.

FRAME_IN_STREAM: Reads all frames received from the network using a single stream. This mode typically is used for analyzing and/or logging all frame traffic in the network.

FRAME_IN_QUEUED: Reads data from a dedicated queue per frame. This mode enables your application to read a sequence of data specific to a frame (for example, CAN identifier).

FRAME_IN_SINGLE_POINT: Reads the most recent value received for each frame. This mode typically is used for control or simulation applications that require lower level access to frames (not signals).

FRAME_OUT_STREAM: Transmits an arbitrary sequence of frame values using a single stream. The values are not limited to a single frame in the database, but can transmit any frame.

FRAME_OUT_QUEUED: Provides a sequence of values for a single frame, for transmit using that frame's timing as the database specifies.

FRAME_OUT_SINGLE_POINT: Writes frame values for the next transmit. This mode typically is used for control or simulation applications that require lower level access to frames (not signals).

SIGNAL_CONVERSION_SINGLE_POINT: This mode does not use any hardware. It is used to convert data between the signal representation and frame representation.

class `nixnet._enums.`**`DevForm`**

Bases: `enum.Enum`

Device physical form factor.

Values: C_SERIES PCI PCIE PXI PXIE USB

class `nixnet._enums.`**`DongleId`**

Bases: `enum.Enum`

Dongle ID

Values:

HSCAN: CAN High Speed

XSCAN: CAN Software-Selectable

LIN: LIN

DONGLE_LESS: Dongle-Less Design

class `nixnet._enums.`**`DongleState`**

Bases: `enum.Enum`

Dongle State.

Values:

NO_DONGLE_NO_EXT_POWER: No dongle, no external power.

NO_DONGLE_EXT_POWER: No dongle, has external power.

DONGLE_NO_EXT_POWER: Has dongle, no external power.

READY: Ready.

BUSY: Busy.

COMM_ERROR: Comm Error.

OVERCURRENT: Overcurrent.

class nixnet._enums.**Err**
Bases: enum.Enum

Error codes returned by NI-XNET.

class nixnet._enums.**FlexRayPocState**
Bases: enum.Enum

An enumeration.

class nixnet._enums.**FlexRaySleep**
Bases: enum.Enum

An enumeration.

class nixnet._enums.**FlexRayTerm**
Bases: enum.Enum

An enumeration.

class nixnet._enums.**FrameType**
Bases: enum.Enum

Frame format type.

class nixnet._enums.**FrmCanTiming**
Bases: enum.Enum

CAN Frame Timing

Values:

CYCLIC_DATA: The transmitting ECU transmits the CAN data frame in a cyclic (periodic) manner. The *Frame.can_tx_time* property defines the time between cycles. The transmitting ECU ignores CAN remote frames received for this frame.

EVENT_DATA: The transmitting ECU transmits the CAN data frame in an event-driven manner. The *Frame.can_tx_time* property defines the minimum interval. For NI-XNET, the event occurs when you write data to a session. The transmitting ECU ignores CAN remote frames received for this frame.

CYCLIC_REMOTE: The receiving ECU transmits the CAN remote frame in a cyclic (periodic) manner. The *Frame.can_tx_time* property defines the time between cycles. The transmitting ECU responds to each CAN remote frame by transmitting the associated CAN data frame.

EVENT_REMOTE: The receiving ECU transmits the CAN remote frame in an event-driven manner. The *Frame.can_tx_time* property defines the minimum interval. For NI-XNET, the event occurs when you write a frame to a session. The transmitting ECU responds to each CAN remote frame by transmitting the associated CAN data frame.

CYCLIC_EVENT: This timing type is a combination of the cyclic and event timing. The frame is transmitted when you write to a session, but also periodically sending the last recent values written. The *Frame.can_tx_time* property defines the cycle period. There is no minimum interval time defined in this mode, so be careful not to write too frequently to avoid creating a high busload.

class nixnet._enums.**FrmFlexRayChAssign**
Bases: enum.Enum

An enumeration.

class nixnet._enums.**FrmFlexRayTiming**
Bases: enum.Enum

An enumeration.

class nixnet._enums.**FrmLinChecksum**

Bases: enum.Enum

LIN Frame Transmitted Checksum

Values:

CLASSIC: Classic checksum.

ENHANCED: Enhanced checksum.

class nixnet._enums.**GetDbcAttributeMode**

Bases: enum.Enum

An enumeration.

class nixnet._enums.**LinCommState**

Bases: enum.Enum

LIN Comm State

Values:

IDLE: This is the LIN interface initial state on power-up. The interface is essentially off, in that it is not attempting to communicate with other nodes (ECUs). When the start trigger occurs for the LIN interface, it transitions from the Idle state to the Active state. When the interface stops due to a call to XNET Stop, the LIN interface transitions from either Active or Inactive to the Idle state.

ACTIVE: This state reflects normal communication. The LIN interface remains in this state as long as bus activity is detected (frame headers received or transmitted).

INACTIVE: This state indicates that no bus activity has been detected in the past four seconds.

Regardless of whether the interface acts as a master or slave, it transitions to this state after four seconds of bus inactivity. As soon as bus activity is detected (break or frame header), the interface transitions to the Active state.

The LIN interface does not go to sleep automatically when it transitions to Inactive. To place the interface into sleep mode, set the XNET Session Interface:LIN:Sleep property when you detect the Inactive state.

class nixnet._enums.**LinDiagnosticSchedule**

Bases: enum.Enum

LIN Diagnostic Schedule

Values:

NULL: The master does not execute any diagnostic schedule. No master request or slave response headers are transmitted on the LIN.

MASTER_REQ: The master executes a diagnostic master request schedule (transmits a master request header onto the LIN) if it can. First, a master request schedule must be defined for the LIN cluster in the imported or in-memory database. Otherwise, error 'nixnet._enums.Err.DIAGNOSTIC_SCHEDULE_NOT_DEFINED' is returned when attempting to set this value. Second, the master must have a frame output queued session created for the master request frame, and there must be one or more new master request frames pending in the queue. If no new frames are pending in the output queue, no master request header is transmitted. This allows the timing of master request header transmission to be controlled by the timing of master request frame writes to the output queue.

If there are no normal schedules pending, the master is effectively in diagnostics-only mode, and master request headers are transmitted at a rate determined by the slot delay defined for the master request frame slot in the master request schedule or the *nixnet._session.intf.Interface.lin_diag_s_tmin* property time, whichever is greater, and the state of the master request frame output queue as described above.

If there are normal schedules pending, the master is effectively in diagnostics-interleaved mode, and a master request header transmission is inserted between each complete execution of a run-once or run-continuous schedule, as long as the *nixnet._session.intf.Interface.lin_diag_s_tmin* property time has been met, and there are one or more new master request frames pending in the master request frame output queue.

SLAVE_RESP: The master executes a diagnostic slave response schedule (transmits a slave response header onto the LIN) if it is able to. A slave response schedule must be defined for the LIN cluster in the imported or in-memory database. Otherwise, error 'nixnet._enums.Err.DIAGNOSTIC_SCHEDULE_NOT_DEFINED' is returned when attempting to set this value.

If there are no normal schedules pending, the master is effectively in diagnostics-only mode, and slave response headers are transmitted at the rate of the slot delay defined for the slave response frame slot in the slave response schedule. The addressed slave may or may not respond to each header, depending on its specified P2min and STmin timings.

If there are normal schedules pending, the master is effectively in diagnostics-interleaved mode, and a slave response header transmission is inserted between each complete execution of a run-once or run-continuous schedule. Here again, the addressed slave may or may not respond to each header, depending on its specified P2min and STmin timings.

class `nixnet._enums.`**`LinLastErr`**

Bases: `enum.Enum`

LIN Comm Last Error Code

Values:

NONE: No bus error has occurred since the previous communication state read.

UNKNOWN_ID: Received a frame identifier that is not valid.

FORM: The form of a received frame is incorrect. For example, the database specifies 8 bytes of payload, but you receive only 4 bytes.

FRAMING: The byte framing is incorrect (for example, a missing stop bit).

READBACK: The interface transmitted a byte, but the value read back from the transceiver was different. This often is caused by a cabling problem, such as noise.

TIMEOUT: Receiving the frame took longer than the LIN-specified timeout.

CRC: The received checksum was different than the expected checksum.

class `nixnet._enums.`**`LinProtocolVer`**

Bases: `enum.Enum`

LIN Protocol Version

Values:

VER_1_2: Version 1.2

VER_1_3: Version 1.3

VER_2_0: Version 2.0

VER_2_1: Version 2.1

VER_2_2: Version 2.2

class nixnet._enums.**LinSchedEntryType**
Bases: enum.Enum

LIN Schedule Entry Type.

Values:

UNCONDITIONAL: A single frame transfers in this slot.

SPORADIC: The master transmits in this slot. The master can select from multiple frames to transmit. Only updated frames are transmitted. When more than one frame is updated, the master decides by priority which frame to send. The other updated frame remains pending and can be sent when this schedule entry is processed the following time. The order of unconditional frames in *LinSchedEntry.frames* (the first frame has the highest priority) determines the frame priority.

EVENT_TRIGGERED: Multiple slaves can transmit an unconditional frame in this slot. The slave transmits the frame only if at least one frame signal has been updated. When a collision occurs (multiple slaves try to transmit in the same slot), this is detected and resolved using a different schedule specified in the *LinSchedEntry.collision_res_sched* property. The resolving schedule runs once, starting in the subsequent slot after the collision, and automatically returns to the previous schedule at the subsequent position where the collision occurred.

NODE_CONFIG_SERVICE: The schedule entry contains a node configuration service. The node configuration service is defined as raw data bytes in *LinSchedEntry.nc_ff_data_bytes*.

class nixnet._enums.**LinSchedRunMode**
Bases: enum.Enum

LIN Schedule Run Mode.

Values:

CONTINUOUS: The master runs the schedule continuously. When the last entry executes, the schedule starts again with the first entry.

ONCE: The master runs the schedule once (all entries), then returns to the previously running continuous schedule (or NULL). If requests are submitted for multiple run-once schedules, each run-once executes in succession based on its *LinSched.priority*, then the master returns to the continuous schedule (or NULL).

NULL: All communication stops immediately. A schedule with this run mode is called a *null schedule*.

class nixnet._enums.**LinSleep**
Bases: enum.Enum

LIN interface sleep/awake state

Values:

REMOTE_SLEEP: Set interface to sleep locally and transmit sleep requests to remote node.

REMOTE_WAKE: Set interface to awake locally and transmit wakeup requests to remote nodes.

LOCAL_SLEEP: Set interface to sleep locally and not to interact with the network.

LOCAL_WAKE: Set interface to awake locally and not to interact with the network.

class nixnet._enums.**LinTerm**
Bases: enum.Enum

LIN Termination

class nixnet._enums.**Merge**
Bases: enum.Enum

Cluster Merge Behavior

Values:

COPY_USE_SOURCE: The target object with all dependent child objects is removed from the target cluster and replaced by the source objects.

COPY_USE_TARGET: The source object is ignored (the target cluster object with child objects remains unchanged).

MERGE_USE_SOURCE: This adds child objects from the source object to child objects from the destination object. If target object contains a child object with the same name, the child object from the source frame replaces it. The source object properties (for example, payload length of the frame) replace the target properties.

MERGE_USE_TARGET: This adds child objects from the source object to child objects from the destination object. If the target object contains a child object with the same name, it remains unchanged. The target object properties remain unchanged (for example, payload length).

class `nixnet._enums.`**`ObjectClass`**
Bases: `enum.Enum`

An enumeration.

class `nixnet._enums.`**`OutStrmTimng`**
Bases: `enum.Enum`

Output Stream Timing

Values:

IMMEDIATE: Frames are dequeued from the queue and transmitted immediately to the bus. The hardware transmits all frames in the queue as fast as possible. There are no restrictions on frames that you use in other sessions.

For replay modes, the hardware is placed into a Replay mode. In this mode, the hardware evaluates the frame timestamps and attempts to maintain the original transmission times as the timestamp stored in the frame indicates. The actual transmission time is based on the relative time difference between the first dequeued frame and the time contained in the dequeued frame.

REPLAY_EXCLUSIVE: The hardware transmits only frames that do not appear in the list. You cannot create any other output sessions. Attempting to create an output session returns an error. Input sessions have no restrictions.

This can be used to test an ECU when the output stream list contains the frames the ECU transmits. You can replay all frames in this mode if the output stream list is unset.

REPLAY_INCLUSIVE: The hardware transmits only frames that appear in the list. You can create output sessions that use frames that do not appear in the Interface:Output Stream List property. Attempting to create an output session that uses a frame from the Interface:Output Stream List property results in an error. Input sessions have no restrictions.

This can be used to emulate an ECU when the output stream list contains the frames the ECU transmits.

class `nixnet._enums.`**`Phase`**
Bases: `enum.Enum`

Version Phase.

Values: RELEASE

class `nixnet._enums.`**`Protocol`**
Bases: `enum.Enum`

Protocol.

Values:

UNKNOWN: Unknown protocol,

CAN: CAN protocol.

FLEX_RAY: FlexRay protocol.

LIN: LIN protocol.

class nixnet._enums.**ReadState**

Bases: `enum.Enum`

An enumeration.

class nixnet._enums.**SessionInfoState**

Bases: `enum.Enum`

State of running session.

Values:

STOPPED: All frames in the session are stopped.

STARTED: All frames in the session are started.

MIX: Some frames in the session are started while other frames are stopped. This state may occur when using `start` or `stop` with `StartStopScope.SESSION_ONLY`.

class nixnet._enums.**SigByteOrdr**

Bases: `enum.Enum`

Signal Byte Order

Values:

Little Endian: Higher significant signal bits are placed on higher byte addresses. In NI-CAN, this was called Intel Byte Order.

Little Endian Signal with Start Bit 12

Big Endian: Higher significant signal bits are placed on lower byte addresses. In NI-CAN, this was called Motorola Byte Order.

Big Endian Signal with Start Bit 12

class nixnet._enums.**SigDataType**

Bases: `enum.Enum`

Signal Data Type

Values:

SIGNED: Signed integer with positive and negative values.

UNSIGNED: Unsigned integer with no negative values.

IEEE_FLOAT: Float value with 7 or 15 significant decimal digits (32 bit or 64 bit).

class nixnet._enums.**StartStopScope**

Bases: `enum.Enum`

Start/Stop Scope enum.

Values:

NORMAL: The session is started followed by starting the interface. This is equivalent to calling *nixnet._session.base.SessionBase.start* with the Session Only Scope followed by calling *nixnet._session.base.SessionBase.start* with the Interface Only Scope.

SESSION_ONLY: The session is placed into the Started state (refer to State Models). If the interface is in the Stopped state before this function runs, the interface remains in the Stopped state, and no communication occurs with the bus. To have multiple sessions start at exactly the same time, start each session with the Session Only Scope. When you are ready for all sessions to start communicating on the associated interface, call *nixnet._session.base.SessionBase.start* with the Interface Only scope. Starting a previously started session is considered a no-op. This operation sends the command to start the session, but does not wait for the session to be started. It is ideal for a real-time application where performance is critical.

INTERFACE_ONLY: If the underlying interface is not previously started, the interface is placed into the Started state (refer to State Models). After the interface starts communicating, all previously started sessions can transfer data to and from the bus. Starting a previously started interface is considered a no-op.

SESSION_ONLY_BLOCKING: The session is placed in the Started state (refer to State Models). If the interface is in the Stopped state before this function runs, the interface remains in the Stopped state, and no communication occurs with the bus. To have multiple sessions start at exactly the same time, start each session with the Session Only Scope. When you are ready for all sessions to start communicating on the associated interface, call nxStart with the Interface Only Scope. Starting a previously started session is considered a no-op. This operation waits for the session to start before completing.

class `nixnet._enums.`**`Warn`**

Bases: `enum.Enum`

Warning codes returned by NI-XNET.

class `nixnet._enums.`**`WriteState`**

Bases: `enum.Enum`

An enumeration.

4.2.6 nixnet.types

class `nixnet.types.`**`DriverVersion`**

Bases: `nixnet.types.DriverVersion_`

Driver Version

The arguments align with the following fields: `[major].[minor].[update][phase][build]`.

`major`

Type int

`minor`

Type int

`update`

Type int

`phase`

Type *nixnet._enums.Phase*

build

Type int

class nixnet.types.**CanComm**

Bases: nixnet.types.CanComm_

CAN Communication State.

state

Communication State

Type *nixnet._enums.CanCommState*

tcvr_err

Transceiver Error. Transceiver error indicates whether an error condition exists on the physical transceiver. This is typically referred to as the transceiver chip NERR pin. False indicates normal operation (no error), and true indicates an error.

Type bool

sleep

Sleep. Sleep indicates whether the transceiver and communication controller are in their sleep state. False indicates normal operation (awake), and true indicates sleep.

Type bool

last_err

Last Error. Last error specifies the status of the last attempt to receive or transmit a frame

Type *nixnet._enums.CanLastErr*

tx_err_count

Transmit Error Counter. The transmit error counter begins at 0 when communication starts on the CAN interface. The counter increments when an error is detected for a transmitted frame and decrements when a frame transmits successfully. The counter increases more for an error than it is decreased for success. This ensures that the counter generally increases when a certain ratio of frames (roughly 1/8) encounter errors. When communication state transitions to Bus Off, the transmit error counter no longer is valid.

Type int

rx_err_count

Receive Error Counter. The receive error counter begins at 0 when communication starts on the CAN interface. The counter increments when an error is detected for a received frame and decrements when a frame is received successfully. The counter increases more for an error than it is decreased for success. This ensures that the counter generally increases when a certain ratio of frames (roughly 1/8) encounter errors.

Type int

class nixnet.types.**LinComm**

Bases: nixnet.types.LinComm_

CAN Communication State.

sleep

Sleep. Indicates whether the transceiver and communication controller are in their sleep state. False indicates normal operation (awake), and true indicates sleep.

Type bool

state

Communication State

Type *nixnet._enums.LinCommState*

last_err
Last Error. Last error specifies the status of the last attempt to receive or transmit a frame

Type *nixnet._enums.LinLastErr*

err_received
Returns the value received from the network when last error occurred.

When `last_err` is `READBACK`, this is the value read back.

When `last_err` is `CHECKSUM`, this is the received checksum.

Type int

err_expected
Returns the value that the LIN interface expected to see (instead of last received).

When `last_err` is `READBACK`, this is the value transmitted.

When `last_err` is `CHECKSUM`, this is the calculated checksum.

Type int

err_id
Returns the frame identifier in which the last error occurred.

This is not applicable when `last_err` is `NONE` or `UNKNOWN_ID`.

Type int

tcvr_rdy
Indicates whether the LIN transceiver is powered from the bus.

True indicates the bus power exists, so it is safe to start communication on the LIN interface.

If this value is false, you cannot start communication successfully. Wire power to the LIN transceiver and run your application again.

Type bool

sched_index
Indicates the LIN schedule that the interface currently is running.

This index refers to a LIN schedule that you requested using the *nixnet._session.base.SessionBase.change_lin_schedule* function. It indexes the array of schedules represented in the *nixnet._session.intf.Interface.lin_sched_names*.

This index applies only when the LIN interface is running as a master. If the LIN interface is running as a slave only, this element should be ignored.

Type int

class nixnet.types.**CanIdentifier**(*identifier*, *extended=False*)
Bases: `object`

CAN frame arbitration identifier.

identifier
CAN frame arbitration identifier

Type int

extended
If the identifier is extended

Type bool

classmethod from_raw(*raw*)
Parse a raw frame identifier into a CanIdentifier

Parameters raw (*int*) – A raw frame identifier

Returns parsed value

Return type *CanIdentifier*

```
>>> CanIdentifier.from_raw(0x1)
CanIdentifier(0x1)
>>> CanIdentifier.from_raw(0x20000001)
CanIdentifier(0x1, extended=True)
```

class nixnet.types.**FrameFactory**
Bases: object

ABC for creating *nixnet.types.Frame* objects.

classmethod from_raw(*frame*)
Convert from RawFrame.

class nixnet.types.**Frame**
Bases: *nixnet.types.FrameFactory*

ABC for frame objects.

to_raw()
Convert to RawFrame.

type
Frame format.

Type *nixnet._enums.FrameType*

class nixnet.types.**RawFrame**(*timestamp*, *identifier*, *type*, *flags=0*, *info=0*, *payload=b”*)
Bases: *nixnet.types.Frame*

Raw Frame.

timestamp
Absolute time the XNET interface received the end-of-frame.

Type int

identifier
Frame identifier.

Type int

type
Frame type.

Type *nixnet._enums.FrameType*

flags
Flags that qualify the type.

Type int

info
Info that qualify the type.

Type int

payload
Payload.

Type bytes

classmethod from_raw(*frame*)
Convert from RawFrame.

to_raw()
Convert to RawFrame.

type
Frame format.

Type *nixnet._enums.FrameType*

class nixnet.types.**CanFrame**(*identifier*, *type=<FrameType.CAN_DATA: 0>*, *payload=b''*)
Bases: *nixnet.types.Frame*

CAN Frame.

identifier
CAN frame arbitration identifier.

Type *nixnet.types.CanIdentifier*

echo
If the frame is an echo of a successful transmit rather than being received from the network.

Type bool

type
Frame type.

Type *nixnet._enums.FrameType*

timestamp
Absolute time the XNET interface received the end-of-frame.

Type int

payload
Payload.

Type bytes

classmethod from_raw(*frame*)
Convert from RawFrame.

```
>>> raw = RawFrame(5, 0x20000001, constants.FrameType.CAN_DATA, _cconsts.NX_
↪FRAME_FLAGS_TRANSMIT_ECHO, 0, b'')
>>> CanFrame.from_raw(raw)
CanFrame(CanIdentifier(0x1, extended=True), echo=True, timestamp=0x5)
```

to_raw()
Convert to RawFrame.

```
>>> CanFrame(CanIdentifier(1, True), constants.FrameType.CAN_DATA).to_raw()
RawFrame(timestamp=0x0, identifier=0x20000001, type=FrameType.CAN_DATA)
>>> c = CanFrame(CanIdentifier(1, True), constants.FrameType.CAN_DATA)
>>> c.echo = True
>>> c.to_raw()
RawFrame(timestamp=0x0, identifier=0x20000001, type=FrameType.CAN_DATA,␣
↪flags=0x80)
```

type
Frame format.

Type *nixnet._enums.FrameType*

class nixnet.types.**CanBusErrorFrame**(*timestamp*, *state*, *tcvr_err*, *bus_err*, *tx_err_count*, *rx_err_count*)

Bases: *nixnet.types.Frame*

Error detected on hardware bus of a *nixnet.session.FrameInStreamSession*.

Note: This requires enabling *nixnet._session.intf.Interface.bus_err_to_in_strm*.

See also *nixnet.types.CanComm*.

timestamp
Absolute time when the bus error occurred.

Type int

state
Communication State

Type *nixnet._enums.CanCommState*

tcvr_err
Transceiver Error.

Type bool

bus_err
Last Error.

Type *nixnet._enums.CanLastErr*

tx_err_count
Transmit Error Counter.

Type int

rx_err_count
Receive Error Counter.

Type int

classmethod from_raw(*frame*)
Convert from RawFrame.

```
>>> raw = RawFrame(0x64, 0x0, constants.FrameType.CAN_BUS_ERROR, 0, 0, b
↪'\x00\x01\x02\x03\x04')
>>> CanBusErrorFrame.from_raw(raw)
CanBusErrorFrame(0x64, CanCommState.ERROR_ACTIVE, True, CanLastErr.ACK, 1, 2)
```

to_raw()
Convert to RawFrame.

```
>>> CanBusErrorFrame(100, constants.CanCommState.BUS_OFF, True, constants.
↪CanLastErr.STUFF, 1, 2).to_raw()
RawFrame(timestamp=0x64, identifier=0x0, type=FrameType.CAN_BUS_ERROR,␣
↪len(payload)=5)
```

type
Frame format.

Type *nixnet._enums.FrameType*

class nixnet.types.**LinFrame**(*identifier*, *type=<FrameType.LIN_DATA: 64>*, *payload=b''*)
Bases: object

LIN Frame.

identifier
LIN frame arbitration identifier.

Type int

echo
If the frame is an echo of a successful transmit rather than being received from the network.

Type bool

type
Frame type.

Type *nixnet._enums.FrameType*

timestamp
Absolute time the XNET interface received the end-of-frame.

Type int

eventslot
Whether the frame was received within an event-triggered slot or an unconditional or sporadic slot.

Type bool

eventid
Identifier for an event-triggered slot.

Type int

payload
A byte string representing the payload.

Type bytes

classmethod from_raw(*frame*)
Convert from RawFrame.

```
>>> raw = RawFrame(5, 2, constants.FrameType.LIN_DATA, 0x81, 1, b'{')
>>> LinFrame.from_raw(raw)
LinFrame(identifier=0x2, echo=True, timestamp=0x5, eventslot=True, eventid=1,␣
↪len(payload)=1)
>>> raw = RawFrame(5, 2, constants.FrameType.LIN_DATA, _cconsts.NX_FRAME_
↪FLAGS_TRANSMIT_ECHO, 0, b'{')
>>> LinFrame.from_raw(raw)
LinFrame(identifier=0x2, echo=True, timestamp=0x5, len(payload)=1)
```

to_raw()
Convert to RawFrame.

```
>>> LinFrame(2, constants.FrameType.LIN_DATA).to_raw()
RawFrame(timestamp=0x0, identifier=0x2, type=FrameType.LIN_DATA)
>>> l = LinFrame(2, constants.FrameType.LIN_DATA)
```

(continues on next page)

(continued from previous page)

```
>>> l.echo = True
>>> l.eventslot = True
>>> l.eventid = 1
>>> l.to_raw()
RawFrame(timestamp=0x0, identifier=0x2, type=FrameType.LIN_DATA, flags=0x81,␣
↪info=0x1)
```

class nixnet.types.**LinBusErrorFrame**(*timestamp*, *state*, *bus_err*, *err_id*, *err_received*, *err_expected*)

Bases: *nixnet.types.Frame*

Error detected on hardware bus of a *nixnet.session.FrameInStreamSession*.

Note: This requires enabling *nixnet._session.intf.Interface.bus_err_to_in_strm*.

See also *nixnet.types.LinComm*.

timestamp
Absolute time when the bus error occurred.

Type int

state
Communication State.

Type *nixnet._enums.LinCommState*

bus_err
Last Error.

Type *nixnet._enums.LinLastErr*

err_id
Identifier on bus.

Type int

err_received
Received byte on bus

Type int

err_expected
Expected byte on bus

Type int

classmethod from_raw(*frame*)
Convert from RawFrame.

```
>>> raw = RawFrame(0x64, 0x0, constants.FrameType.LIN_BUS_ERROR, 0, 0, b
↪'\x00\x01\x02\x03\x04')
>>> LinBusErrorFrame.from_raw(raw)
LinBusErrorFrame(0x64, LinCommState.IDLE, LinLastErr.UNKNOWN_ID, 0x2, 3, 4)
```

to_raw()
Convert to RawFrame.

```
>>> LinBusErrorFrame(100, constants.LinCommState.INACTIVE, constants.
↪LinLastErr.UNKNOWN_ID, 2, 3, 4).to_raw()
RawFrame(timestamp=0x64, identifier=0x0, type=FrameType.LIN_BUS_ERROR,␣
↪len(payload)=5)
```

type
Frame format.

Type *nixnet._enums.FrameType*

class nixnet.types.**DelayFrame**(*offset*)
Bases: *nixnet.types.Frame*

Delay hardware when DelayFrame is outputted.

Note: This requires *nixnet._session.intf.Interface.out_strm_timng* to be in replay mode.

offset
Time to delay in milliseconds.

Type int

classmethod from_raw(*frame*)
Convert from RawFrame.

```
>>> raw = RawFrame(5, 0, constants.FrameType.SPECIAL_DELAY, 0, 0, b'')
>>> DelayFrame.from_raw(raw)
DelayFrame(5)
```

to_raw()
Convert to RawFrame.

```
>>> DelayFrame(250).to_raw()
RawFrame(timestamp=0xfa, identifier=0x0, type=FrameType.SPECIAL_DELAY)
```

type
Frame format.

Type *nixnet._enums.FrameType*

class nixnet.types.**LogTriggerFrame**(*timestamp*)
Bases: *nixnet.types.Frame*

Timestamp of when a trigger occurred.

This frame is generated on input sessions when a rising edge is detected on an external connection.

Note: This requires using *nixnet._session.base.SessionBase.connect_terminals* to connect an external connection to the internal `LogTrigger` terminal.

timestamp
Absolute time that the trigger occurred.

Type int

classmethod from_raw(*frame*)
Convert from RawFrame.

```
>>> raw = RawFrame(5, 0, constants.FrameType.SPECIAL_LOG_TRIGGER, 0, 0, b'')
>>> LogTriggerFrame.from_raw(raw)
LogTriggerFrame(0x5)
```

to_raw()
Convert to RawFrame.

```
>>> LogTriggerFrame(250).to_raw()
RawFrame(timestamp=0xfa, identifier=0x0, type=FrameType.SPECIAL_LOG_TRIGGER)
```

type
Frame format.

Type *nixnet._enums.FrameType*

class nixnet.types.**StartTriggerFrame**(*timestamp*)
Bases: *nixnet.types.Frame*

Timestamp of *nixnet.session.FrameInStreamSession* start.

Note: This requires enabling *nixnet._session.intf.Interface.start_trig_to_in_strm*.

timestamp
Absolute time that the trigger occurred.

Type int

classmethod from_raw(*frame*)
Convert from RawFrame.

```
>>> raw = RawFrame(5, 0, constants.FrameType.SPECIAL_START_TRIGGER, 0, 0, b'')
>>> StartTriggerFrame.from_raw(raw)
StartTriggerFrame(0x5)
```

to_raw()
Convert to RawFrame.

```
>>> StartTriggerFrame(250).to_raw()
RawFrame(timestamp=0xfa, identifier=0x0, type=FrameType.SPECIAL_START_TRIGGER)
```

type
Frame format.

Type *nixnet._enums.FrameType*

class nixnet.types.**XnetFrame**
Bases: *nixnet.types.FrameFactory*

Create *Frame* based on *RawFrame* content.

classmethod from_raw(*frame*)
Convert from RawFrame.

class nixnet.types.**PduProperties**
Bases: nixnet.types.PDU_PROPERTIES_

Properties that map a PDU onto a frame.

Mapping PDUs to a frame requires setting three frame properties that are combined into this tuple.

pdu
Defines the sequence of values for the other two properties.

Type *Pdu*

start_bit
Defines the start bit of the PDU inside the frame.

Type int

update_bit
Defines the update bit for the PDU inside the frame. If the update bit is not used, set the value to `-1`.

Type int

4.2.7 nixnet.errors

exception `nixnet.errors.`**`XnetError`**(*message*, *error_code*)
Bases: `nixnet.errors.Error`

Error raised by any NI-XNET method.

error_code
Error code reported by NI-XNET.

Type int

error_type
Error type reported by NI-XNET.

Type *nixnet._enums.Err*

exception `nixnet.errors.`**`XnetWarning`**(*message*, *warning_code*)
Bases: `Warning`

Warning raised by any NI-XNET method.

warning_code
Warning code reported by NI-XNET.

Type int

warning_type
Warning type reported by NI-XNET.

Type *nixnet._enums.Warn*

`nixnet.errors.`**`XnetResourceWarning`**
alias of `builtins.ResourceWarning`

4.3 Examples

4.3.1 Queued I/O Example

This example uses *nixnet.session.FrameInQueuedSession* and *nixnet.session.FrameOutQueuedSession* to demonstrate how queued sessions work.

CAN Queued I/O

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import time

import six

import nixnet
from nixnet import constants
from nixnet import types


def main():
    database_name = 'NIXNET_example'
    cluster_name = 'CAN_Cluster'
    input_frame = 'CANEventFrame1'
    output_frame = 'CANEventFrame1'
    interface1 = 'CAN1'
    interface2 = 'CAN2'

    with nixnet.FrameInQueuedSession(
            interface1,
            database_name,
            cluster_name,
            input_frame) as input_session:
        with nixnet.FrameOutQueuedSession(
                interface2,
                database_name,
                cluster_name,
                output_frame) as output_session:
            terminated_cable = six.moves.input('Are you using a terminated cable (Y␣
↪or N)? ')
            if terminated_cable.lower() == "y":
                input_session.intf.can_term = constants.CanTerm.ON
                output_session.intf.can_term = constants.CanTerm.OFF
            elif terminated_cable.lower() == "n":
                input_session.intf.can_term = constants.CanTerm.ON
                output_session.intf.can_term = constants.CanTerm.ON
            else:
                print("Unrecognised input ({}), assuming 'n'".format(terminated_
↪cable))
                input_session.intf.can_term = constants.CanTerm.ON
                output_session.intf.can_term = constants.CanTerm.ON

            # Start the input session manually to make sure that the first
            # frame value sent before the initial read will be received.
            input_session.start()

            user_value = six.moves.input('Enter payload [int, int]: ')
            try:
                payload_list = [int(x.strip()) for x in user_value.split(",")]
            except ValueError:
                payload_list = [2, 4, 8, 16]
                print('Unrecognized input ({}). Setting data buffer to {}'.
↪format(user_value, payload_list))
```

(continues on next page)

(continued from previous page)

```
            id = types.CanIdentifier(0)
            payload = bytearray(payload_list)
            frame = types.CanFrame(id, constants.FrameType.CAN_DATA, payload)

            i = 0
            while True:
                for index, byte in enumerate(payload):
                    payload[index] = byte + i

                frame.payload = payload
                output_session.frames.write([frame])
                print('Sent frame with ID: {} payload: {}'.format(frame.identifier,
                                                                  list(frame.
↪payload)))

                # Wait 1 s and then read the received values.
                # They should be the same as the ones sent.
                time.sleep(1)

                count = 1
                frames = input_session.frames.read(count)
                for frame in frames:
                    print('Received frame with ID: {} payload: {}'.format(frame.
↪identifier,
                                                                          list(six.
↪iterbytes(frame.payload))))

                i += 1
                if max(payload) + i > 0xFF:
                    i = 0

                inp = six.moves.input('Hit enter to continue (q to quit): ')
                if inp.lower() == 'q':
                    break

            print('Data acquisition stopped.')


if __name__ == '__main__':
    main()
```

Refer to *Adapting CAN examples to LIN* for how to adapt from CAN to LIN.

4.3.2 Stream I/O Example

This example uses *nixnet.session.FrameInStreamSession* and *nixnet.session.FrameOutStreamSession* to demonstrate how streamed sessions work.

CAN Stream I/O

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
```

(continues on next page)

(continued from previous page)

```
import time

import six

import nixnet
from nixnet import constants
from nixnet import types


def main():
    interface1 = 'CAN1'
    interface2 = 'CAN2'

    with nixnet.FrameInStreamSession(interface1) as input_session:
        with nixnet.FrameOutStreamSession(interface2) as output_session:
            terminated_cable = six.moves.input('Are you using a terminated cable (Y␣
↪or N)? ')
            if terminated_cable.lower() == "y":
                input_session.intf.can_term = constants.CanTerm.ON
                output_session.intf.can_term = constants.CanTerm.OFF
            elif terminated_cable.lower() == "n":
                input_session.intf.can_term = constants.CanTerm.ON
                output_session.intf.can_term = constants.CanTerm.ON
            else:
                print("Unrecognised input ({}), assuming 'n'".format(terminated_
↪cable))
                input_session.intf.can_term = constants.CanTerm.ON
                output_session.intf.can_term = constants.CanTerm.ON

            input_session.intf.baud_rate = 125000
            output_session.intf.baud_rate = 125000

            # Start the input session manually to make sure that the first
            # frame value sent before the initial read will be received.
            input_session.start()

            user_value = six.moves.input('Enter payload [int, int]: ')
            try:
                payload_list = [int(x.strip()) for x in user_value.split(",")]
            except ValueError:
                payload_list = [2, 4, 8, 16]
                print('Unrecognized input ({}). Setting data buffer to {}'.
↪format(user_value, payload_list))

            id = types.CanIdentifier(0)
            payload = bytearray(payload_list)
            frame = types.CanFrame(id, constants.FrameType.CAN_DATA, payload)

            print('The same values should be received. Press q to quit')
            i = 0
            while True:
                for index, byte in enumerate(payload):
                    payload[index] = byte + i

                frame.payload = payload
                output_session.frames.write([frame])
```

(continues on next page)

(continued from previous page)

```
                print('Sent frame with ID: {} payload: {}'.format(frame.identifier,
                                                                  list(frame.
↪payload)))

                # Wait 1 s and then read the received values.
                # They should be the same as the ones sent.
                time.sleep(1)

                count = 1
                frames = input_session.frames.read(count)
                for frame in frames:
                    print('Received frame with ID: {} payload: {}'.format(frame.
↪identifier,
                                                                          list(six.
↪iterbytes(frame.payload))))

                i += 1
                if max(payload) + i > 0xFF:
                    i = 0

                inp = six.moves.input('Hit enter to continue (q to quit): ')
                if inp.lower() == 'q':
                    break

            print('Data acquisition stopped.')


if __name__ == '__main__':
    main()
```

LIN Stream I/O

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import time

import six

import nixnet
from nixnet import constants
from nixnet import types


def main():
    interface1 = 'LIN1'
    interface2 = 'LIN2'
    database = 'NIXNET_exampleLDF'

    with nixnet.FrameInStreamSession(interface1, database) as input_session:
        with nixnet.FrameOutStreamSession(interface2, database) as output_session:
            terminated_cable = six.moves.input('Are you using a terminated cable (Y␣
↪or N)? ')
            if terminated_cable.lower() == "y":
```

(continues on next page)

(continued from previous page)

```
                input_session.intf.lin_term = constants.LinTerm.ON
                output_session.intf.lin_term = constants.LinTerm.OFF
            elif terminated_cable.lower() == "n":
                input_session.intf.lin_term = constants.LinTerm.ON
                output_session.intf.lin_term = constants.LinTerm.ON
            else:
                print("Unrecognised input ({}), assuming 'n'".format(terminated_
↪cable))
                input_session.intf.lin_term = constants.LinTerm.ON
                output_session.intf.lin_term = constants.LinTerm.ON

            output_session.intf.lin_master = True

            # Start the input session manually to make sure that the first
            # frame value sent before the initial read will be received.
            input_session.start()

            # Set the schedule. This will also automatically enable master mode.
            output_session.change_lin_schedule(0)

            user_value = six.moves.input('Enter payload [int, int]: ')
            try:
                payload_list = [int(x.strip()) for x in user_value.split(",")]
            except ValueError:
                payload_list = [2, 4, 8, 16]
                print('Unrecognized input ({}). Setting data buffer to {}'.
↪format(user_value, payload_list))

            id = 0
            payload = bytearray(payload_list)
            frame = types.LinFrame(id, constants.FrameType.LIN_DATA, payload)

            print('The same values should be received. Press q to quit')
            i = 0
            while True:
                for index, byte in enumerate(payload):
                    payload[index] = byte + i

                frame.payload = payload
                output_session.frames.write([frame])
                print('Sent frame with ID: {} payload: {}'.format(frame.identifier,
                                                                  list(frame.
↪payload)))

                # Wait 1 s and then read the received values.
                # They should be the same as the ones sent.
                time.sleep(1)

                count = 1
                frames = input_session.frames.read(count)
                for frame in frames:
                    print('Received frame with ID: {} payload: {}'.format(frame.
↪identifier,
                                                                          list(six.
↪iterbytes(frame.payload))))

                i += 1
```

(continues on next page)

(continued from previous page)

```
                if max(payload) + i > 0xFF:
                    i = 0

                inp = six.moves.input('Hit enter to continue (q to quit): ')
                if inp.lower() == 'q':
                    break

            print('Data acquisition stopped.')


if __name__ == '__main__':
    main()
```

Refer to *Adapting CAN examples to LIN* for how to adapt from CAN to LIN.

4.3.3 Single-Point I/O Example

This example uses *nixnet.session.SignalInSinglePointSession* and *nixnet.session.SignalOutSinglePointSession* to demonstrate how single-point sessions work.

To adapt this to Frames, just change the sessions to *nixnet.session.FrameInSinglePointSession* and *nixnet.session.FrameOutSinglePointSession* with frames instead of signals. Then adjust read/write to take a frame object per frame configured in the session rather than signals.

This works for both CAN and LIN frames. LIN frames also require change_lin_sched to write a request for the LIN interface to change the running schedule. See *Queued I/O Example* to see how to read and write frames.

CAN Single-Point I/O

```
def main():
    database_name = 'NIXNET_example'
    cluster_name = 'CAN_Cluster'
    input_signals = ['CANEventSignal1', 'CANEventSignal2']
    output_signals = ['CANEventSignal1', 'CANEventSignal2']
    interface1 = 'CAN1'
    interface2 = 'CAN2'

    with nixnet.SignalInSinglePointSession(
            interface1,
            database_name,
            cluster_name,
            input_signals) as input_session:
        with nixnet.SignalOutSinglePointSession(
                interface2,
                database_name,
                cluster_name,
                output_signals) as output_session:
            terminated_cable = six.moves.input('Are you using a terminated cable (Y␣
↪or N)? ')
            if terminated_cable.lower() == "y":
                input_session.intf.can_term = constants.CanTerm.ON
                output_session.intf.can_term = constants.CanTerm.OFF
            elif terminated_cable.lower() == "n":
                input_session.intf.can_term = constants.CanTerm.ON
```

(continues on next page)

(continued from previous page)

```
                output_session.intf.can_term = constants.CanTerm.ON
            else:
                print("Unrecognised input ({}), assuming 'n'".format(terminated_
↪cable))
                input_session.intf.can_term = constants.CanTerm.ON
                output_session.intf.can_term = constants.CanTerm.ON

            # Start the input session manually to make sure that the first
            # signal value sent before the initial read will be received.
            input_session.start()

            user_value = six.moves.input('Enter {} signal values [float, float]: '.
↪format(len(input_signals)))
            try:
                value_buffer = [float(x.strip()) for x in user_value.split(",")]
            except ValueError:
                value_buffer = [24.5343, 77.0129]
                print('Unrecognized input ({}). Setting data buffer to {}'.
↪format(user_value, value_buffer))

            if len(value_buffer) != len(input_signals):
                value_buffer = [24.5343, 77.0129]
                print('Invalid number of signal values entered. Setting data buffer␣
↪to {}'.format(value_buffer))

            print('The same values should be received. Press q to quit')
            i = 0
            while True:
                for index, value in enumerate(value_buffer):
                    value_buffer[index] = value + i
                output_session.signals.write(value_buffer)
                print('Sent signal values: {}'.format(value_buffer))

                # Wait 1 s and then read the received values.
                # They should be the same as the ones sent.
                time.sleep(1)

                signals = input_session.signals.read()
                for timestamp, value in signals:
                    date = convert_timestamp(timestamp)
                    print('Received signal with timestamp {} and value {}'.
↪format(date, value))

                i += 1
                if max(value_buffer) + i > sys.float_info.max:
                    i = 0

                inp = six.moves.input('Hit enter to continue (q to quit): ')
                if inp.lower() == 'q':
                    break

            print('Data acquisition stopped.')
```

4.3.4 Signal/Frame Conversion Example

This example uses *nixnet.convert.SignalConversionSinglePointSession* to take signal values from the user, converts them to frames, and converts them back.

To adapt this example to LIN frames, reference signals in a database that use LIN:

convert_frames_to_signals: Accepts any frame type.

convert_signals_to_frames: Chooses the frame object to create based on the `frame_type` field in the raw data. This can be overridden by passing a custom *nixnet.types.FrameFactory* in the `frame_type` parameter.

```
def main():
    database_name = 'NIXNET_example'
    cluster_name = 'CAN_Cluster'
    signal_names = ['CANEventSignal1', 'CANEventSignal2']

    with convert.SignalConversionSinglePointSession(
            database_name,
            cluster_name,
            signal_names) as session:

        user_value = six.moves.input('Enter {} signal values [float, float]: '.
↪format(len(signal_names)))
        try:
            expected_signals = [float(x.strip()) for x in user_value.split(",")]
        except ValueError:
            expected_signals = [24.5343, 77.0129]
            print('Unrecognized input ({}). Setting data buffer to {}'.format(user_
↪value, expected_signals))

        if len(expected_signals) != len(signal_names):
            expected_signals = [24.5343, 77.0129]
            print('Invalid number of signal values entered. Setting data buffer to {}
↪'.format(expected_signals))

        frames = session.convert_signals_to_frames(expected_signals)
        print('Frames:')
        for frame in frames:
            print('    {}'.format(frame))
            print('        payload={}'.format(list(six.iterbytes(frame.payload))))

        converted_signals = session.convert_frames_to_signals(frames)
        print('Signals: {}'.format([v for (_, v) in converted_signals]))
```

4.3.5 Adapting CAN examples to LIN

To adapt the examples from CAN to LIN, reference signals in a database that use LIN:

write: Accepts any frame type.

read: Chooses the frame object to create based on the `frame_type` field in the raw data. This can be overridden by passing a custom *nixnet.types.FrameFactory* in the `frame_type` parameter.

change_lin_sched: Writes a request for the LIN interface to change the running schedule.

This displays the diff of `can_frame_stream_io.py` and `lin_frame_stream_io.py` to demonstrate the changes required to update CAN example code for LIN.

```
--- /home/docs/checkouts/readthedocs.org/user_builds/nixnet/checkouts/latest/nixnet_
↪examples/can_frame_stream_io.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/nixnet/checkouts/latest/nixnet_
↪examples/lin_frame_stream_io.py
@@ -12,29 +12,32 @@


 def main():
-    interface1 = 'CAN1'
-    interface2 = 'CAN2'
+    interface1 = 'LIN1'
+    interface2 = 'LIN2'
+    database = 'NIXNET_exampleLDF'

-    with nixnet.FrameInStreamSession(interface1) as input_session:
-        with nixnet.FrameOutStreamSession(interface2) as output_session:
+    with nixnet.FrameInStreamSession(interface1, database) as input_session:
+        with nixnet.FrameOutStreamSession(interface2, database) as output_session:
             terminated_cable = six.moves.input('Are you using a terminated cable (Y␣
↪or N)? ')
             if terminated_cable.lower() == "y":
-                input_session.intf.can_term = constants.CanTerm.ON
-                output_session.intf.can_term = constants.CanTerm.OFF
+                input_session.intf.lin_term = constants.LinTerm.ON
+                output_session.intf.lin_term = constants.LinTerm.OFF
             elif terminated_cable.lower() == "n":
-                input_session.intf.can_term = constants.CanTerm.ON
-                output_session.intf.can_term = constants.CanTerm.ON
+                input_session.intf.lin_term = constants.LinTerm.ON
+                output_session.intf.lin_term = constants.LinTerm.ON
             else:
                 print("Unrecognised input ({}), assuming 'n'".format(terminated_
↪cable))
-                input_session.intf.can_term = constants.CanTerm.ON
-                output_session.intf.can_term = constants.CanTerm.ON
+                input_session.intf.lin_term = constants.LinTerm.ON
+                output_session.intf.lin_term = constants.LinTerm.ON

-            input_session.intf.baud_rate = 125000
-            output_session.intf.baud_rate = 125000
+            output_session.intf.lin_master = True

             # Start the input session manually to make sure that the first
             # frame value sent before the initial read will be received.
             input_session.start()
+
+            # Set the schedule. This will also automatically enable master mode.
+            output_session.change_lin_schedule(0)

             user_value = six.moves.input('Enter payload [int, int]: ')
             try:
@@ -43,9 +46,9 @@
                 payload_list = [2, 4, 8, 16]
                 print('Unrecognized input ({}). Setting data buffer to {}'.
↪format(user_value, payload_list))

-            id = types.CanIdentifier(0)
```

(continues on next page)

(continued from previous page)

```
+             id = 0
              payload = bytearray(payload_list)
-             frame = types.CanFrame(id, constants.FrameType.CAN_DATA, payload)
+             frame = types.LinFrame(id, constants.FrameType.LIN_DATA, payload)

              print('The same values should be received. Press q to quit')
              i = 0
```

4.3.6 Programmatic Database Usage

This example uses *nixnet.system._databases.AliasCollection* to demonstrate how databases can be programmatically added and used in a system.

```
def main():
    with system.System() as my_system:
        database_alias = 'custom_database'
        database_filepath = os.path.join(os.path.dirname(__file__), 'databases\custom_
↪database.dbc')
        default_baud_rate = 500000
        my_system.databases.add_alias(database_alias, database_filepath, default_baud_
↪rate)

    database_name = 'custom_database'
    cluster_name = 'CAN_Cluster'
    output_frame = 'CANEventFrame1'
    interface = 'CAN1'

    with nixnet.FrameOutQueuedSession(
            interface,
            database_name,
            cluster_name,
            output_frame) as output_session:
        terminated_cable = six.moves.input('Are you using a terminated cable (Y or N)?
↪ ')
        if terminated_cable.lower() == "y":
            output_session.intf.can_term = constants.CanTerm.OFF
        elif terminated_cable.lower() == "n":
            output_session.intf.can_term = constants.CanTerm.ON
        else:
            print("Unrecognised input ({}), assuming 'n'".format(terminated_cable))
            output_session.intf.can_term = constants.CanTerm.ON

        user_value = six.moves.input('Enter payload [int, int]: ')
        try:
            payload_list = [int(x.strip()) for x in user_value.split(",")]
        except ValueError:
            payload_list = [2, 4, 8, 16]
            print('Unrecognized input ({}). Setting data buffer to {}'.format(user_
↪value, payload_list))

        id = types.CanIdentifier(0)
        payload = bytearray(payload_list)
        frame = types.CanFrame(id, constants.FrameType.CAN_DATA, payload)

        print("Writing CAN frames using {} alias:".format(database_name))
```

(continues on next page)

(continued from previous page)

```
        i = 0
        while i < 3:
            for index, byte in enumerate(payload):
                payload[index] = byte + i

            frame.payload = payload
            output_session.frames.write([frame])
            print('Sent frame with ID: {} payload: {}'.format(frame.identifier,␣
↪list(frame.payload)))
            i += 1

    with system.System() as my_system:
        del my_system.databases[database_name]
```

4.3.7 Dynamic Database Creation

This example programmatically modifies the in-memory database to contain a cluster, a frame, and two signals. The database is then used in a `nixnet.session.SignalOutSinglePointSession` and `nixnet.session.SignalInSinglePointSession` to write and then read a pair of signals.

CAN Dynamic Database Creation

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from random import randint
import six
import time

import nixnet
from nixnet import constants
from nixnet import database


def main():
    database_name = ':memory:'
    cluster_name = 'CAN_Cluster'
    frame_name = 'CAN_Event_Frame'
    signal_1_name = 'CAN_Event_Signal_1'
    signal_2_name = 'CAN_Event_Signal_2'
    signal_list = [signal_1_name, signal_2_name]
    output_interface = 'CAN1'
    input_interface = 'CAN2'

    # Open the default in-memory database.
    # Database.close will be called by Database.__exit__ when exiting the 'with'␣
↪block.
    with database.Database(database_name) as db:

        # Add a CAN cluster, a frame, and two signals to the database.
        cluster = db.clusters.add(cluster_name)
```

(continues on next page)

(continued from previous page)

```
        cluster.protocol = constants.Protocol.CAN
        cluster.baud_rate = 125000
        frame = cluster.frames.add(frame_name)
        frame.id = 1
        frame.payload_len = 2
        signal_1 = frame.mux_static_signals.add(signal_1_name)
        signal_1.byte_ordr = constants.SigByteOrdr.BIG_ENDIAN
        signal_1.data_type = constants.SigDataType.UNSIGNED
        signal_1.start_bit = 0
        signal_1.num_bits = 8
        signal_2 = frame.mux_static_signals.add(signal_2_name)
        signal_2.byte_ordr = constants.SigByteOrdr.BIG_ENDIAN
        signal_2.data_type = constants.SigDataType.UNSIGNED
        signal_2.start_bit = 8
        signal_2.num_bits = 8

        # Using the database we just created, write and then read a pair of signals.
        with nixnet.SignalOutSinglePointSession(
                output_interface,
                database_name,
                cluster_name,
                signal_list) as output_session:
            with nixnet.SignalInSinglePointSession(
                    input_interface,
                    database_name,
                    cluster_name,
                    signal_list) as input_session:
                terminated_cable = six.moves.input('Are you using a terminated cable␣
↪(Y or N)? ')
                if terminated_cable.lower() == "y":
                    input_session.intf.can_term = constants.CanTerm.ON
                    output_session.intf.can_term = constants.CanTerm.OFF
                elif terminated_cable.lower() == "n":
                    input_session.intf.can_term = constants.CanTerm.ON
                    output_session.intf.can_term = constants.CanTerm.ON
                else:
                    print("Unrecognised input ({}), assuming 'n'".format(terminated_
↪cable))
                    input_session.intf.can_term = constants.CanTerm.ON
                    output_session.intf.can_term = constants.CanTerm.ON

                # Start the input session manually to make sure that the first
                # signal values sent before the initial read will be received.
                input_session.start()

                # Generate a pair of random values and send out the signals.
                output_values = [randint(0, 255), randint(0, 255)]
                output_session.signals.write(output_values)
                print('Sent signal values: {}'.format(output_values))

                # Wait 1 s and then read the received values.
                # They should be the same as the ones sent.
                time.sleep(1)

                input_signals = input_session.signals.read()
                input_values = [int(value) for timestamp, value in input_signals]
                print('Received signal values: {}'.format(input_values))
```

(continues on next page)

(continued from previous page)

```python


if __name__ == '__main__':
    main()
```

LIN Dynamic Database Creation

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from random import randint
import six
import time

import nixnet
from nixnet import constants
from nixnet import database


def main():
    database_name = ':memory:'
    cluster_name = 'LIN_Cluster'
    ecu_1_name = 'LIN_ECU_1'
    ecu_2_name = 'LIN_ECU_2'
    schedule_name = 'LIN_Schedule_1'
    schedule_entry_name = 'LIN_Schedule_Entry'
    frame_name = 'LIN_Frame'
    signal_1_name = 'LIN_Signal_1'
    signal_2_name = 'LIN_Signal_2'
    signal_list = [signal_1_name, signal_2_name]
    output_interface = 'LIN1'
    input_interface = 'LIN2'

    # Open the default in-memory database.
    # Database.close will be called by Database.__exit__ when exiting the 'with'␣
↪block.
    with database.Database(database_name) as db:

        # Add a LIN cluster, a frame, and two signals to the database.
        cluster = db.clusters.add(cluster_name)
        cluster.protocol = constants.Protocol.LIN
        cluster.baud_rate = 19200
        frame = cluster.frames.add(frame_name)
        frame.id = 1
        frame.payload_len = 2
        signal_1 = frame.mux_static_signals.add(signal_1_name)
        signal_1.byte_ordr = constants.SigByteOrdr.BIG_ENDIAN
        signal_1.data_type = constants.SigDataType.UNSIGNED
        signal_1.start_bit = 0
        signal_1.num_bits = 8
        signal_2 = frame.mux_static_signals.add(signal_2_name)
        signal_2.byte_ordr = constants.SigByteOrdr.BIG_ENDIAN
        signal_2.data_type = constants.SigDataType.UNSIGNED
        signal_2.start_bit = 8
```

(continues on next page)

(continued from previous page)

```
        signal_2.num_bits = 8

        # Add a LIN ECU and LIN Schedule to the cluster.
        ecu_1 = cluster.ecus.add(ecu_1_name)
        ecu_1.lin_protocol_ver = constants.LinProtocolVer.VER_2_2
        ecu_1.lin_master = True
        ecu_2 = cluster.ecus.add(ecu_2_name)
        ecu_2.lin_protocol_ver = constants.LinProtocolVer.VER_2_2
        ecu_2.lin_master = False
        cluster.lin_tick = 0.01
        schedule = cluster.lin_schedules.add(schedule_name)
        schedule.priority = 0
        schedule.run_mode = constants.LinSchedRunMode.CONTINUOUS
        schedule_entry = schedule.entries.add(schedule_entry_name)
        schedule_entry.delay = 1000.0
        schedule_entry.type = constants.LinSchedEntryType.UNCONDITIONAL
        schedule_entry.frames = [frame]

        # Using the database we just created, write and then read a pair of signals.
        with nixnet.SignalOutSinglePointSession(
                output_interface,
                database_name,
                cluster_name,
                signal_list) as output_session:
            with nixnet.SignalInSinglePointSession(
                    input_interface,
                    database_name,
                    cluster_name,
                    signal_list) as input_session:
                terminated_cable = six.moves.input('Are you using a terminated cable␣
↪(Y or N)? ')
                if terminated_cable.lower() == "y":
                    input_session.intf.lin_term = constants.LinTerm.ON
                    output_session.intf.lin_term = constants.LinTerm.OFF
                elif terminated_cable.lower() == "n":
                    input_session.intf.lin_term = constants.LinTerm.ON
                    output_session.intf.lin_term = constants.LinTerm.ON
                else:
                    print("Unrecognised input ({}), assuming 'n'".format(terminated_
↪cable))
                    input_session.intf.lin_term = constants.LinTerm.ON
                    output_session.intf.lin_term = constants.LinTerm.ON

                # Start the input session manually to make sure that the first
                # signal values sent before the initial read will be received.
                input_session.start()

                # Set the schedule. This will also automatically enable master mode.
                output_session.change_lin_schedule(0)

                # Generate a pair of random values and send out the signals.
                output_values = [randint(0, 255), randint(0, 255)]
                output_session.signals.write(output_values)
                print('Sent signal values: {}'.format(output_values))

                # Wait 1 s and then read the received values.
                # They should be the same as the ones sent.
```

(continues on next page)

(continued from previous page)

```
                time.sleep(1)

                input_signals = input_session.signals.read()
                input_values = [int(value) for timestamp, value in input_signals]
                print('Received signal values: {}'.format(input_values))


if __name__ == '__main__':
    main()
```

4.4 Contributing to nixnet

Contributions to **nixnet** are welcome from all!

nixnet is managed via git, with the canonical upstream repository hosted on GitHub.

nixnet follows a pull-request model for development. If you wish to contribute, you will need to create a GitHub account, fork this project, push a branch with your changes to your project, and then submit a pull request.

See GitHub's official documentation for more details.

4.4.1 Getting Started

To contribute to this project, it is recommended that you follow these steps:

1. Fork the repository on GitHub.
2. Run the unit tests on your system (see Testing section). At this point, if any tests fail, do not begin development. Try to investigate these failures. If you're unable to do so, report an issue through our GitHub issues page.
3. Write new tests that demonstrate your bug or feature. Ensure that these new tests fail.
4. Make your change.
5. Run all the unit tests again (which include the tests you just added), and confirm that they all pass.
6. Send a GitHub Pull Request to the main repository's master branch. GitHub Pull Requests are the expected method of code collaboration on this project.

4.4.2 Testing

In order to be able to run the **nixnet** unit tests, your setup should meet the following minimum requirements:

- Setup has a machine with NI-XNET or the NI-XNET Runtime installed.
- Machine has a supported version of CPython or PyPy installed.
- **TODO** Document required hardware and system setup.

nixnet relies on tox and pytest for testing

```
$ pip install tox
```

To run the tests:

```
$ # Unit tests:
$ tox
$ # Integration tests:
$ tox -c tox-integration.ini -- --can-in-interface CAN1 --can-out-interface CAN2 --
↪lin-in-interface LIN1 --lin-out-interface LIN2
$ # Integration tests (no LIN board):
$ tox -c tox-integration.ini -- --can-in-interface CAN1 --can-out-interface CAN2
```

Examples for debugging failures:

```
$ # Only run python3 unit tests
$ tox -e py3-test
$ # Further filter those tests to all starting with test_frames
$ tox -e py3-test -- -k test_frames
$ # Drop into PDB on first failure and quit when done
$ tox -e py3-test -- -x --pdb
```

4.4.3 Developer Certificate of Origin (DCO)

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

(taken from developercertificate.org)

See LICENSE for details about how **nixnet** is licensed.

CHAPTER 5

Indices and Tables

- genindex
- modindex

Python Module Index

Index