NI-XNET User Manual

2025-03-13

Contents

Welcome to the NI-XNET User Manual

The NI-XNET User Manual provides detailed descriptions of product functionality and step-by-step processes for use.

Looking for something else?

For information not found in the User Manual for your product, such as specifications and API reference, browse Related Information.

Related information:

- NI-XNET API Reference
- NI-XNET Tools and Utilities
- NI-XNET Database Editor
- NI-XNET Release Notes
- Download NI-XNET
- NI Learning Center
- NI Community

NI-XNET Overview

NI-XNET provides support for developing applications for prototyping, simulating, and testing Automotive Ethernet, CAN, LIN, and FlexRay networks.

NI-XNET is an NI instrument driver that enables test engineers and technicians to monitor, simulate, and automate controls for frames and signals on automotive bus systems. Use NI-XNET to develop applications for prototyping, simulating, and testing Automotive Ethernet, CAN, LIN, and FlexRay networks.

NI-XNET Key Features

NI-XNET driver software enables you to work with Automotive Ethernet, CAN, LIN, and FlexRay frames and signals on multiple platforms.

- Provides functions for reading and writing Automotive Ethernet, CAN, LIN, and FlexRay frames and signals on multiple platforms including PXI, PCI, NI CompactDAQ, and NI CompactRIO.
- Provides support for developing applications to prototype, simulate, and test Automotive Ethernet, CAN, LIN, and FlexRay networks.
- Enables real-time, high-speed manipulation of hundreds of CAN frames and signals, such as hardware-in-the-loop simulation, rapid control prototyping, bus monitoring, and automation control.

Components of an NI-XNET System

NI-XNET is designed for use in a test system that requires hardware, drivers, and software to optimize LabVIEW for your application. Use the following list of typical system components as a starting point for building your NI-XNET test system.

- Application development environment, such as LabVIEW
- NI-XNET driver software and database editor
- Software suite, including NI TestStand, Instrument Studio, and Switch Executive)

- NI-XNET high-performance Automotive Ethernet, CAN, LIN, or FlexRay interfaces (configuration determined based on customer needs)
- PXI chassis and controller
- Power supply
- Windows or Linux OS computer

NI-XNET Requirements

Your system must meet the following minimum requirements to run and use NI-XNET.

- Supported Operating Systems
  - Windows 11 or Windows 10 (64-bit)
  - Windows Server 2022 (64-bit)
  - Windows Server 2019 (64-bit)
  - Windows Server 2016 (64-bit)
  - RedHat Enterprise Linux 8 or 9.0
  - openSUSE Leap 15.3 or 15.4
  - Ubuntu 22.04 or 20.04

Note NI-XNET requires a 64-bit distribution and does not support 32-bit applications.

LabVIEW Project Provider

You can use NI-XNET features to create NI-XNET sessions within your LabVIEW project. You can drag these preconfigured NI-XNET sessions from the project to the block diagram and wire them directly to the XNET Read and XNET Write VIs.

You typically use a LabVIEW project when your application accesses the network using a fixed configuration. For example, if you are testing a single product, and your VI reads/writes a predetermined set of signals, a LabVIEW project is ideal.

Follow these steps to use NI-XNET within a LabVIEW project:

1. Right-click on the LabVIEW target you plan to use with NI-XNET. For Windows, this is **My Computer**. For LabVIEW Real-Time (RT), this is an RT target, such as a PXI controller.
2. Select **New»NI-XNET Session**.
3. Use the wizard and setup dialog to configure the session. Each configuration step has online help. When you are done, click **OK** to close the setup dialog.
4. If you do not have a VI already, add a VI under the LabVIEW target. You must use the new session within a VI listed under the same target.
5. Drag the new session to the VI block diagram. NI-XNET creates an XNET Read or XNET Write VI that matches the session mode. You need to make some changes to the block diagram, such as creating a loop. You now can run the VI.

If you require configuration of NI-XNET sessions at run time, you can use the XNET Create Session VI as an alternative to a LabVIEW project. For example, if your application tests a wide variety of products, and the end user of your application must select a database and its signals using the front panel, the XNET Create Session VI is ideal.

Troubleshooting and Common Questions

Where is my database on my disk?

The NI-XNET driver works with database aliases, which can cause some confusion when trying to share the actual database file. This also can cause problems if the database file is deleted on the disk, but the alias remains in the editor. There are two ways to find the path of your database on your disk:

- In the NI-XNET database editor, select **File » Manage Aliases**.
- In LabVIEW, right-click the I/O control and select **Manage Aliases**.

The NIXNET_example database is at `C:\Documents and Settings\All Users\Documents\National Instruments\NI-XNET\Examples`.

How is the example database alias automatically added?

NI-XNET is hard coded to detect whether you are trying to open a session using the NIXNET_example database and programmatically adds the alias for you if it is not already present.

How is the example database automatically deployed on LabVIEW RT?

The NI-XNET LabVIEW RT installer automatically deploys the NIXNET_example database during the installation. This makes it easier to test the example on your LabVIEW RT system.

The example database is added automatically on Windows and LabVIEW RT. Can I erase all traces of it?

Yes. Complete the following steps to erase all traces of the example database.

On Windows:

1. Open the **Manage NI-XNET Databases** dialog (see above), select the NIXNET_example alias on your local machine, and select **Remove Alias**.
2. Browse to `C:\Documents and Settings\All Users\Documents\National Instruments\NI-XNET\Examples` on your local machine and delete the `nixnet_example.xml` file.

Note The NI-XNET LabVIEW, CVI, and C examples work with this database file and therefore are not guaranteed to work if you delete the database file. The NI-XNET database is installed automatically with NI-XNET.

Can I permanently set the baud rate setting for my device as in NI-CAN?

There is no way to set the baud rate permanently in NI-XNET. The cluster in the FIBEX database file sets the baud rate. If you are using a frame streaming session without a database, you must set the baud rate programmatically.

Can I permanently set the transceiver type for my CAN XS device as in NI-CAN?

There is no way to set the transceiver type permanently in NI-XNET. The NI-XNET CAN XS device always defaults to a High Speed (HS) transceiver type. If you want a different transceiver type, you always must set it programmatically. You can set it programmatically in the following ways.

- (LabVIEW) Change the value (e.g., LS, HS) of the XNET Session property `Intf.CAN.Tcvr.Type`.
- (C) Use the following code:

```
Property = nxCANTcvrType_LS;
//(or Property = nxCANTcvrType_HS or Property = nxCANTcvrType_SW)
nxGetPropertySize (SessionRef, nxPropSession_IntfCANTcvrType, &PropertySize);
nxSetProperty (SessionRef, nxPropSession_IntfCANTcvrType, PropertySize,
&Property);
```

Can I change the database or object properties setting programmatically (for example, change the cycle time of a cyclic frame)?

Yes. You can open an object and change its properties programmatically. This has no effect on the actual database. It only changes the properties of the objects loaded in memory until the session is closed and the objects are released from memory. An example of how to do this is in the example finder at **Hardware Input and Output** » **CAN** » **NI-XNET** » **Intro to Sessions** » **Frame Sessions** » **CAN Change Frame Properties Dynamically**.

Why is there no XNET Clear VI at the end of the examples?

When the VI or application is stopped, NI-XNET takes care of closing all references for you. This makes programming simpler and more robust, as you do not need to ensure all references are closed.

Installation and Configuration

NI application software, such as LabVIEW, or another application development environment (ADE) such as Visual Studio should already be installed before you begin installing NI-XNET driver software. Refer to the ***NI-XNET Readme*** on your installation media for details about supported application software and ADE versions.

Install your NI-XNET software before connecting new hardware so that Windows can detect the devices; back up any applications before upgrading software or modifying the application.

Install NI-XNET Driver Software

NI automates installation using NI Package Manager. If not already installed, or if an upgrade is required, NI Package Manager is automatically installed. You can also download NI Package Manager at ni.com/r/NIPMDownload. Refer to the NI Package Manager Manual for information about installing, removing, and upgrading NI software.

To install NI-XNET software, you must log on as an administrator or as a user with administrator privileges.

1. Launch the NI-XNET installer.

If the installer displays the Package Manager license agreement, review the content, select **I accept the above license agreement**, and then click **Next**. Review the summary, and click **Next**. NI-XNET driver installation begins once NI Package Manager is installed.

2. Select any additional components that you want to install, and click **Next**. Typically, some suggestions are already selected. Click **Deselect All** to clear the selections, **Select All** to select everything in the list, or select only the additional items you want to install.
3. Review the NI-XNET license agreement, select **I accept the above license**

agreement, and then click **Next**.
4. Review the summary, and click **Next**.
5. When installation is complete, click **Reboot Now** and allow the computer to restart. Proceed to Install Your Hardware.

> **Note** Refer to ***LabVIEW Real-Time (RT) Configuration*** in the ***NI-XNET Hardware and Software Help*** for information about installing NI-XNET software on your LabVIEW Real-Time system.

Related concepts:

- Install Your Hardware
- LabVIEW Real-Time (RT) Configuration

Related information:

- Download NI Package Manager
- NI Package Manager Manual

Install Your Hardware

This section describes how to install your NI Automotive Ethernet, CAN, FlexRay, and LIN hardware on PCI/PCI Express, PXI/PXI Express, and USB buses, as well as how to install XNET C Series modules.

Related tasks:

- Install NI-XNET Driver Software

Safety Information

The following section contains important safety information that you must follow when installing and using the module.

Do ***not*** operate the module in a manner not specified in this document. Misuse of the module can result in a hazard. You can compromise the safety protection built into the

module if the module is damaged in any way. If the module is damaged, return it to National Instruments (NI) for repair.

Do ***not*** substitute parts or modify the module except as described in this document. Use the module only with the chassis, modules, accessories, and cables specified in the installation instructions. You ***must*** have all cover sand filler panels installed during operation of the module.

Do ***not*** operate the module in an explosive atmosphere or where there maybe flammable gases or fumes. If you must operate the module in such an environment, it must be in a suitably rated enclosure.

If you need to clean the module, use a soft, nonmetallic brush. Make sure that the module is completely dry and free from contaminants before returning it to service.

Operate the module only at or below Pollution Degree 2. Pollution is foreign matter in a solid, liquid, or gaseous state that can reduce dielectric strength or surface resistivity. The following is a description of pollution degrees:

- Pollution Degree 1 means no pollution or only dry, nonconductive pollution occurs. The pollution has no influence.
- Pollution Degree 2 means that only non conductive pollution occurs in most cases. Occasionally, however, a temporary conductivity caused by condensation must be expected.
- Pollution Degree 3 means that conductive pollution occurs, or dry,non conductive pollution occurs that becomes conductive due to condensation.

You ***must*** insulate signal connections for the maximum voltage for which the module is rated. Do ***not*** exceed the maximum ratings for Do ***not*** remove or add connector blocks when power is connected to the system. Avoid contact between your body and the connector block signal when hot swapping modules. Remove power from signal lines before connecting them to or disconnecting them from the module.

Operate the module at or below the ***installation category***[1] marked on the hardware label. Measurement circuits are subjected to ***working voltages***[2] and transient stresses (overvoltage) from the circuit to which they reconnected during measurement or test. Installation categories establish standard impulse withstand voltage levels that

commonly occur in electrical distribution systems. The following is a description of installation categories:

- Installation Category I is for measurements performed on circuits not directly connected to the electrical distribution system referred to as MAINS [3] voltage. This category is for measurements of voltages from specially protected secondary circuits. Such voltage measurements include signal levels, special equipment, limited-energy parts of equipment, circuits powered by regulated low-voltage sources, and electronics.
- Installation Category II is for measurements performed on circuits directly connected to the electrical distribution system. This category refers to local-level electrical distribution, such as that provided by standard wall outlet (for example, 115 AC voltage for U.S. or 230 AC voltage for Europe). Examples of Installation Category II are measurements performed on household appliances, portable tools, and similar modules.
- Installation Category III is for measurements performed in the building installation at the distribution level. This category refers to measurements on hard-wired equipment such as equipment in reinstallation, distribution boards, and circuit breakers. Other examples are wiring, including cables, bus bars, junction boxes, switches, socket outlets in the fixed installation, and stationary motors with permanent connections to fixed installations.
- Installation Category IV is for measurements performed at the primary electrical supply installation (<1,000 V). Examples include electricity meters and measurements on primary overcurrent protection device sand on ripple control units.

[1] Installation categories, also referred to as measurement categories, are defined in electrical safety standard IEC 61010–1.

[2] Working voltage is the highest rms value of an AC or DC voltage that can occur across any particular insulation.

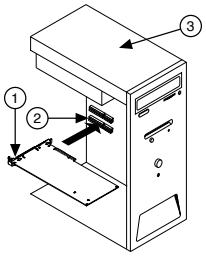
[3] MAINS is defined as a hazardous live electrical supply system that powers equipment. Suitably rated measuring circuits maybe connected to the MAINS for measuring purposes.

Install Your PCI/PCI Express Hardware

Caution Before you remove the device from its package, touch the antistatic plastic package to a metal part of the computer chassis to discharge electrostatic energy, which can damage components on your device.

1. Power off and unplug the computer.
2. Access the computer system expansion slots. This step might require you to remove one or more access panels on the computer case.

   Figure 1. Installing a PCI/PCI Express Device

   1. PCI/PCI Express Device
   2. PCI/PCI Express System Slot
   3. PC with PCI/PCI Express Slot
3. Locate a compatible slot and remove the corresponding slot cover on the computer back panel.

   Note Some motherboards reserve the x16 slot for onboard graphics.

4. Touch a metal part of the computer to discharge any static electricity.
5. Insert the device into the slot with the bus connector(s) sticking out of the opening on the back panel. Gently rock the device into place. Do not force the device into place.
6. Secure the mounting bracket to the computer back panel rail.
7. You can use a RTSI cable to connect your device RTSI interface to other NI RTSI-equipped hardware. Refer to ***Synchronization*** in the ***NI-XNET Hardware and Software Help*** for more information about the RTSI interface on your PCI/PCI Express device.
8. Replace any access panels on the computer case.
9. Proceed to Verify Your Installation.

Related concepts:

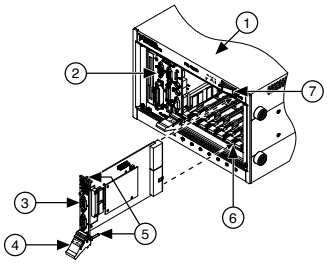
- Synchronization

Related tasks:

- Verify Your Installation

Install Your PXI/PXI Express Hardware

> **Caution** Before you remove the device from its package, touch the antistatic plastic package to a metal part of the computer chassis to discharge electrostatic energy, which can damage components on your device.

1. Power off the chassis, leaving the AC power source connected. Refer to your chassis manual to install or configure the chassis.
   The AC power cord grounds the chassis and protects it from electrostatic discharge (ESD) while you install the module.
2. Locate a compatible PXI or PXI Express peripheral slot and remove the corresponding filler panel.
3. Touch any metal part of the chassis to discharge static electricity.
   Figure 2. Installing a PXI/PXI Express Device in the Chassis
   

   1. PXI/PXI Express Chassis
   2. PXI/PXI Express System Controller
   3. PXI/PXI Express Module
   4. Injector/Ejector Handle
   5. Front-Panel Mounting Screws
   6. Module Guides
   7. Power Switch
4. Insert the module into the selected slot.

a. Ensure that the PXI/PXI Express module injector/ejector handle is not latched and swings freely.
b. Place the edges of the module into the slot guides at the top and bottom of the chassis.
c. Slide the module into the slot to the rear of the chassis.
d. When you begin to feel resistance, pull up on the injector/ejector handle to latch the device.

5. Screw the front panel of the module to the front panel mounting rail of the chassis.
6. Proceed to Verify Your Installation.

Related tasks:

- Verify Your Installation

Install Your USB Hardware

> **Caution** Before you remove the device from its package, touch the antistatic plastic package to a metal part of the computer chassis to discharge electrostatic energy, which can damage components on your device.

1. Connect the cable of your NI USB device to a High-Speed USB 2.0 port on your computer.
2. If applicable, connect accessories as described in the quick start guides.
3. You can use a synchronization cable to connect your module to other NI synchronization port-equipped hardware. Refer to ***Synchronization*** in the ***NI-XNET Hardware and Software Help*** for more information.
4. Proceed to Verify Your Installation.

Related concepts:

- Synchronization

Related tasks:

- Verify Your Installation

Install Your C Series Hardware

> **Caution** Before you remove the module from the package, touch the antistatic plastic package to a metal part of your system chassis to discharge electrostatic energy, which can damage components on your module.

1. When using your C Series hardware with a CompactDAQ chassis, refer to the chassis user manual for detailed installation instructions.
2. When using your C Series hardware with a CompactRIO chassis, refer to ***Installing C Series I/O Modules in the Chassis*** in the ***NI cRIO-9101/9102/9103/9104 User Manual and Specifications*** document for detailed installation instructions.
3. Connect the power source to your C Series module. The C Series modules require an external power supply that meets the specifications listed in the respective operating instructions.
4. Proceed to Verify Your Installation.

Related tasks:

- Verify Your Installation

Measurement & Automation Explorer (MAX)

You can use Measurement & Automation Explorer (MAX) to access all National Instruments products. Like other National Instruments hardware products, NI-XNET uses NI MAX as the centralized location for XNET device configuration.

To launch MAX, click the **Measurement & Automation** shortcut on the desktop or select **Start»Programs»National Instruments»Measurement & Automation**.

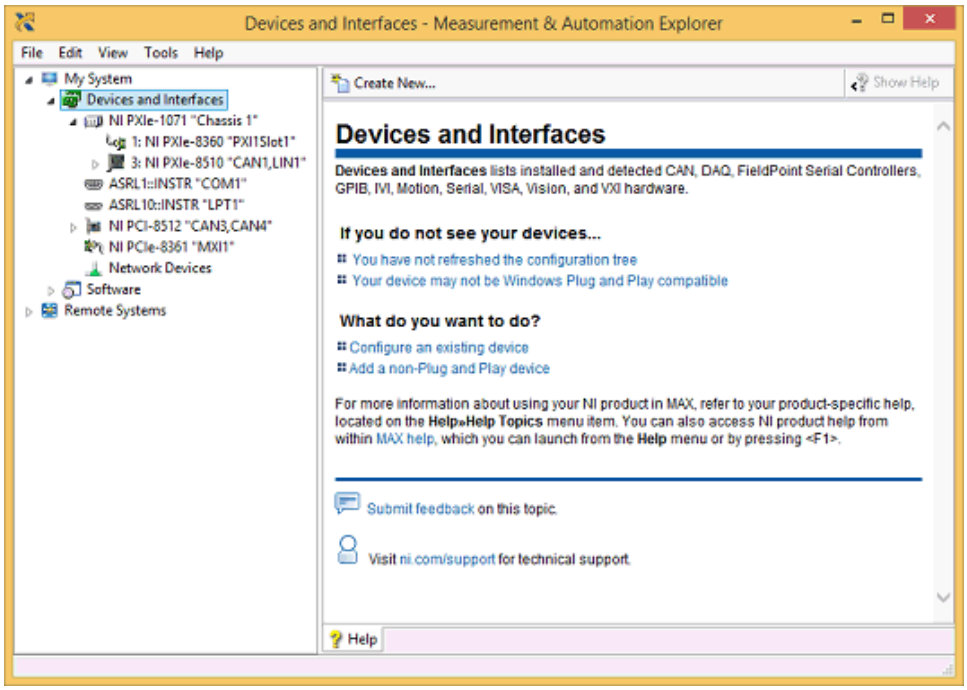
For information about the NI-XNET software in MAX, consult the online help at **Help»Help Topics»NI-XNET**.

You can view help for NI MAX Configuration tree items using the built-in MAX help pane. If this help pane does not appear on the right side of the MAX window, click the

Show Help button in the upper right corner.

Verifying NI-XNET Hardware Installation

The NI MAX Configuration tree **Devices and Interfaces** branch lists NI-XNET hardware (along with other local computer system hardware), as shown in the following figure.



NI-XNET Hardware Listed in MAX

If the NI-XNET hardware is not listed here, NI MAX is not configured to search for new devices on startup. To search for the new hardware, press <F5>.

To verify installation of the NI-XNET hardware, right-click the NI-XNET device and select **Self-Test**. If the self-test passes, the card icon shows a checkmark. If the self-test fails, the card icon shows an X mark, and the **Test Status** in the right pane describes the problem. Refer to Troubleshooting and Common Questions for information about resolving hardware installation problems.

Verify Your Installation

1. Power on your system and start Windows.

Windows may display a New Hardware Found dialog box. In this case, choose the default option, **Install the Software Automatically (Recommended)**, and let the operating system install the driver files.

2. Launch Measurement & Automation Explorer (MAX) and select **View** » **Refresh** from the menu or press <F5> to refresh.
   Your Automotive Ethernet, CAN, FlexRay, and LIN hardware should be listed now under **Devices and Interfaces**. To test detected NI-XNET hardware, right-click each device and select **Self Test**.
   If you are using a C Series module with CompactRIO, refer to ***Getting Started with CompactRIO*** in the ***NI-XNET Hardware and Software Help***.
3. Proceed to Connect the Cables.

Related concepts:

- Getting Started with CompactRIO

Related tasks:

- Install Your C Series Hardware
- Install Your PCI/PCI Express Hardware
- Install Your PXI/PXI Express Hardware
- Install Your USB Hardware

Related reference:

- Connect the Cables

Troubleshooting

If you have problems installing your software, visit ni.com/xnet. For hardware troubleshooting, go to ni.com/support and enter your device name, or go to ni.com/kb to search for knowledgebase articles.

If you think you have damaged your device and need to return your NI hardware for repair or device calibration, go to ni.com/info and enter the Info Code `rdsenn` for information about the Return Merchandise Authorization (RMA) process.

Related information:

- Download NI-XNET
- NI Support
- NI Knowledgebase
- Using an Info Code

Connect the Cables

After you have installed the hardware, connect your cables to the hardware. Refer to ***Cabling Requirements*** in the ***NI-XNET Hardware and Software Help*** for information about cabling requirements for your NI-XNET hardware.

Related concepts:

- Cabling Requirements for Automotive Ethernet
- Cabling Requirements for FlexRay
- Cabling Requirements for High-Speed CAN
- Cabling Requirements for LIN
- Cabling Requirements for Low-Speed/Fault-Tolerant CAN
- Cabling Requirements for Single Wire CAN

Related tasks:

- Verify Your Installation

Uninstalling NI-XNET Software

Complete the following steps to uninstall the NI-XNET software.

1. Open the NI Package Manager, either from the Start Menu or by searching for NI Package Manager.
2. Select the **Installed** tab at the top of the window to open the Installed page.
3. In the list of installed software, select **NI-XNET**, and then click **Remove**.
   Use the search field at the top of the page to filter the list. For example, type `XNET` in the search field and press **Enter**.

4. Review the summary, and click **Next**.
5. When removal is complete, click **Close**.

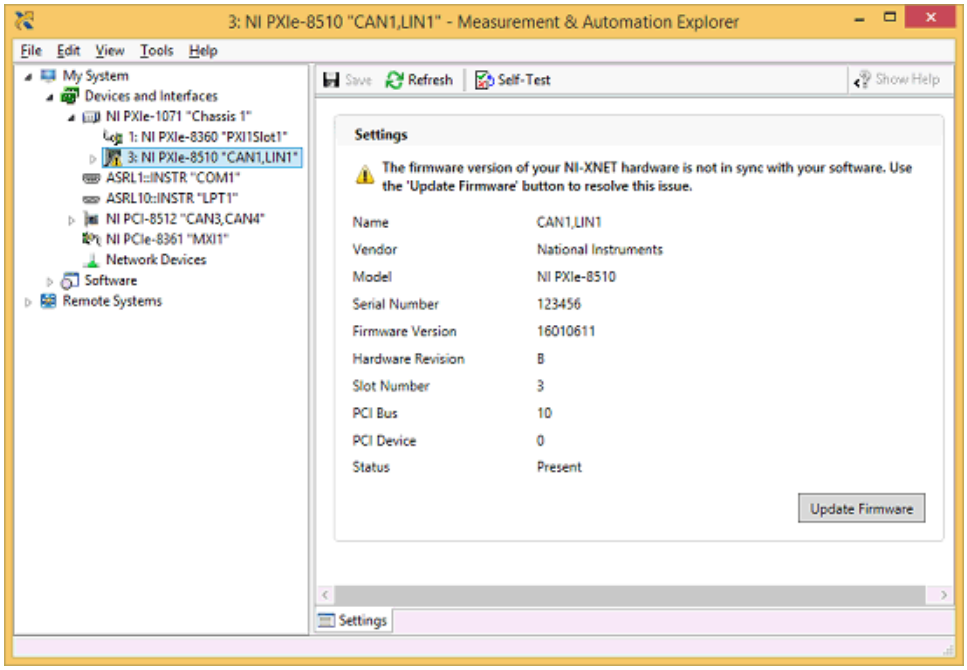
> **Note** NI Package Manager removes only components that were installed with the software (folders, utilities, device drivers, DLLs, and registry entries). If you have added anything to a directory created when the software was installed, NI Package Manager cannot delete that directory because it is not empty after the software is removed. You may need to manually remove any remaining components.

6. Restart your computer.

XNET Device Firmware Update

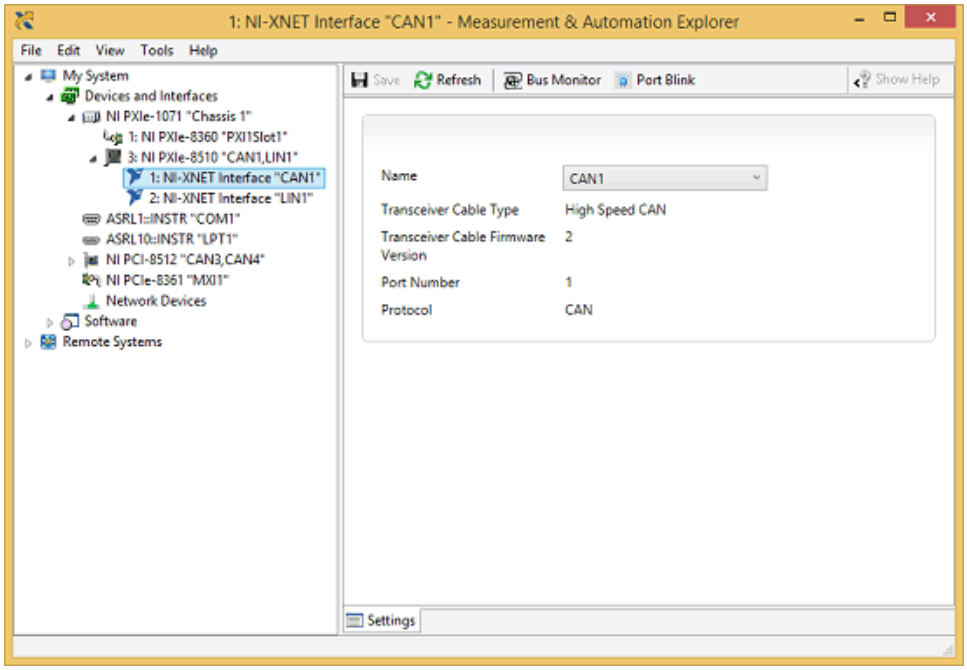
For PXI Express devices and C Series modules, the firmware is not updated automatically when you open an XNET session. The right pane in NI MAX displays the firmware status.

If the firmware on the XNET device does not match the version the XNET software expects, a yellow warning is displayed on the device icon, and a message is displayed in the right pane, as shown below. In this case, you can use the **Update Firmware** button to apply the proper firmware version to the device.



Configuring NI-XNET Interfaces

The NI-XNET hardware interfaces are listed under the device name. To change the interface name, select a new one from the **Name** box in the right pane, as shown below.



LabVIEW Real-Time (RT) Configuration

LabVIEW Real-Time (RT) combines easy-to-use LabVIEW programming with the power of real-time systems. When you use an NI PXI controller, you can install an NI-XNET card and use the NI-XNET API to develop real-time applications. For example, you can simulate the behavior of a control algorithm within an XNET device, using data from received NI-XNET messages to generate outgoing NI-XNET messages with deterministic response times.

When you install the NI-XNET software, the installer copies components for LabVIEW RT to the Windows system. As with any other NI product for LabVIEW RT, you then download the NI-XNET software to the LabVIEW RT system using the **Remote Systems** branch in NI MAX. For more information, refer to the LabVIEW RT documentation.

After you install the NI-XNET hardware and download the NI-XNET software to the

LabVIEW RT system, you can verify the installation. Find your RT target under **Remote Systems** and open the **Devices and Interfaces** item. Perform a self test for all installed NI-XNET devices.

Related tasks:

- Install NI-XNET Driver Software

Getting Started with CompactRIO

When you use a C Series NI-XNET module in a CompactRIO chassis, the NI-XNET features on LabVIEW RT are the same as on other LabVIEW RT targets, such as PXI. Nevertheless, the communication between the NI-XNET RT driver and module does not exist in the default FPGA VI that ships with CompactRIO. Prior to using NI-XNET features, you must use LabVIEW FPGA to compile and run an FPGA VI that contains the required communication logic.

The following steps describe how to use a C Series NI-XNET module in a CompactRIO chassis from its out-of-box configuration.

1. Install the required software to the host computer.
   a. LabVIEW (Including RT and FPGA)

      Install LabVIEW, LabVIEW Real-Time, LabVIEW FPGA, and NI-RIO. For supported versions of the software mentioned above, refer to the Supported Platforms section in the NI-XNET readme file.

   b. NI-XNET

      Install NI-XNET after the required LabVIEW components.

2. Install NI-XNET to the CompactRIO RT controller. Use MAX to find your CompactRIO controller under **Remote Systems**, then right-click **Software** and select **Change/ Remove Software**. There are two ways to install the required components:
   - **NI-RIO with NI Scan Engine Support** If this selection is dimmed, refer to the explanation on the right to resolve the problem, or use custom installation. After selecting this item, the next page displays a list of add-ons. Scroll down to

the bottom of the add-on list to check **NI-XNET**.

- **Custom Software Installation** Custom installation can be useful on controllers with small amounts of memory because you can use it to avoid installing unused components. Select the **NI-XNET** item, which in turn selects the required dependencies (for example, NI-RIO IO Scan).

3. Add modules to the LabVIEW project.
   To compile an FPGA VI with the required communication logic, you must add NI-XNET modules in a LabVIEW project.
   a. Add the controller.

      Assuming your controller is online, you can right-click the project item and select **New»Targets and Devices»Existing target or device**, then select your controller under **Real-Time CompactRIO**. If your controller is offline, you can add it by selecting **New target or device**.

   b. Select the chassis programming mode.

      When you add the controller, a dialog asks you to select the programming mode for the chassis. Although NI-XNET uses scan engine components, you must select **LabVIEW FPGA Interface** as the chassis mode. This configures the chassis to support compiling an FPGA VI.

      If a **Discover C Series Modules?** dialog appears, click the **Do Not Discover** button and proceed to step d.

   c. Ignore errors for discovered NI-XNET modules.

      LabVIEW 2010 may report an error for NI-XNET modules, stating that LabVIEW FPGA is not supported. LabVIEW 2011 or later does not report this error. Do not change the chassis to Scan Interface mode. Ignore this error message and click **Continue**.

   d. Add NI-XNET modules.

      Right-click the chassis item under the controller (not FPGA) and select **New»C Series Modules»Existing target or device** . Select the plus sign to discover and then hold <Shift> to select all NI-XNET modules in the list. Click **OK** to add the modules to the project. You also can add NI-XNET modules offline by selecting

New target or device, then **C Series Module**, and in the next dialog select the appropriate **Module Type** (for example, NI 9862). When you use an NI-XNET module in a project, you do not necessarily need to have that module installed physically. For NI-XNET, the module in the project is simply a signal to the FPGA VI that NI-XNET communication is required for that slot.

4. Compile and run the FPGA VI.If you are new to CompactRIO, you can use an empty FPGA VI to get started quickly with NI-XNET tools and examples. Select the FPGA target in the LabVIEW project, and then select **New»VI**. When the front panel opens, click the LabVIEW run button (the arrow) to compile and run the VI. Although the VI is empty, it loads the required NI-XNET support. When compilation completes, and the VI runs the first time, you can close the front panel and proceed to the next step.

   If you have an existing FPGA VI in your project, you must recompile the FPGA VI to incorporate NI-XNET support for the configured slots. When the FPGA VI is recompiled, you run it using the same methods you used previously. This typically is done using **Open FPGA VI Reference** from a host VI.

   The following tables provide a detailed list of actions that cause NI-XNET to load and unload. NI-XNET must be loaded for its hardware to be detected. Within the tables, the term XNET-enabled FPGA VI refers to an FPGA VI compiled with a project that contains at least one NI-XNET module. The term XNET-disabled FPGA VI refers to an FPGA VI compiled with no NI-XNET modules.

Table 1. Actions That Cause NI-XNET to Load

Action	Comment
Invoke Open FPGA VI Reference with an XNET-enabled FPGA VI.	NI-XNET loads regardless of whether **Run the FPGA VI** is checked in the configuration dialog.
Run the XNET-enabled FPGA VI using Interactive Front Panel Communication.	—

Note NI-XNET does not load when the CompactRIO system powers up. Even if you configure an XNET-enabled FPGA VI to load automatically on power on, you must perform an action from Table 1 prior to using NI-XNET.

Table 2. Actions That Cause NI-XNET to Unload

Action	Comment
Invoke Close FPGA VI Reference with the shortcut option **Close and Reset if Last Reference (default)**.	If the reference is not the last to close, NI-XNET remains loaded. The shortcut options **Close** and **Close and Abort without Reference Counting** do not unload NI-XNET.
Power down CompactRIO.	—
Run XNET-disabled FPGA VI.	This applies to Open FPGA VI Reference or Interactive Front Panel Communication.
Invoke Reset using the Invoke Method node of the FPGA interface.	Reset of an open FPGA reference causes NI-XNET to unload, and then immediately load again. If you are using NI-XNET sessions during the reset, the sessions are invalidated. Other methods such as Abort do not unload NI-XNET.
Run a different XNET-enabled FPGA VI from the XNET-enabled FPGA VI currently loaded.	When you change FPGA VIs, the effect is the same as the reset method. NI-XNET unloads and then immediately loads again.

Note When using FPGA Interactive Front Panel Communication, stopping the FPGA VI does not unload NI-XNET. This applies to stopping the VI normally (for example, from the front panel button), or using the LabVIEW abort button (the stop sign).

5. Wait for interfaces to be detected. After the FPGA runs with NI-XNET support, it may take a few seconds for the new FPGA features to be detected, appropriate RT drivers to load, and NI-XNET modules to be detected. This delay occurs only after you perform the action from Table 1.There are several options for detecting NI-XNET interface hardware:
   - **MAX Devices & Interfaces** You can detect the interfaces visually by opening the **Devices & Interfaces** tree under the RT controller in NI MAX. Once the hardware is detected, you can perform a self test to confirm that all hardware and software is ready to use.
   - **LabVIEW Interface I/O Name** When you drop an XNET interface I/O name control on the front panel of an RT VI, the control uses features similar to NI MAX to display available interfaces. For interface detection to operate, you must right-click the RT controller in the LabVIEW project and select **Connect**

(or **Deploy**). Once connected, you can use the interface I/O name to select an interface prior to running the RT VI.

- **System API** If you need to detect interfaces programmatically within a running RT VI, National Instruments provides APIs for this purpose. The NI System Configuration API can detect any NI hardware product, including NI-XNET interfaces. NI-XNET also provides a System API with properties specific to NI-XNET hardware. If you run your RT VI as a startup VI (for example, after power on), you must perform an action from Table 1, then use a System API to wait for the required interfaces prior to calling **XNET Create Session**. If you create an I/O session prior to detecting the specified interface, an interface-not-found error can occur.

6. Use NI-XNET. Once the interfaces are detected, you are ready to use them. Within your RT VI, NI-XNET sessions are used to read and write I/O data. For more information, refer to the Sessions topics.

Related tasks:

- Verify Your Installation

CAN Overview

CAN FD, ISO Versus Non-ISO

Bosch published several versions of the CAN specification, such as CAN 2.0, published in 1991. This specification has two parts; part A is for the standard format with an 11-bit identifier, and part B is for the extended format with a 29-bit identifier. CAN 2.0 supports frames with payload up to 8 bytes and transmission speed up to 1 Mbaud.

To allow faster transmission rates, in 2012 Bosch released CAN FD 1.0 (CAN with Flexible Data-Rate), supporting a payload length up to 64 bytes and faster baud rates. ISO later standardized CAN FD. ISO CAN FD 11898-1:2015 introduced some changes to the original CAN FD 1.0 protocol from Bosch, which made the CAN FD 1.0 (non-ISO CAN FD) and ISO CAN FD protocols incompatible. These changes are now available under ISO 11898-1:2015. The standards cannot communicate with each other.

NI-XNET supports both ISO CAN FD and non-ISO CAN FD. The default is ISO CAN FD. The NI-XNET API behavior supporting ISO CAN FD mode has been changed slightly to allow new features compared to the Non-ISO FD mode. In Non-ISO CAN FD mode, you must use the Interface:CAN:Transmit I/O Mode session property to switch the CAN I/O mode of transmitted frames. In ISO CAN FD mode, the transmission mode is specified in the database (CAN:I/O Mode property) or, when the database is not used, in the frame type field of the frame header.

Received data frames in Non-ISO CAN FD mode always have the type CAN Data, while in ISO CAN FD mode the type is more specific, indicating the protocol in which the frame has been transmitted (CAN 2.0, CAN FD, or CAN FD+BRS).

Because an existing CAN FD application developed with NI-XNET 15.0 (which supported non-ISO CAN FD only) might not work with the API changes for ISO CAN FD, NI-XNET 15.5 has introduced a Legacy ISO mode. In this mode, the API behavior is the same as in Non-ISO CAN FD mode, but it communicates on the bus using ISO CAN FD mode.

You define the ISO CAN FD mode when you add an alias for a database supporting CAN FD. In a dialog box (or the XNET Database Add Alias VI), you define whether the mode

default is ISO CAN FD, Non-ISO CAN FD, or Legacy ISO mode. In the session, you still can change the ISO mode with an Interface:CAN:FD ISO Mode property.

Understanding CAN Frame Timing

When you use an NI-XNET database for CAN, the properties of each CAN frame specify the CAN data transfer timing. To understand how the CAN frame timing properties apply to NI-XNET sessions, refer to CAN Timing Type and Session Mode.

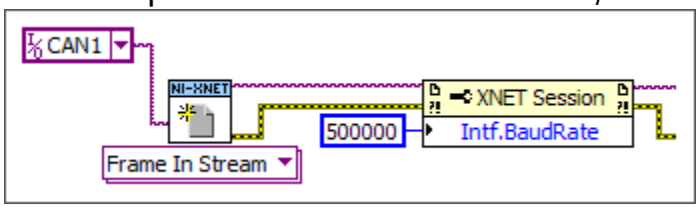
Configuring Frame I/O Stream Sessions

As described in Database Programming, you typically need to specify database objects when creating an NI-XNET session.

The CAN protocol supports an exception that makes some applications easier to program. In sessions with Frame Input Stream or Frame Output Stream mode, you can read or write arbitrary frames. Because these modes do not use specific frames, only the database cluster properties apply. For CAN, the only required cluster property is the baud rate. If the I/O mode of your cluster is CAN FD or CAN FD+BRS, the FD baud rate also is required.

Although the CAN baud rate applies to all hardware on the bus (cluster), NI-XNET also provides the baud rate properties as interface properties. You can set these interface properties using the session property node.

If your application uses only Frame I/O Stream sessions, no database object is required (no cluster). You simply can call the XNET Create Session VI and then set the baud rate using the session property node. The following figure shows an example diagram that creates a Frame Input Stream session and sets the baud rate to 500 kbps. The resulting session operates in the standard CAN I/O mode.

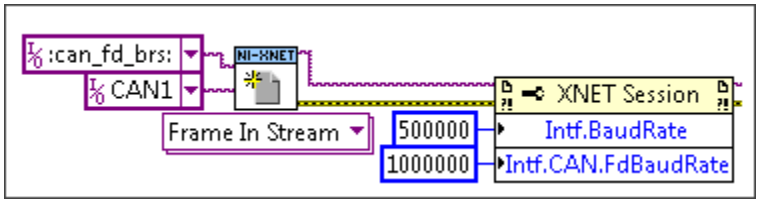


Configure CAN Frame Input Stream

If your application uses only Frame I/O Stream sessions, but you want to connect to a

CAN FD bus, use the in-memory database :can_fd: or :can_fd_brs: as shown in the following figure. These databases are configured as a CAN cluster with the CAN:I/O Mode set to CAN FD or CAN FD+BRS, as appropriate. If you use either database, you must set the Interface:CAN:64bit FD Baud Rate property.

Configure CAN Frame Input Stream for a CAN FD Session



Related concepts:

- CAN Timing Type and Session Mode
- Database Programming for the C API

Low-Speed CAN

Low-Speed CAN is commonly used to control "comfort" devices in an automobile, such as seat adjustment, mirror adjustment, and door locking. It differs from High-Speed CAN in that the maximum baud rate is 125K and it utilizes CAN transceivers that offer fault-tolerant capability. This enables the CAN bus to keep operating even if one of the wires is cut or short-circuited because it operates on relative changes in voltage, and thus provides a much higher level of safety. The transceiver solves many common and frequent wiring problems such as poor connectors, and also overcomes short circuits of either transmission wire to ground or battery voltage, or the other transmission wire. The transceiver resolves the fault situation without involvement of external hardware or software. On the detection of a fault, the transceiver switches to a one wire transmission mode and automatically switches back to differential mode if the fault is removed.

Special resistors are added to the circuitry for the proper operation of the fault-tolerant transceiver. The values of the resistors depend on the number of nodes and the resistance values per node. For guidelines on selecting the resistor, refer to Cabling Requirements for Low-Speed/Fault-Tolerant CAN.

Related concepts:

- Cabling Requirements for Low-Speed/Fault-Tolerant CAN

Single Wire CAN

Single wire CAN is found primarily in specialty automotive applications and emphasizes low cost. Defined in the SAE 2411 specification, single wire CAN uses only one single-ended CAN data wire, as opposed to the differential CAN wires found in most applications. The reduced noise immunity of single wire CAN limit its speed compared to the other CAN physical layers.

Single wire CAN offers four communication modes. The first two modes relate the CAN bus speed. The first mode, Normal Mode, allows the controller to run at 33.333 Kbits/s and is the mode the bus runs in when conducting in-vehicle traffic. The second mode, High Speed Mode, allows the controller to run at 83.333 Kbits/s and is for data download when attached to an offboard tester ECU.

When running in either of the first two modes, the nominal voltage levels are 0 V and 4 V. If a controller goes into Sleep Mode, it ignores all traffic running at these voltage levels. The final mode is called High Voltage Wakeup mode and transmits only at normal communication speeds at nominal voltage levels of 0 V and 12 V (actual high voltage is typically close to V_{bat}). If a controller goes into Sleep Mode, it wakes up when receiving a CAN frame at the high-voltage signaling levels.

For cabling guidelines and other information, refer to Single Wire Physical Layer.

Related concepts:

- Single Wire CAN Physical Layer

CAN Error Detection and Confinement

One of the most important and useful features of CAN is its high reliability, even in extremely noisy environments. CAN provides a variety of mechanisms to detect errors in frames. This error detection is used to retransmit the frame until it is received successfully. CAN also provides an error confinement mechanism used to remove a

malfunctioning device from the CAN network when a high percentage of its frames result in errors. This error confinement prevents malfunctioning devices from disturbing the overall network traffic.

Error Detection

Whenever any CAN device detects an error in a frame, that device transmits a special sequence of bits called an error flag. This error flag is normally detected by the device transmitting the invalid frame, which then retransmits to correct the error. The retransmission starts over from the start of frame, and thus arbitration with other devices can occur again.

CAN devices detect the following errors, which are described in the following topics:

- Bit error
- Stuff error
- CRC error
- Form error
- Acknowledgment error

Related concepts:

- Bit Error
- Stuff Error
- CRC Error
- Form Error
- Acknowledgment Error

Bit Error

During frame transmissions, a CAN device monitors the bus on a bit-by-bit basis. If the bit level monitored is different from the transmitted bit, a bit error is detected. This bit error check applies only to the Data Length Code, Data Bytes, and Cyclic Redundancy Check fields of the transmitted frame.

Related concepts:

- Error Detection

Stuff Error

Whenever a transmitting device detects five consecutive bits of equal value, it automatically inserts a complemented bit into the transmitted bit stream. This stuff bit is automatically removed by all receiving devices. The bit stuffing scheme is used to guarantee enough edges in the bit stream to maintain synchronization within a frame.

A stuff error occurs whenever six consecutive bits of equal value are detected on the bus.

Related concepts:

- Error Detection

CRC Error

A CRC error is detected by a receiving device whenever the calculated CRC differs from the actual CRC in the frame.

Related concepts:

- Error Detection

Form Error

A form error occurs when a violation of the fundamental CAN frame encoding is detected. For example, if a CAN device begins transmitting the Start Of Frame bit for a new frame before the End Of Frame sequence completes for a previous frame (does not wait for bus idle), a form error is detected.

Related concepts:

- Error Detection

Acknowledgment Error

An acknowledgment error is detected by a transmitting device whenever it does not detect a dominant Acknowledgment Bit (ACK).

Related concepts:

- Error Detection

Error Confinement

To provide for error confinement, each CAN device must implement a transmit error counter and a receive error counter. The transmit error counter is incremented when errors are detected for transmitted frames, and decremented when a frame is transmitted successfully. The receive error counter is used for received frames in much the same way. The error counters are increased more for errors than they are decreased for successful reception/transmission. This ensures that the error counters will generally increase when a certain ratio of frames (roughly 1/8) encounter errors. By maintaining the error counters in this manner, the CAN protocol can generally distinguish temporary errors (such as those caused by external noise) from permanent failures (such as a broken cable). For complete information about the rules used to increment/decrement the error counters, refer to the CAN specification (ISO 11898).

With regard to error confinement, each CAN device may be in one of three states: error active, error passive, and bus off.

Related concepts:

- Error Active State
- Error Passive State
- Bus Off State

Error Active State

When a CAN device is powered on, it begins in the error active state. A device in error active state can normally take part in communication, and transmits an active error flag when an error is detected. This active error flag (sequence of dominant 0 bits) causes the current frame transmission to abort, resulting in a subsequent retransmission. A CAN device remains in the error active state as long as the transmit and receive error counters are both below 128. In a normally functioning network of CAN devices, all devices are in the error active state.

Related concepts:

- CAN FD Frames
- Error Confinement

Error Passive State

If either the transmit error counter or the receive error counter increments above 127, the CAN device transitions into the error passive state. A device in error passive state can still take part in communication, but transmits a passive error flag when an error is detected. This passive error flag (sequence of recessive 1 bits) generally does not abort frames transmitted by other devices. Because passive error flags cannot prevail over any activity on the bus line, they are noticed only when the error passive device is transmitting a frame. Thus, if an error passive device detects a receive error on a frame which is received successfully by other devices, the frame is not retransmitted.

One special rule to keep in mind: When an error passive device detects an acknowledgment error, it does not increment its transmit error counter. Thus, if a CAN network consists of only one device (for example, if you do not connect a cable to the National Instruments CAN interface), and that device attempts to transmit a frame, it retransmits continuously but never goes into bus off state (although it eventually reaches error passive state).

Related concepts:

- CAN FD Frames
- Error Confinement

Bus Off State

If the transmit error counter increments above 255, the CAN device transitions into the bus off state. A device in the bus off state does not transmit or receive any frames, and thus cannot have any influence on the bus. The bus off state disables a malfunctioning CAN device that frequently transmits invalid frames, so that the device does not adversely affect other devices on the network. When a CAN device transitions to bus off, it can be placed back into error active state (with both counters reset to zero) only by manual intervention. For sensor/actuator types of devices, this often involves powering the device off then on. For NI-XNET network interfaces, communication can be started again using an API function.

Related concepts:

- Error Confinement

CAN Identifiers and Message Priority

When a CAN device transmits data onto the network, an identifier that is unique throughout the network precedes the data. The identifier defines not only the content of the data, but also the priority.

When a device transmits a message onto the CAN network, all other devices on the network receive that message. Each receiving device performs an acceptance test on the identifier to determine if the message is relevant to it. If the received identifier is not relevant to the device (such as RPM received by an air conditioning controller), the device ignores the message.

When more than one CAN device transmits a message simultaneously, the identifier is used as a priority to determine which device gains access to the network. The lower the numerical value of the identifier, the higher its priority.

The following figure shows two CAN devices attempting to transmit messages, one using identifier 647 hex, and the other using identifier 6FF hex. As each device transmits the 11 bits of its identifier, it examines the network to determine if a higher-priority identifier is being transmitted simultaneously. If an identifier collision is detected, the losing device(s) immediately stop transmission and wait for the higher-priority message to complete before automatically retrying. Because the highest priority identifier continues its transmission without interruption, this scheme is referred to as nondestructive bitwise arbitration , and CAN's identifier is often referred to as an arbitration ID. This ability to resolve collisions and continue with high-priority transmissions is one feature that makes CAN ideal for real-time applications.

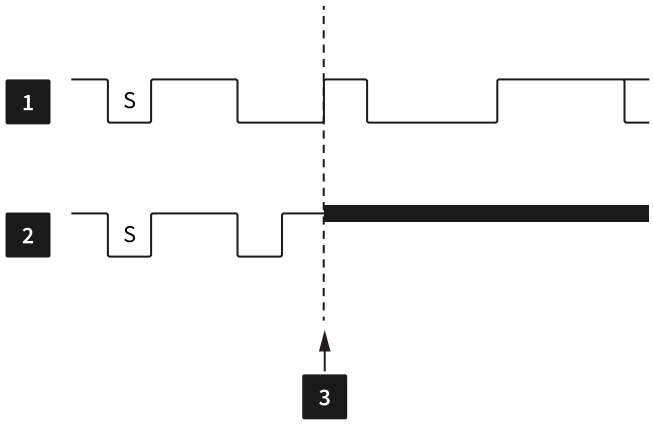
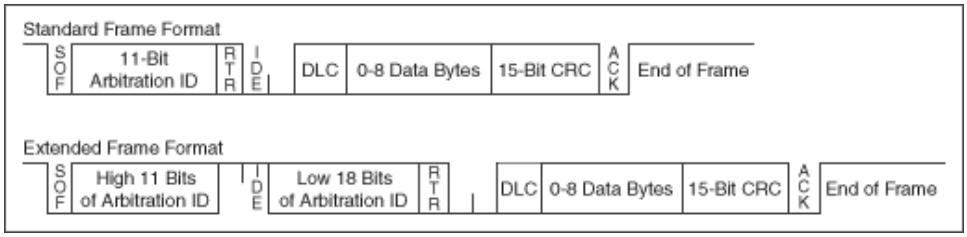


Table 3. Example of CAN Arbitration

1 Device A: ID = 11001000111 (647 hex)
2 Device B: ID = 11011111111 (6FF hex)
3 Device B Loses Arbitration; Device A Wins Arbitration and Proceeds
S = Start Frame Bit

CAN Frames

In a CAN network, the messages transferred across the network are called frames. The CAN protocol supports two frame formats as defined in the Bosch version 2.0 specifications, the essential difference being in the length of the arbitration ID. In the standard frame format (also known as 2.0A), the length of the ID is 11 bits. In the extended frame format (also known as 2.0B), the length of the ID is 29 bits. The following figure shows the essential fields of the standard and extended frame formats, and the following topics describe each field.



Start of Frame (SOF)

Start of Frame is a single bit (0) that marks the beginning of a CAN frame.

Arbitration ID

The arbitration ID fields contain the identifier for a CAN frame. The standard format has one 11-bit field, and the extended format has two fields, which are 11 and 18 bits in length. In both formats, bits of the arbitration ID are transmitted from high to low order.

Remote Transmit Request (RTR)

The Remote Transmit Request bit is dominant (0) for data frames, and recessive (1) for remote frames. Data frames are the fundamental means of data transfer on a CAN network, and are used to transmit data from one device to one or more receivers. A device transmits a remote frame to request transmission of a data frame for the given arbitration ID. The remote frame is used to request data from its source device, rather than waiting for the data source to transmit the data on its own.

Identifier Extension (IDE)

The Identifier Extension bit differentiates standard frames from extended frames. Because the IDE bit is dominant (0) for standard frames and recessive (1) for extended frames, standard frames are always higher priority than extended frames.

Data Length Code (DLC)

The Data Length Code is a 4-bit field that indicates the number of data bytes in a data frame. In a remote frame, the Data Length Code indicates the number of data bytes in the requested data frame. Valid Data Length Codes range from zero to eight.

Data Bytes

For data frames, this field contains from 0 to 8 data bytes. Remote CAN frames always contain zero data bytes.

Cyclic Redundancy Check (CRC)

The 15-bit Cyclic Redundancy Check detects bit errors in frames. The transmitter calculates the CRC based on the preceding bits of the frame, and all receivers recalculate it for comparison. If the CRC calculated by a receiver differs from the CRC in the frame, the receiver detects an error.

Acknowledgment Bit (ACK)

All receivers use the Acknowledgment Bit to acknowledge successful reception of the frame. The ACK bit is transmitted recessive (1), and is overwritten as dominant (0) by all devices that receive the frame successfully. The receivers acknowledge correct frames regardless of the acceptance test performed on the arbitration ID. If the transmitter of the frame detects no acknowledgment, it could mean that the receivers detected an error (such as a CRC error), the ACK bit was corrupted, or there are no receivers (for example, only one device on the network). In such cases, the transmitter automatically retransmits the frame.

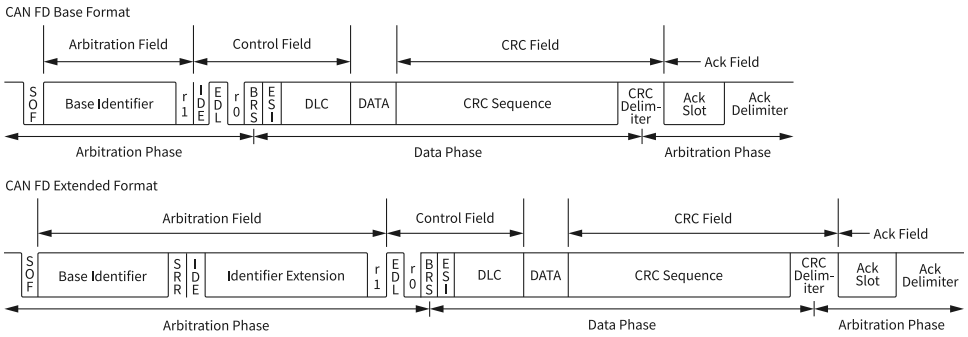
End of Frame

Each frame ends with a sequence of recessive bits. After the required number of recessive bits, the CAN bus is idle, and the next frame transmission can begin.

CAN FD Frames

CAN FD Standard and Extended Frame Formats

The CAN FD standard supports the same two frame formats as defined in the Bosch version 2.0 specification, as well as two additional frame formats. The essential difference between the original and new format is the addition of a few bits to redefine the DLC and increase the data phase speed. The following figure shows the essential fields of the standard and extended FD frame formats, and the following sections describe each field that differs from the CAN 2.0 specification.



Extended Data Length Bit (EDL)

The EDL bit indicates the frame is a CAN FD frame. This is the r0 bit in a standard frame

and is transmitted dominate. For a CAN FD frame, the EDL bit is transmitted recessive.

When this bit is set, the DLC is interpreted differently than when the frame is a standard CAN 2.0 frame. as shown in the following table:

DLC	CAN 2.0	CAN FD
0..8	0..8	0..8
9	8	12
10	8	16
11	8	20
12	8	24
13	8	32
14	8	48
15	8	64

Bit Rate Switch Bit (BRS)

The BRS bit indicates whether the bit rate of the non-arbitration portion of the CAN frame is transmitted at the standard data rate or the fast CAN FD rate. This bit is transmitted dominate to transmit at the standard rate and recessive to transmit at the CAN FD rate.

Error State Indicator Bit (ESI)

The ESI bit is transmitted dominate by a node in the Error Active State and recessive by a node in the Error Passive State.

Cyclic Redundancy Check Sequence (CRC)

The CAN FD standard uses a different CRC polynomial than the CAN 2.0 standard. The CAN 2.0 standard uses a 15-bit CRC, while the CAN FD standard uses two separate CRC polynomials. The first CRC is 17 bits, for frames with a payload of 0–16 bytes. The second CRC is 21 bits, for frames larger than 16 bytes.

Related concepts:

- Error Active State
- Error Passive State

CAN FD, ISO Versus Non-ISO

Bosch published several versions of the CAN specification, such as CAN 2.0, published in 1991. This specification has two parts; part A is for the standard format with an 11-bit identifier, and part B is for the extended format with a 29-bit identifier. CAN 2.0 supports frames with payload up to 8 bytes and transmission speed up to 1 Mbaud.

To allow faster transmission rates, in 2012 Bosch released CAN FD 1.0 (CAN with Flexible Data-Rate), supporting a payload length up to 64 bytes and faster baud rates. ISO later standardized CAN FD. ISO CAN FD 11898-1:2015 introduced some changes to the original CAN FD 1.0 protocol from Bosch, which made the CAN FD 1.0 (non-ISO CAN FD) and ISO CAN FD protocols incompatible. These changes are now available under ISO 11898-1:2015. The standards cannot communicate with each other.

NI-XNET supports both ISO CAN FD and non-ISO CAN FD. The default is ISO CAN FD. The NI-XNET API behavior supporting ISO CAN FD mode has been changed slightly to allow new features compared to the Non-ISO FD mode. In Non-ISO CAN FD mode, you must use the Interface:CAN:Transmit I/O Mode session property to switch the CAN I/O mode of transmitted frames. In ISO CAN FD mode, the transmission mode is specified in the database (CAN:I/O Mode property) or, when the database is not used, in the frame type field of the frame header.

Received data frames in Non-ISO CAN FD mode always have the type CAN Data, while in ISO CAN FD mode the type is more specific, indicating the protocol in which the frame has been transmitted (CAN 2.0, CAN FD, or CAN FD+BRS).

Because an existing CAN FD application developed with NI-XNET 15.0 (which supported non-ISO CAN FD only) might not work with the API changes for ISO CAN FD, NI-XNET 15.5 has introduced a Legacy ISO mode. In this mode, the API behavior is the same as in Non-ISO CAN FD mode, but it communicates on the bus using ISO CAN FD mode.

You define the ISO CAN FD mode when you add an alias for a database supporting CAN FD. In a dialog box (or `nxdbAddAlias64`), you define whether the mode default is

ISO CAN FD, Non-ISO CAN FD, or Legacy ISO mode. In the session, you still can change the ISO mode with an Interface:CAN:FD ISO Mode property.

FlexRay Overview

The FlexRay communications network is a new, deterministic, fault-tolerant, and high-speed bus system developed in conjunction with automobile manufacturers and leading suppliers.

FlexRay delivers the error tolerance and time-determinism performance requirements for X-by-wire applications (for example, drive-by-wire, steer-by-wire, brake-by-wire, etc.). The FlexRay protocol serves as a communication infrastructure for future generation high-speed control applications in vehicles by providing the following services:

- **Message exchange service**—Provides deterministic cycle-based message transport.
- **Synchronization service**—Provides a common timebase to all nodes.
- **Start-up service**—Provides an autonomous start-up procedure.
- **Error management service**—Provides error handling and error signaling.
- **Symbol service**—Allows the realization of a redundant communication path.
- **Wakeup service**—Addresses power management needs.

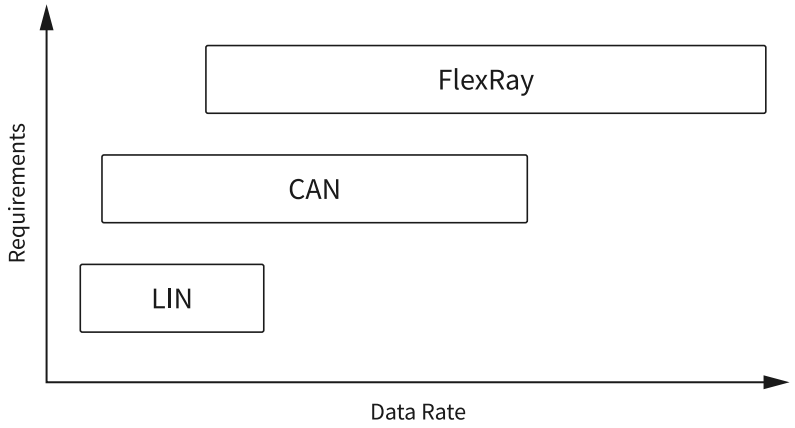
Increasing Communications Demands

In recent years, the amount of electronics introduced into automobiles has increased significantly. This trend is expected to continue as automobile manufacturers initiate further advances in safety, reliability, and comfort. The introduction of advanced control systems—combining multiple sensors, actuators, and electronic control units—is placing boundary demands on the existing Controller Area Network (CAN) communication bus.

Requirements for future in-car control applications include the combination of higher data rates, deterministic behavior, and the support of fault tolerance. For example, drive-by-wire, which replaces direct mechanical control of a vehicle with CPU-generated bus commands, demands high-speed bus systems that are fault tolerant,

are deterministic, and can support distributed control systems.

Increased functionality requires more flexibility in both bandwidth and system extension. Communications availability, reliability, and data bandwidth are the keys for targeted applications in power train, chassis, and body control.



Requirements Comparison

As shown in the figure above, the FlexRay bus addresses the significant increase in requirements for in-vehicle applications. As the amount of electronics in automobiles increases, high-bandwidth, deterministic, and redundant communications are available through the FlexRay communications bus.

FlexRay Bus Benefits

The FlexRay Communications System Specification Version 2.0 outlines many key bus network benefits:

- Provides up to 10 Mbits/s data rate on each channel, or a gross data rate up to 20 Mbits/s.
- Significantly increases Frame Length (compared to CAN—8 bytes per frame).
- Makes synchronous and asynchronous data transfer possible.
- Guarantees frame latency and jitter during synchronous transfer (real-time capabilities).
- Provides prioritization of messages during asynchronous transfer.
- Provides fault-tolerant clock synchronization via a global timebase.
- Gives error detection and signaling.
- Enables error containment on the physical layer through the use of an

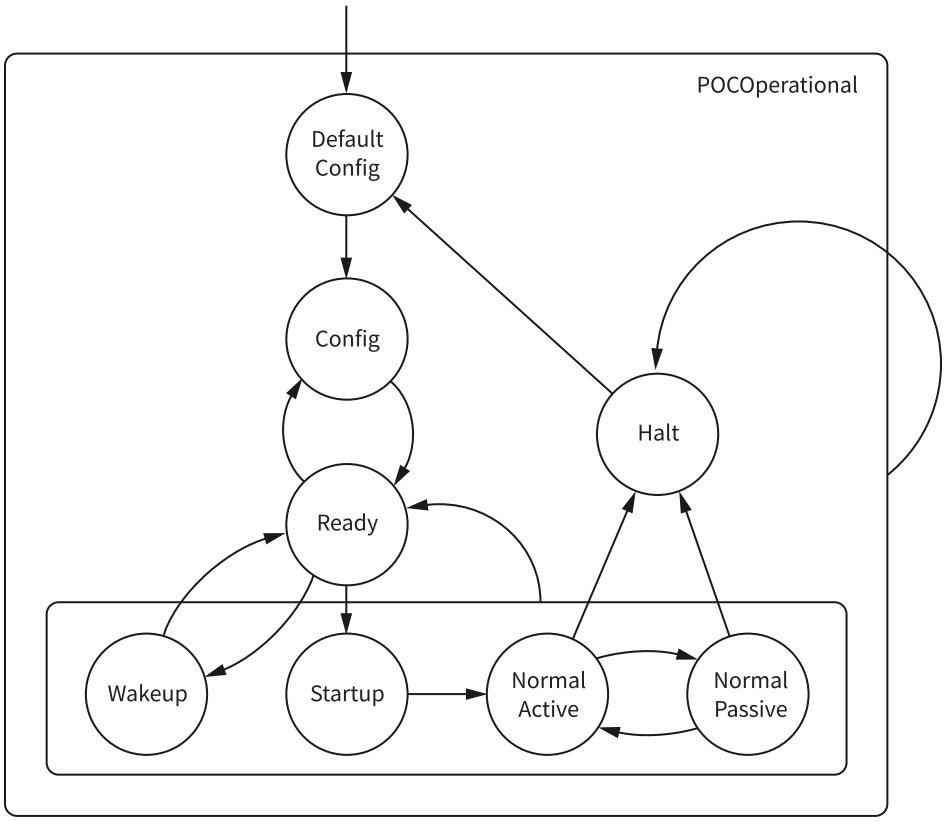
independent Bus Guardian mechanism.
- Provides scalable fault tolerance through single or dual-channel communication.

Data Security and Error Handling

The FlexRay network provides scalable fault tolerance by allowing single or dual-channel communication. For security-critical applications, the devices connected to the bus may use both channels for transferring data. However, you also can connect only one channel when redundancy is not needed, or to increase the bandwidth by using both channels for transferring non-redundant data.

Within the physical layer, FlexRay provides fast error detection and signaling, as well as error containment through an independent Bus Guardian. The Bus Guardian is a mechanism on the physical layer that protects a channel from interference caused by communication not aligned with the cluster communication schedule.

Protocol Operation Control



In the default config state, the controller is stopped. This is the power-on state.

In the config state, the controller is stopped. You can configure the controller in this state.

In the ready state, the controller can transition to the wakeup or startup states to perform a coldstart (startup of a bus) or integrate into a running cluster.

In the wakeup state, the controller can wake up nodes that are sleeping while the rest of the cluster is active.

The startup state is not a single state, but represents a state machine that is used for bus startup. The state machine has three different paths, depending on how the interface will participate in the startup process. The leading coldstart node is the interface that is initiating the schedule synchronization. The following coldstart node(s) are other coldstart-capable interfaces joining the leading coldstarter in starting up the FlexRay bus. The non-coldstart nodes connect to a currently running bus.

After properly integrating onto the bus, the controller transitions through the three operating states (Normal Active, Normal Passive, and Halt), which are similar to the CAN operating states of Error Active, Error Passive, and Bus Off.

When the interface is in Normal Active state, it is fully synchronized and supports cluster-wide clock synchronization.

When the interface is in Normal Passive state, it stops transmitting frames and symbols, but received frames are still processed. It still can perform clock synchronization based on received frames, but it does not contribute to the clock synchronization.

When the interface is in Halt state, all frame and symbol processing is stopped, as is macrotick generation.

Communication Cycle

The Communication Cycle is the fundamental element of the media-access scheme within FlexRay. A cycle duration is fixed when the network becomes configured. A FlexRay schedule has 64 cycles, numbered 0–63. After cycle 63, the schedule restarts at

cycle 0. The time window the Communication Cycle defines has two parts, a static segment and dynamic segment. The configuration also defines the segment lengths.

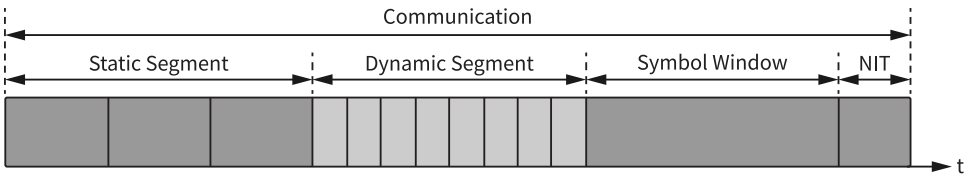
The Static Segment's purpose is to provide a time window for scheduling a number of time-triggered messages. This part of the Communication Cycle is reserved for the synchronous communication, which guarantees a specified frame latency and jitter through fault-tolerant clock synchronization. You must configure the messages to be transferred in the Static Segment before starting the communication, and the maximum amount of data transferred in the Static Segment cannot exceed the Static Segment duration. This provides for bus determinism, because each static slot is given a guaranteed time on the bus, and only one device may transfer data within a given slot.

In the Dynamic Segment, each device may transfer event-triggered messages, which its Frame ID prioritizes. This part of the cycle forms a communication scheme similar to the CAN bus. The Frame ID is for controlling the media access.

The Symbol Window is an optional part of the communication cycle where you can transmit a special symbol (Media Access Test Symbol (MTS)) on the network to test the Bus Guardian.

The Network Idle Time (NIT) is the part of the communication cycle where the node calculates and applies clock correction to maintain synchronization with the FlexRay bus.

The following figure shows the communication cycle of a given time period. The figure shows that the bandwidth used for time-triggered and event-triggered messages is scalable.



Startup

The action of initiating a startup process is called a coldstart. Only a subset of nodes, called coldstart nodes, may initiate a startup.

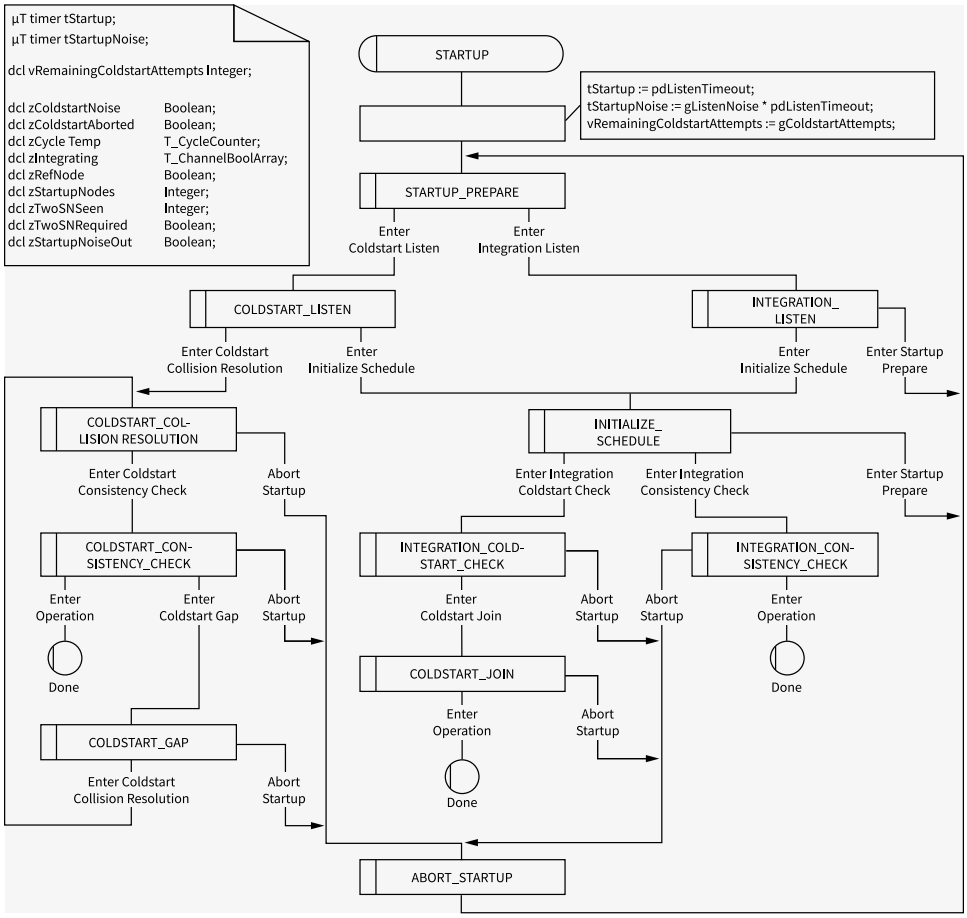
A coldstart attempt begins with the transmission of the collision avoidance symbol (CAS). Only the coldstart node that transmits the CAS can transmit frames in the four cycles that follow the CAS. During the fifth cycle, other coldstart nodes can join it; later on, all other nodes can join it also.

In each cluster consisting of at least three nodes, at least three nodes must be configured as coldstart nodes. If a cluster has only two nodes, both of them must be configured as coldstart nodes.

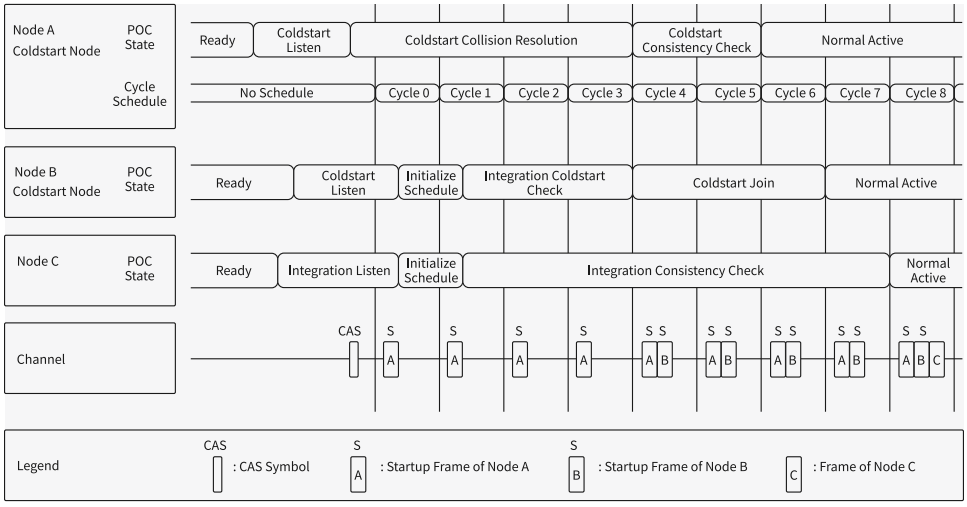
The coldstart node that transmits the CAS is called a leading coldstart node. The other coldstart nodes are called following coldstart nodes.

During the startup process, a node can transmit only startup frames. A startup frame has an indicator in the header segment (refer to Frame Format) that indicates it is a startup frame. All startup frames are also sync frames, which contain an indicator that nodes use to assist with clock correction.

The following diagram shows the startup state machine as the ***FlexRay Protocol Specification v. 2.1*** defines it.



The following diagram shows the state transitions for a leading coldstart node (Node A), following coldstart node (Node B), and non-coldstart node (Node C).



Path of the Leading Coldstart Node

When a coldstart node enters startup, it listens to the FlexRay bus to make sure the bus

is idle before commencing a coldstart attempt. If no communication is detected, the node transmits a CAS symbol followed by the first regular cycle, numbered cycle zero. From cycle zero onward, the node transmits its startup frame. During this time, only one node (the leading coldstart node) can transmit startup frames. If two nodes happen to transmit the CAS at the same time, both would transmit startup nodes during this time, and both would detect the error and restart the coldstart process.

Starting in cycle four, other coldstart nodes begin to transmit their startup frames. The leading coldstart node collects startup frames in cycles four and five and performs clock correction. If there are no errors, the node leaves startup and enters normal active.

Path of a Following Coldstart Node

When a coldstart node enters startup, it listens to the FlexRay bus to make sure the bus is idle before commencing a coldstart attempt. If communication is detected, the node tries to receive a valid pair of startup frames to derive its schedule and calculate its initial clock correction.

After successfully receiving these frames, it collects all sync frames during the following two cycles and performs clock correction. If there are no errors during the clock correction, the node begins to transmit its own startup frames.

If there still are no errors after three cycles of transmitting startup frames, the node leaves startup and enters normal active.

Path of a Non-Coldstart Node

When a non-coldstart node enters startup, it listens to the FlexRay bus and tries to receive FlexRay frames. If communication is detected, the node tries to receive a valid pair of startup frames to derive its schedule and clock correction from the coldstart nodes.

In the following two cycles, the node receives startup frames. After receiving valid startup frames during four consecutive cycles from at least two different coldstart nodes, the node leaves startup and enters normal active.

Related concepts:

- Frame Format

Clock Synchronization

FlexRay is a time-triggered bus, requiring every node in the cluster to have approximately the same view of time. Time in FlexRay is based on cycles, macroticks, and microticks. A cycle is composed of an integer number of macroticks, and a macrotick is composed of an integer number of microticks.

A cycle consists of an integer number of macroticks, which must be identical for all nodes in the cluster. This value remains the same for each cycle. The duration of a macrotick also is identical (within tolerances) for all nodes in a cluster. However, each node derives the macrotick from its microtick, which is derived from a local oscillator. The number of microticks per macrotick may differ for each node on the cluster (because they may use different local oscillators). In addition, the number of microticks per macrotick may differ from macrotick to macrotick within the same node, if required.

Clock synchronization is required to ensure that the time differences between the nodes of a cluster remain consistent. There are two types of time differences—phase (offset) differences and frequency (rate) differences. FlexRay nodes perform both offset and rate correction to remain synchronized.

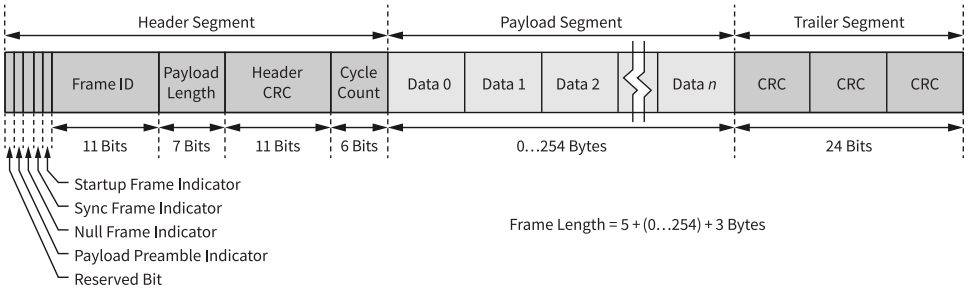
Rate correction is performed during the entire cycle. A positive or negative integer number of microticks are added to the configured number of microticks in a communication cycle. The actual number is determined by a clock synchronization algorithm computed after the static segment of every odd cycle.

Offset correction is performed only during the NIT of every odd cycle. A positive or negative integer number of microticks are added during the NIT offset correction segment. The actual number is determined by a clock synchronization algorithm computed during every cycle (but as mentioned above, the correction actually is performed only during odd cycles).

Frame Format

The following figure shows the FlexRay frame format. The FlexRay frame has three

segments: header, payload, and trailer.



- **Header**—Includes the Frame ID, Payload Length, Header CRC, and Cycle Count. The Frame ID identifies a frame and is for prioritizing event-triggered frames. The Payload Length contains the number of words transferred in the frame. The Header CRC is for detecting errors during the transfer. The Cycle Count contains the value of a counter that advances incrementally each time a Communication Cycle starts. Additionally, the header includes some indicators to help identify the frame type. The Payload Preamble indicator indicates whether an optional vector is contained within the payload segment of the transmitted frame (for example, a network management vector). The Null Frame indicator indicates whether the frame is a normal or null frame (a frame that does not contain a valid payload). The Sync Frame indicator indicates whether the frame is a special sync frame used for clock synchronization. Finally, the Startup Frame indicator indicates whether the frame is a startup frame to help start the FlexRay cluster.
- **Payload**—Contains the data the frame transfers. The FlexRay payload or data frame length is up to 127 words (254 bytes), which is more than 30 times greater than CAN.
- **Trailer**—Contains three 8-bit CRCs to detect errors.

Related concepts:

- Startup

LIN Overview

Local Interconnect Network (LIN) was developed to create a standard for low-cost, low-end multiplex communication in automotive networks. While Controller Area Network (CAN) addresses the need for high-bandwidth, advanced error-handling networks, LIN provides cost-efficient communications in applications where the

bandwidth and versatility of CAN are not required, such as power window and power seat controllers.

LIN can be implemented relatively inexpensively using the standard serial UART embedded in most modern low-cost 8-bit microcontrollers.

The LIN bus connects a single master device (node) and one or more slave devices (nodes) together in a LIN cluster. A node capability file describes the behavior of each node. The node capability files are inputs to a system defining tool, which generates a LIN description file (LDF) that describes the behavior of the entire cluster. You can parse the LDF to generate the specified behavior in the desired nodes. At this point, the master device's master task starts transmitting headers on the bus, and all the slave tasks in the cluster (including the master devices's own slave task) respond, as specified in the LDF.

In general terms, you use the LDF to configure and create the LIN cluster's scheduling behavior. For example, it defines the cluster's baud rate, the ordering and time delays for the master task's transmission of headers, and the behavior of each slave task in response.

LIN Topology and Behavior

The LIN bus connects a single master device (node) and one or more slave devices (nodes) together in a LIN cluster. A node capability file describes the behavior of each node. The node capability files are inputs to a system defining tool, which generates a LIN description file (LDF) that describes the behavior of the entire cluster. You can parse the LDF to generate the specified behavior in the desired nodes. At this point, the master device's master task starts transmitting headers on the bus, and all the slave tasks in the cluster (including the master device's own slave task) respond, as specified in the LDF.

In general terms, you use the LDF to configure and create the LIN cluster's scheduling behavior. For example, it defines the cluster's baud rate, the ordering and time delays for the master task's transmission of headers, and the behavior of each slave task in response.

LIN Frame Format

LIN is a polled bus with a single master node and one or more slave nodes. The master node contains both a master task and a slave task. Each slave node contains only a slave task. The master task in the master node controls all communication over LIN.

The basic unit of transfer on the LIN bus is the frame, which is divided into a header and a response. The master node always transmits the header, which consists of three distinct fields: the Break , the Synchronization Field (Sync), and the Identifier Field (ID). A slave task (which can reside in either the master node or a slave node) always transmits the response; a response consists of a data payload and a checksum.

Normally, the master task runs a predefined schedule, which describes the headers to transmit on the bus, in a continuously repeating loop. Prior to starting the LIN, each slave task is configured either to publish data to the bus or subscribe to data in response to each received header ID. On receiving the header, each slave task verifies ID parity and then checks the ID to determine whether it needs to publish or subscribe during the response portion of the frame. If the slave task needs to publish a response, it transmits one to eight data bytes to the bus, followed by a checksum byte. If the slave task needs to subscribe, it reads the data payload and checksum byte from the bus and takes appropriate internal action. For standard slave-to-master communication, the master broadcasts the identifier to the network, and one and only one slave responds with a data payload.

A separate slave task that exists in the master node accomplishes master-to-slave communication. This task self-receives all headers transmitted on the bus and responds as if it were an independent slave. To transmit data bytes, the master first must update its internal slave task's response with the data values it wants to transmit. The master then transmits the appropriate header, and the internal slave task transmits its response to the bus.

Break

Every LIN frame begins with the Break, comprised of at least 13 dominant bits followed by a break delimiter of at least one recessive bit. This serves as a start-of-frame notice to all nodes on the bus.

Sync

The Sync field is the second field that the master task transmits in the header. Sync is

defined as the character x55. The Sync field allows slave nodes that perform automatic baud rate detection to measure the baud rate period and adjust their internal baud rate to synchronize with the bus.

ID

The ID field is the final field in the header transmitted by the master task. This field provides identification for each message on the network and ultimately determines which devices in the network receive or respond to each transmission. All slave tasks continually listen for Identifier Fields, verify their parity, and determine whether they are publishers or subscribers for this particular identifier. LIN provides 64 IDs. IDs 0–59 (0x3B) are for signal-carrying (data) frames, 60 (0x3C) and 61 (0x3D) carry diagnostic data, and 62 (0x3E) and 63 (0x3F) are reserved for future protocol enhancements. The ID is protected, as it is transmitted over the bus by performing a 2-bit parity calculation on the 6-bit ID and combining the parity and the ID into a single byte called the protected ID. This protected ID has the lower 6 bits containing the raw ID and the upper two bits containing the parity.

The following figure shows how parity is calculated using the raw ID and how the protected ID is formed from the combination of the parity bits and raw ID.

Protected ID(7:6)		Protected ID(5:0)
P(1)	P(0)	Raw ID(5:0)
$\neg$ (ID(1) $\oplus$ ID(3) $\oplus$ ID(4) $\oplus$ ID(5))	ID(0) $\oplus$ ID(1) $\oplus$ ID(2) $\oplus$ ID(4)	0–63

Data Payload

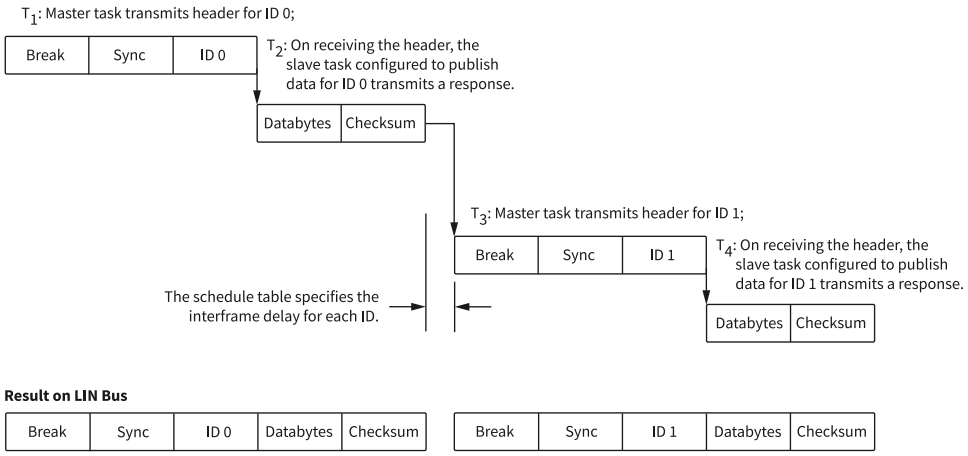
The slave task transmits the Data Payload field in the response. This field contains one to eight bytes of data.

Checksum

The slave task transmits the Checksum field as the last byte in the response. The message portion included in the checksum can differ based on the checksum mode in use. The classic checksum is calculated using the data bytes. The enhanced checksum is calculated using the data bytes and protected ID.

The LIN 2.1 specification defines the checksum calculation process as the summing of all values, subtracting 255 every time the sum is greater than or equal to 256, then inverting the result. Per the LIN 2.1 specification, classic checksum is for use with LIN 1. x slave devices and enhanced checksum with LIN 2. x slave devices. It further specifies that IDs 60–61 always use classic checksum. NI-XNET uses the checksum configuration obtained from the database to determine which checksum algorithm to use for a particular frame. Per the LIN 2.1 specification, IDs 60–61 always use classic checksum, regardless of the setting of the checksum attribute.

The following figure shows how a master task header and slave task response combine to create a LIN full frame.



LIN Bus Timing

A nominal time for a LIN frame to be transmitted across the bus is the number of bits multiplied by the time for each bit. Because different entities transmit the two LIN frame fields, the timing breaks down into the time for the header to be transmitted and the time for the response to be transmitted, as shown below.

T_{Bit} [s] = Time it takes to transmit 1 bit (1/Baud_Rate)

N_{Data} = Number of data bytes in response

$T_{Header\_Nominal}$ [s] = 34 * T_{Bit}

$T_{Response\_Nominal}$ [s] = 10 * (N_{Data} + 1) * T_{Bit}

$T_{Frame\_Nominal}\ [s] = T_{Header\_Nominal} + T_{Response\_Nominal}$

However, to allow for byte processing and other delays within a device, each segment is allocated an additional 40 percent as compared to the nominal time for the frame to transmit.

$T_{Header\_Maximum}\ [s] = 1.4 * T_{Header\_Nominal}$

$T_{Response\_Maximum}\ [s] = 1.4 * T_{Response\_Nominal}$

$T_{Frame\_Maximum}\ [s] = T_{Header\_Maximum} + T_{Response\_Maximum}$

LIN Error Detection and Confinement

The LIN 2.1 specification specifies that slave tasks should handle error detection and that error monitoring by the master task is not required. The LIN 2.1 specification does not require handling of multiple errors within one LIN frame or the use of error counters. On encountering the first error in a frame, the slave task aborts processing of the frame until detection of the next Break-Sync sequence (in the next header the master transmits). With NI-XNET, you can determine whether any of these errors have occurred by checking the Last Error Code (LEC) field by reading XNET Read (State LIN Comm).vi.

LIN also provides a mechanism for slave nodes to report errors to the master node. The LIN 2.1 specification defines a 1-bit scalar signal named response_error , which each slave publishes to the master in one of its unconditional frames. This bit is set whenever a frame that a slave node receives or transmits (except for an event-triggered response) contains an error in the response field. The bit is cleared after the frame containing the signal is successfully published to the master.

LIN Sleep and Wakeup

LIN provides a mechanism for devices to enter sleep state and potentially conserve power. Per the LIN 2.1 specification, the master may force all slaves into sleep mode by sending a diagnostic master request frame (ID=60, 0x3C) with the first data byte equal to 0 and the remaining bytes set to 0xFF. This special frame is called the go-to-sleep

command. Slaves also enter sleep mode automatically if LIN is inactive for more than 4 seconds.

LIN also provides a mechanism for waking devices on the bus. Wakeup is one task that any node on the bus (a slave as well as the master) may initiate. Per the LIN 2.1 specification, force the bus dominant for 250 µs to 5 ms to issue the wakeup request. Each slave should detect the wakeup request and be ready to process headers within 100 ms. The master also should detect the wakeup request and start sending headers when the slave nodes are ready (within 100–150 ms after receiving the wakeup request). If the master does not issue headers within 150 ms after receiving the first wakeup request, the slave requesting wakeup may try issuing a second wakeup request (and waiting for another 150 ms). If the master still does not respond, the slave may issue the wakeup request and wait 150 ms a third time. If there still is no response, the slave must wait for 1.5 seconds before issuing a fourth wakeup request.

The master may wake up the bus just by starting to send a normal break. However, if this happens, the slaves may not be awake, and the slave nodes may not process the first header transmitted.

Advanced Frame Types

The LIN 2.1 specification classifies LIN frames into five types: ***unconditional***, ***event triggered***, ***sporadic***, ***diagnostic***, and ***reserved***. It is important to note that the differences in these frame types are due to either the timing of how they are transmitted or the data bytes' content. Regardless of frame classification, a LIN frame always consists of a header that the master task transmits and a response that a slave task transmits.

The unconditional frame type is most commonly used. Unconditional frames carry signals (data), and their identifiers are 0–59 (0x3B). Whenever the publisher of an unconditional frame receives the header, it always transmits a response.

The event-triggered frame type attempts to conserve bus bandwidth by requesting an unconditional frame response from multiple slaves within one frame slot time. The event-triggered frame may have an ID of 0–59 (0x3B). When an unconditional frame is used as an event frame, the bytes of data are restricted to 1–7 bytes instead of 1–8 bytes. This is because the first data byte must be loaded with the protected ID of the

slave's unconditional frame.

The event-triggered frame works as follows: The master writes an event-triggered ID in a header. The slaves respond to the event-triggered ID only if their data has been updated. If only one slave publishes a response, the master receives it and looks at the first data byte, which indicates which slave (through the protected ID) published the response. If multiple slaves publish a response, a collision occurs. When the master detects this collision, it invokes a new schedule to resolve the collision. This collision resolving schedule queries each unconditional frame associated with the event-triggered frame to get the responses from all objects. Afterward, the original schedule is continued.

Sporadic frames attempt to provide some dynamic behavior to LIN. Sporadic frames always carry signals (data), and their IDs are 0–59 (0x3B). Only the slave task associated with the master node can send sporadic frames. The header of a sporadic frame is sent in its frame slot only when the master task knows that a data value (signal) within the frame has been updated. If multiple unconditional frames associated with a sporadic slot have updated data, the master transmits only the highest priority frame, which the order that the frames appear in the sporadic frame list determines.

Diagnostic frames are always eight data bytes in length and always carry diagnostic or configuration data. Their ID is either 60 (0x3C) for a master request frame or 61 (0x3D) for a slave response frame.

Reserved frames have an ID of 62 (0x3E) and 63 (0x3F). You must not use them in a LIN 2. x cluster.

Automotive Ethernet Overview

Automotive Ethernet refers to Ethernet-based communication used as an in-vehicle networking technology, especially communication among electronic control units (ECUs) of a vehicle. Benefits to using Automotive Ethernet include faster data communication for in-vehicle networking; cost-saving, lighter weight cabling; and software interfaces for upper layers of the Ethernet stack that are the same as those utilized for standard Ethernet.

The Ethernet protocol enables an open technology, high bandwidth network for in-vehicle communication and improves the ability to share data from a common source to an entire network. One critical element of Ethernet is the Ethernet frame (also known as an Ethernet packet), which includes such data as destination MAC address, source MAC address, 802.1Q header (optional), EtherType or length of frame, payload, and a Cyclic Redundancy Check (CRC) called the Frame Check Sequence (FCS). A minimum inter-frame gap of 12 bytes must follow the termination of the Ethernet frame or packet.

IEEE 802 is a family of IEEE standards related to the Data Link and Physical (PHY) layers of the Open Systems Interconnection (OSI) network communication model. The IEEE 802 standards are specific to networks that transport variable-size packets. Within this family are the IEEE 802.1 standards that provide specifications for local area network (LAN), metropolitan area network (MAN), and bridging architecture and network management. Also in this family are the IEEE 802.3 standards, which define the physical layer (PHY) and the media access control (MAC) of the data link layer of wired Ethernet.

The Ethernet standard as it applies to in-vehicle networking is especially influenced by the following specifications:

- **IEEE 802.1AS-2011**: Specifies the protocol and procedures used to ensure time synchronization for time sensitive applications, such as audio and video, over a virtual bridged local area network.
- **IEEE 802.1Q-2018**: Specifies how MAC service is supported by bridged networks, principles of operation, and the operation of MAC and VLAN bridges, including management, protocols, and algorithms.
- **IEEE 802.3bw**: 100BASE-T1 Physical Layer (PHY) specifications and management parameters for full duplex 100 Mb/s communication over single twisted pair cabling.
- **IEEE 802.3bp**: 1000BASE-T1 Physical Layer (PHY) specifications and management parameters for full duplex 1 Gb/s communication over single twisted pair cabling.
- **IEEE 1722-2016**: Transport protocol standard for time-sensitive applications on bridged local area networks. This standard enables interoperable audio and video streaming by defining raw and compressed audio/video formats, synchronization mechanisms, and address assignments.

PDF versions of the IEEE 802 standards are available on the Institute of Electrical and

Electronics Engineers (IEEE) website, ieee.org.

Hardware Overview

NI-XNET CAN, FlexRay, LIN, and Ethernet interfaces are optimized for applications requiring real-time, high-speed manipulation of hundreds of CAN frames and signals, such as hardware-in-the-loop simulation, rapid control prototyping, bus monitoring, and automation control.

NI-XNET CAN Hardware

NI-XNET CAN devices interface the CAN protocol controller to the physical bus wires using one of the following physical layers:

- High-Speed CAN
- Low-Speed/Fault Tolerant CAN
- Single Wire CAN

The XS Software Selectable Physical Layer enables you to select each port individually in the physical layer for High-Speed or Low-Speed/Fault Tolerant transceivers.

Topics in this section describe the following characteristics of NI-XNET CAN hardware:

- Transceiver
- Bus Power Requirements
- Cabling Requirements
- Termination

High-Speed CAN Physical Layer

The High-Speed CAN physical layer circuitry interfaces the CAN protocol controller to the physical bus wires.

High-Speed CAN Transceiver

NI-XNET CAN High-Speed hardware uses either the NXP TJA1041 or NXP TJA1043 High-Speed CAN transceiver.

The NI-XNET CAN HS/FD Transceiver Cable uses the TJA1043 transceiver. All PXI and PCI NI-XNET High-Speed CAN interfaces Revision F or earlier use the TJA1041. All PXI and PCI NI-XNET High-Speed CAN interfaces Revision G or later use the TJA1043. All USB NI-XNET High-Speed CAN interfaces use the TJA1043.

Both the TJA1041 and TJA1043 are fully compatible with the ISO 11898 standard and support baud rates of 40 kbps to 1 Mbps. These devices also support advanced power management through a low-power sleep mode. Refer to the NI-XNET Session Interface:CAN:Transceiver State property for more information. For detailed transceiver specifications, refer to the NXP TJA1041 or NXP TJA1043 product data sheet.

CAN High Speed Bus Power Requirements

The High-Speed physical layer on PXI, PCI, USB, and Transceiver Cable NI-XNET interfaces is internally powered. As such, there is no need to supply bus power. The COM pin serves as the reference ground for the bus signals.

The High-Speed physical layer on C Series NI 9862 requires external power supply of +9 V to +30 V to operate. Connect the external power supply to the Vsup pin on the module. The COM pins are for reference ground.

Cabling Requirements for High-Speed CAN

Cables should meet the physical medium requirements specified in ISO 11898, shown in the following table.

Belden cable (3084A) meets all these requirements and should be suitable for most applications.

Table 4. ISO 11898 Specifications for Characteristics of a CAN_H and CAN_L Pair of Wires

Characteristic	Value
Impedance	108 Ω minimum, 120 Ω nominal, 132 Ω maximum

Characteristic	Value
Length-related resistance	70 mΩ/m nominal
Specific line delay	5 ns/m nominal

Related reference:

- Connect the Cables

Cable Lengths

The cabling characteristics and desired bit transmission rate affect the allowable cable length. Detailed cable length recommendations are in the ISO 11898 and CiA DS 102 specifications. ISO 11898 specifies 40 m total cable length with a maximum stub length of 0.3 m for a bit rate of 1 Mbps. The ISO 11898 specification says that significantly longer cable lengths may be allowed at lower bit rates, but each node should be analyzed for signal integrity problems.

Number of Devices

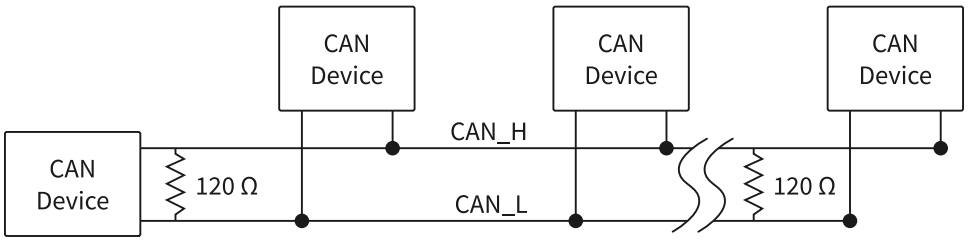
The maximum number of devices depends on the electrical characteristics of the devices on the network. If all devices meet the requirements of ISO 11898, you can connect at least 30 devices to the bus. You can connect higher numbers of devices if the device electrical characteristics do not degrade signal quality below ISO 11898 signal level specifications. The NI-XNET CAN hardware electrical characteristics allow at least 110 CAN ports on the network.

Cable Termination

The pair of signal wires (CAN_H and CAN_L) constitutes a transmission line. If the transmission line is not terminated, each signal change on the line causes reflections that may cause communication failures.

Because communication flows both ways on the CAN bus, CAN requires that both ends of the cable be terminated. However, this requirement does not mean that every device should have a termination resistor. If multiple devices are placed along the cable, only the devices on the ends of the cable should have termination resistors.

Refer to the following figure for an example of where termination resistors should be placed in a system with more than two devices.



Termination Resistor Placement

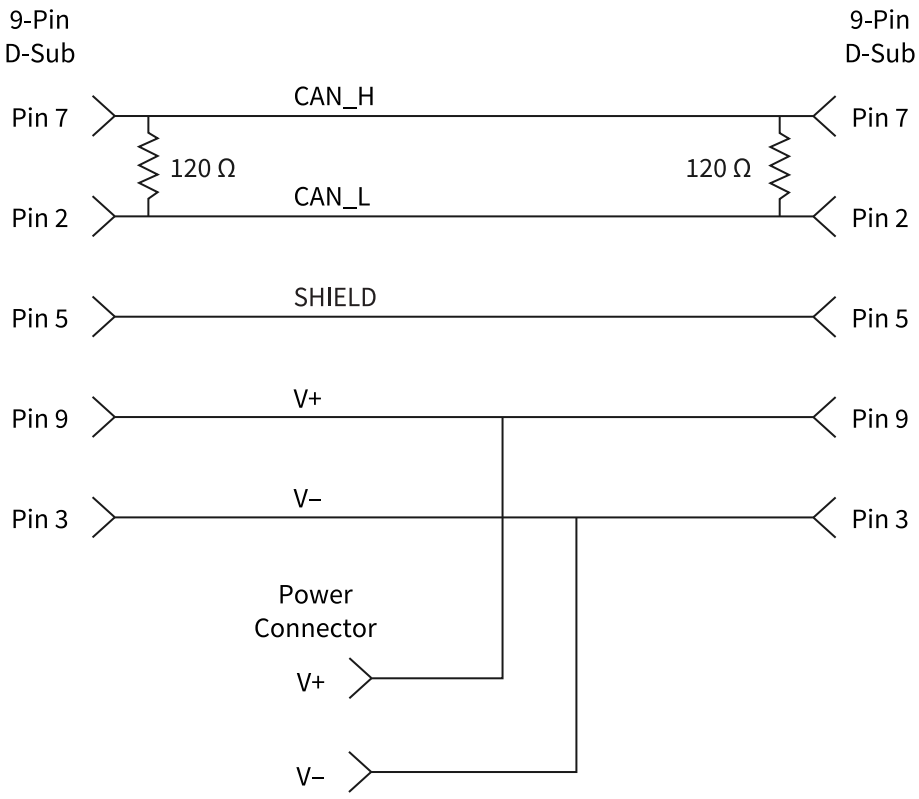
The termination resistors on a cable should match the nominal impedance of the cable. ISO 11898 requires a cable with a nominal impedance of 120 Ω, so you should use a 120 Ω resistor at each end of the cable. Each termination resistor should be capable of dissipating 0.25 W of power.

NI-XNET devices feature software selectable bus termination for High-Speed CAN transceivers. On the USB-8502, PXI-8512, PCI-8512, PXI-8513 (in high-speed mode), PCI-8513 (in high-speed mode), USB-8502, and on CAN HS/FD and CAN XS Transceiver Cables, you can enable 120 Ω termination resistors between CAN_H and CAN_L through an API call.

Refer to the NI-XNET Session Interface:CAN:Termination property in the API reference for more information.

Cabling Example

Cable Connecting Two CAN Devices



Low-Speed/Fault-Tolerant CAN Physical Layer

The Low-Speed/Fault-Tolerant CAN physical layer circuitry interfaces the CAN protocol controller to the physical bus wires.

Low-Speed/Fault Tolerant CAN Transceiver

NI-XNET CAN Low-Speed/Fault-Tolerant hardware uses either the NXP TJA1054A or NXP TJA1055T Low-Speed/Fault-Tolerant transceiver.

NI PXI and PCI XNET interfaces revision E and higher use the TJA1055T transceiver, while revision D and lower use the TJA1054A transceiver.

To identify your PCI/PXI NI-XNET hardware revision, refer to the ***19xxxx<rev>-4xL*** text on the green label in the top left corner on the secondary side of the board; ***<rev>*** indicates the hardware revision.

Both the TJA1054A and TJA1055T support baud rates up to 125 kbps. The transceiver can detect and automatically recover from the following CAN bus failures:

- CAN_H wire interrupted
- CAN_L wire interrupted
- CAN_H short-circuited to battery
- CAN_L short-circuited to battery
- CAN_H short-circuited to VCC
- CAN_L short-circuited to VCC
- CAN_H short-circuited to ground
- CAN_L short-circuited to ground
- CAN_H and CAN_L mutually short-circuited

The TJA1054A and TJA1055T support advanced power management through a low-power sleep mode. Refer to the NI-XNET Session Interface:CAN:Transceiver State property for more information. For detailed specifications for the transceivers, refer to the NXP TJA1054 and NXP TJA1055T product data sheets.

CAN Low Speed/Fault Tolerant Bus Power Requirements

The Low-Speed/Fault-Tolerant physical layer on PXI and PCI NI-XNET interfaces is internally powered. As such, there is no need to supply bus power. The COM pin serves as the reference ground for the bus signals.

The Low-Speed/Fault-Tolerant physical layers on the C Series NI 9861 and the TRC-8543 Transceiver Cable require an external power supply of +9 V to +30 V to operate. Connect the external power supply to the VSUP pin on the module. The COM pins are for reference ground.

Cabling Requirements for Low-Speed/Fault-Tolerant CAN

Cables should meet the physical medium requirements shown in the following table. Belden cable (3084A) meets all of those requirements and should be suitable for most applications.

Table 5. Specifications for Characteristics of a CAN_H and CAN_L Pair of Wires

Characteristic	Value
Length-related resistance	90 mΩ/m nominal
Length-related capacitance: CAN_L and ground, CAN_H and ground, CAN_L and CAN_H	30 pF/m nominal

Related concepts:

- Low-Speed CAN

Related reference:

- Connect the Cables

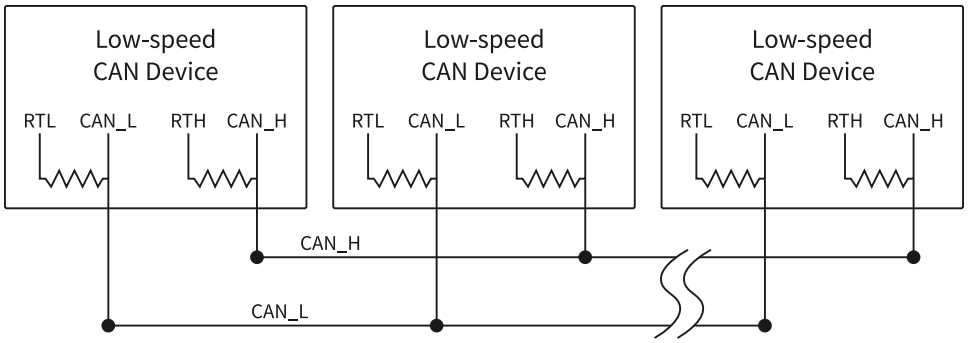
Number of Devices (LS/FT CAN)

The maximum number of devices depends on the electrical characteristics of the devices on the network. If all devices meet the requirements of typical Low-Speed/Fault-Tolerant CAN, you can connect up to 32 devices to the bus. You can connect higher numbers of devices if the electrical characteristics of the devices do not degrade signal quality below Low-Speed/Fault-Tolerant signal level specifications.

Low-Speed CAN Termination

Every device on the Low-Speed CAN network requires a termination resistor for each CAN data line: RRTH for CAN_H and RRTL for CAN_L.

The following figure shows termination resistor placement in a Low-Speed CAN network.



Termination Resistor Placement for Low-Speed CAN

The Determining the Necessary Termination Resistance for the Board section explains how to determine the correct termination resistor values for the Low-Speed CAN transceiver.

Refer to the NI-XNET Session Interface:CAN:Termination property for more information.

Related concepts:

- NI-XNET FlexRay Hardware
- NI-XNET LIN Hardware
- Determining the Necessary Termination Resistance for the Board

Determining the Necessary Termination Resistance for the Board

Unlike High-Speed CAN, Low-Speed CAN requires termination at the Low-Speed CAN transceiver instead of on the cable. The termination requires two resistors: RTH for AN_H and RTL for CAN_L. This configuration allows the NXP fault-tolerant CAN transceiver to detect and recover from bus faults. You can use the NI-XNET Low-Speed/Fault-Tolerant CAN hardware to connect to a Low-Speed CAN network having from two to 32 nodes as specified by NXP (including the port on the CAN Low-Speed/Fault-Tolerant interface). You also can use the Low-Speed/Fault-Tolerant interface to communicate with individual Low-Speed CAN devices. It is important to determine the overall termination of the existing network, or the individual device termination, before connecting it to a Low-Speed/Fault-Tolerant port.

NXP recommends an overall RTH and RTL termination of 100 Ω to 500 Ω (each) for a properly terminated low-speed network. You can determine the overall network

termination as follows:

$$\frac{1}{R_{\text{RTH overall}}} = \frac{1}{R_{\text{RTH node 1}}} + \frac{1}{R_{\text{RTH node 2}}} + \frac{1}{R_{\text{RTH node 3}}} + \frac{1}{R_{\text{RTH node n}}}$$

NXP also recommends an individual device RTH and RTL termination of 500 Ω to –16 KΩ. After determining the existing network or device termination, you can use the following formula to indicate which nearest value the termination property needs to be set to produce the proper overall RTH and RTL termination of 100 Ω to 500 Ω upon connection of the card:

$$R_{\text{RTH overall}} = \frac{1}{\left(\frac{1}{R_{\text{RTH of low-speed CAN interface}}} + \frac{1}{R_{\text{RTH of existing network or device}}}\right)}$$

where R_{RTH} overall should be 100 Ω to 500 Ω.

NI-XNET Low-Speed/Fault-Tolerant CAN hardware features software selectable bus termination resistors, allowing you to adjust the overall network termination through an API call. In general, if the existing network has an overall network termination of 125 Ω or less, you should select the 5 KΩ option for your NI-XNET device. For existing overall network termination above 125 Ω, you should select the 1 KΩ termination option for your NI-XNET device.

Related concepts:

- Low-Speed CAN Termination

Single Wire CAN Physical Layer

The Single Wire CAN physical layer circuitry interfaces the CAN protocol controller to the physical bus wires.

Related concepts:

- Single Wire CAN

CAN Transceiver for Single Wire Hardware

NI-XNET Single Wire hardware uses either the NXP AU5790 or ON Semiconductor NCV7356 Single Wire CAN transceiver.

NI PCI-8513 and NI PCI-8513/2 software-selectable NI-XNET PCI CAN interfaces (revision D and higher) use the ON Semiconductor NCV7356 Single Wire transceiver, while revision C (and lower) uses the NXP AU5790 Single Wire transceiver.

NI PXI-8513 and NI PXI-8513/2 software-selectable NI-XNET PXI CAN interfaces (revision E and higher) use the ON Semiconductor NCV7356 Single Wire transceiver, while revision D (and lower) uses the NXP AU5790 Single Wire transceiver.

To identify the your PCI/PXI NI-XNET hardware revision, refer to the 19xxxx<rev>–4xL text on the green label in the top left corner on the secondary side of the board; <rev> indicates the hardware revision.

The NI-XNET Single Wire hardware supports baud rates up to 33.3 kbps in normal transmission mode and 83.3 kbps in High-Speed transmission mode. The achievable baud rate is primarily a function of the network characteristics (termination and number of nodes on the bus), and assumes bus loading as per SAE J2411. Each Single Wire CAN port has a local bus load resistance of 9.09 kΩ between the CAN_H and RTH pins of the transceiver to provide protection against the loss of ground. NI-XNET Single Wire hardware also supports advanced power management through low-power sleep and wake up modes. Refer to the NI-XNET Session Interface:CAN:Transceiver State property for more information.

For detailed transceiver specifications, refer to their respective data sheets.

Related concepts:

- NI-XNET FlexRay Hardware
- NI-XNET LIN Hardware

CAN Single Wire Bus Power Requirements

The Single Wire physical layer on PXI and PCI NI-XNET interfaces is internally powered; supplying bus power is optional. Supplying power externally requires a power supply

of +8 V to +18 V (+12 V recommended) to operate. Connect the external power supply to the Ext_Vbat pin on the module. The COM pin serves as reference ground for the bus signals.

Related concepts:

- NI-XNET FlexRay Hardware
- NI-XNET LIN Hardware

Cabling Requirements for Single Wire CAN

The number of nodes on the network, total system cable length, bus loading of each node, and clock tolerance are all interrelated. It is therefore the system designer's responsibility to factor in all the above parameters when designing a Single Wire CAN network. The SAE J2411 standard includes some recommended specifications that can help in making these decisions.

Related reference:

- Connect the Cables

Cable Length

There can be no more than 60 m between any two ECU nodes.

Number of Devices

The maximum number of Single Wire CAN nodes allowed on the network depends on the device and cable electrical characteristics. If all devices and cables meet the requirements of J2411, between 2 and 32 devices may be networked together.

Termination (Bus Loading)

All NI Single Wire CAN hardware includes a built-in 9.09 kΩ load resistor, as specified by J2411.

XS Software Selectable Physical Layer

XNET CAN XS hardware allows you to select each port individually in the physical layer for one of the following transceivers:

- High-Speed
- Low-Speed/Fault-Tolerant
- Single Wire
- External Transceiver

When an XS port is selected as High-Speed, it behaves exactly as a dedicated High-Speed interface. When an XS port is selected as Low-Speed/Fault-Tolerant, it behaves exactly as a dedicated Low-Speed/Fault-Tolerant interface. When an XS port is selected as Single Wire, it behaves exactly as a dedicated Single Wire interface. The bus power requirements depend on the mode selected. Refer to the appropriate High-Speed, Low-Speed/Fault-Tolerant, or Single Wire physical layer section to determine the behavior for the mode selected. For example, the bus power requirements for an XS port configured for Single Wire mode are identical to those of a dedicated Single Wire node. This feature is provided as the Interface:CAN:Transceiver Type property.

When an XS port is selected as External, all onboard transceivers are bypassed, and the CAN controller signals are routed directly to the 9-pin D-SUB connector. External mode is intended for interfacing custom physical layer circuits to NI XNET CAN hardware. Refer to External CAN Transceiver for more details.

Related concepts:

- External CAN Transceiver

External CAN Transceiver

The external CAN transceiver mode on the PXI-8513 and PCI-8513 XS software selectable interfaces allows you to connect custom CAN transceivers to the NI-XNET CAN hardware. The DB-9 connector on the PXI-8513 and PCI-8513 interfaces includes five different pins to connect with the custom transceiver. Refer to Pinouts for the DB-9 pinout for external CAN transceiver. Refer to Interface:CAN:External Transceiver Config for more information about configuring the NI-XNET hardware to communicate with the custom transceiver.

Related concepts:

- XS Software Selectable Physical Layer

NI-XNET Transceiver Cables

NI-XNET transceiver cables are designed to provide flexibility in connecting a CAN or LIN bus to multiprotocol interfaces such as the NI 9860, PCIe-8510, and PXIe-8510, or to the native NI-XNET port on integrated controllers such as the cDAQ-9134/9135. The isolated transceiver cable implements the physical layer of the interface.

Related concepts:

- Isolation

CAN Interface Pinouts

PCI and PXI CAN Interface

The following table describes the CAN DB-9 pinout on PCI and PXI CAN interfaces, such as PCI-8511/8512 and PXI-8511/8512.

D-SUB PIN	Signal	Description
1	NC	No connection
2	CAN_L	CAN_L bus line
3	COM	CAN reference ground
4	NC	No connection
5	(SHLD)	Optional CAN shield
6	(COM)	Optional CAN reference ground
7	CAN_H	CAN_H bus line
8	NC	No connection
9	(Ext_Vbat)	Optional CAN power supply if bus power/external VBAT is required (single-wire CAN on XS

D-SUB PIN	Signal	Description
		hardware only)

External CAN Transceiver

The following table describes the CAN DB-9 pinout on PCI and PXI CAN interfaces that provide external CAN transceivers, such as PCI-8513 and PXI-8513.

D-SUB Pin	Signal	Description
1	Output1	Generic output used to configure the transceiver mode
2	Ext_RX	Data received from the CAN Bus
3	COM	CAN reference ground
4	Output0	Generic output used to configure the transceiver mode
5	(SHLD)	Optional CAN shield
6	COM	CAN reference ground
7	Ext_TX	Data to transmit on the CAN Bus
8	NERR	Input to connect to the nERR pin of your transceiver to route status back from the transceiver to the hardware
9	NC	No connection

C Series CAN Interface

The following table describes the CAN DB-9 pinout on C Series CAN interfaces, such as NI 9861 and NI 9862.

D-SUB Pin	Signal	Description
1	NC	No connection
2	CAN_L	CAN_L bus line
3	COM	CAN reference ground
4	NC	No connection
5	(SHLD)	Optional CAN shield
6	(COM)	Optional CAN reference ground

D-SUB Pin	Signal	Description
7	CAN_H	CAN_H bus line
8	NC	No connection
9	VSUP	External power supply (+9 V to +30 V) required

CAN HS/FD Transceiver Cable

The following table describes the CAN DB-9 pinout on the TRC-8542 CAN HS/FD Transceiver Cable.

D-SUB Pin	Signal	Description
1	NC	No connection
2	CAN_L	CAN_L bus line
3	COM	CAN reference ground
4	NC	No connection
5	NC	No connection
6	COM	CAN reference ground
7	CAN_H	CAN_H bus line
8	NC	No connection
9	NC	No connection

CAN HS/FD or LS/FT Transceiver Cable

The following table describes the CAN DB-9 pinout on the TRC-8543 CAN HS/FD Transceiver Cable.

D-SUB Pin	Signal	Description
1	NC	No connection
2	CAN_L	CAN_L bus line
3	COM	CAN reference ground
4	NC	No connection
5	NC	No connection

D-SUB Pin	Signal	Description
6	COM	CAN reference ground
7	CAN_H	CAN_H bus line
8	NC	No connection
9	VSUP	External power supply (+9 V to +30 V) required

USB CAN Interface Devices

The following table describes the CAN DB-9 pinout on USB CAN interfaces, such as USB-8501 and USB-8502.

D-SUB Pin	Signal	Description
1	NC	No connection
2	CAN_L	CAN_L bus line
3	COM	CAN reference ground
4	NC	No connection
5	NC	No connection
6	COM	CAN reference ground
7	CAN_H	CAN_H bus line
8	NC	No connection
9	NC	No connection

NI-XNET FlexRay Hardware

The FlexRay physical layer circuitry interfaces the FlexRay protocol controller to the physical bus wires. Refer to the following topics, which describe NI-XNET FlexRay hardware:

- Transceiver
- Bus Power Requirements
- Cabling Requirements for FlexRay
- Termination

Related concepts:

- CAN Transceiver for Single Wire Hardware
- CAN Single Wire Bus Power Requirements
- Cabling Requirements for FlexRay
- Low-Speed CAN Termination

FlexRay Transceiver

NI-XNET FlexRay hardware uses a pair of NXP TJA1080 FlexRay transceivers per port. The TJA1080 is fully compatible with the FlexRay standard and supports baud rates up to 10 Mbps. This device also supports advanced power management through a low-power sleep mode. Refer to the NI-XNET Session Interface:FlexRay:Sleep property for more information. For detailed TJA1080 specifications, refer to the NXP TJA1080 product data sheet.

NI-XNET FlexRay Bus Power Requirements

The FlexRay physical layer on PXI and PCI NI-XNET interfaces is internally powered. As such, there is no need to supply bus power. The COM pin serves as the reference ground for the bus signals.

Cabling Requirements for FlexRay

Cables may be shielded or unshielded and should meet the physical medium requirements described in the following table.

Characteristic	Value
Differential mode impedance @ 10 MHz	80–110 Ω
Specific line delay	10 ns/m
Cable attenuation @ 5 MHz (sine wave)	82 dB/km

The cabling characteristics, cabling topology, and desired bit transmission rates affect the allowable cable length. Detailed recommendations for cable length and number of

devices are in the ***FlexRay Electrical Physical Layer Specification*** available from the FlexRay Consortium. In general, the maximum electrical length for a passive bus topology is 24 m, with the number of devices limited to 22.

Related concepts:

- NI-XNET FlexRay Hardware

Related reference:

- Connect the Cables

FlexRay Termination

The simplest way to terminate FlexRay networks is with a single termination resistor between the bus wires Bus Plus and Bus Minus. The specific network topology determines the optimal termination values.

For all XNET devices, the termination is software selectable. XNET provides the option of 80 Ω between Bus Plus and Bus Minus or no termination. You cannot set termination for channel A and channel B independently. Refer to the Termination attribute in the XNET API for more details. To determine the appropriate termination for your network, refer to the ***FlexRay Electrical Physical Layer Specification*** for more information.

Refer to the NI-XNET Session Interface:FlexRay:Termination property for more information.

FlexRay Pinout

PCI and PXI FlexRay Interface

The following table describes the CAN DB-9 pinout on PCI and PXI FlexRay interfaces, such as PCI-8517 and PXI-8517.

PIN	Signal	Description
1	NC	No connection

PIN	Signal	Description
2	FlexRay A BM	FlexRay channel A bus minus
3	COM	FlexRay reference ground
4	FlexRay B BM	FlexRay channel B bus minus
5	SHLD	FlexRay shield
6	(COM)	Optional FlexRay reference ground
7	FlexRay A BP	FlexRay channel A bus plus
8	FlexRay B BP	FlexRay channel B bus plus
9	(Ext_VBat)	Optional external bus voltage

NI-XNET LIN Hardware

The NI-XNET LIN physical layer circuitry interfaces the LIN protocol controller to the physical bus wires. NI-XNET LIN Interfaces are fully compliant with the LIN 1.3/2.0/2.1/2.2 specification. Refer to the following topics, which describe NI-XNET LIN hardware:

- Transceiver
- Bus Power Requirements
- Cabling Requirements for LIN
- Termination

Related concepts:

- CAN Transceiver for Single Wire Hardware
- CAN Single Wire Bus Power Requirements
- Cabling Requirements for LIN
- Low-Speed CAN Termination

LIN Transceiver

NI-XNET LIN hardware uses the Atmel ATA6620 or ATA6625 LIN transceiver for PCI-XNET and PXI-XNET LIN Interfaces, and the NXP TJA1028 transceiver for USB, C Series, and

Transceiver Cable XNET LIN interfaces.

NI PXI-8516 and PCI-8516 XNET interfaces revision F and higher use the ATA6625 LIN transceiver, while revision E and lower use the ATA6620 LIN transceiver.

To identify your PCI/PXI NI-XNET hardware revision, refer to the ***19xxxx<rev>-4xL*** text on the green label in the top left corner on the secondary side of the board; ***<rev>*** indicates the hardware revision.

These transceivers are fully compatible with the ISO-9141 standard and support baud rates up to 20 kbps. For detailed information, refer to their respective data sheets.

NI-XNET LIN Bus Power Requirements

The LIN physical layer on NI-XNET interfaces requires an external power supply of +8 to +18 V, as the following table specifies. Connect the external power supply to the VBat/Vsup pin on the interface. The COM pins are for reference ground.

Characteristic	Specification
Voltage	+8 VDC to +18 VDC on VBat connector pin (referenced to COM)
Current	55 mA maximum

Cabling Requirements for LIN

LIN cables should meet the physical medium requirement of a bus RC time constant of 5 µs. For detailed formulas for calculating this value, refer to the Line Characteristics section of the LIN specification. Belden cable (3084A) and other unterminated CAN/Serial quality cables meet these requirements and should be suitable for most applications.

According to the local interconnect network (LIN) specification, the maximum allowable cable length is 40 m and the maximum number of devices on a LIN bus is 16.

Related concepts:

- NI-XNET LIN Hardware

Related reference:

- Connect the Cables

LIN Termination

LIN cables require no termination, as nodes are terminated at the transceiver. Slave nodes typically are pulled up from the LIN bus to VBat with a 30 kΩ resistance and a serial diode. This termination usually is integrated into the transceiver package. The master node requires a 1 kΩ resistor and serial diode between the LIN bus and VBat. On NI-XNET LIN products, master termination is software selectable; you can enable it in the API with the NI-XNET Session Interface:LIN:Termination property.

LIN Interface Pinouts

PCI and PXI LIN Interface

The following table describes the CAN DB-9 pinout on PCI and PXI LIN interfaces, such as PCI-8516 and PXI-8516.

PIN	Signal	Description
1	NC	No connection
2	NC	No connection
3	COM	LIN reference ground
4	NC	No connection
5	SHLD	Optional LIN shield. Connecting the optional shield may improve signal integrity in a noisy environment.
6	(COM)	Optional LIN reference ground
7	LIN	LIN data line
8	NC	No connection
9	VBat	Supplies bus power to the LIN physical layer, as the LIN

PIN	Signal	Description
		specification requires. All NI-XNET LIN interfaces require bus power of +8 to +18 VDC.

C Series CAN Interface

The following table describes the CAN DB-9 pinout on C Series LIN interfaces, such as NI 9866.

D-SUB Pin	Signal	Description
1	NC	No connection
2	NC	No connection
3	COM	LIN reference ground
4	NC	No connection
5	(SHLD)	Optional LIN shield
6	(COM)	Optional LIN reference ground
7	LIN	LIN data line
8	NC	No connection
9	VSUP	External power supply +8 V to +18 V) required

LIN Transceiver Cable

The following table describes the CAN DB-9 pinout on the TRC-8546 LIN Transceiver Cable.

D-SUB Pin	Signal	Description
1	NC	No connection
2	NC	No connection
3	COM	LIN reference ground
4	NC	No connection
5	NC	No connection
6	(COM)	Optional LIN reference ground

D-SUB Pin	Signal	Description
7	LIN	LIN data line
8	NC	No connection
9	VSUP	External power supply (+8 V to +18 V) required

USB LIN Interface Devices

The following table describes the CAN DB-9 pinout on USB LIN interfaces, such as USB-8506.

D-SUB Pin	Signal	Description
1	NC	No connection
2	NC	No connection
3	COM	LIN reference ground
4	NC	No connection
5	(SHLD)	Optional LIN shield
6	(COM)	Optional LIN reference ground
7	LIN	LIN data line
8	NC	No connection
9	VSUP	External power supply +8 V to +18 V) required

NI Automotive Ethernet Hardware

NI Automotive Ethernet hardware enables you to use the NI-XNET driver or standard networking drivers to develop applications to test and validate automotive electronic control units (ECUs) over automotive Ethernet networks.

The following topics describe NI Automotive Ethernet hardware:

- Cabling Requirements
- Synchronization

Note This document often uses the term **Automotive Ethernet** to refer to all Ethernet hardware supported by NI-XNET. Unless otherwise stated, assume this term refers to hardware with either Automotive Ethernet or standard Ethernet physical interfaces.

Related concepts:

- Synchronization

Cabling Requirements for Automotive Ethernet

Automotive Ethernet uses single unshielded twisted pair copper wiring. As specified in IEEE 802.3ch-2020, the balanced pair operates at full duplex and has a maximum length of 15 m. The physical layer requires the twisted pair to have an impedance of 100 Ω.

Related reference:

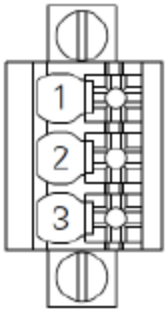
- Connect the Cables

NI-XNET Automotive Ethernet Pinout

Ethernet Interface

The following table describes the pinout signals for NI Automotive Ethernet hardware, such as PXIe-8521.

Pin Number	Signal Type	Required Signal Direction	Signal Description	Signal Required
1	TRX_P	Bidirectional	Transceiver plus	Required
2	TRX_M	Bidirectional	Transceiver minus	Required
3	Shield	—	Shield	Optional



Note Traditional Ethernet hardware such as the PXIe-8623 use the standard RJ45 pinout for Ethernet.

Synchronization

PXI, PXI Express, PCI, and PCI Express NI-XNET

The PXI and PXI Express chassis features a dedicated synchronization bus integrated into the backplane. NI-XNET products support use of this bus to synchronize with other NI hardware products such as DAQ, IMAQ, and motion. The PXI synchronization bus consists of a flexible interconnect scheme for sharing timing and triggering signals in a system.

For PCI and PCI Express hardware, the RTSI bus interface is a connector at the top of the card. You can synchronize multiple National Instruments PCI/PCIe cards by connecting a RTSI ribbon cable between the cards that need to share timing and triggering signals.

CAN/XS, PCIe-8510 (4-Port), and FlexRay XNET products also feature two configurable timing and triggering ports on the device front panel. These ports are TTL-tolerant user-configurable for inputting and outputting timebases and triggers. These signals are not electrically isolated from the backplane. Refer to the XNET Connect Terminals function documentation for more details.

C Series NI-XNET

All NI-XNET ports on a particular C Series chassis share a common timebase, allowing a

better correlation of data on the ports. NI-XNET products support use of this timebase to synchronize with other National Instruments hardware products such as DAQ modules.

Moreover, on a CompactRIO system, the module's timebase is corrected for drift with respect to the RT controller's timebase, allowing the capability to correlate data with other modules in the chassis.

On a CompactDAQ system, you can route the Start Trigger between multiple DAQmx and XNET modules. For information about performing this routing in LabVIEW, refer to the LabVIEW API Interface:Source Terminal:Start Trigger property. For information about performing this routing in C/C++, refer to the C API Interface:Source Terminal:Start Trigger property.

USB NI-XNET

USB-850x 2-port devices can synchronize with external trigger or clock sources. Synchronization occurs through a 3-pin Combicon connection allowing for a shared timestamp clock, start trigger, and ground. USB-850x 2-port devices can synchronize to timestamp clocks of 20 MHz, 10 MHz, or 1 MHz. For 20 MHz synchronization, ensure that the synchronization cable is shielded and grounded. Clock frequency is detected automatically by the hardware, and illegal clock frequencies are reported as an error. USB-850 x 2-port devices can also generate a clock of 1 MHz, allowing for accurate CAN-CAN, CAN-LIN, and LIN-LIN synchronization.

Automotive Ethernet NI-XNET

- **Local Time**—

  NI Automotive Ethernet modules use PXI_Clk10, a 10 MHz PXI backplane clock provided by the chassis, to drive the local time keeper and to synchronize with other modules in the PXI chassis. If the PXI backplane clock is not available, the module uses its own internal oscillator.

  PXI_Clk10 provides frequency but not date/time information. When an NI-XNET session is created, XNET initializes the date/time information for the local clock using host time.

- **Network Time**—

  NI Automotive Ethernet modules can also maintain network time (IEEE 802.1AS) for each port. When Ethernet frames are received, each packet is time stamped with network time as well as with local time.

  When a port acts as a master, the network time is initialized from host time and is synchronized to local time.

  When a port acts as a slave in an electronic control unit (ECU) network, local time and network time can eventually drift, relative to each other. The date/time information for network time is obtained from the ECU that acts as the grandmaster clock.

  Both local and network time can be adjusted using the NI-XNET API.

- **Host Time**—

  Host time is the clock of the operating system where LabVIEW is running. The host time can obtain time/date information using a real time clock (RTC) or a network time protocol (NTP) server.

  Although host time provides accurate date/time information, the accuracy and resolution of its clock can often be in tens of milliseconds. In contrast, NI Automotive Ethernet modules provide resolution for local time and network time in nanoseconds. Although local time and network time use host time to initialize their date/time information, they do not use the same physical clock as host time. Therefore, both local time and network time can eventually drift relative to host time.

- **Triggers**—

  Triggers can be simultaneously time stamped by the local time keeper and the network time keeper for each port. PXI triggers can be used to synchronize the NI Automotive Ethernet module's time keepers with trigger events on other PXI modules.

Related concepts:

- NI Automotive Ethernet Hardware

Related tasks:

- Install Your PCI/PCI Express Hardware
- Install Your USB Hardware

Isolation

All NI-XNET hardware products protect your equipment from being damaged by high-voltage spikes on the target bus. Bus ports on PCI, PXI, and NI-XNET products support channel-to-channel and channel-to-bus isolation, and are galvanically isolated up to 60 VDC; the isolation is intended to prevent ground loops.

Bus ports on C Series NI-XNET products support channel-to-bus isolation, and are galvanically isolated up to 500 Vrms (5 s max withstand).

Bus ports on NI-XNET Transceiver Cable products support channel-to-bus isolation, and are galvanically isolated up to 1000 Vrms (5 s max withstand).

Note For Multiprotocol Interface products such as PXIe-8510, PCIe-8510, and NI-9860, isolation is provided through the NI-XNET Transceiver Cable.

Related concepts:

- NI-XNET Transceiver Cables

Interfaces

The interface represents a single CAN, FlexRay, LIN, or Ethernet connector on an NI hardware device. Within NI-XNET, the interface is the object used to communicate with external hardware described in the database.

Each interface name uses the following syntax:

<protocol><n>

The <protocol> is one of the following:

- CAN for a CAN interface
- FlexRay for a FlexRay interface
- LIN for a LIN interface
- ENET for an Ethernet interface

The number <n> identifies the specific interface within the <protocol> scope. The numbering starts at 1. For example, if you have a two-port CAN device, a two-port FlexRay device, a two-port LIN device, and a two-port Ethernet device in your system, the interface names are CAN1, CAN2, FlexRay1, FlexRay2, LIN1, LIN2, ENET1, and ENET2, respectively. Devices that use a transceiver cable receive an interface name only when the transceiver cable is connected and identified.

Although you can change the interface number <n> within Measurement & Automation Explorer (MAX), the typical practice is to allow NI-XNET to select the number automatically. NI-XNET always starts at 1 and increments for each new interface found. If you do not change the number in MAX, and your system always uses a single two-port CAN device, you can write all your applications to assume CAN1 and CAN2. For as long as that CAN card exists in your system, NI-XNET uses the same interface numbers for that device, even if you add new CAN cards.

NI-XNET also uses the term port to refer to the connector on an NI hardware device. This physical connector includes the transceiver cable if applicable. The difference between the terms is that port refers to the hardware object (physical), and interface refers to the software object (logical). The benefit of this separation is that you can use

the interface name as an alias to any port, so that your application does not need to change when your hardware configuration changes. For example, if you have a PXI chassis with a single CAN PXI device in slot 3, the CAN port labeled Port 1 is assigned as interface CAN1. Later on, if you remove the CAN PXI card and connect a USB device for CAN, the CAN port on the USB device is assigned as interface CAN1. Although the physical port is in a different place, VIs or programs written to use CAN1 work with either hardware configuration without change.

For Ethernet interfaces, a special suffix "/monitor" appended to the interface name indicates the use of a monitor path. For example, "ENET1" specifies use of the endpoint path, and "ENET1/monitor" specifies use of the monitor path. The monitor path is used to read Ethernet frames that are received or transmitted on each port. When Tap is enabled, data received via the monitor path by a Tap pair will be identical on each port in the pair. Additional information on the monitor and endpoint paths is provided in Using Ethernet.

Related concepts:

- Using NI-CAN
- Sessions
- Using Ethernet

Displaying Available Interfaces

Measurement and Automation Explorer (MAX)

Use NI MAX to view your available NI-XNET hardware, including all devices and interfaces.

To view hardware in your local Windows system, select **Devices and Interfaces** under **My System**. Each NI-XNET device is listed by hardware model name followed by port name, for example, NI PCI-8517 "FlexRay1, FlexRay2".

Select each NI-XNET device to view its physical ports. Each port is listed with the current interface name assignment, such as FlexRay1.

In the selected port's window on the right, you can change one property: the interface

name. Therefore, you can assign a different interface name than the default. For example, you can change the interface for physical port 2 of a PCI-8517 to FlexRay1 instead of FlexRay2. The blinking LED test panel assists in identifying a specific port when your system contains multiple instances of the same hardware product (for example, a chassis with five CAN devices).

To view hardware in a remote LabVIEW Real-Time system, find the desired system under **Remote Systems** and select **Devices and Interfaces** under that system. The features of NI-XNET devices and interfaces are the same as the local system.

Related concepts:

- How Do I Create a Session?

Databases

Databases are the means of choice for managing embedded networks. Although it is possible (and supported) in principle to run a network without a database, using a database is highly recommended to maintain a consistent set of network parameters for all nodes in the network. This is especially true for FlexRay, which requires you to set up about 30 parameters consistently to get a running network.

Additionally, a database can manage the contents of the data exchanged over the network. You can store frames and signals running on the network in a database, as well as information about which ECU is transmitting or receiving which data. This information also is needed for each node in the network.

For an NI-XNET interface to communicate with hardware products on the external network, NI-XNET must understand the communication in the actual embedded system, such as the vehicle. This embedded communication is described within a standardized file, such as CANdb (`.dbc`), FIBEX (`.xml`), AUTOSAR (`.arxml`), or LIN Description File (`.ldf`). Within NI-XNET, this file is referred to as a database. The database contains many object classes, each of which describes a distinct entity in the embedded system.

- **Database**: Each database is represented as a distinct instance in NI-XNET. Although the database typically is a file, you also can create the database at run time (in

memory).

- **Cluster**: Each database contains one or more clusters, where the cluster represents a collection of hardware products connected over a shared cabling harness. In other words, each cluster represents a single CAN, FlexRay, or LIN network. For example, the database may describe a single vehicle, where the vehicle contains one CAN cluster Body, another CAN cluster Powertrain, one FlexRay cluster Chassis, and a LIN cluster PowerSeat.
- **ECU**: Each Electronic Control Unit (ECU) represents a single hardware product in the embedded system. The cluster contains one or more ECUs connected over a CAN, FlexRay, or LIN cable. It is possible for a single ECU to be contained in multiple clusters, in which case it behaves as a gateway between the clusters.)
- **Frame**: Each frame represents a unique unit of data transfer over the cluster cable. The frame bits contain payload data and an identifier that specifies the data (signal) content. Only one ECU in the cluster transmits (sends) each frame, and one or more ECUs receive each frame.
- **Signal**: Each frame contains zero or more values, each of which is called a signal. Within the database, each signal specifies its name, position, length of the raw bits in the frame, and a scaling formula to convert raw bits to/from a physical unit. The physical unit uses a double-precision floating-point numeric type.

Other object classes include the Subframe, LIN Schedule, and LIN Schedule Entry.

Note that Ethernet interfaces currently do not support databases.

Related concepts:

- Using NI-CAN
- Sessions

Creating a Database Alias

When using a database file with NI-XNET, you can specify the file path or an alias to the file. The alias provides a shorter, easier-to-read name for use within your application.

For example, for the file path

```
C:\Documents and Settings\All Users\Documents\Vehicle5\
```

```
MyDatabase.dbc
```

you can add an alias named MyDatabase. In addition to improving readability, the alias concept isolates your application from the specific file path. For example, if your application uses the alias MyDatabase and you change its file path to

```
C:\Embedded\Vehicle5\MyDatabase.dbc
```

your application continues to run without change.

After you create an alias, it exists until you explicitly delete it. If you uninstall NI-XNET, the aliases are deleted; however, if you reinstall (upgrade) NI-XNET, the aliases from the previous installation remain. Deleting an alias does not delete the database file itself, but merely the association within NI-XNET.

Related concepts:

- XNET Database I/O Name

Sessions

The NI-XNET session represents a connection between your National Instruments CAN, FlexRay, LIN, or Ethernet hardware and hardware products on the external network.

Each session configuration includes:

- Interface : This specifies the National Instruments hardware to use.
- Database objects : These describe how external hardware communicates.
- Mode : This specifies the direction and representation of I/O data.

The links above link to detailed information about each configuration topic. The mode topic has additional links to topics that explain how to read or write I/O data for each mode. The I/O data consists of values for frames or signals.

In addition to read/write of I/O data, you can use the session to interact with the network in other ways. For example, `nxReadState` includes selections to read the state of communication, such as whether communication has stopped due to error

detection defined by the protocol standard.

You can use sessions for multiple hardware interfaces. For each interface, you can use multiple input sessions and multiple output sessions simultaneously. The sessions can use different modes. For example, you can use a Signal Input Single-Point session at the same time you use a Frame Input Stream session.

The limitations on sessions relate primarily to a specific frame or its signals. For example, if you create a Frame Output Queued session for frameA, then create a Signal Output Single-Point session for frameA.signalB (a signal in frameA), NI-XNET returns an error. This combination of sessions is not allowed, because writing data for the same frame with two sessions would result in inconsistent sequences of data on the network.

Related concepts:

- Using NI-CAN
- Interfaces
- Databases
- Session Modes

Session Modes

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.

The mode is an enumeration of the following:

- **Signal Input 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 Input 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 Input 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 Output 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 Output 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 Output 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 Input 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 Input 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 Input 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 Output 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 Output Queued** : Provides a sequence of values for a single frame, for transmit using that frame's timing as the database specifies.
- **Frame Output 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).
- **Conversion** : This mode does not use any hardware. It is used to convert data between the signal representation and frame representation.

Note that Ethernet is supported by only two modes, Frame Input Stream and Frame Output Stream.

Related concepts:

- CAN Timing Type and Session Mode
- FlexRay Timing Type and Session Mode

- LIN Frame Timing and Session Mode
- Signal Input Single-Point Mode
- Signal Input Waveform Mode
- Signal Input XY Mode
- Signal Output Single-Point Mode
- Signal Output Waveform Mode
- Signal Output XY Mode
- Frame Input Stream Mode
- Frame Input Queued Mode
- Frame Input Single-Point Mode
- Frame Output Stream Mode
- Frame Output Queued Mode
- Frame Output Single-Point Mode
- Conversion Mode
- Sessions

Frame Input Queued Mode

This mode 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 XNET Read VI (LabVIEW) or `nxReadFrame` function (C) 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 an array of frame values. These values represent all values received for the frame since the previous call to XNET Read VI or `nxReadFrame`.

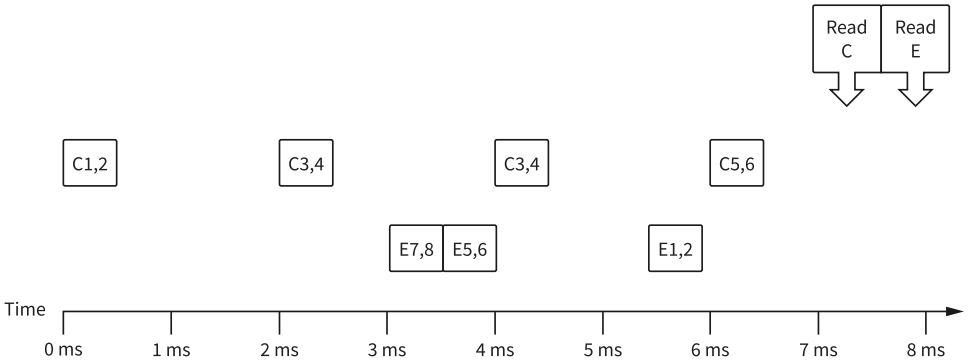
In LabVIEW, if the session uses a CAN interface, the XNET Read (Frame CAN) VI is the recommended way to read data for this mode. This VI returns an array of frames, where each frame is a LabVIEW cluster specific to the CAN protocol. If the session uses a FlexRay or LIN interface, the read selection for that protocol is recommended. For more advanced applications, use the XNET Read (Frame Raw) VI, which returns frames in an optimized, protocol-independent format.

Example

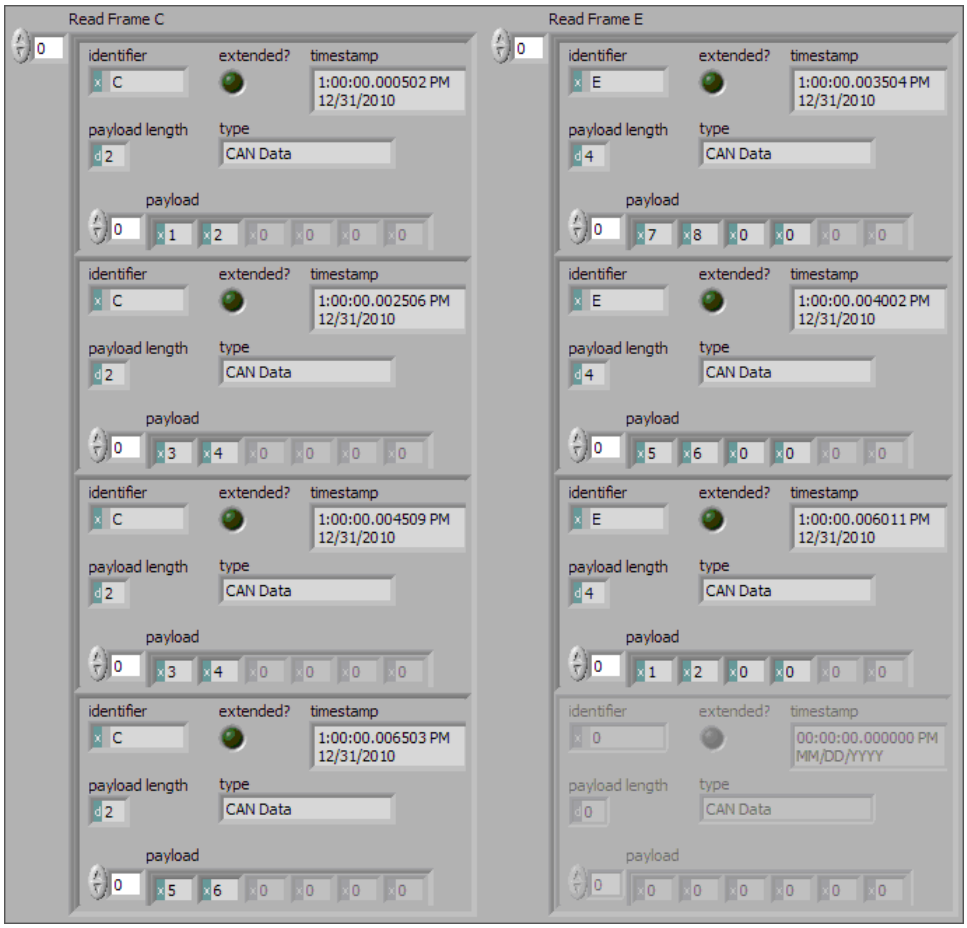
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing .

Each frame contains two signals, one in the first byte and another in the second byte.

This example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network, followed by two calls to XNET Read VI or `nxReadFrame`, one for frame C and one for frame E.



The following figure shows the data returned from the two calls to XNET Read VI or `nxReadFrame` separate sessions for frame C and frame E.



The first call returned an array of values for frame C, and the second call returned an array for frame E. Each frame is displayed with CAN-specific elements. For information about the data returned from the read function, refer to Raw Frame Format. The example uses hexadecimal C and E as the identifier of each frame. The first two payload bytes contain the signal data. The timestamp represents the absolute time when the NI-XNET interface received the frame (end of frame), accurate to microseconds.

Compared to the example for the Frame Input Stream mode, this mode effectively sorts received frames so you can process them on an individual basis.

Related concepts:

- Cyclic and Event Timing
- Raw Frame Format
- Frame Input Stream Mode
- Session Modes

Frame Input Single-Point Mode

This mode reads the most recent value received for each frame. It typically is used for control or simulation applications that require lower level access to frames (not signals).

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

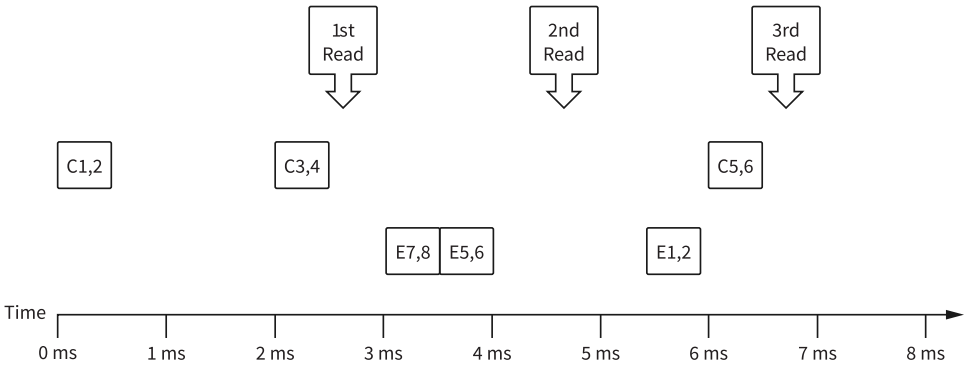
The input data is returned as an array of frames, one for each frame specified for the session.

Example

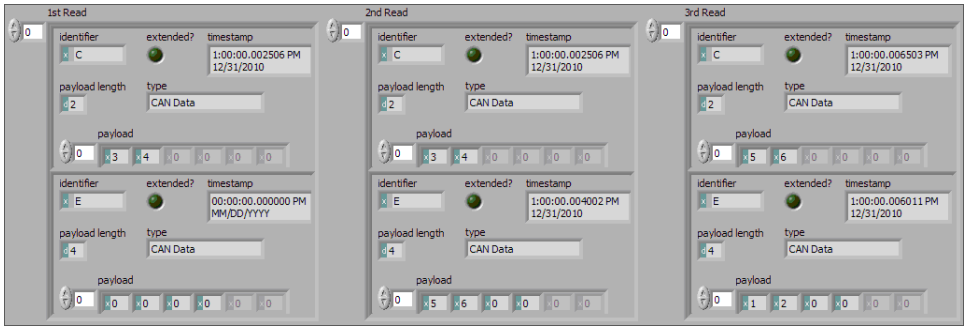
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing.

Each frame contains two signals, one in the first byte and another in the second byte.

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network, followed by a single call to `nxReadFrame`. Each frame contains its name (C or E), followed by the value of its two signals.



The following figure shows the data returned from each of the three calls to `nxReadFrame`. Each frame is displayed with CAN-specific elements. For information about the data returned from the read function, refer to Raw Frame Format. The session contains frame data for two frames: C and E.



In the data returned from the first call to `nxReadFrame`, frame C contains values 3 and 4 in its payload. The first reception of frame C values (1 and 2) were lost, because this mode returns the most recent values.

In the frame timeline, Time of 0 ms indicates the time at which the session started to receive frames. For frame E, no frame is received prior to the first call to `nxReadFrame`, so the timestamp is invalid, and the payload is the Default Payload. For this example we assume that the Default Payload is all 0.

In the data returned from the second call to `nxReadFrame`, payload values 3 and 4 are returned again for frame C, because no new frame has been received since the previous call to `nxReadFrame`. The timestamp for frame C is the same as the first call to `nxReadFrame`.

In the data returned from the third call to `nxReadFrame`, both frame C and frame E are received, so both elements return new values.

Related concepts:

- Cyclic and Event Timing
- Raw Frame Format
- Session Modes

Frame Input Stream Mode

This mode reads all frames received from the network using a single stream. It typically is used for analyzing and/or logging all frame traffic in the network.

The input data is returned as an array 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 FrameA, then create a Frame Input Stream session, when FrameA is received, its data is returned from the call to `nxReadFrame` 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.

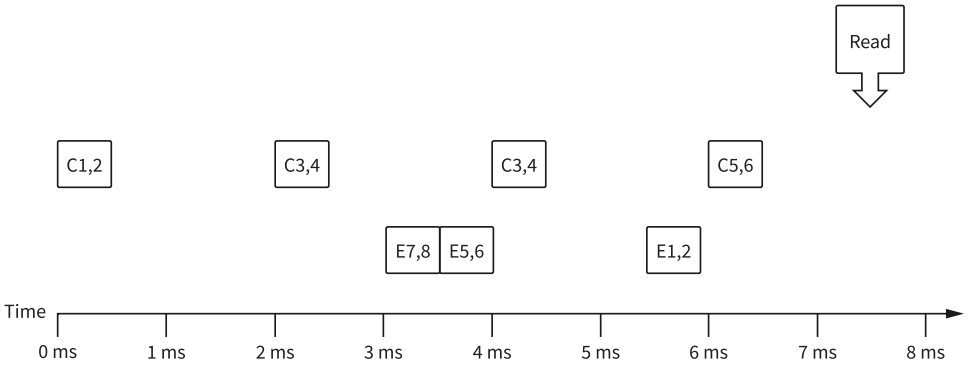
When used with a FlexRay interface, frames from both channels are returned. For example, if a frame is received in a static slot on both channel A and channel B, two frames are returned from `nxReadFrame`.

Example

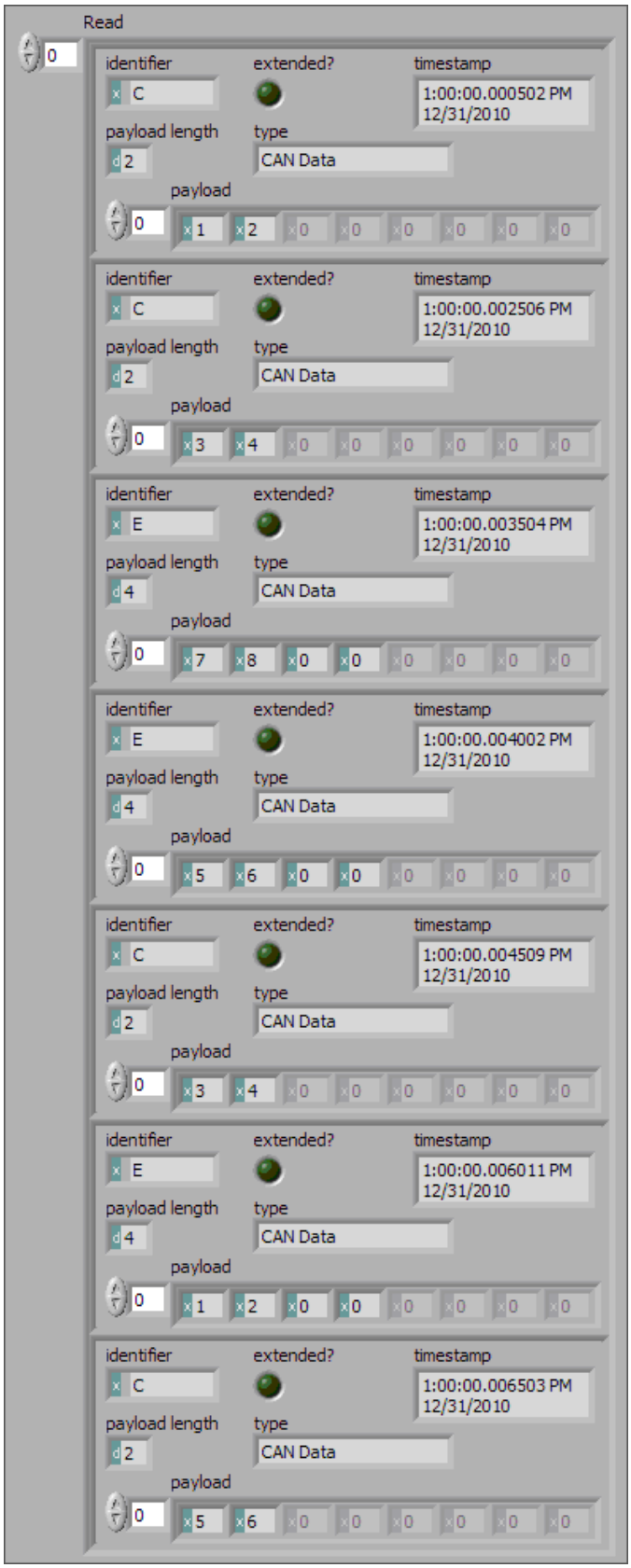
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing.

Each frame contains two signals, one in the first byte and another in the second byte.

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network, followed by a single call to `nxReadFrame`. Each frame contains its name (C or E), followed by the value of its two signals.



The following figure shows the data returned from `nxReadFrame`.



Frame C and frame E are returned in a single array of frames. Each frame is displayed with CAN-specific elements. For information about the data returned from the read function, refer to Raw Frame Format. This example uses hexadecimal C and E as the identifier of each frame. The signal data is contained in the first two payload bytes. The timestamp represents the absolute time when the XNET interface received the frame (end of frame), accurate to microseconds.

Related concepts:

- Basic Programming Model
- Frame Input Queued Mode
- Cyclic and Event Timing
- Raw Frame Format
- Session Modes

Frame Output Queued Mode

This mode 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 an array of frame values, to be transmitted sequentially for the frame specified in the session.

This mode allows you to specify only one frame for the session. To transmit sequential values for multiple frames, use a different Frame Output Queued session for each frame or use the Frame Output Stream mode.

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

For this mode, NI-XNET transmits each frame according to its properties in the database. Therefore, when you call `nxWriteFrame`, 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

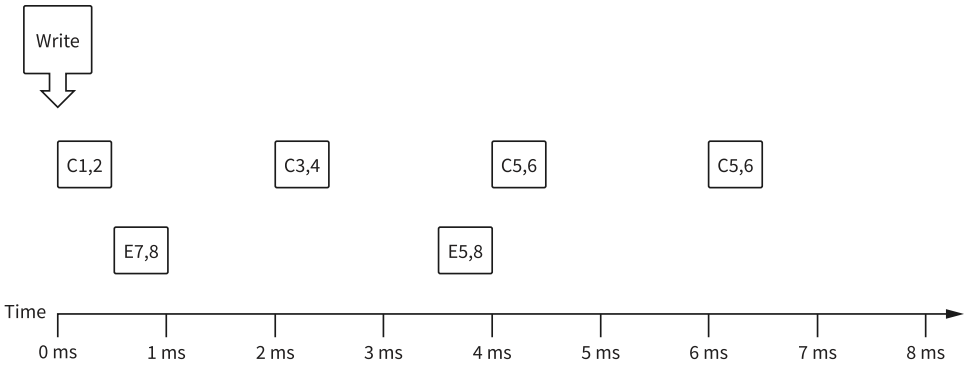
bytes different than the frame's payload may cause unexpected results on the bus.

Examples

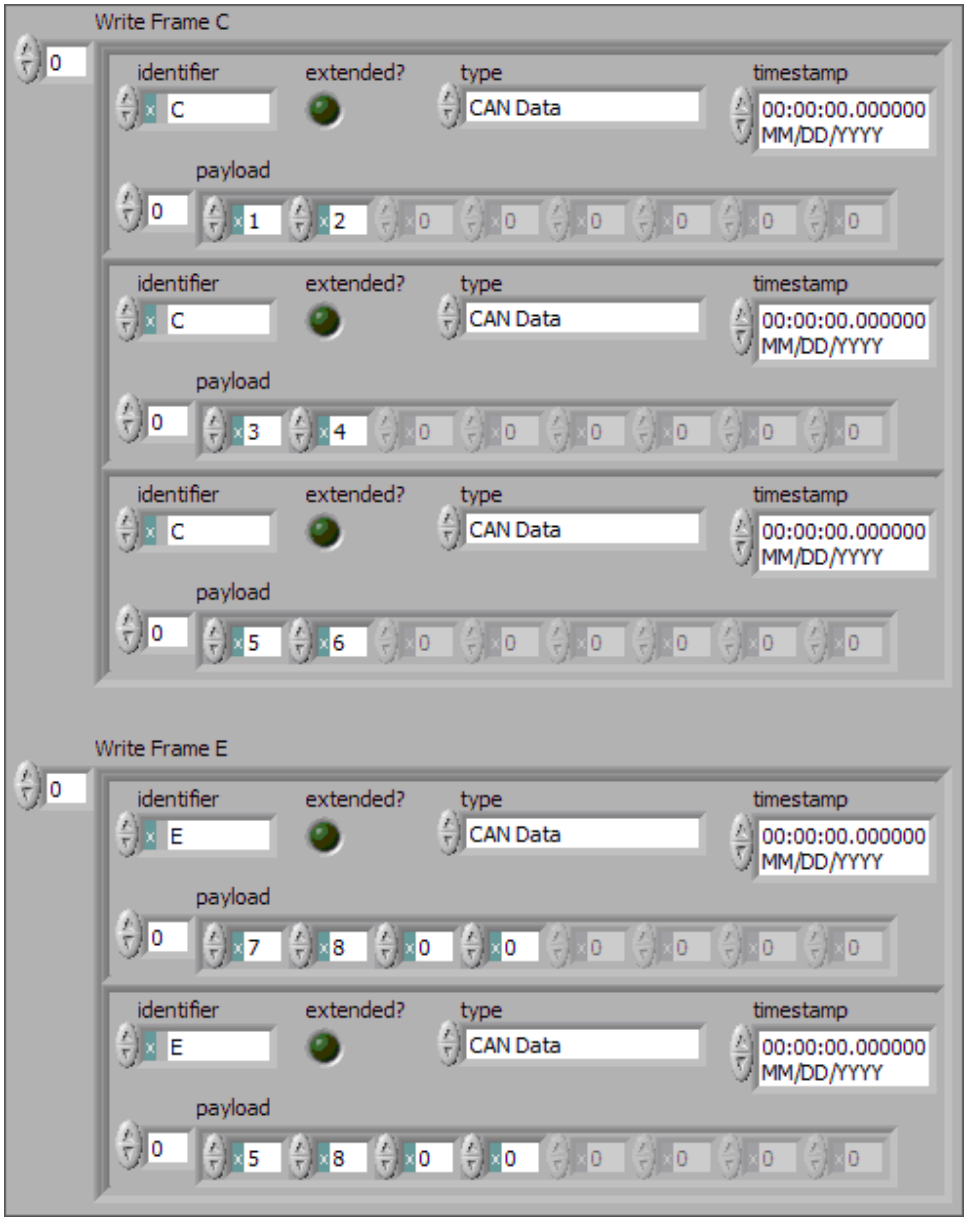
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing.

Each frame contains two signals, one in the first byte and another in the second byte.

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network. Each frame contains its name (C or E), followed by the value of its two signals. The timeline begins with two calls to `nxWriteFrame`, one for frame C, followed immediately by another call for frame E.



The following figure shows the data provided to each call to `nxWriteFrame`. Each frame is displayed with CAN-specific elements. For information about the data returned from the write function, refer to Raw Frame Format. The first array shows data for the session with frame C. The second array shows data for the session with frame E.



Assuming the Auto Start? property uses the default of true, each session starts within the call to `nxWriteFrame`. Frame C transmits followed by frame E, both using the frame values from the first element (index 0 of each array).

According to the database, frame C transmits once every 2.0 ms, and frame E is limited to an event-driven transmit once every 2.5 ms.

At 2.0 ms in the timeline, the frame value with bytes 3, 4 is taken from index 1 of the frame C array and used for transmit of frame C.

When 2.5 ms have elapsed after acknowledgment of the previous transmit of frame E,

the frame value with bytes 5, 8, 0, 0 is taken from index 1 of frame E array and used for transmit of frame E.

At 4.0 ms in the timeline, the frame value with bytes 5, 6 is taken from index 2 of the frame C array and used for transmit of frame C.

Because there are no more frame values for frame E, this frame no longer transmits. Frame E is event-driven, so new frame values are required for each transmit.

Because frame C is a cyclic frame, it transmits repeatedly. Although there are no more frame values for frame C, the previous frame value is used again at 6.0 ms in the timeline, and every 2.0 ms thereafter. If `nxWriteFrame` is called again, the new frame value is used.

Related concepts:

- Frame Output Stream Mode
- Cyclic and Event Timing
- Raw Frame Format
- Session Modes

Frame Output Single-Point Mode

This mode writes frame values for the next transmit. It typically is used for control or simulation applications that require lower level access to frames (not signals).

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

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

For this mode, NI-XNET transmits each frame according to its properties in the database. Therefore, when you call `nxWriteFrame`, 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

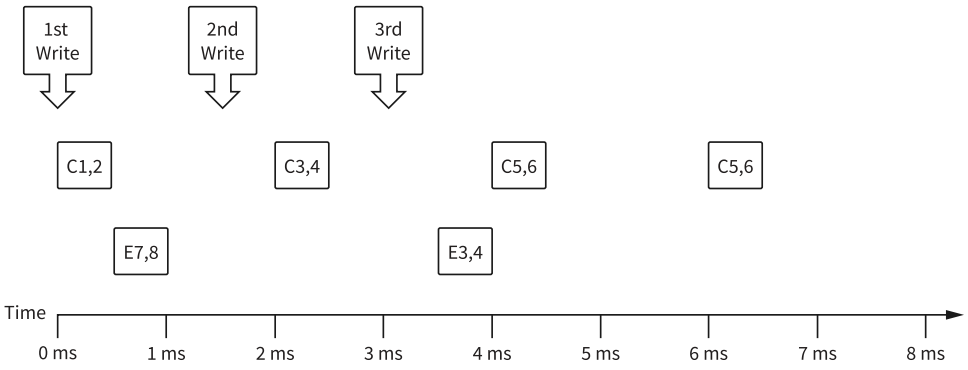
in the database, the requested number of bytes transmit. If the number of payload bytes is larger than the Payload Length configured in the database, the queue is flushed and no frames transmit. For other interfaces, transmitting a number of payload bytes different than the frame payload may cause unexpected results on the bus.

Example

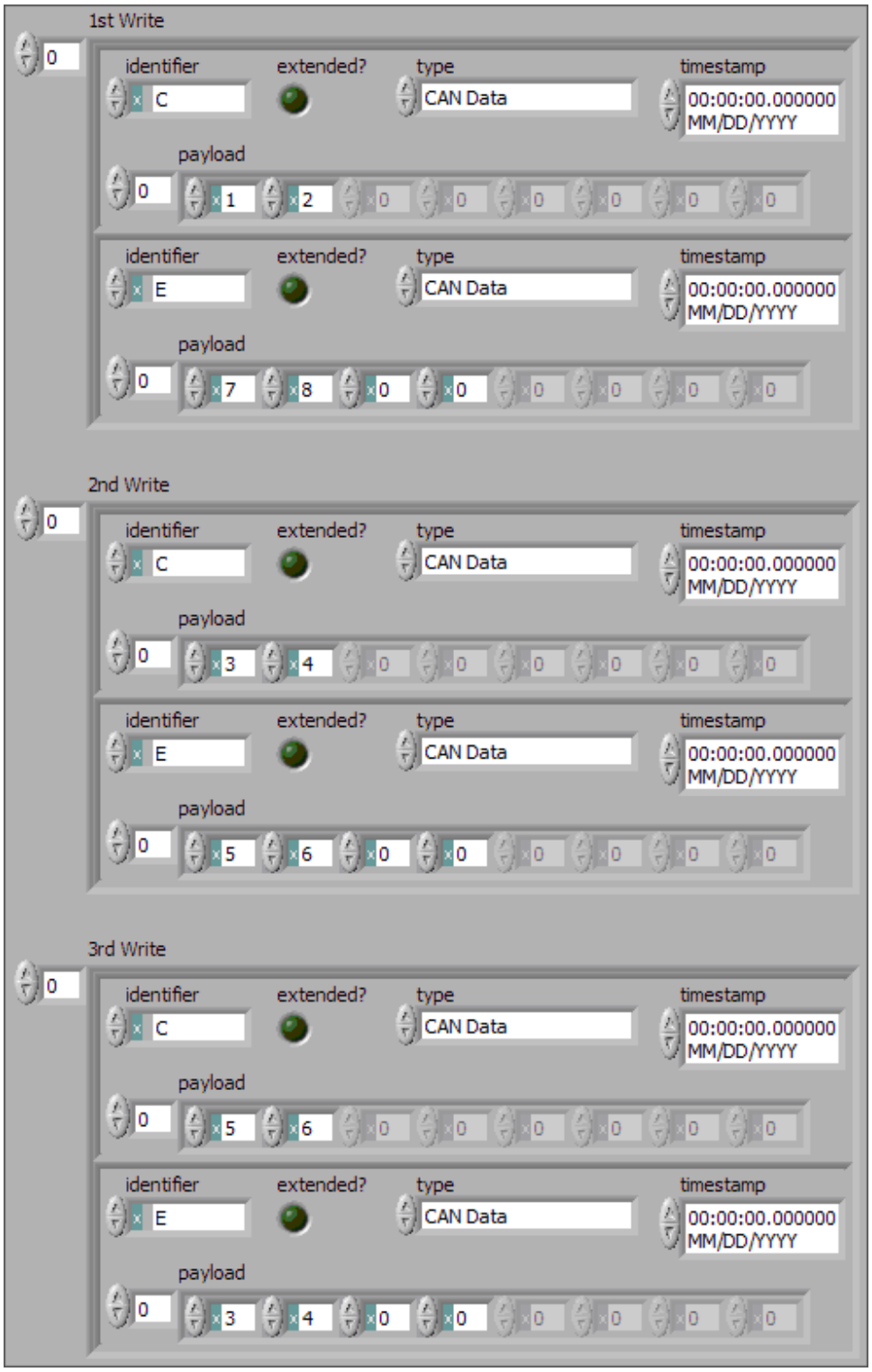
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing.

Each frame contains two signals, one in the first byte and another in the second byte.

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network. Each frame contains its name (C or E), followed by the value of its two signals. The timeline shows three calls to `nxWriteFrame`.



The following figure shows the data provided to each of the three calls to `nxWriteFrame`. Each frame is displayed with CAN-specific elements. For information about the data returned from the write function, refer to Raw Frame Format. The session contains frame values for two frames: C and E.



Assuming the Auto Start? property uses the default of true, the session starts within the first call to `nxWriteFrame`. Frame C transmits followed by frame E, both using frame values from `nxWriteFrame`.

After the second call to `nxWriteFrame`, frame C transmits using its value (bytes 3, 4), but frame E does not transmit, because its minimal interval of 2.5 ms has not elapsed since acknowledgment of the previous transmit.

Because the third call to `nxWriteFrame` occurs before the minimum interval elapses for frame E, its next transmit uses its value (bytes 3, 4, 0, 0). The value for frame E in the second call to `nxWriteFrame` is not used.

Frame C transmits the third time using the value from the third call to `nxWriteFrame` (bytes 5, 6). Because frame C is cyclic, it transmits again using the same value (bytes 5, 6).

Related concepts:

- Cyclic and Event Timing
- Raw Frame Format
- Session Modes

Frame Output Stream Mode

This mode 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 `nxWriteFrame` is an array of frame values, each of which transmits as soon as possible. Frames transmit sequentially (one after another).

This mode is not supported for FlexRay.

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 `nxWriteFrame`. The timing of each frame as specified in the database is ignored. For example, if you provide four frame values to the `nxWriteFrame`, the first frame value transmits immediately, followed by the next three values transmitted back to back. For this mode, the CAN frame payload length in the database is ignored, and `nxWriteFrame` is always used.

Similarly for LIN, frame values transmit on the network based entirely on the time when you call `nxWriteFrame`. The timing of each frame as specified in the database is ignored. The LIN frame payload length in the database is ignored, and `nxWriteFrame` is always used. For LIN, this 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 non-empty 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.

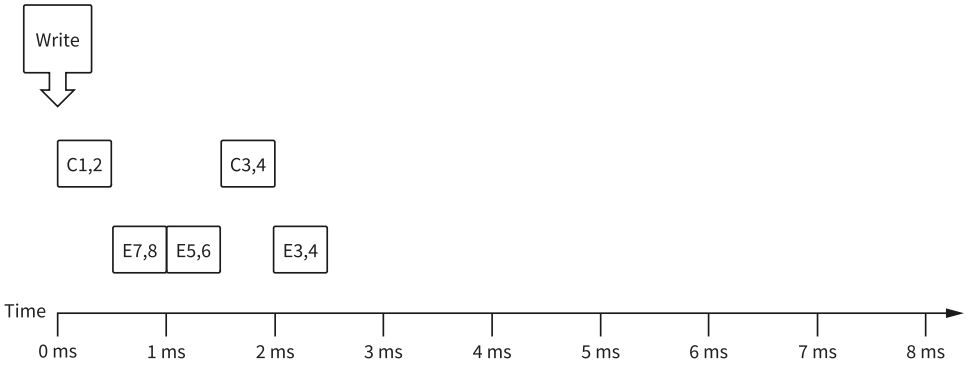
The frame values for this mode are stored in a queue, such that every value provided is transmitted.

Example

In this example CAN database, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven CAN frames, refer to Cyclic and Event Timing.

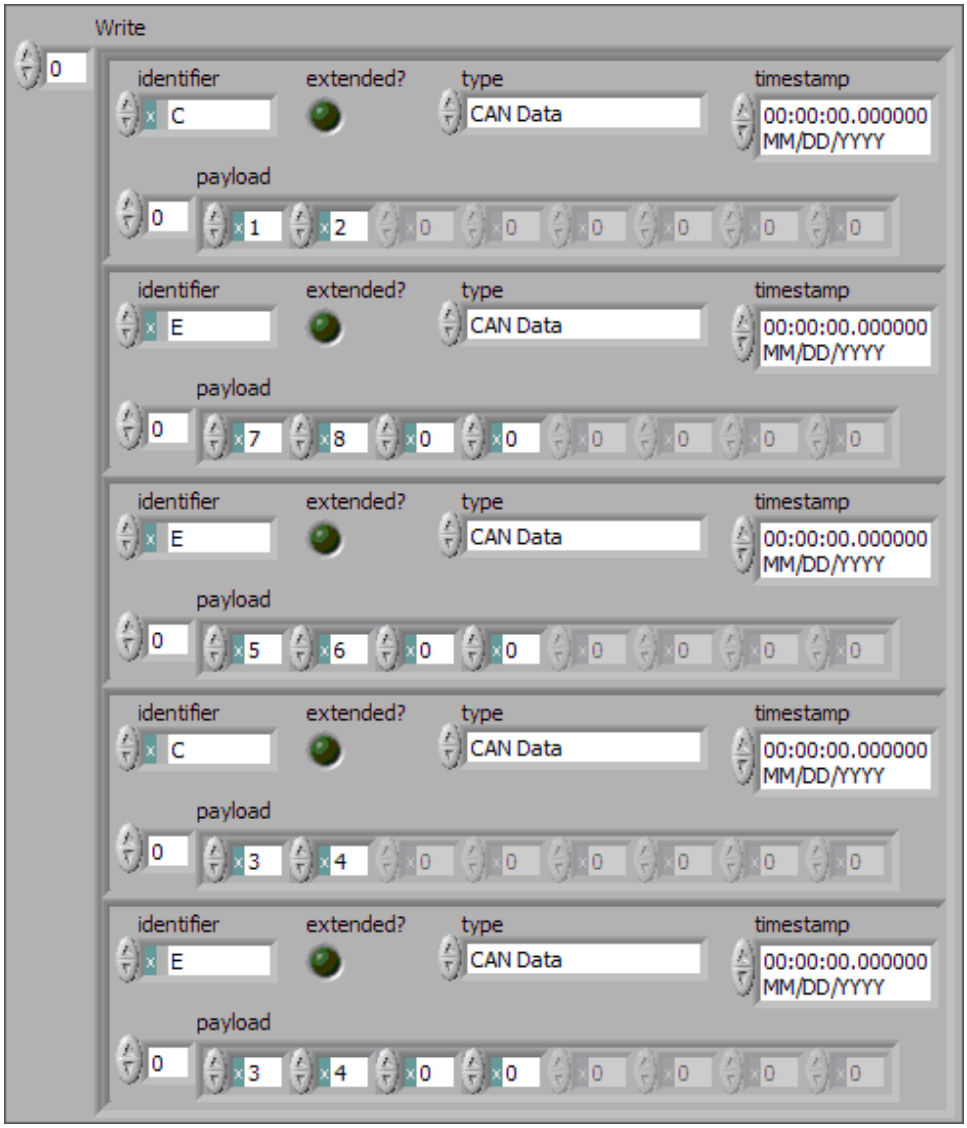
Each frame contains two signals, one in the first byte and another in the second byte.

The following figure shows a timeline of a frame transfer on the CAN network. Each frame contains its name (C or E), followed by the value of its two signals. The timeline begins with a single call to `nxWriteFrame`.



The following figure shows the data provided to the single call to `nxWriteFrame`. Each frame is displayed with CAN-specific elements. For information about the data returned from the write function, refer to Raw Frame Format. The array provides

values for frames C and E.



Assuming the Auto Start? property uses the default of true, each session starts within the call to `nxWriteFrame`. All frame values transmit immediately, using the same sequence as the array.

Although frame C and E specify a slower timing in the database, the Frame Output Stream mode disregards this timing and transmits the frame values in quick succession.

Within each frame values, this example uses an invalid timestamp value (0). This is acceptable, because each frame value timestamp is ignored for this mode.

Although frame C is specified in the database as a cyclic frame, this mode does not repeat its transmit. Unlike the Frame Output Queued mode, the Frame Output Stream mode does not use CAN frame properties from the database.

Related concepts:

- Frame Output Queued Mode
- Cyclic and Event Timing
- Raw Frame Format
- Session Modes
- Synchronized Replay

Signal Input Single-Point Mode

This mode reads the most recent value received for each signal. It typically is used for control or simulation applications, such as Hardware In the Loop (HIL).

This mode does not use queues to store each received frame. If the interface receives two frames prior to calling `nxReadSignalSinglePoint`, that call to `nxReadSignalSinglePoint` returns signals for the second frame.

Use `nxReadSignalSinglePoint` for this mode.

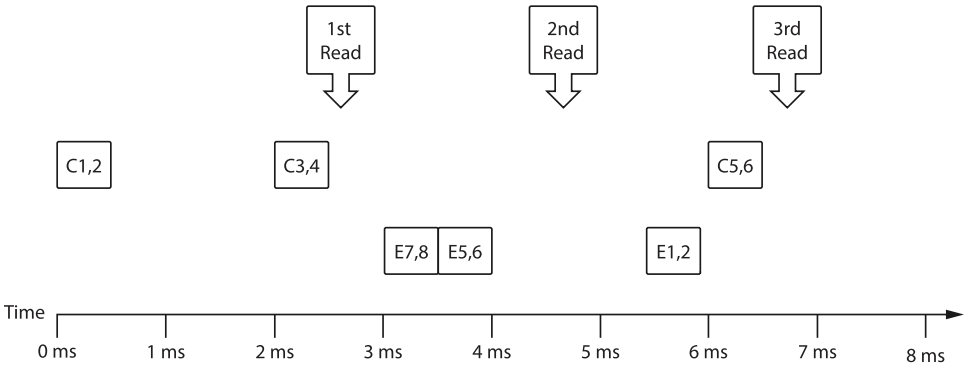
You also can specify a trigger signal for a frame. This signal name is `:trigger:.<frame name>`, and once it is specified in the `nxCreateSession` signal list, it returns a value of 0.0 if the frame did not arrive since the last Read (or Start), and 1.0 if at least one frame of this ID arrived. You can 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.

Example

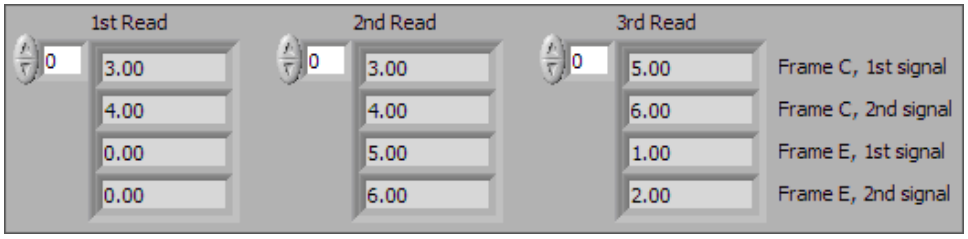
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing.

Each frame contains two signals, one in the first byte and another in the second byte.

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network. Each frame contains its name (C or E), followed by the value of its two signals. The timelines shows three calls to `nxReadSignalSinglePoint`.



The following figure shows the data returned from each of the three calls to `nxReadSignalSinglePoint`. The session contains all four signals.



In the data returned from the first call to `nxReadSignalSinglePoint` , values 3 and 4 are returned for the signals of frame C. The values of the first reception of frame C (1 and 2) were lost, because this mode returns the most recent values.

In the frame timeline, Time of 0 ms indicates the time at which the session started to receive frames. For frame E, no frame is received prior to the first call to `nxReadSignalSinglePoint` , so the last two values return the signal Default Values. For this example, assume that the Default Value is 0.0.

In the data returned from the second call to `nxReadSignalSinglePoint` , values 3 and 4 are returned again for the signals of frame C, because no new frame has been received since the previous call to `nxReadSignalSinglePoint`. New values are returned for frame E (5 and 6).

In the data returned from the third call to `nxReadSignalSinglePoint`, both frame C and frame E are received, so all signals return new values.

Related concepts:

- Basic Programming Model
- Session Modes
- Cyclic and Event Timing

Signal Input Waveform Mode

Using the time when the signal frame is received, this mode 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.

Use `nxReadSignalWaveform` for this mode.

You specify the resample rate using the XNET Session Resample Rate property.

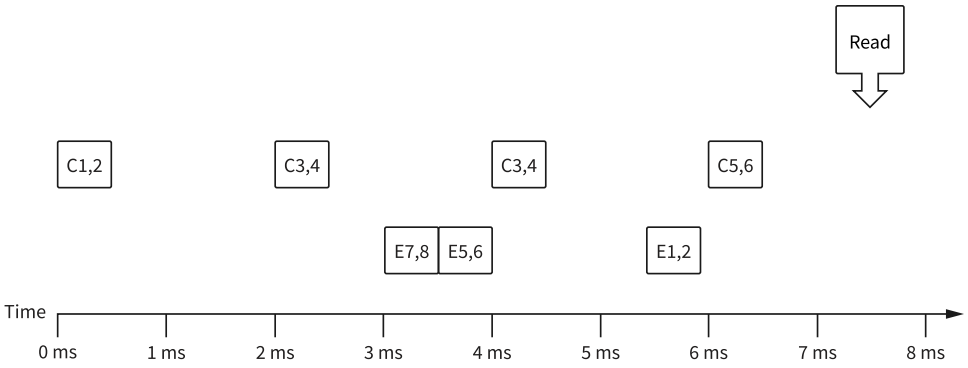
Starting a Signal Input Waveform session discards any previous samples and frames (the same result as running nxFlush). Note that when calling nxReadSignalWaveform for the first time on the session, the session will be started if it was not already. Stopping the session after the first start requires the session to be explicitly started in the future.

Signal Input Waveform Example

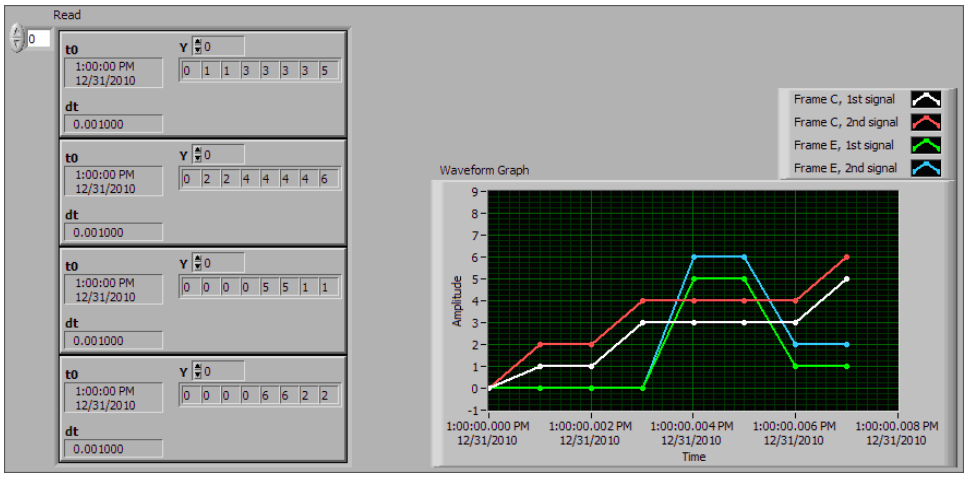
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing.

Each frame contains two signals, one in the first byte and another in the second byte.

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network, followed by a single call to `nxReadSignalWaveform`. Each frame contains its name (C or E), followed by the value of its two signals.



The following figure shows the data returned from `nxReadSignalWaveform`. The session contains all four signals and uses the default resample rate of 1000.0.



In the data returned from `nxReadSignalWaveform`, t0 provides an absolute timestamp for the first sample. Assuming this is the first call to `nxReadSignalWaveform` after starting the session, this t0 reflects that start of the session, which corresponds to Time 0 ms in the frame timeline. At time 0 ms, no frame has been received. Therefore, the first sample of each waveform uses the signal default value. For this example, assume the default value is 0.0.

In the frame timeline, frame C is received twice with signal values 3 and 4. In the waveform diagram, you cannot distinguish this from receiving the frame only once, because the time of each frame reception is resampled into the waveform timing.

In the frame timeline, frame E is received twice in fast succession, once with signal values 7 and 8, then again with signals 5 and 6. These two frames are received within one sample of the waveform (within 1 ms). The effect on the data from `nxReadSignalWaveform` is that values for the first frame (7 and 8) are lost.

You can avoid the loss of signal data by setting the session resample rate to a high rate. NI-XNET timestamps receive frames to an accuracy of 100 ns. Therefore, if you use a resample rate of 1000000 (1 MHz), each frame's signal values are represented in the waveforms without loss of data. Nevertheless, using a high resample rate can result in a large amount of duplicated (redundant) values. For example, if the resample rate is 1000000, a frame that occurs once per second results in one million duplicated signal values. This tradeoff between accuracy and efficiency is a disadvantage of the Signal Input Waveform mode.

The Signal Input XY mode does not have the disadvantages mentioned previously. The signal value timing is a direct reflection of received frames, and no resampling occurs. Signal Input XY mode provides the most efficient and accurate representation of a sequence of received signal values.

One of the disadvantages of Signal Input XY mode is that the samples are not equidistant in time.

In summary, when reading a sequence of received signal values, use Signal Input Waveform mode when you need to synchronize CAN/FlexRay/LIN data with DAQmx analog/digital input waveforms or display CAN/FlexRay/LIN data. Use Signal Input XY mode when you need to analyze CAN/FlexRay/LIN data, for validation purposes.

Related concepts:

- Session Modes
- Cyclic and Event Timing
- Signal Input XY Mode
- State Models

Signal Input XY Mode

For each frame received, this mode provides the frame signals as a timestamp/value pair. This is the recommended mode for reading a sequence of all signal values.

The timestamp represents the absolute time when the XNET interface received the frame (end of frame), accurate to microseconds.

Use `nxReadSignalXY` for this mode.

The data consists of two two-dimensional arrays, one for timestamp and one for value.

Each timestamp/value pair represents a value from a received frame. When signals exist in different frames, the array size may be different from one signal to another.

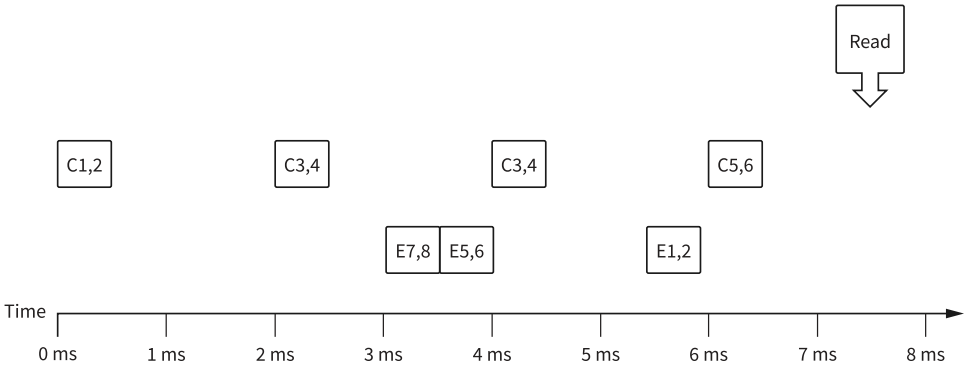
The received frames for this mode are stored in queues to avoid signal data loss.

Example

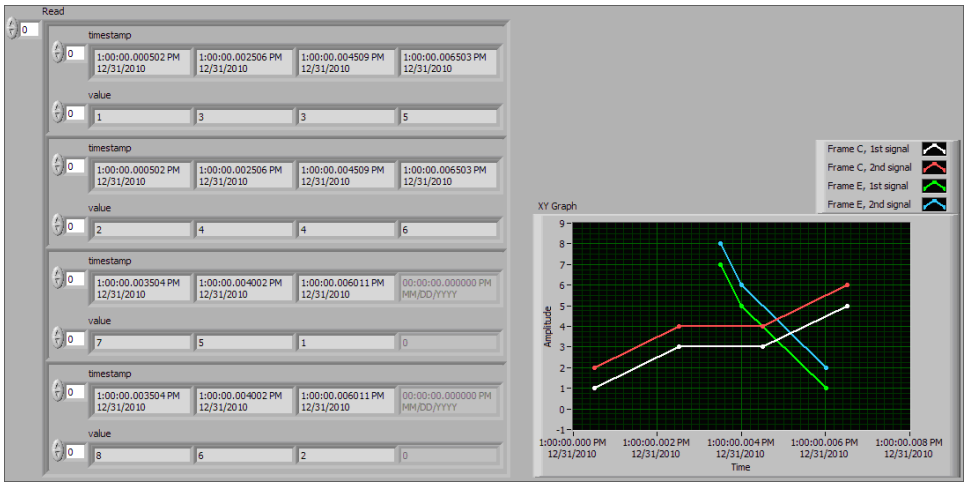
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing.

Each frame contains two signals, one in the first byte and another in the second byte.

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network, followed by a single call to `nxReadSignalXY`. Each frame contains its name (C or E), followed by the value of its two signals.



The following figure shows the data returned from `nxReadSignalXY`. The session contains all four signals.



Frame C was received four times, resulting in four valid values for the first two signals. Frame E was received three times, resulting in three valid values for the second two signals. The timestamp and value arrays are the same size for each signal. The timestamp represents the end of frame, to microsecond accuracy.

The XY Graph displays the data from `nxReadSignalXY`. This display is an accurate representation of signal changes on the network.

Related concepts:

- Session Modes
- Signal Input Waveform Mode
- Cyclic and Event Timing

Signal Output Single-Point Mode

This mode writes signal values for the next frame transmit. It typically is used for control or simulation applications, such as Hardware In the Loop (HIL).

This mode does not use queues to store signal values. If `nxWriteSignalSinglePoint` is called twice before the next transmit, the transmitted frame uses signal values from the second call to `nxWriteSignalSinglePoint`.

Use `nxWriteSignalSinglePoint` for this mode.

You also can specify a trigger signal for a frame. This signal name is `:trigger:`.

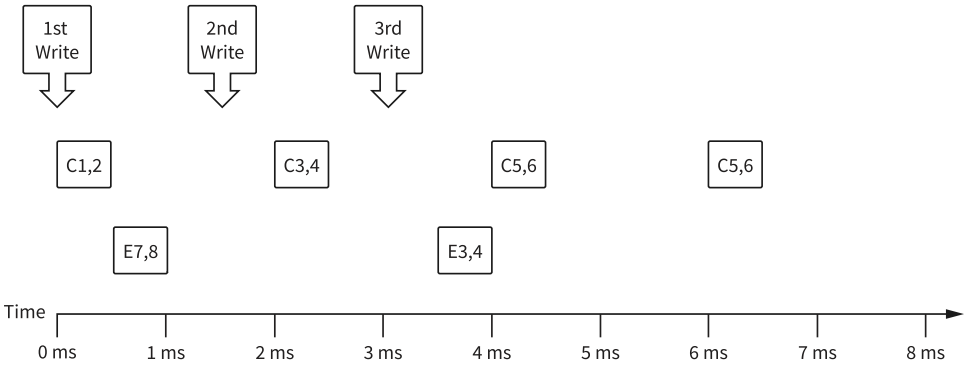
<frame name> , and once it is specified in the nxCreateSession signal list, you can write a value of 0.0 to suppress writing of that frame, or any value not equal to 0.0 to write the frame. You can specify multiple trigger signals for different frames in the same session.

Example

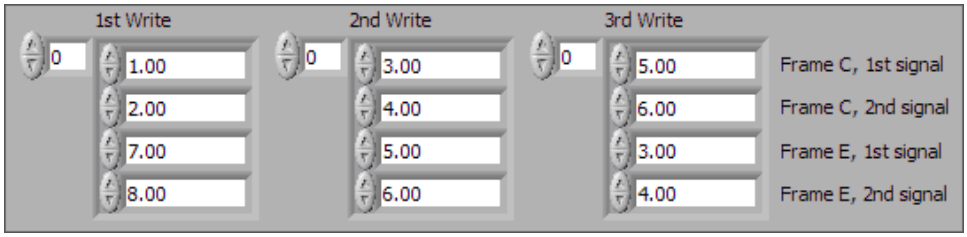
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing.

Each frame contains two signals, one in the first byte and another in the second byte.

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network. Each frame contains its name (C or E), followed by the value of its two signals. The timeline shows three calls to nxWriteSignalSinglePoint.



The following figure shows the data provided to each of the three calls to nxWriteSignalSinglePoint. The session contains all four signals.



Assuming the Auto Start? property uses the default of true, the session starts within the first call to nxWriteSignalSinglePoint. Frame C transmits followed by

frame E, both using signal values from the first call to `nxWriteSignalSinglePoint`.

If a transmitted frame contains a signal not included in the output session, that signal transmits its default value. If a transmitted frame contains bits no signal uses, those bits transmit the default payload.

After the second call to `nxWriteSignalSinglePoint` , frame C transmits using its values (3 and 4), but frame E does not transmit, because its minimal interval of 2.5 ms has not elapsed since acknowledgment of the previous transmit.

Because the third call to `nxWriteSignalSinglePoint` occurs before the minimum interval elapses for frame E, its next transmit uses its values (3 and 4). The values for frame E in the second call to `nxWriteSignalSinglePoint` are not used.

Frame C transmits the third time using values from the third call to the `nxWriteSignalSinglePoint` (5 and 6). Because frame C is cyclic, it transmits again using the same values (5 and 6).

Related concepts:

- Session Modes
- Cyclic and Event Timing

Signal Output Waveform Mode

Using the time when the signal frame is transmitted according to the database, this mode 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.

The resampling translates from the waveform timing to each frame's transmit timing. When the time for the frame to transmit occurs, it uses the most recent signal values in the waveform that correspond to that time.

Use `nxWriteSignalWaveform` for this mode.

You specify the resample rate using the Resample Rate property.

The frames for this mode are stored in queues.

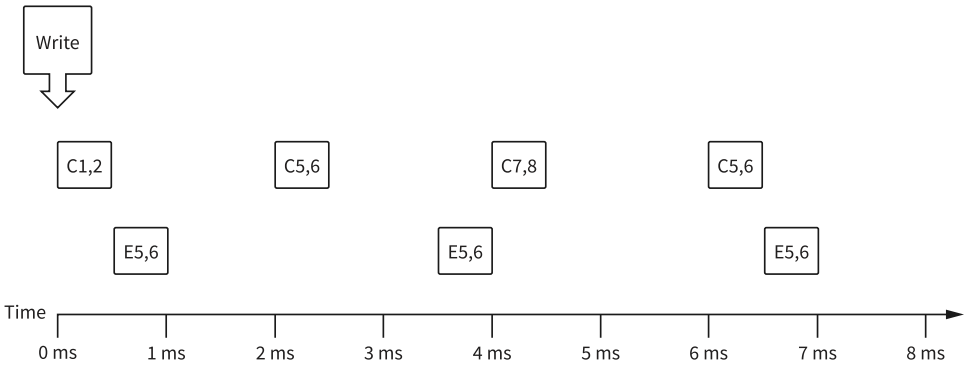
This mode is not supported for a LIN interface operating as slave. For more information, refer to LIN Frame Timing and Session Mode.

Example

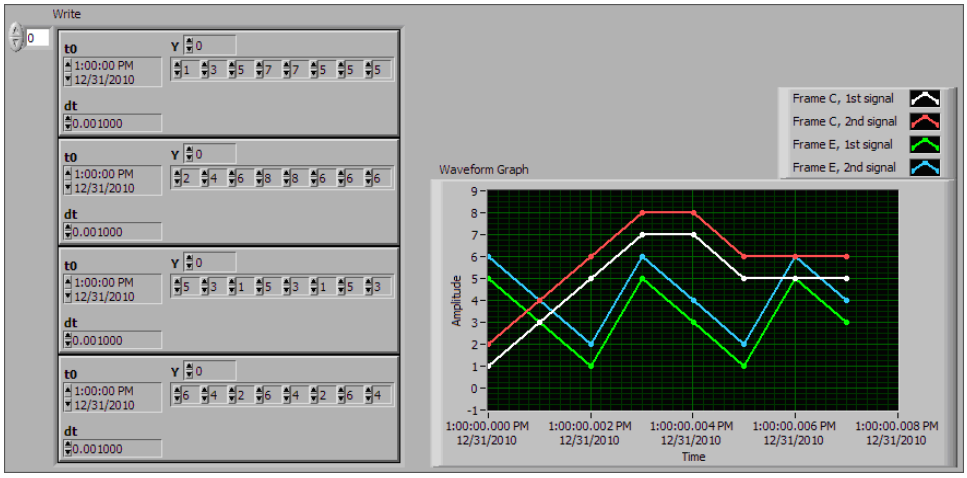
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing.

Each frame contains two signals, one in the first byte and another in the second byte.

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network. Each frame contains its name (C or E), followed by the value of its two signals. The timeline begins with a single call to `nxWriteSignalWaveform`.



The following figure shows the data provided to the call to `nxWriteSignalWaveform`. The session contains all four signals and uses the default resample rate of 1000.0 samples per second.



Assuming the Auto Start? property uses the default of true, the session starts within the call to `nxWriteSignalWaveform`. Frame C transmits followed by frame E, both using signal values from the first sample (index 0 of all four Y arrays).

The waveform elements `t0` (timestamp of first sample) and `dt` (time between samples in seconds) are ignored for the call to `nxWriteSignalWaveform`. Transmit of frames starts as soon as the XNET session starts. The frame properties in the database determine the each frame's transmit time. The session resample rate property determines the time between waveform samples.

In the waveforms, the sample at index 1 occurs at 1.0 ms in the frame timeline. According to the database, frame C transmits once every 2.0 ms, and frame E is limited to an event-driven transmit with interval 2.5 ms. Therefore, the sample at index 1 cannot be resampled to a transmitted frame and is discarded.

Index 2 in the waveforms occurs at 2.0 ms in the frame timeline. Frame C is ready for its next transmit at that time, so signal values 5 and 6 are taken from the first two Y arrays and used for transmit of frame C. Frame E still has not reached its transmit time of 2.5 ms from the previous acknowledgment, so signal values 1 and 2 are discarded.

At index 3, frame E is allowed to transmit again, so signal values 5 and 6 are taken from the last two Y arrays and used for transmit of frame E. Frame C is not ready for its next transmit, so signal values 7 and 8 are discarded.

This behavior continues for Y array indices 4 through 7. For the cyclic frame C, every second sample is used to transmit. For the event-driven frame E, every sample is interpreted as an event, such that every third sample is used to transmit.

Although not shown in the frame timeline, frame C transmits again at 8.0 ms and every 2.0 ms thereafter. Frame C repeats signal values 5 and 6 until the next call to `nxWriteSignalWaveform`. Because frame E is event driven, it does not transmit after the timeline shown, because no new event has occurred.

Because the waveform timing is fixed, you cannot use it to represent events in the data. When used for event driven frames, the frame transmits as if each sample was an event. This mismatch between frame timing and waveform timing is a disadvantage of the Signal Output Waveform mode.

When you use the Signal Output XY mode, the signal values provided to `nxWriteSignalXY` are mapped directly to transmitted frames, and no resampling occurs. Unless your application requires correlation of output data with DAQmx waveforms, Signal Output XY is the recommended mode for writing a sequence of signal values.

Related concepts:

- Session Modes
- LIN Frame Timing and Session Mode
- Cyclic and Event Timing
- Signal Output XY Mode

Signal Output XY Mode

This mode provides a sequence of signal values for transmit using each frame's timing as specified in the database. This is the recommended mode for writing a sequence of all signal values.

Use `nxWriteSignalXY` for this mode. The timestamp array is unused (reserved).

Each signal value is mapped to a frame for transmit. Therefore, the array of signal values is mapped to an array of frames to transmit. When signals exist in the same frame, signals at the same index in the arrays are mapped to the same frame. When signals exist in different frames, the array size may be different from one cluster (signal) to another.

The frames for this mode are stored in queues, such that every signal provided is

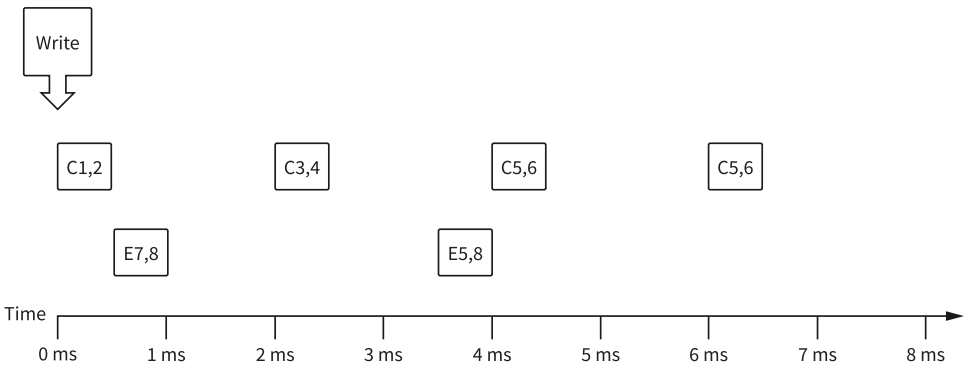
transmitted in a frame.

Examples

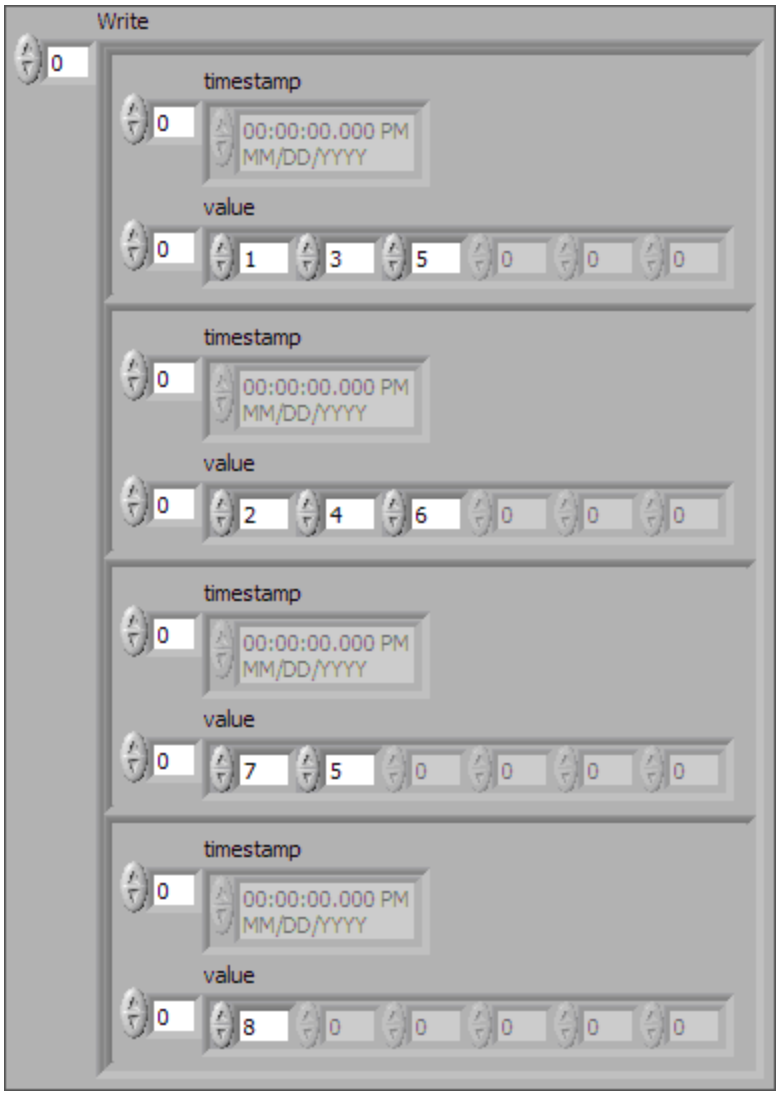
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to Cyclic and Event Timing.

Each frame contains two signals, one in the first byte and another in the second byte.

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network. Each frame contains its name (C or E), followed by the value of its two signals. The timeline begins with a single call to `nxWriteSignalXY`.



The following figure shows the data provided to `nxWriteSignalXY`. The session contains all four signals.



Assuming the Auto Start? property uses the default of true, the session starts within a call to `nxWriteSignalXY`. This occurs at 0 ms in the timeline. Frame C transmits followed by frame E, both using signal values from the first sample (index 0 of all four Y arrays).

According to the database, frame C transmits once every 2.0 ms, and frame E is limited to an event-driven interval of 2.5 ms.

At 2.0 ms in the timeline, signal values 3 and 4 are taken from index 1 of the first two Y arrays and used for transmit of frame C.

At 3.5 ms in the timeline, signal value 5 is taken from index 1 of the third Y array. Because this is a new value for frame E, it represents a new event, so the frame transmits again. Because no new signal value was provided at index 1 in the fourth

array, the second signal of frame E uses the value 8 from the previous transmit.

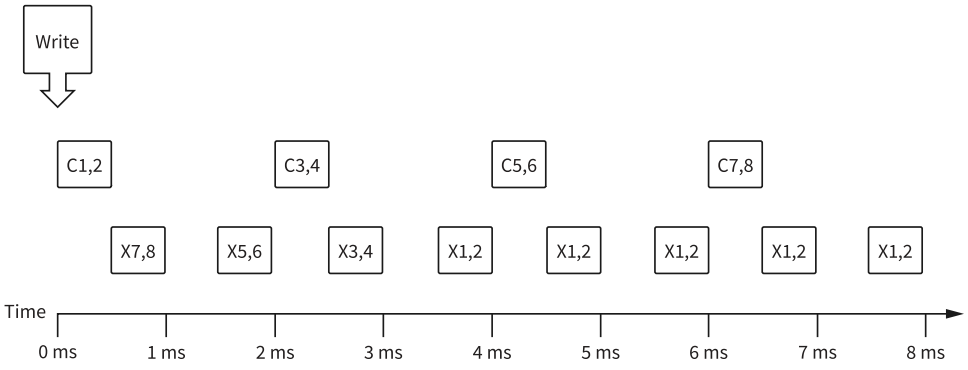
At 4.0 ms in the timeline, signal values 5 and 6 are taken from index 2 of the first two Y arrays and used for transmit of frame C.

Because there are no more signal values for frame E, this frame no longer transmits. Frame E is event driven, so new signal values are required for each transmit.

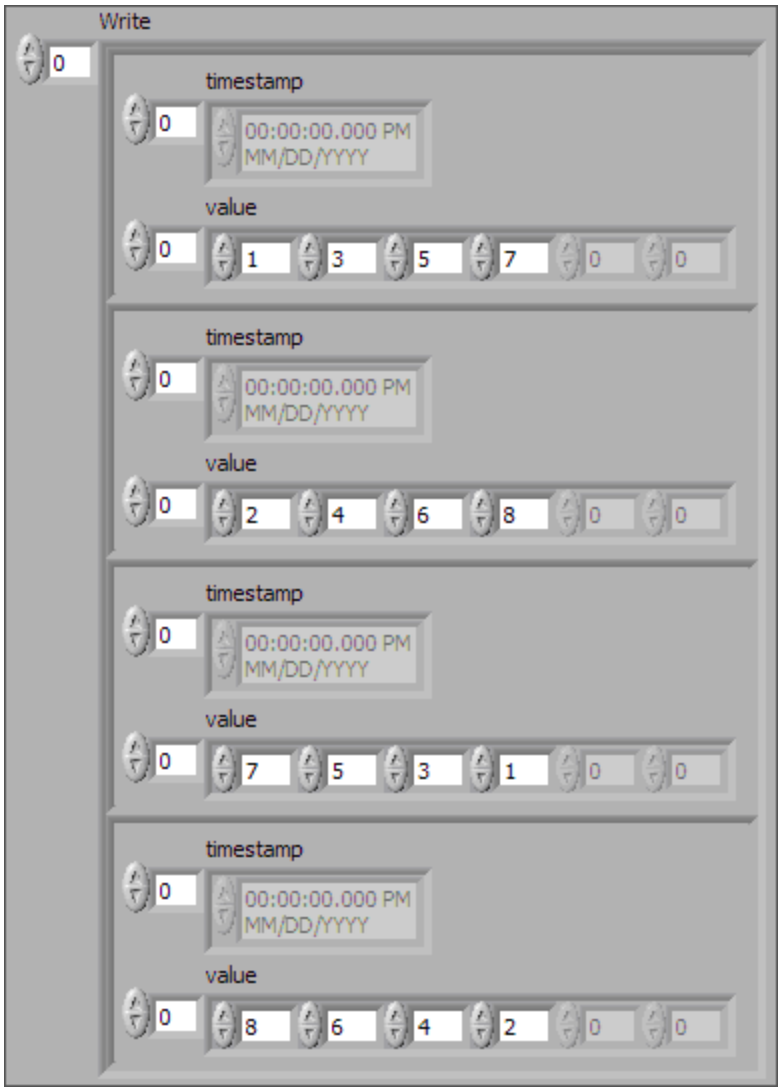
Because frame C is a cyclic frame, it transmits repeatedly. Although there are no more signal values for frame C, the values of the previous frame are used again at 6.0 ms in the timeline and every 2.0 ms thereafter. If `nxWriteSignalXY` is called again, the new signal values are used.

The next example network demonstrates a potential problem that can occur with Signal Output XY mode.

In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame X is a cyclic frame that transmits on the network once every 1.0 ms. Each frame contains two signals, one in the first byte and another in the second byte. The timeline begins with a single call to `nxWriteSignalXY`.



The following figure shows the data provided to `nxWriteSignalXY`. The session contains all four signals.



The number of signal values in all four Y arrays is the same. The four elements of the arrays are mapped to four frames. The problem is that because frame X transmits twice as fast as frame C, the frames for the last two arrays transmit twice as fast as the frames for the first two arrays.

The result is that the last pair of signals for frame X (1 and 2) transmit over and over, until the timeline has completed for frame C. This sort of behavior usually is unintended. The Signal Output XY mode goal is to provide a complete sequence of signal values for each frame.

The best way to resolve this issue is to provide a different number of values for each signal, such that the number of elements corresponds to the timeline for the corresponding frame. If the previous call to `nxWriteSignalXY` provided eight elements for frame X (last two Y arrays) instead of just four elements, this would have

created a complete 8.0 ms timeline for both frames.

Although you need to resolve this sort of timeline for cyclic frames, this is not necessarily true for event-driven frames. For an event-driven frame, you may decide simply to pass either zero or one set of signal values to `nxWriteSignalXY`. When you do this, each call to `nxWriteSignalXY` can generate a single event, and the overall timeline is not a major consideration.

Related concepts:

- Session Modes
- Signal Output Waveform Mode
- Cyclic and Event Timing

Conversion Mode

This mode is intended to convert NI-XNET signal data to frame data or vice versa. It does not use any NI-XNET hardware, and you do not specify an interface when creating this mode.

Conversion occurs with the `nxConvertFramesToSignalsSinglePoint` or `nxConvertSignalsToFramesSinglePoint` functions. None of the Read or Write functions work with this mode; they return an error because hardware I/O is not permitted.

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.

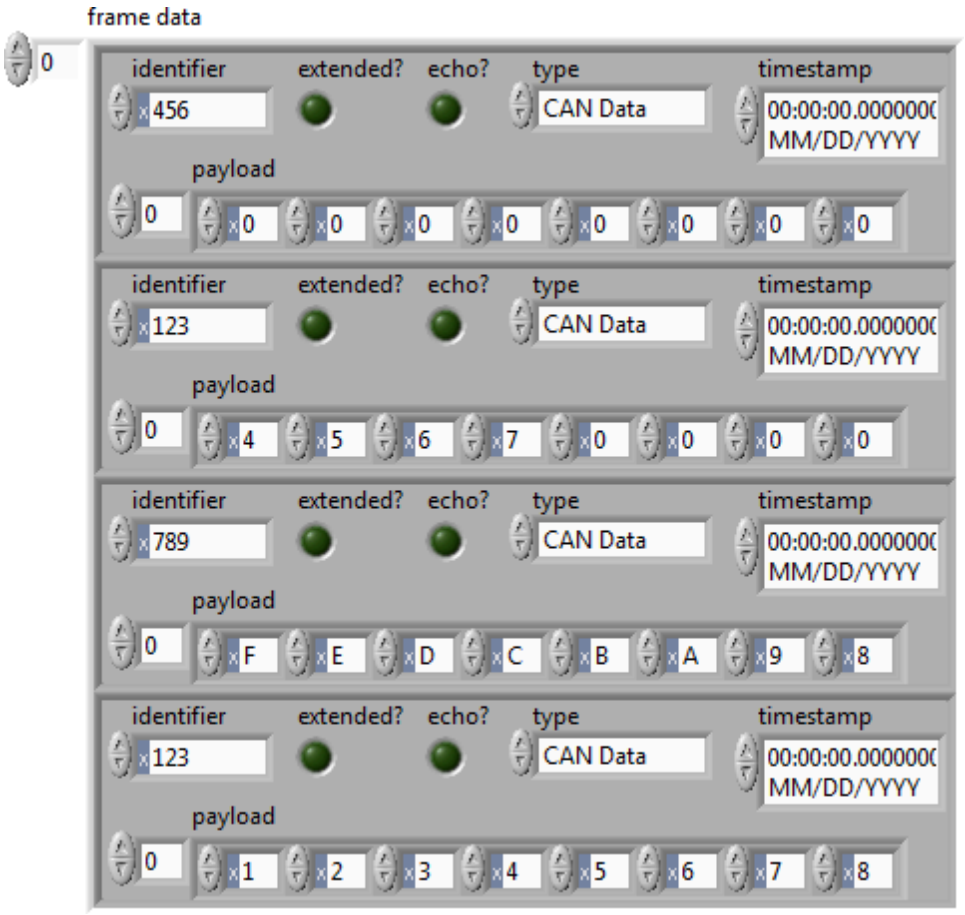
In addition, the `nxConvertFramesToByteArraySinglePoint` and `nxConvertByteArrayToFramesSinglePoint` functions allow for raw byte extraction/insertion of the signal bytes from/to a frame. For this mode, the conversion session must only span one signal, and this signal must be byte aligned (both start bit and number of bits). If these conditions are not met, the functions will return an error. Byte ordering is ignored in this case; the bytes are transferred in ascending order from/to the frame. This mode will work for signals >64 bits as well; it is the only way of

accessing such signals.

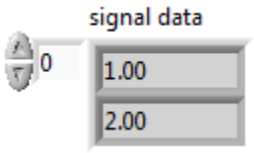
Example 1: Conversion of CAN Frames to Signals

Suppose you have a database with a CAN frame with ID 0x123 and two unsigned byte signals assigned to it (byte 1 and byte 2).

Creating an appropriate conversion session and calling `nxConvertFramesToSignalsSinglePoint` with the following input



results in the following signal values being returned:



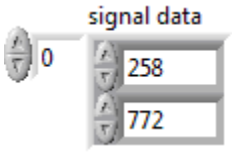
Explanation: The data are taken from frame 4. Frames 1 and 3 are ignored because

they have a wrong (unmatched) ID. Frame 2 is ignored because its data are overwritten later with the values from frame 4, because frames are processed in the order of input.

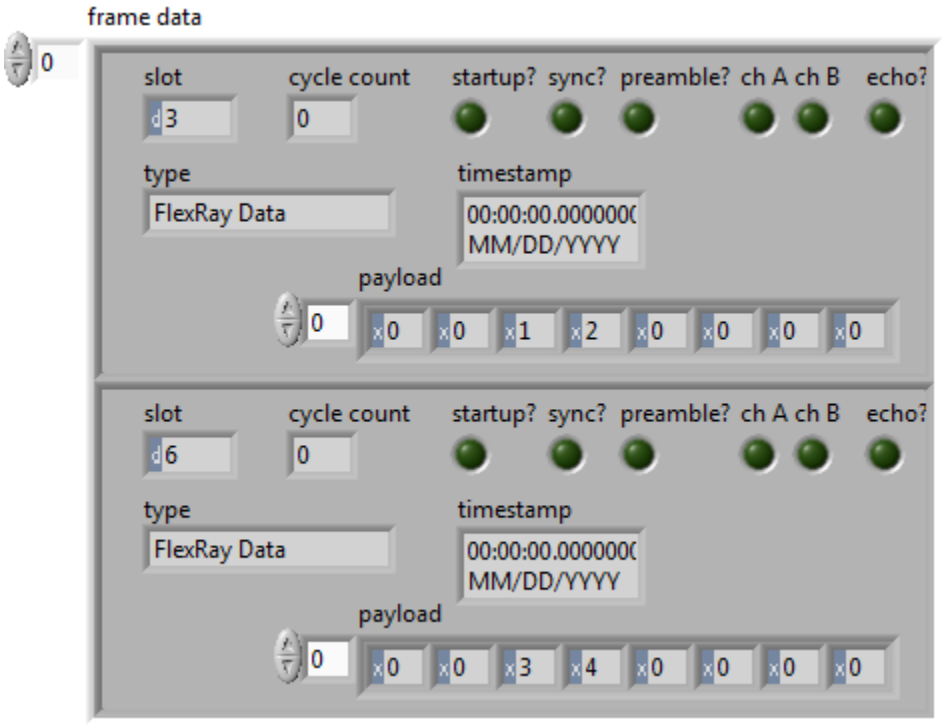
Example 2: Conversion of Signals to FlexRay Frames

Suppose you have two FlexRay frames with slot ID 3 and 6, and each one has assigned a two-byte, Big Endian signal at byte 2 and 3 (zero based). Suppose also that all relevant default values of other signals in the frame are 0.

Creating an appropriate conversion session and calling `nxConvertSignalsToFramesSinglePoint` with the following input



causes the following frames to be generated:



Explanation: The first signal is converted to the byte sequence 0x01, 0x02 (1 x 256 + 2), and the byte sequence is placed at byte 2 of the frame with slot ID 3. The second signal is converted to byte sequence 0x03, 0x04 (3 x 256 + 4) and placed at byte 2 of the frame

with slot ID 6. All other data are filled with the default values (0).

Related concepts:

- Session Modes

How Do I Create a Session?

There are two methods for creating a session: a LabVIEW project and the XNET Create Session VI. You typically use only one method to create all sessions for your application.

LabVIEW Project

Using LabVIEW project sessions is best suited for applications that are static, in that the network data does not change from one execution to the next. Refer to Getting Started for a description of creating a session in a LabVIEW project.

When you configure the session in a LabVIEW project, you select the interface, mode, and database objects with the NI-XNET user interface. The database objects (cluster, frames, and signals) must exist in a file. If you do not already have a database file, you can create one using the NI-XNET Database Editor , which you can launch from NI-XNET user interface.

XNET Create Session VI

You can use the XNET Create Session VI to create NI-XNET sessions at run time. This run-time creation has advantages over a LabVIEW project, because the end user of your application can configure sessions from the front panel. The disadvantage is that the VI diagram is more complex.

If your application is used for a specific product (for example, an instrument panel for a specific make/model/year car), and the front panel must be simple (for example, a test button with a pass/fail LED), a LabVIEW project is the best method to use for NI-XNET sessions. Because the configuration does not change, a LabVIEW project provides the easiest programming model.

If your application is used for many different products (for example, a test system for

an engine in any make/model/year car), the XNET Create Session VI is the best method to use for NI-XNET sessions. On the front panel, the application end user can provide a database file and select the specific frames or signals to read and/or write.

The XNET Create Session VI takes inputs for the interface, mode, and database objects. You select the interface using techniques described in How Do I View Available Interfaces?. The database objects depend on the mode (for example, Signal Input Waveform requires an array of signals). You select the database objects using techniques described in Database Programming.

Related concepts:

- Getting Started with NI-XNET C API
- Displaying Available Interfaces
- Database Programming for the C API

State Models

The following figures show the state model for the NI-XNET session and the associated NI-XNET interface.

The session controls the transfer of frame values between the interface (network) and the data structures that Read or Write access. In other words, the session controls the receipt or transmission of specific frames for the session.

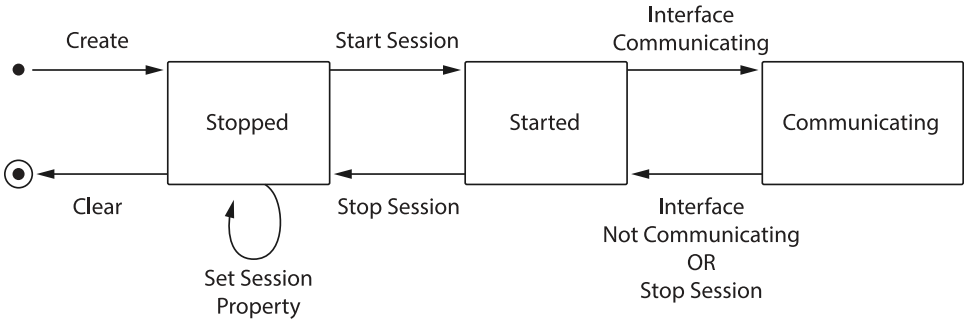
The interface controls communication on the physical network cluster. Multiple sessions can share the interface. For example, you can use one session for input on interface CAN1 and a second session for output on interface CAN1.

Although most state transitions occur automatically when you call the XNET Read or Write VI (LabVIEW) or the appropriate `nxRead` or `nxWrite` function (C), you can perform a more specific transition using XNET Start and XNET Stop IVs (LabVIEW) or `nxStart` and `nxStop` (C). If you invoke a transition that has already occurred, the transition is not repeated, and no error is returned.

Session State Model

For a description of each state, refer to Session States. For a description of each

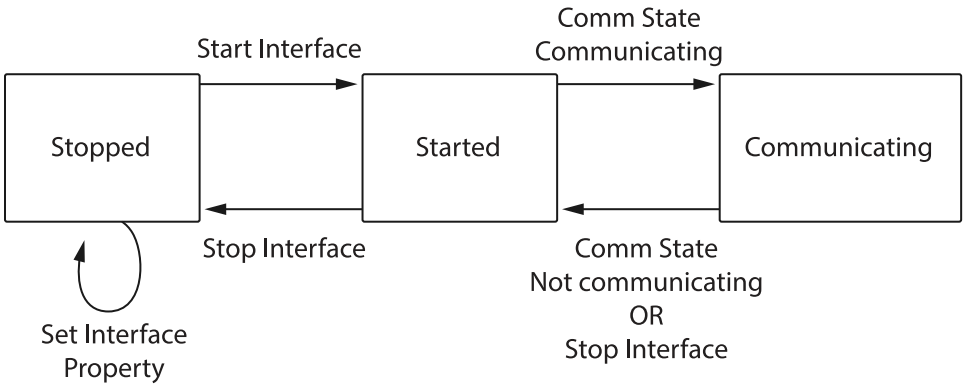
transition, refer to Session Transitions.



Note Starting a Signal Input Waveform session discards any previous samples and frames--the same result as running the XNET Flush VI (LabVIEW) or nxFlush (C). Note that when calling the XNET Read (Signal Waveform) VI (LabVIEW) or nxReadSignalWaveform function for the first time on the session, the session will be started if it was not already. Stopping the session after the first start requires the session to be explicitly started in the future.

Interface State Model

For a description of each state, refer to Interface States. For a description of each transition, refer to Interface Transitions.



Related concepts:

- Session States
- Session Transitions (C API)
- Signal Input Waveform Mode
- Interface States

- Interface Transitions

Session States

Stopped

The session initially is created in the stopped state. In the stopped state, the session does not transfer frame values to or from the interface.

While the session is stopped, you can change properties specific to this session. You can set any Session property except those in the Interface category (refer to Stopped in Interface States).

While the session is Started, you cannot change properties of objects in the database, such as frames or signals. The properties of these objects are committed when the session is created.

Started

In the started state, the session is started, but is waiting for the associated interface to be started also. The interface must be communicating for the session to exchange data on the network.

For most applications, the Started state is transitory in nature. When you call the XNET Read, Write, or Start VI (LabVIEW) or the appropriate `nxRead` or `nxWrite` function or `nxStart` using defaults (C), the interface is started along with the session. Once the interface is communicating, the session automatically transitions to communicating without interaction by your application.

If you call the XNET Start VI or `nxStart` function with the scope of Session Only, the interface is not started. You can use this advanced feature to prepare multiple sessions for the interface, then start communication for all sessions together by starting the interface (XNET Start VI or `nxStart` with scope of Interface Only).

Communicating

In the Communicating state, the session is communicating on the network with remote ECUs. Frame or signal values are received for an input session. Frame or signal

values are transmitted for an output session. Your application accesses these values using the appropriate or XNET Read or Write VI or function.

Related concepts:

- Interface States
- State Models

Session Transitions in LabVIEW

Create

When the session is created, the database, cluster, and frame properties are committed to the interface. For this configuration to succeed, the interface must be in the Stopped state. There is one exception: You can create a Frame Stream Input session while the interface is communicating.

There are two ways to create a session:

- **Create Session VI method**: When your application calls the XNET Create Session VI, the session is created. To ensure that all sessions for the interface are created prior to start, you typically wire all Create Session VIs in sequence prior to the first use of the XNET Read or XNET Write VI (for example, prior to the main loop).
- **LabVIEW project method**: Although you specify the session properties in the LabVIEW project user interface, the session is not created at that time. When you run a VI that uses the session with an XNET node (property node or VI), the session is created. In addition, all other sessions in the LabVIEW project that use the same interface and cluster (database) are created at that time. This ensures that all project-based sessions your application uses are created before the interface starts (for example, the first call to the XNET Read or XNET Write VI).

Clear

When the session is cleared, it is stopped (no longer communicates), and then all its resources are removed.

There are two ways to clear a session:

- **Application stop method**: The typical way to clear a session is to do nothing explicit in your application. When the application stops execution, NI-XNET automatically clears all sessions that application uses. When using the LabVIEW development environment, the application stops when the top-level VI goes idle, including when you select the LabVIEW abort button in that VI's toolbar. When using an application built using a LabVIEW project, the application stops when the executable exits.
- **XNET Clear VI method**: This clears the session explicitly. To change the properties of database objects that a session uses, you may need to call the XNET Clear VI to change those properties, then recreate the session.

Set Session Property

While the session is stopped, you can change properties specific to this session. You can set any property in the XNET Session Node except those in the Interface category (refer to Stopped in Interface States).

You cannot set properties of a session in the Started or Communicating state. If there is an exception for a specific property, the property help states this.

Start Session

For an input session, you can start the session simply by calling the XNET Read VI. To read received frames, the XNET Read VI performs an automatic Start of scope Normal, which starts the session and interface.

For an output session, if you leave the Auto Start? property at its default value of true, you can start the session simply by calling the XNET Write VI. The auto-start feature of Write performs a Start of scope Normal, which starts the session and interface.

To start the session prior to calling the XNET Read VI or XNET Write VI, you can call the XNET Start VI. The XNET Start VI default scope is Normal, which starts the session and interface. You also can use the XNET Start VI with scope of Session Only (this Start Session transition) or Interface Only (the interface Start Interface transition).

Stop Session

You can stop the session by calling the XNET Clear or XNET Stop VI. The XNET Stop VI

provides the same scope as the XNET Start VI, allowing you to stop the session, interface, or both (normal scope).

When the session stops, the underlying queues are not flushed. For example, if an input session receives frames, and then you call the XNET Stop VI, you still can call the XNET Read VI to read the frame values from the queues. To discard session frame queues, call the XNET Flush VI (or XNET Clear VI).

Interface Communicating

This transition occurs when the session interface enters the Communicating state.

Interface Not Communicating

This transition occurs when the session interface exits the Communicating state.

The session also exits its Communicating state when the session stops due to the XNET Clear or XNET Stop VI.

Related concepts:

- Interface States

Session Transitions (C API)

Create

When the session is created, the database, cluster, and frame properties are committed to the interface. For this configuration to succeed, the interface must be in the Stopped state. There is one exception: You can create a Frame Stream Input session while the interface is communicating.

When your application calls `nxCreateSession`, the session is created. To ensure that all sessions for the interface are created prior to start, you typically place all calls to `nxCreateSession` in sequence prior to the first use of the appropriate `nxRead` or `nxWrite` function (for example, prior to the main loop).

Clear

When the session is cleared, it is stopped (no longer communicates), and then all its resources are removed. This clears the session explicitly. To change the properties of database objects that a session uses, you may need to call `nxdbSetProperty` to change those properties, then recreate the session.

Set Session Property

While the session is Stopped, you can change properties specific to this session. You can set any XNET Session Properties except those in the Interface category (refer to Stopped in Interface States).

You cannot set properties of a session in the Started or Communicating state. If there is an exception for a specific property, the property help states this.

Start Session

For an input session, you can start the session simply by calling the appropriate `nxRead` function. To read received frames, the appropriate `nxRead` function performs an automatic Start of scope Normal, which starts the session and interface.

For an output session, if you leave the Auto Start? property at its default value of true, you can start the session simply by calling the appropriate `nxWrite` function. The auto-start feature of the appropriate `nxWrite` function performs a Start of scope Normal, which starts the session and interface.

To start the session prior to calling the appropriate `nxRead` or `nxWrite` function, you can call `nxStart`. The `nxStart` default scope is Normal, which starts the session and interface. You also can use `nxStart` with scope of Session Only (this Start Session transition) or Interface Only (the interface Start Interface transition).

Stop Session

You can stop the session by calling `nxStop`. `nxStop`provides the same scope as `nxStart`, allowing you to stop the session, interface, or both (normal scope).

When the session stops, the underlying queues are not flushed. For example, if an

input session receives frames, and then you call `nxStop`, you still can call the appropriate `nxRead` function to read the frame values from the queues. To discard session frame queues, call `nxFlush`.

Interface Communicating

This transition occurs when the session interface enters the Communicating state.

Interface Not Communicating

This transition occurs when the session interface exits the Communicating state.

The session also exits its Communicating state when the session stops due to `nxStop`.
Related concepts:

- Interface States
- State Models

Interface States

Stopped

The interface always exists, because it represents the communication controller of the NI-XNET hardware product port. This physical port is wired to a cable that connects to one or more remote ECUs.

The NI-XNET interface initially powers on in the Stopped state. In the Stopped state, the interface does not communicate on its port.

While the interface is stopped, you can change properties specific to the interface. These properties are contained within the Session Node (LabVIEW) or in the Session Property Interface Properties. When more than one session exists for a given interface, the Interface category properties provide shared access to the interface configuration. For example, if you set an interface property using one session, then get that same property using a second session, the returned value reflects the change.

Properties that you change in the interface are not saved from one execution of your application to another. When the last session for an interface is cleared, the interface

properties are restored to defaults.

Started

In the Started state, the interface is started, but it is waiting for the associated communication controller to complete its integration with the network.

This state is transitory in nature, in that your application does not control transition out of the Started state. For CAN and LIN, integration with the network occurs in a few bit times, so the transition is effectively from Stopped to Communicating. For FlexRay, integration with the network entails synchronization with global FlexRay time, which can take as long as hundreds of milliseconds.

Communicating

In the Communicating state, the interface is communicating on the network. One or more communicating sessions can use the interface to receive and/or transmit frame values.

The interface remains in the Communicating state as long as communication is feasible. For information about how the interface transitions in and out of this state, refer to "Comm State Communicating" and "Comm State Not Communicating" sections of ***Interface Transitions*** in this help.

In LabVIEW, the Communicating state behaves differently for Ethernet as compared to other XNET protocols (e.g., CAN). For more information, refer to the Ethernet Operational Status property in the NI-XNET API reference.

Related concepts:

- Session States
- Session Transitions (C API)
- State Models
- Session Transitions in LabVIEW

Interface Transitions

Set Interface Property

While the interface is stopped, you can change interface-specific properties. These properties are in the Session Node category in LabVIEW or the Session Property Interface Properties (C). When more than one session exists for a given interface, the Interface category properties provide shared access to the interface configuration. For example, if you set an interface property using one session, then get that same property using a second session, the returned value reflects the change.

You cannot set properties of the interface while it is in Started or Communicating state. If there is an exception for a specific property, the property help states this.

Start Interface

You can request the interface start in two ways:

- XNET Read or XNET Write VI or the appropriate `nxRead` or `nxWrite` function method: The automatic start described for the Start Session transition uses a scope of Normal, which requests the interface and session start.
- XNET Start VI or `nxStart` method: If you call this function with scope of Normal or Interface Only, you request the interface start.

After you request the interface start, the actual transition depends on whether you have connected the interface start trigger. You connect the start trigger by calling the XNET Connect Terminals VI or `nxConnectTerminals` with a destination of Interface Start Trigger with a destination of Interface Start Trigger or by writing the XNET Session Interface:Source Terminal:Start Trigger property.

The Start Interface transition occurs as follows, based on the start trigger connection:

- **Disconnected (default)**: Start Interface occurs as soon as it is requested (a Read, Write, or Start VI or function).
- **Connected**: Start Interface occurs when the connected source terminal transitions low-to-high (for example, pulses). Every Start Interface transition requires a new low-to-high transition, so if your application stops the interface (for example, Stop VI or `nxStop`), then restarts the interface, the connected source terminal must transition low-to-high again.

Stop Interface

Under normal conditions, the interface is stopped when the last session is stopped (or cleared). In other words, the interface communicates as long as at least one session is in use.

If a significant number of errors occur on the network, the communication controller may stop the interface on its own. For more information, refer to ***Comm State Not Communicating***.

If your application calls XNET Stop VI or `nxStop` with scope of **Interface Only**, that immediately transitions the interface to the Stopped state. Use this feature with care, because it affects all sessions that use the interface and is not limited to the session passed to the stop method. In other words, using XNET Stop VI or `nxStop` with a scope of **Interface Only** stops communication by all sessions simultaneously.

Comm State Communicating

This transition occurs when the interface is integrated with the network.

For CAN, this occurs when communication enters Error Active or Error Passive state. For information about the specific CAN interface communication states, refer to XNET Read VI or `nxReadState` in the NI-XNET API reference.

For FlexRay, this occurs when communication enters one Normal Active or Normal Passive state. For information about the specific FlexRay interface communication states, refer to XNET Read VI or `nxReadState` in the NI-XNET API reference.

For LIN, this occurs when communication enters the Active state. The interface remains communicating while in the Active or Inactive state (not affected by bus activity). For more information about the specific LIN interface communication states, refer to XNET Read VI or `nxReadState` in the NI-XNET API reference..

Comm State Not Communicating

This transition occurs when the interface is no longer integrated with the network.

For CAN, this occurs when communication enters Bus Off or Idle state. For information

about the specific CAN interface communication states, refer to XNET Read (State CAN Comm) VI or `nxReadState` in the NI-XNET API reference.

For FlexRay, this occurs when communication enters the Halt, Config, Default Config, or Ready state. For information about the specific FlexRay interface communication states, refer to XNET Read (State CAN Comm) VI or `nxReadState` in the NI-XNET API reference.

For LIN, this occurs when communication enters the Idle state. For more information about the specific LIN interface communication states, refer to XNET Read VI or `nxReadState` in the NI-XNET API reference.

Related concepts:

- State Models

PDUs

Introduction to Protocol Data Units

Protocol Data Units (PDUs) are encapsulated network data that are a way to communicate information between independent protocols, such as in a CAN-FlexRay gateway. You can think of them as containers of signals. The container (PDU) can be in multiple frames. A single frame can contain multiple PDUs.

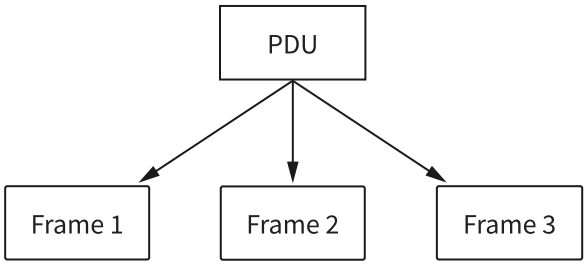
NI-XNET supports only a one-to-one relationship between frames and PDUs for CAN and LIN protocols, and does not support an update bit for PDUs.

Signals returned from the frame are the same as signals returned from the mapped PDU. In this case, you can deactivate the **Use PDUs** option in the NI-XNET Database Editor tool to hide PDUs. But if the database file contains frames with advanced PDU configuration (using a one-to-n or n-to-one relationship or update bits), the **Use PDUs** option cannot be deactivated.

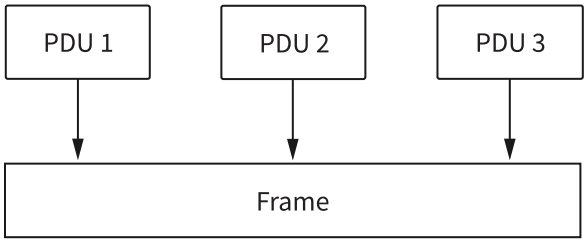
Note FIBEX files prior to version 3.0,.DBC files, and .NCD files cannot contain an advanced PDU configuration.

Frames and PDUs

The frame element contains an arbitrary number of non-overlapping PDUs. A frame can have multiple PDUs, and the same PDU can exist in different frames. The following figure shows the one-to-n (one PDU in n number of frames) and n-to-one (n number of PDUs in one frame) relationships.



One PDU in *n* (Three) Frames

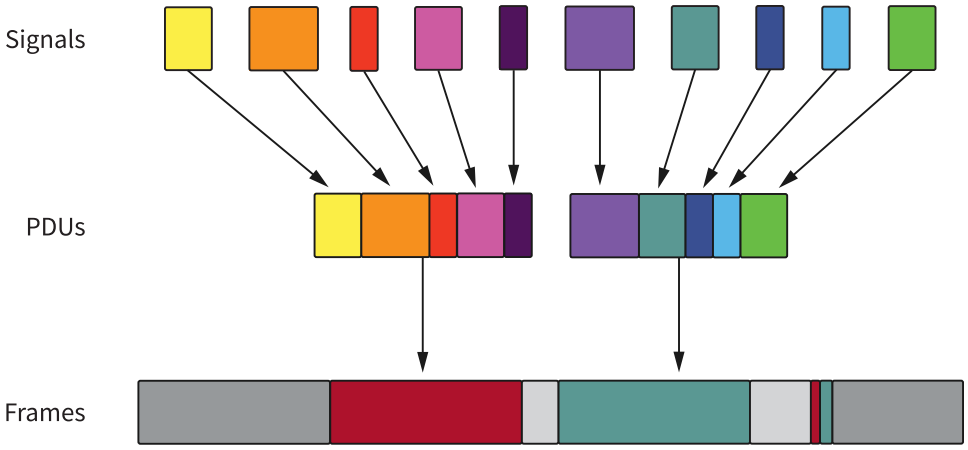


***n* (Three) PDUs in One Frame**

Signals and PDUs

A PDU acts like a container for a logical group of signals.

The following figure represents the relationship between frames, PDUs, and signals.



Protocol Data Unit Properties

Start Bit: The start bit of the PDU within the frame indicates where in the frame the particular PDU data starts.

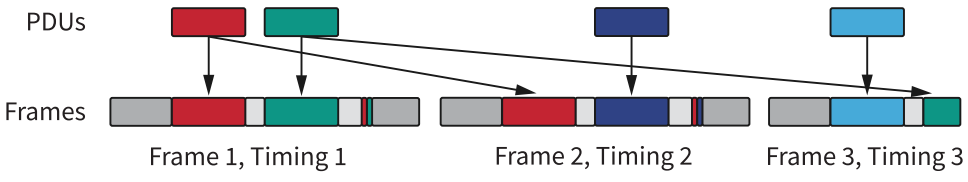
Note PDUs used in FlexRay I/O are required to have byte-aligned start bits.

Length: The PDU length defines the PDU size in bytes.

Update Bit: The receiver uses the update bit to determine whether the frame sender has updated data in a particular PDU. Update bits allow for the decoupling of a signal update from a frame occurrence. Update bits is an optional PDU property.

PDU Timing and Frame Timing

Because the same PDU can exist in multiple Frames, PDUs can have flexible transmission schedules. For example, if PDU A is present in Frame 1 (Timing 1) as well as in Frame 2 (Timing 2), the receiving node receives it as per the different timings of the containing frames. (Refer to the following figure.)



Programming PDUs with NI-XNET

You can use PDUs in two ways to create a session for read/write:

- Create a signal I/O session using signals within the PDU. To do this, use the signal name as you would with signals contained within a frame.
- Create an I/O session to read/write the raw PDU data. To do this, pass (in LabVIEW, wire) the PDU(s) to the special Create Session modes for PDU. These modes operate like the equivalent frame modes.

Important points to consider while programming with PDUs:

- PDUs currently are supported only on FlexRay interfaces.
- On the receive side, if the PDU has an update bit associated with it, the NI-XNET driver sets the update bit when new data is received for the particular PDU from the bus. Otherwise, if no new data is received for this PDU, the PDU is discarded. On the transmit side, the NI-XNET driver sets the update bit when it detects that new data is available for the particular PDU in the PDUs queue or table. The NI-XNET driver clears the bit if no new data is detected in the PDU queue or table. If the frame containing the PDUs has cyclic timing, even if no new data is available for any of the PDUs in the frame, the frame is transmitted across the bus with the update bits all cleared. However, if the PDU containing the frame has event timing, it is transmitted across the bus only if at least one PDU that it contains has new data (with update bit set).
- The read-only XNET Cluster PDUs Required? property is useful when programming traversal through the database, as it indicates whether to consider PDUs in the traversal.

Related concepts:

- Using FlexRay
- XNET PDU I/O Name

Frames

Each cluster can contain an arbitrary number of frames. A frame is a single message that is exchanged on the cluster. In NI-CAN, this is equivalent to an NI-CAN message.

The basic properties of a frame are its identifier (Arbitration ID for CAN, Slot ID for FlexRay) and the payload length, which can be any value between 0 and 8 for CAN and any even value between 0 and 254 for FlexRay.

In addition, several protocol-specific properties exist. You can use the NI-XNET Database Editor to edit these properties in a protocol type-specific way.

Raw Frame Format

This topic describes the raw data format for frames. The `nxReadFrame` and `nxWriteFrame` functions and XNET Read (Frame Raw) and XNET Write (Frame Raw) VIs use this format

The raw frame format is ideal for log files, because you can transfer the data between NI-XNET and the file with very little conversion.

The raw frame format consists of one or more frames encoded in a sequence of bytes. The encoding can be different for each protocol supported by NI-XNET.

Related concepts:

- Frame Input Queued Mode
- Frame Input Single-Point Mode
- Frame Input Stream Mode
- Frame Output Queued Mode
- Frame Output Single-Point Mode
- Frame Output Stream Mode
- TDMS

CAN, FlexRay, and LIN

This format is used for CAN, FlexRay, and LIN interfaces. This includes frames for SAE J1939 and CAN FD. Refer to the NI-XNET log file examples for functions and VIs that convert raw frame data to/from LabVIEW clusters for CAN, FlexRay, or LIN frames. Each frame is encoded as one Base Unit, followed by zero or more Payload Units.

Base Unit

In the following table, ***Byte Offset*** refers to the offset from the frame start. For example, if the first frame is in raw data bytes 0–23, and the second frame is in bytes 24–47, the second frame Identifier starts at byte 32 (24 + Byte Offset 8).

Element	Byte Offset	Description
Timestamp	0 to 7	64-bit timestamp in 100 ns increments. The timestamp format is absolute. The 64-bit element contains the number of 100 ns intervals that have elapsed since 1 January 1601 00:00:00 Coordinated Universal Time (UTC). IN previous releases, this timestamp was called nxTimestamp_t. This element contains a 64-bit unsigned integer (U64) in native byte order. For little-endian computing platforms (for example, Windows), Byte Offset 0 is the least significant byte. For more information, refer to the NI-XNET examples for log file access.
Identifier	8 to 11	The frame identifier. This element contains a 32-bit unsigned integer (U32) in native byte order. When Type specifies a CAN frame, bit 29 (hex 20000000) indicates the CAN identifier format: set for extended, clear for standard. If bit 29 is clear, the lower 11 bits (0–10) contain the CAN frame identifier. If bit 29 is set, the lower 29 bits (0–28) contain the CAN frame identifier. When Type specifies a FlexRay frame, the lower 16 bits contain the slot number. When Type specifies a LIN frame, this element contains a number in the range 0–63 (inclusive). This number is the LIN frame's ID (unprotected).

<table>
<tr><th>Element</th><th>Byte Offset</th><th>Description</th></tr>
<tr><td></td><td></td><td>For SAE J1939 frames, the PGN and address fields are mapped to the Extended CAN identifier and written in the same way as for CAN.

All unused bits are 0.</td></tr>
<tr><td>Type</td><td>12</td><td>The frame type.

This element specifies the fundamental frame type. The Identifier, Flag, and Info element interpretations are different for each type.

The upper 3 bits of this element specify the protocol. The valid values in decimal are:

<table>
<tr><td>0</td><td>CAN</td></tr>
<tr><td>1</td><td>FlexRay</td></tr>
<tr><td>2</td><td>LIN</td></tr>
<tr><td>6</td><td>J1939</td></tr>
<tr><td>7</td><td>Special</td></tr>
</table>

The lower 5 bits of this element contain the specific type.

Table 6. CAN Type Values
<table>
<tr><td>CAN Data (0)</td><td>The CAN data frame contains payload data. This is the most commonly used frame type for CAN.</td></tr>
<tr><td>CAN 2.0 Data (8)</td><td>The CAN frame contains payload data. It has been transmitted in ISO CAN FD mode as a CAN 2.0</td></tr>
</table>
</td></tr>
</table>

Element	Byte Offset	Description
		<table><tr><td></td><td>frame.</td></tr><tr><td>CAN FD Data (16)</td><td>The CAN frame contains payload data. It has been transmitted in ISO CAN FD+BRS mode as a CAN FD frame.</td></tr><tr><td>CAN FD+BRS Data (24)</td><td>The CAN frame contains payload data. It has been transmitted in ISO CAN FD+BRS mode as a CAN FD+BRS frame.</td></tr><tr><td>CAN Remote (1)</td><td>A CAN remote frame. An ECU transmits a CAN remote frame to request data for the corresponding identifier. Your application can respond by writing a CAN data frame for the identifier.</td></tr><tr><td>Delay (224)</td><td>The Delay frame is used with the replay feature to insert a relative time delay between frame transmissions. For information about this frame, including the other frame fields, refer to Special Frames.</td></tr><tr><td>Log Trigger (225)</td><td>A Log Trigger frame. This frame is generated when a trigger occurs on an external connection (for example, PXI_Trig0). For information about this frame, including the</td></tr></table>

Element	Byte Offset	Description

	other frame fields, refer to Special Frames.
Start Trigger (226)	A Start Trigger frame is generated when the interface is started. (Refer to Start Interface for more information.) For information about this frame, including the other frame fields, refer to Special Frames.
CAN Bus Error (2)	A CAN Bus Error frame is generated when a bus error is detected on the CAN bus. For information about this frame, including the other frame fields, refer to Special Frames.

Table 7. FlexRay Type Values

FlexRay Data (32)	FlexRay data frame. The frame contains payload data. This is the most commonly used frame type for FlexRay. All elements in the frame are applicable.
FlexRay Null (33)	FlexRay null frame. When a FlexRay null frame is received, it indicates that the transmitting ECU did not have new data for the current cycle.

Element	Byte Offset	Description
		<table><tr><td></td><td>Null frames occur in the static segment only. This frame type does not apply to frames in the dynamic segment. This frame type occurs only when you set the XNET Session Interface:FlexRay:Null Frames To Input Stream? property to true. This property enables logging of received null frames to a session with the Frame Input Stream Mode. Other sessions are not affected. For this frame type, the payload array is empty (size 0), and preamble? and echo? are false. The remaining elements in the frame reflect the data in the received null frame and the timestamp when it was received.</td></tr><tr><td>FlexRay Symbol (34)</td><td>FlexRay symbol frame. The frame contains a symbol received on the FlexRay bus. For this frame type, the</td></tr></table>

Element	Byte Offset	Description
		first payload byte (offset 0) specifies the type of symbol: 0 for MTS, 1 for wakeup. The frame payload length is 1 or higher, with bytes beyond the first byte reserved for future use. The frame timestamp specifies when the symbol window occurred. The cycle count, channel A indicator, and channel B indicator are encoded the same as FlexRay data frames. All other fields in the frame are unused (0).
		Log Trigger (225) — A Log Trigger frame. This frame is generated when a trigger occurs on an external connection (for example, PXI_Trig0). For information about this frame, including the other frame fields, refer to ***Special Frames***.
		Start Trigger (226) — A Start Trigger frame is generated when the interface is started. For information about this frame, including the other frame fields, refer to ***Special Frames***.

<table>
<tr><th>Element</th><th>Byte Offset</th><th>Description</th></tr>
<tr><td></td><td></td><td>

Table 8. LIN Type Values

<table>
<tr><td>LIN Data (64)</td><td>The LIN data frame contains payload data.</td></tr>
<tr><td>Log Trigger (225)</td><td>A Log Trigger frame. This frame is generated when a trigger occurs on an external connection (for example, PXI_Trig0). For information about this frame, including the other frame fields, refer to Special Frames.</td></tr>
<tr><td>Start Trigger (226)</td><td>A Start Trigger frame is generated when the interface is started. (Refer to Start Interface for more information.) For information about this frame, including the other frame fields, refer to Special Frames</td></tr>
<tr><td>LIN Bus Error (65)</td><td>A LIN Bus Error frame is generated when a bus error is detected on the LIN bus. For information about this frame, including the other frame fields, refer to Special Frames.</td></tr>
<tr><td>LIN No Response (66)</td><td>A LIN No Response frame is generated when a header with no response is detected on the LIN bus. For information about this frame, including the other</td></tr>
</table>

</td></tr>
</table>

Element	Byte Offset	Description
		frame fields, refer to ***Special Frames***.
Flags	13	Eight Boolean flags that qualify the frame type. Bit 7 (hex 80) is protocol independent (supported in CAN, FlexRay, and LIN frames). If set, the frame is echoed (returned from the appropriate `nxRead` function or XNET Read VI after NI-XNET transmitted on the network). If clear, the frame was received from the network (from a remote ECU). For FlexRay frames: • Bit 0 is set if the frame is a Startup frame. • Bit 1 is set if the frame is a Sync frame. • Specifies the frame Preamble bit. • Specifies if the frame transfers on Channel A. • Specifies if the frame transfers on Channel B. For LIN frames: • Bit 0 is set if the frame occurred in an event-triggered entry (slot). When bit 0 is set, the Info element contains the event-triggered frame ID, and the Identifier element contains the Unconditional ID from the first payload byte. All unused bits are zero.
Info	14	Information that qualifies the frame Type. This element is not used for CAN. For FlexRay frames, this element provides the frame cycle count (0–63). For LIN frames read for a non-stream input session, if bit 0 of the Flags element is clear, the Info element is

Element	Byte Offset	Description
		unused (0). If bit 0 of the Flags element is set (event-triggered entry), the Info element contains the event-triggered frame ID, and the Identifier element contains the Unconditional ID from the first payload byte. For LIN frames read for a stream input session, if Interface:LIN:Checksum to Input Stream? is false (default), the Info element contains 0 for each frame. If true, the Info element contains the received checksum for each frame. For SAE J1939 frames, the three lowest bits of this element contain the three highest bits of the PayloadLength.
Payload Length	15	The PayloadLength indicates the number of valid data bytes in Payload. For all standard CAN and LIN frames, PayloadLength cannot exceed 8. Because this base unit always contains 8 bytes of payload data, the entire frame is contained in the base unit, and no additional payload units exist. For CAN FD frames, PayloadLength can be 0–8, 12, 16, 20, 24, 32, 48, or 64. For FlexRay frames, PayloadLength is 0–254 bytes. For SAE J1939 frames, PayloadLength is 0–1785 bytes; the low 8 bits are in this element, and the high three bits are found in the low bits of the Info field. If PayloadLength is 0–8, only the base unit exists. If PayloadLength is 9 or greater, one or more payload units follow the base unit. Additional payload units are provided in increments of 8 bytes, to optimize efficiency for DMA transfers. For example, if PayloadLength is 12, bytes 0–7 are in the base unit

Element	Byte Offset	Description

		Payload, bytes 8–11 are in the first half of the next payload unit, and the last 4 bytes of the next payload unit are ignored.
		
		In other words, each raw data frame can vary in length. You can calculate each frame size (in bytes) using the following pseudocode:
		
		`U16 FrameSize; // maximum 272 for largest FlexRay frame`
		`FrameSize = 24; // 24 byte base unit`
		`if (PayloadLength > 8)`
		`FrameSize = FrameSize +`
		`(U16)(PayloadLength - 1) AND 0xFFF8;`
		
		The last pseudocode line subtracts 1 and truncates to the nearest multiple of 8 (using bitwise AND). This adds bytes for additional payload units. For example, PayloadLength of 9 through 16 requires one additional payload unit of 8 bytes.
		
		The NI-XNET example code helps you handle the variable-length frame encoding details.

Payload	16 to 23	This element always uses 8 bytes in the log file, but PayloadLength determines the number of valid bytes.

Payload Unit

The base unit PayloadLength element determines the number of additional payload units (0–31).

Element	Byte Offset	Description
Payload	0 to 7	This element always uses 8 bytes in the log file, but PayloadLength determines the number of valid bytes.

Ethernet

The following format is used for Ethernet interfaces. Ethernet has a raw frame format that is different from what CAN, LIN, and FlexRay use.

In the following table, Byte Offset refers to the offset from the beginning of the frame. For example, if the first frame is in raw data bytes 0 127, and the second frame is in bytes 128 255, the second frame's Source MAC Address starts at offset 156 (128 + Byte Offset 28).

The following table specifies the overall frame format, including header fields that are specific to XNET (e.g., timestamps).

Field	Byte Offset	Description
Length	0 to 1	This unsigned 16-bit integer provides the length of the entire frame, including two bytes for the Length field itself. The length of Frame Data (IEEE Std 802.3 frame data) can be computed by subtracting 28 from this Length, to account for the fields that are specific to NI (and the FCS). This field uses big-endian byte order (most significant byte first; also known as network order) when used for writing a frame, but it uses host-byte order when reading a frame.
Type	2 to 3	This unsigned 16-bit integer provides the type of the Ethernet frame. The type is an enumerated value:

<table>
<tr><th>Field</th><th>Byte Offset</th><th>Description</th></tr>
<tr><td></td><td></td><td>
Frame Data (value 0): Ethernet frame received or transmitted.

The upper 3 bits of this element specify the protocol. The valid values in decimal are:

<table>
<tr><td>0</td><td>Ethernet</td></tr>
<tr><td>7</td><td>Special</td></tr>
</table>

The lower 5 bits of this element contain the specific type.

Special values specify features that are not related to the protocol or bus traffic. For more information about special frames, refer to Special Frames.

Table 9. Ethernet Type Values
<table>
<tr><td>Delay Frame (225)</td><td>The Delay frame is used with the replay feature to insert a relative time delay between frame transmissions. For information about this frame, including the other frame</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Field</th><th>Byte Offset</th><th>Description</th></tr>
<tr><td></td><td></td><td>
<table>
<tr><td></td><td>fields, refer to Special Frames.</td></tr>
<tr><td>Future Time Wait (226)</td><td>The Future Time Wait frame is used with the replay feature to cause the output stream to wait until a particular time occurs. For information about this frame, including the other frame fields, refer to Special Frames.</td></tr>
</table>
</td></tr>
<tr><td>Local Timestamp</td><td>4 to 11</td><td>This timestamp uses XNET local time.

This is an absolute timestamp in 1 nanosecond increments. This 64-bit type contains the number of 1 ns intervals that have elapsed since 1 January 1970 00:00:00 International Atomic Time (TAI). The time represented by zero corresponds to the PTP epoch as specified in time synchronization protocols (e.g.,</td></tr>
</table>

Field	Byte Offset	Description
		IEEE Std 802.1AS). The timestamp point in the Ethernet frame occurs at the beginning of the first symbol following the start of frame delimiter. **Note** As of 00:00:00, 1 January 2018 UTC, UTC was behind TAI by 37 seconds. The location of the timestamp point depends on the Port Mode of the session's interface. When Port Mode is Direct, the timestamp point's location corresponds to time synchronization protocols, using the reference plane marking the boundary between the port's connector (copper wire) and PHY. When Port Mode is Tap, the timestamp point's location is the midpoint between the connector/PHY reference plane of this session's interface and the connector/PHY reference plane of the Tap partner.
Network Timestamp	12 to 19	This timestamp uses network time (clock of the network's time synchronization protocol, such as IEEE Std 802.1AS). This is an absolute timestamp in 1 nanosecond increments. This 64-bit type contains the

Field	Byte Offset	Description
		number of 1 ns intervals that have elapsed since 1 January 1970 00:00:00 International Atomic Time (TAI). The time represented by zero corresponds to the PTP epoch as specified in time synchronization protocols (e.g., IEEE Std 802.1AS). The timestamp point in the Ethernet frame occurs at the beginning of the first symbol following the start of frame delimiter. **Note** As of 00:00:00, 1 January 2018 UTC, UTC was behind TAI by 37 seconds. The location of the timestamp point depends on the Port Mode of the session's interface. When Port Mode is Direct, the timestamp point's location corresponds to time synchronization protocols, using the reference plane marking the boundary between the port's connector (copper wire) and PHY. When Port Mode is Tap, the timestamp point's location is the midpoint between the connector/PHY reference plane of this session's interface and the connector/PHY reference plane of the Tap partner.
Flags	20 to 23	This 32-bit field provides

Field	Byte Offset	Description
		Boolean flags that qualify the frame. Bit 0 corresponds to the lowest bit (i.e., hex 00000001). • Transmit (bit 31): Boolean value that indicates whether the frame occurred due to transmit (true) or not (false). For nxRead on the monitor path: ◦ When Port Mode of this session's interface is Direct, the monitor path echoes each transmit that was submitted to nxWriteFrame on the endpoint path. ◦ When Port Mode of this session's interface is Tap, the value true indicates that the frame was received by the Tap partner, and transmitted on this interface. For nxReadFrame on the endpoint path, this flag is always false. • Receive (bit 30): Boolean value that indicates whether the frame occurred due to receive (true) or not (false). For nxReadFrame on the monitor path: ◦ When Port Mode of this session's interface

Field	Byte Offset	Description
		is Direct, this flag is true when a frame is received on the interface. ◦ When Port Mode of this session's interface is Tap, the value true indicates that the frame was received by this interface, and will be transmitted on the Tap partner. For nxReadFrame on the endpoint path, this flag is always true. • Network Synced (bit 23): Contains the value of the Synced property at the time that both timestamps are acquired, to specify whether the Network Timestamp is synchronized to the network (true) or not (false). • Error (bit 16): Indicates that an error occurred during reception/transmission of the frame (false = good frame, true = bad frame). All unused bits are 0. This field is ignored by nxWriteFrame.
Frame Data	24 (Length-5)	Data of the IEEE Std 802.3 frame. The Frame Data begins with the destination MAC address, and ends with the

Field	Byte Offset	Description
		frame's last byte of MSDU. The maximum length of this array is provided in the Payload Length Maximum property. This field uses big-endian byte order.
FCS	(Length-4) to (Length-1)	IEEE Std 802.3 Frame Check Sequence (FCS) that was received with the Frame Data. This field uses big-endian byte order, and it is ignored by `nxWriteFrame`.

The following tables provide examples of the two most commonly used formats for Frame Data on Ethernet, as specified in IEEE Std 802.3 and IEEE Std 802.1Q. Unless otherwise indicated, fields in the following tables all use big-endian byte order.

The following table shows Frame Data for an untagged frame. An untagged frame uses the default Priority 0, default Drop Eligible false, and the default VLAN Identifier (VID) 1.

Field	Byte Offset	Description
Destination MAC Address	24 to 29	This is the destination MAC address as specified in IEEE Std 802 and IEEE Std 802.3. The MAC address consists of 6 bytes.
Source MAC Address	30 to 35	This is the source MAC address as specified in IEEE Std 802 and IEEE Std 802.3. The MAC address consists of 6 bytes. For Write, XNET can automatically populate this field (see Source MAC Address Auto).
EtherType	36 to 37	This 16-bit unsigned integer

Field	Byte Offset	Description
		specifies the protocol that is used to encode/decode bytes in the MSDU. In other words, the EtherType determines what the frame contains. EtherType values are assigned by the IEEE Registration Authority (IEEE-RA). Examples include hex 0800 for Internet Protocol version 4 (IPv4), hex 08DD for Internet Protocol version 6 (IPv6), and hex 22F0 for IEEE Std 1722.
MSDU	38 to (Length-5)	The remaining bytes of the Frame Data contain the frame's payload, which IEEE 802 standards refer to as the mac_service_data_unit (MSDU). IEEE Std 802.3 specifies that the minimum length of the MSDU is 46 bytes (padded as necessary), and the maximum length of the MSDU is 1500 bytes. Another term used for the maximum length of the MSDU is the Maximum Transmission Unit (MTU).

The following table shows Frame Data for a frame with a VLAN tag.

Field	Byte Offset	Description
Destination MAC Address	24 to 29	This is the destination MAC address as specified in IEEE Std 802 and IEEE Std 802.3. The MAC address consists of 6 bytes.
Source MAC Address	30 to 35	This is the source MAC address as specified in IEEE Std 802 and IEEE Std 802.3. The MAC address consists of 6 bytes. For Write, XNET can automatically

Field	Byte Offset	Description
		populate this field. For more information, refer to Source MAC Address Auto in the API reference.
Tag Protocol ID	36 to 37	IEEE Std 802.1Q specifies a tag that adds information to the frame without changing its content (i.e., EtherType or MSDU). Use of the tag is optional. If a frame contains a tag, this Tag Protocol Identification (TPID) field specifies the encoding of the tag's information (Tag Control Info). TPID of hex 8100 is the Customer VLAN Tag (C-TAG), which is the general-purpose tag format commonly known as a VLAN tag.
Tag Control Info	38 to 39	IEEE Std 802.1Q specifies the 16-bit Tag Control Info for a C-TAG as follows: • Bits 13-15 (upper 3 bits): Priority Code Point (PCP). This field is commonly known as the Priority of the frame. The Priority is mapped to a traffic class, and that traffic class determines the timing and importance of the frame as it egresses from a queue at each port in the switched Ethernet network. In other words, the Priority determines how the frame travels through queues. • Bit 12: Drop Eligibility Indicator (DEI): This field is

Field	Byte Offset	Description
		commonly known as the Drop Eligible indicator. If Drop Eligible is true, the frame can be discarded by metering algorithms in preference to frames in which Drop Eligible is false. • Bits 0-11 (lower 12 bits): VLAN Identifier (VID): This VLAN Identifier specifies where the frame travels through the network (i.e., on which ports of a switch it egresses). Within a frame, VID value 0 indicates a null VID, meaning that the tag contains only priority information (commonly known as priority-tag). The default VID value for all ports is 1, and therefore untagged and priority-tagged frames use the default VID of 1.
EtherType	40 to 41	This 16-bit unsigned integer specifies the protocol that is used to encode/decode bytes in the MSDU. In other words, the EtherType determines what the frame contains. EtherType values are assigned by the IEEE Registration Authority (IEEE-RA). Examples include hex 0800 for Internet Protocol version 4 (IPv4), hex 08DD for Internet Protocol version 6 (IPv6), and hex 22F0 for IEEE Std 1722.
MSDU	42 to (Length-5)	The remaining bytes of the Frame Data contain the frame's

Field	Byte Offset	Description
		payload, which IEEE 802 standards refer to as the mac_service_data_unit (MSDU). IEEE Std 802.3 specifies that the minimum length of the MSDU is 46 bytes (padded as necessary), and the maximum length of the MSDU is 1500 bytes. Another term used for the maximum length of the MSDU is the Maximum Transmission Unit (MTU).

Special Frames

The NI-XNET driver offers some special frames not directly used in bus communication.

Future Time Wait Frame

When a frame with a Future Time Wait frame type is received, hardware waits until the absolute time associated with this frame occurs before continuing on to process the subsequent frames. If the Future Time Wait timestamp was in the past when this frame is processed, hardware immediately continues processing the subsequent frames. This frame type always resets the start time. The next frame to be dequeued is treated as a new first frame and transmitted immediately. Future Time Wait is only supported with Ethernet.

Future Time Wait is particularly useful for synchronizing with other devices that also support a common notion of time. For example, this frame type can be used to synchronize the start of replay across multiple Ethernet ports. It can also be used in conjunction with nxFutureTimeTrigger to synchronize with other devices that require a physical signal. Multiple Future Time Wait frames can be written into a replay stream. Hardware will compensate for front-end delays (from the MAC through the PHY) when determining how long to wait. This will cause data from the next frame following Future Time Wait to hit the wire at the same time that was just waited for (hardware subtracts the front-end delay from the actual time that is waited for).

Delay Frame

When a frame with a Delay Frame frame type is received, the hardware delays for the requested time. The next frame to be dequeued is treated as a new first frame and transmitted immediately. You can use a Delay Frame with a time of 0 to restart time quickly. If you replay a logfile of frames repeatedly, you can insert a Delay Frame at the start of each replay to insert a delay between each iteration through the file.

The Delay frame fields are as follows:

Element	Description
Identifier	0 (Ignored)
Extended	False (Ignored)
Echo	False (Ignored)
Type	Delay
Timestamp	Amount of time to delay. Note that this is not an absolute time and is not related to any other time in the replay frames. A time of 0.25 (that is, LabVIEW absolute time of 6:00:00.250PM 12/31/1903) will delay 250 ms.
Payload Length	0
Payload	Ignored

Log Trigger Frame

A Log Trigger frame is a special frame that can be received by a Frame Stream Input session. This frame is generated when a rising edge is detected on an external connection (PXI_Trig or FrontPanel trigger). To enable the hardware to log this frame, you must use the Connect Terminals function (`nxConnectTerminals` in C or the XNET Connect Terminals VI in LabVIEW) to connect the external connection to the internal LogTrigger terminal. A Log Trigger frame is applicable to CAN, FlexRay, and LIN.

The fields of the Log Trigger frame are as follows:

Table 10. CAN Frame

Element	Description
identifier	0
extended?	False
echo?	False
type	Log Trigger
timestamp	Time when the trigger occurred
payload length	0 (may increase in the future)
payload	N/A

Table 11. FlexRay Frame

Element	Description
slot	0
cycle count	0
startup?	False
sync?	False
preamble?	False
ch A	False
ch B	False
echo?	False
Type	Log Trigger
Timestamp	Time when the trigger occurred
Payload length	0 (may increase in the future)
Payload	N/A

Table 12. LIN Frame

Element	Description
identifier	0
event slot?	False

Element	Description
event ID	0
echo?	False
type	Log Trigger
timestamp	Time when the trigger occurred
payload length	0 (may increase in the future)
payload	N/A

Start Trigger Frame

A Start Trigger frame is a special frame that a Frame Stream Input session can receive. This frame is generated when the interface is started. To enable the hardware to log this frame, you must enable the Interface:Start Trigger Frames to Input Stream? property. A Start Trigger frame is applicable to CAN, FlexRay, and LIN.

Start Trigger frames are not supported on Ethernet devices, as they are not needed. Ethernet devices use Future Time Wait frames for synchronization use cases.

The fields of the Start Trigger frame are as follows:

Table 13. CAN Frame

Element	Description
identifier	0
extended?	False
echo?	False
type	Start Trigger
timestamp	Time when the interface started
payload length	0 (may increase in the future)
payload	N/A

Table 14. FlexRay Frame

Element	Description
slot	0
cycle count	0
startup?	False
sync?	False
preamble?	False
ch A	False
ch B	False
echo?	False
Type	Start Trigger
Timestamp	Time when the interface started
Payload Length	0 (may increase in the future)
Payload	N/A

Table 15. LIN Frame

Element	Description
identifier	0
event slot?	False
event ID	0
echo?	False
type	Start Trigger
timestamp	Time when the interface started
payload length	0 (may increase in the future)
payload	N/A

Bus Error Frame

A CAN Bus Error frame is a special that can be received by a Frame Stream Input session. This frame is generated when a bus error is detected on the CAN bus. To

enable the hardware to log this frame, you must enable the Interface:Bus Error Frames to Input Stream? property. A Bus Error frame is applicable to CAN and LIN. The fields of the Bus Error frame are as follows:

Table 16. CAN Frame

<table>
<tr><th>Element</th><th>Description</th></tr>
<tr><td>identifier</td><td>0</td></tr>
<tr><td>extended?</td><td>False</td></tr>
<tr><td>echo?</td><td>False</td></tr>
<tr><td>type</td><td>CAN Bus Error</td></tr>
<tr><td>timestamp</td><td>Time when the bus error was detected</td></tr>
<tr><td>payload length</td><td>5 (may increase in future)</td></tr>
<tr><td>payload</td><td>Byte 0: CAN Comm State
<table><tr><td>0</td><td>Error Active</td></tr><tr><td>1</td><td>Error Passive</td></tr><tr><td>2</td><td>Bus Off</td></tr></table>
Byte 1: TX Error Counter

Byte 2: RX Error Counter

Byte 3: Detected Bus Error
<table><tr><td>0</td><td>None (never returned)</td></tr><tr><td>1</td><td>Stuff</td></tr><tr><td>2</td><td>Form</td></tr><tr><td>3</td><td>Ack</td></tr><tr><td>4</td><td>Bit 1</td></tr><tr><td>5</td><td>Bit 0</td></tr><tr><td>6</td><td>CRC</td></tr></table>
Byte 4: Transceiver Error?</td></tr>
</table>

Element	Description
	0 — no error 1 — error

Table 17. LIN Frame

Element	Description
identifier	0
event slot?	False
event ID	0
echo?	False
type	LIN Bus Error
timestamp	Time when the interface started
payload length	0 (may increase in the future)
payload	Byte 0: LIN Comm State 0 — Idle 1 — Active 2 — Inactive Byte 1: Detected Bus Error 0 — None (never returned) 1 — UnknownId 2 — Form 3 — Framing 4 — Readback 5 — Timeout

Element	Description
	6 | CRC Byte 2: Identifier on bus Byte 3: Received byte on bus Byte 4: Expected byte on bus

LIN No Response Frame

A LIN No Response frame is a special frame that a Frame Stream Input session can receive. This frame is generated when a header with no response is detected on the LIN bus. To enable the hardware to log this frame, you must enable the Interface:LIN:No Response Frames to Input Stream? property. The No Response frame fields are as follows:

Element	Description
Identifier	Unprotected version of header ID
Event Slot?	0
Event ID	0
Echo?	False
Type	LIN No Response
Timestamp	Time when the end of the header (ID) was detected
Payload Length	0
Payload	N/A

Related concepts:

- Handling Timestamps

Signal

Each frame contains an arbitrary number of signals, which are the basic data exchange units on the network. These signals are equivalent to NI-CAN channels.

Some of the signal properties are:

- **Start bit**: signal start position within the frame
- **Number of bits**: signal length within the frame
- **Data type**: data type (signed, unsigned, or float)
- **Byte order**: little or big endian
- **Scaling factor and offset**: for converting physical data to binary representation

Multiplexed Signals

Multiplexed signals do not appear in every instance of a frame; they appear only if the frame indicates this. For this reason, a frame can contain a multiplexer signal and several subframes. The multiplexer signal is at most 16 bits long and contains an unsigned integer number that identifies the subframe instance in the instance of a frame. The subframes contain the multiplexed signals.

This means the frame signal content is not fixed (static), but can change depending on the multiplexer signal (dynamic) value. A frame can contain both a static and a dynamic part.

Creating Multiplexed Signals

- **In the API** —
  Creating multiplexed signals in the API is a two-step process:
  1. Create the multiplexer signal and subframes as children of the frame object. The subframes are assigned the mode value; that is, the value of the multiplexer signal for which this subframe becomes active.
  2. Create the multiplexed signals as children of their respective subframes. This automatically assigns the signals as dynamic signals to the subframe's parent frame.
- **In the NI-XNET Database Editor**—

  You create multiplexed signals by changing their Signal Type to `Multiplexed` and assigning them mode values. The Database Editor handles subframe manipulation completely behind the scenes.

Reading Multiplexed Signals

You can read multiplexed signals like static signals without any additional effort. Because the frame read also contains the multiplexer signal, the NI-XNET driver can decide which signals are present in the frame and return new values for only those signals.

Writing Multiplexed Signals

Writing multiplexed signals needs additional consideration. As writing signals results in a frame being created and sent over the network, writing multiplexed signals requires the multiplexer signal be part of the writing session. This is needed for the NI-XNET driver to decide which set of dynamic signals a certain frame contains. Only the subframe dynamic signals selected with the multiplexer signal value are written to the frame; the values for the other dynamic signals of that frame are ignored.

Support for Multiplexed Signals

Multiplexed signals are currently supported for CAN only. FlexRay does not support them.

J1939 Sessions

If you use a DBC file defining a J1939 database or create a stream session with the cluster name :can_j1939:, you will create a J1939 XNET session. If the session is running in J1939 mode, the session property application protocol returns nxAppProtocol_J1939 instead of nxAppProtocol_None. This property is read only, as you cannot change the application protocol while the session is running.

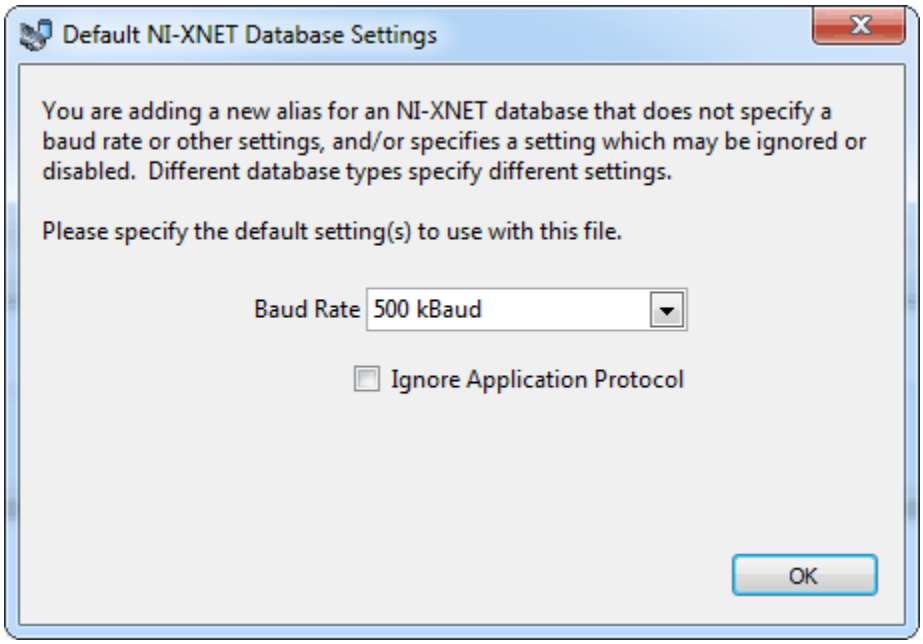
FIBEX databases do not define support for J1939 in the standard. If you save a J1939 database to FIBEX in the NI-XNET Database Editor or with the `nxdbSaveDatabase` API function, the J1939 properties are saved in a FIBEX extension defined by National Instruments in the FIBEX XML file.

Compatibility Issue

If you have used a J1939 database with a version of NI-XNET that does not support J1939, the session now opens in J1939 mode, which defines a different behavior than a non-J1939 session. This may break the compatibility of your application. To avoid issues, you can ignore the application protocol for the database alias in question.

Complete the following steps to set whether the database application protocol is used or ignored when the alias is added:

1. Launch the NI-XNET Database Editor.
2. From the main menu, select **File»Manage Aliases**, which opens the **Manage NI-XNET Databases** dialog.
3. In the **Manage NI-XNET Databases** dialog, click the **Add Alias** button, which opens the **Add Alias to NI-XNET Database...** dialog.
4. Browse to the database file to add, then click **OK** to continue. If the protocol for the selected database is CAN and the application protocol is J1939, an **Ignore Application Protocol** checkbox is displayed, as shown in the following figure. (The **Baud Rate** control may or may not be displayed, depending on whether the database specifies it.)



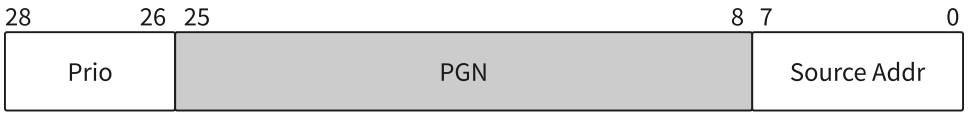
5. To have NI-XNET interpret the alias as an alias for a J1939 database, leave **Ignore Application Protocol** unchecked. To have NI-XNET interpret the alias as an alias for a plain CAN database, check **Ignore Application Protocol**.
6. Click **OK** to complete the alias addition.

J1939 Basics

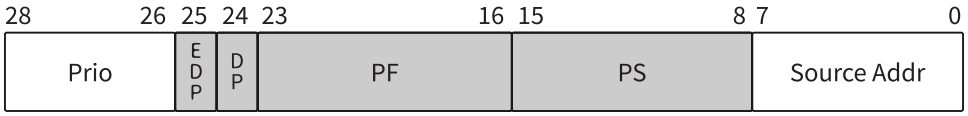
A J1939 network consists of ECUs connected by a CAN bus running at 250 k baud rate. Some newer networks might use a 500 k baud rate. A physical ECU can contain one or more logical ECUs called nodes or Controller Applications. This description refers to it as a node or ECU.

J1939 application protocol uses a 29-bit extended frame identifier. The ID is divided into several parts:

- **Source Address (8 bits):** Determines the address of the node transmitting the frame. By examining the Source Address part of the ID, the receiving session can recognize which node has sent the frame.
- **PGN (18 bits):** Identifies the frame and defines which signals it contains.
- **Priority (3 bits):** Priority is used when multiple CAN frames are sent on the bus at exactly the same time. In this case, the CAN frame with the higher priority (lower number) is transmitted before the lower priority frame. The CAN standard defines the CAN frames priority (lower IDs have higher priority). Therefore, the J1939 priority bits are the most significant bits in the ID. This ensures that the ID value with a higher priority is always lower, independent of the PGN and Source Address, as shown in the following figure.



You can send a frame to a global address (all nodes) or a specific address (node with this address). This information is coded inside the PGN, as shown in the following figure.



The PF value in the identifier defines whether the message has a global or specific destination:

- 0–239 (0x00–0xEF): specific destination
- 240–255 (0xF0–0xFF): global destination

In the CAN identifier, this looks like the following (X = don't care):

- 0xXXF0XXXX to 0xXXFFXXXX are messages with global destination (broadcast)
- 0xXX00XXXX to 0xXXEFXXXX are messages with specific destination

For global messages, the PS byte of the ID defines group extension. This extends the number of possible global PGNs to 4096 (0xF000 to 0xFFFF).

For destination-specific messages, PS defines the destination address, so PF defines only 240 destination-specific PGNs (0–239).

DP and EDP bits increase the number of possible PGNs by defining data pages. EDP, however, always is set to 0 in J1939, so only DP can be set to 0 or 1, which doubles the number of PGNs described above. The maximum number of possible PGNs (and so, different messages) in J1939 is 2*(4096 + 240) = 8672.

For node addresses (source address and destination address), the ID reserves 8 bit, which allows values from 0 to 255. Two values have a special meaning:

- 254 is the null address. This means there is no valid address assigned to a node yet.
- 255 is the global address. This allows sending even PGNs with PF 0 to 239 to a global destination.

Node Addresses in NI-XNET

A newly created XNET session has no node address. If you read the J1939 Node Address property after creating a session, it returns the value 254 (null address).

A receiving XNET session without address can read all frames from the bus. A receiving XNET session with an assigned address can read only frames with a global destination address (255) and frames sent to this address, but not frames sent to other nodes. A read session with a null (254) or global (255) address observes all messages on the bus, without participating in any J1939 handshakes.

A transmitting XNET session requires a node address. A write session with a null (254) or global (255) address transmits messages only if a valid source address is set in the frame identifier. A write session with a valid claimed address always substitutes the

source address portion of the frame identifier with the node's claimed address.

All nodes in the network must have different node addresses; otherwise, two nodes could send a frame with the same CAN identifier, which is not allowed by the CAN standard. To ensure that each node has a different address, J1939 defines a procedure called address claiming to obtain an address on the network. There are two properties required for address claiming:

- Node name (64 bit value)
- Node address

The node name identifies a node (ECU) and usually is saved in the database. Each ECU in the network has a unique node name. For the address claiming procedure, there are two important features of the node name value:

- Priority: The lower name value has the higher priority.
- Arbitrary address capability (bit 63 = 1): This node can use a different address than specified in case of conflict.

The arbitrary address capability is defined in the highest significant bit of the value (bit 63). All arbitrary-capable names have a lower priority than nonarbitrary-capable names.

Transmitting Frames

When transmitting frames, the granted address of the node automatically replaces the source address portion of the frame identifier.

In your application, you may want a session to transmit frames using the source address provided in the identifier in the database or the frame data. If you do not assign a valid address to a session (or set the address to 254 explicitly), NI-XNET does not change the address in your frame identifier before transmitting. An error is returned when a transmitting session without an address tries to send a frame without a valid address in the identifier.

Address Claiming Procedure

To obtain an address on the network, set the J1939/Node Name and J1939/Node

Address properties or set the J1939/ECU property (which is equivalent to setting the other properties using the values in the ECU object in the database). After setting the Node Address (to a value less than 254), XNET sends an address claimed message and waits 300 ms for the response from the network. If no other node is using this address, there is no response to the message; after the timeout, the address is granted to the session and the session can transmit frames on the network.

Setting the Node Address causes NI-XNET to start the interface; you must set any properties that are to be set before the interface starts before setting Node Address. Setting the Node Address does not start the session. J1939 traffic is not retained by an input session until Start or Read are explicitly called.

During the claiming procedure, the node address property returns the null address (254), so you can poll this address until it gets a valid value.

If the address cannot be granted to the session (for example, when the name is not arbitrary and another node with higher priority uses the node address), the address is not granted. After timeout, the J1939 CommState indicates the reason for failed address claiming. If the node name is arbitrary address capable, NI-XNET tries to find another address and claim it. This procedure can take some time depending on how fast the other nodes respond to the address claimed message.

NI-XNET examples contain the address claiming procedure, which you can use in your applications.

The frames transmitted during address claiming are not passed to the J1939 input session. To see those frames, open a non-J1939 CAN session, which can be running parallel with a J1939 session on the same interface.

Mixing J1939 and CAN Messages

J1939 frames in the database and CAN frames data in XNET include the Application Protocol property. This means you can mix J1939 and standard CAN messages in one session. Standard CAN messages cannot exceed 8 bytes and do not use the node address.

In standard CAN frames, the complete identifier is considered as the CAN message

identifier; in J1939, only the PGN determines the message. Frames with the same PGN but different priority or source address are considered the same message.

Received frames with extended identifier always are considered J1939 frames. If you use extended CAN frames as non-J1939 frames, you must process the received data to update the Application Protocol property.

Transport Protocol (TP)

When you use frames with more than 8 bytes, NI-XNET automatically uses the J1939 transport protocol to transmit and receive the frames. You do not receive any transport protocol management messages in the sessions. When this is required, you must open a non-J1939 CAN session, which can be running parallel to a J1939 session on the same interface.

Transport protocol defines many properties used to change the behavior (for example, timing).

If errors occur in the transport protocol, they are not reported directly from the read function. You can monitor errors in the TP by reading the J1939 CommState function.

Note that the transport protocol is not using the priority in the identifier, and the priority value is not transmitted with the TP. Received TP messages have the priority always set to 0.

NI-XNET Sessions

You can use all NI-XNET session modes with J1939 protocol, whether or not the frames use transport protocol. This includes frame and signal sessions in queued, single point, or stream mode.

Not Supported in the Current NI-XNET Version

Signal Ranges

For coded signal values in frames, J1939 reserves special values to transmit specific

indicators (for example, the error indicator). The current NI-XNET version does not support this; those values are converted to signal values. This behavior may change in a future NI-XNET version.

Cyclic and Event Timing

For all embedded network protocols (for example, CAN, FlexRay, and LIN), the transmit of a specific frame is classified as one of the following:

- **Cyclic**: The frame transmits at a cyclic (periodic) rate, regardless of whether the application has updated its payload data. The advantage of cyclic behavior is that the application does not need to worry about when to transmit, yet data changes arrive at other ECUs within a well-defined deadline.
- **Event**: The frame transmits when a specific event occurs. This event often is simply that the application updated the payload data, but other events are possible. The advantage is that the frame transmits on the network only as needed.

The following sections describe how the cyclic and event concept apply to each protocol.

Within NI-XNET, a Cyclic frame begins transmit as soon as the session starts, regardless of whether you called the appropriate XNET Write VI (LabVIEW) or `nxWrite` function (C). The call to the appropriate write function is the event that drives an Event frame transmit.

CAN

For each frame, the XNET Frame CAN:Timing Type property determines whether the network transfer is cyclic or event:

- **Cyclic Data**: This is typical Cyclic frame behavior.
- **Event Data**: This is typical Event frame behavior.
- **Cyclic Remote**: Because one ECU in the network transmits the CAN remote frame at a cyclic (periodic) rate, the resulting CAN data frame also is cyclic.
- **Event Remote**: One ECU in the network transmits the CAN remote frame based on an event. Another ECU responds with the corresponding CAN data frame. In NI-XNET, the appropriate XNET Write VI or `nxWrite`function generates the event to

transmit the CAN remote frame.

FlexRay

For each frame, the XNET Frame FlexRay:Timing Type property determines whether the network transfer is cyclic or event:

- **Cyclic (in static segment)**: No null frame transmits, so this is typical Cyclic frame behavior.
- **Event (in static segment)**: The null frame indicates no event.
- **Cyclic (in dynamic segment)**: The frame transmits each FlexRay cycle. This configuration is not common for the dynamic segment, which typically is for Event frames only.
- **Event (in dynamic segment)**: This is typical Event frame behavior.

LIN

As described in the Using LIN topic, the currently running schedule entries determine each LIN frame's timing. In each schedule entry, the master transmits a single frame header, and the payload of one (or more) frames can follow.

For each schedule entry, the XNET LIN Schedule Entry Type property determines how the associated frames transmit. The schedule run mode also contributes to the cyclic or event behavior.

- **Cyclic: Unconditional type, Continuous run mode**: This is typical Cyclic frame behavior.
- **Event: Unconditional type, Once run mode**: Although the frame transmits unconditionally, the schedule runs once based on an event, so this is Event frame behavior. In NI-XNET, the XNET Write (State LIN Schedule Change) VI or `nxWriteState(nxState_LINScheduleChange)` function changes the mode to the run-once schedule. This effectively generates the event to transmit the LIN frame.
- **Event: Sporadic type**: In this schedule entry, the master can transmit one of multiple Event-driven frames. In NI-XNET, the appropriate XNET Write VI or `nxWrite` function writes signal or frame values to generate the event to transmit. Because the entry itself is Event, this behavior applies regardless of the schedule's run mode.

- **Event: Event-triggered type**: In this schedule entry, multiple slave ECUs can transmit in the entry, each using an Event-driven frame. In NI-XNET, the appropriate XNET Write VI or `nxWrite` function writes signal or frame values to generate the event to transmit. Because the entry itself is Event, this behavior applies regardless of the schedule's run mode.

Related concepts:

- Using LIN
- Frame Input Queued Mode
- Frame Input Single-Point Mode
- Frame Input Stream Mode
- Frame Output Queued Mode
- Frame Output Single-Point Mode
- Frame Output Stream Mode
- LIN Frame Timing and Session Mode
- Signal Input Single-Point Mode
- Signal Input Waveform Mode
- Signal Input XY Mode
- Signal Output Single-Point Mode
- Signal Output Waveform Mode
- Signal Output XY Mode

Required Properties

When you create a new object, the properties may be:

- **Optional** : The property has a default value after creation, and the application does not need to set the property when the default value is desired for the session.
- **Required** : The property has no default value after creation. An undefined required property returns an error from `nxCreateSession`. A required property means you must provide a value for the property after you create the object.

The following NI-XNET object classes have no required properties:

- Session
- System

- Device
- Interface
- Database
- ECU
- LIN Schedule

This topic lists all required database properties. Properties with a protocol prefix (for example, FlexRay:) in the property name apply only to a session that uses the specified protocol.

The Cluster object class requires the following properties:

- 64bit Baud Rate *
- FlexRay:Action Point Offset
- FlexRay:CAS Rx Low Max
- FlexRay:Channels
- FlexRay:Cluster Drift Damping
- FlexRay:Cold Start Attempts
- FlexRay:Cycle
- FlexRay:Dynamic Slot Idle Phase
- FlexRay:Listen Noise
- FlexRay:Macro Per Cycle
- FlexRay:Max Without Clock Correction Fatal
- FlexRay:Max Without Clock Correction Passive
- FlexRay:Minislot Action Point Offset
- FlexRay:Minislot
- FlexRay:Network Management Vector Length
- FlexRay:NIT
- FlexRay:Number of Minislots
- FlexRay:Number of Static Slots
- FlexRay:Offset Correction Start
- FlexRay:Payload Length Static
- FlexRay:Static Slot
- FlexRay:Symbol Window
- FlexRay:Sync Node Max
- FlexRay:TSS Transmitter
- FlexRay:Wakeup Symbol Rx Idle
- FlexRay:Wakeup Symbol Rx Low

- FlexRay:Wakeup Symbol Rx Window
- FlexRay:Wakeup Symbol Tx Idle
- FlexRay:Wakeup Symbol Tx Low
- Tick

The Frame object class requires the following properties:

- FlexRay:Base Cycle
- FlexRay:Channel Assignment
- FlexRay:Cycle Repetition
- Identifier
- Payload Length

The Subframe object class requires the following properties:

- Multiplexer Value

The Signal object class requires the following properties:

- Byte Order
- Data Type
- Number of Bits
- Start Bit

The LIN Schedule Entry object class requires the following properties:

- Delay
- Event Identifier
- Frames

For FlexRay, Baud Rate always is required. For CAN and LIN, when you use a Frame I/O Stream session, you can specify Baud Rate using either the XNET Cluster 64bit Baud Rate property or XNET Session Interface:64bit Baud Rate property. For CAN and LIN with other session modes, the XNET Cluster Baud Rate property is required.

Synchronized Replay

You can synchronize replay across multiple devices using either Local Time or Network Time timescales. The replay timescale is selected via the Interface:Ethernet:Output Stream Timescale property. This property is Ethernet-specific, because only Ethernet supports both Local Time and Network Time. CAN and LIN support only Local Time.

To synchronize frame replay across multiple XNET devices, perform the following steps:

1. Create a Frame Output Stream session on each device. Use this session to configure each device in the following steps.
2. Synchronize all devices for the desired replay timescale. For example, if using Network Time, ensure that Network Time within all devices is synchronized to a master time source. If using Local Time, ensure that all devices are synchronized to the local chassis time. If all devices are within the same PXI chassis, Local Time is automatically frequency-locked, but there will be a phase offset. You can use Future Time Triggers and Timestamp Triggers on XNET Ethernet devices to measure the phase offset. Ethernet devices allow you to remove the phase offset of Network Time via the Interface:Ethernet:Time Sync:Adjust Network Time property. CAN and LIN devices do not use time to synchronize, so time synchronization is not required, per se, but their Local Time counters must be frequency-locked across all devices. This happens automatically as long as they are in the same PXI chassis. All devices allow you to remove the phase offset of Local Time via the Interface:Adjust Local Time property.
3. Configure all Ethernet sessions to use the same timescale for replay, using the Interface:Ethernet:Output Stream Timescale property. Local Time must be used when synchronizing with CAN or LIN.
4. If synchronizing CAN or LIN with Ethernet, do the following:
   - Configure the CAN/LIN sessions to use a start trigger that is asserted from a future time event on one of the Ethernet devices.
   - With the CAN/LIN sessions, use nxConnectTerminals to connect a PXI trigger to Start Trigger.
   - With one of the Ethernet sessions, use XNET Connect Terminals to connect Time Trigger to the same PXI trigger used on the CAN/LIN devices.
5. Set the Auto Start? property to False on all sessions.
6. On the Ethernet sessions, write a Future Time Wait frame into each session with an absolute timestamp that is the start time when the session should start outputting data. Each session must use a timestamp that represents a common moment in time across all devices, but from within the view of that individual device. Whether

or not all devices use the same absolute timestamp depends on whether there is a phase offset in the time counters within each device. If no phase offset, the timestamp must be the same. Otherwise, the timestamp must compensate for the measured phase offset.

7. Write the respective CAN/LIN/Ethernet data frames to be replayed into each session. The absolute time of each frame timestamp is not important, but the relative timing is. Hardware will use the timing rules described in Handling Timestamps for the relative timing of each data frame.
8. If synchronizing CAN or LIN with Ethernet, use the same Ethernet session from step 4 to pulse Time Trigger at the correct start time, using XNET Future Time Trigger. Use the same start time value that was used for the Future Time Wait frame on the same session.
9. Start all sessions. The sessions must be started before the start time occurs. Hardware will wait until the start time occurs, and then it will immediately start outputting the subsequent data frames.

Related concepts:

- Handling Timestamps
- Timescales
- Frame Output Stream Mode

Getting Started with NI-XNET LabVIEW API

NI-XNET driver software provides the ability to build projects using NI-XNET, learn from and build onto NI-XNET examples, and use the NI-XNET palette to create your own virtual instrument (VI).

LabVIEW Project

Using NI-XNET for LabVIEW, you can start with a LabVIEW project and create NI-XNET sessions to use within a VI to read or write network data.

Within a LabVIEW project, you can create NI-XNET sessions used within a VI to read or write network data. Using LabVIEW project sessions is best suited for static applications, in that the network data does not change from one execution to the next. Even if your application is more dynamic, a LabVIEW project is an excellent introduction to NI-XNET concepts.

To get started, open a new LabVIEW project, right-click **My Computer**, and select **New»NI-XNET Session**. In the resulting dialog, the window on the left provides an introduction to the NI-XNET session in the LabVIEW project. The introduction links to help topics that describe how to create a session in the project, including a description of the session modes.

NI-XNET Examples

NI-XNET includes LabVIEW examples that demonstrate a wide variety of use cases. The examples build on the basic concepts to demonstrate more in-depth use cases. Most of the examples create a session at run time rather than a LabVIEW project.

To view the NI-XNET examples, select **Find Examples...** from the LabVIEW Help menu. When you browse examples by task, NI-XNET examples are under Hardware Input and Output. The examples are grouped by protocol in Automotive Ethernet, CAN, FlexRay, and LIN folders, and each folder contains shared examples. You can write NI-XNET applications for any of these protocols; this organization helps you to find examples for your specific hardware product.

Open an example VI by double-clicking its name in the NI Example Finder. To run the example, select values using the front panel controls, then read the instructions on the front panel to run the examples. A few examples are suggested to get started with NI-XNET:

Automotive Ethernet (**Hardware Input and Output»Automotive Ethernet**)

- Ethernet Basic Input and Output.lvproj (Ethernet Reader.vi with Ethernet Writer.vi)
- NI-XNET IP Stack Simple TCP.lvproj (Simple TCP - Server.vi with Simple TCP - Client.vi)

CAN (**Hardware Input and Output»CAN»NI-XNET»Intro to Sessions**)

Signal Sessions

- CAN Signal Input Single Point.vi with CAN Signal Output Single Point.vi
- CAN Signal Input Waveform.vi with CAN Signal Output Waveform.vi

Frame Sessions

- CAN Frame Input Stream.vi with any output example.

FlexRay (**Hardware Input and Output»FlexRay»Intro to Sessions**)

Signal Sessions

- FlexRay Signal Input Single Point.vi with FlexRay Signal Output Single Point.vi
- FlexRay Signal Input Waveform.vi with FlexRay Signal Output Waveform.vi

Frame Sessions

- FlexRay Frame Input Stream.vi with any output example.

LIN (**Hardware Input and Output»LIN»NI-XNET»Intro to Sessions**)

Signal Sessions

- LIN Signal Input Single Point.vi with LIN Signal Output Single Point.vi
- LIN Signal Input Waveform.vi with LIN Signal Output Waveform.vi

Frame Sessions

- LIN Frame Input Stream.vi with any output example.

NI-XNET Function Palette

After learning the fundamentals of NI-XNET with a LabVIEW project and the examples, you can begin to write your own application.

The NI-XNET functions palette includes nodes that you drag to your VI block diagram. When your VI block diagram is open, this palette is in the **Measurement I/O»XNET** functions palette.

To view help for each node in the NI-XNET functions palette, open the context help window by selecting **Show Context Help** from the LabVIEW **Help** menu (or pressing <Ctrl-H>). As you hover over each node or subpalette, a brief summary appears. To open the complete help, click the **Detailed help** link in the summary.

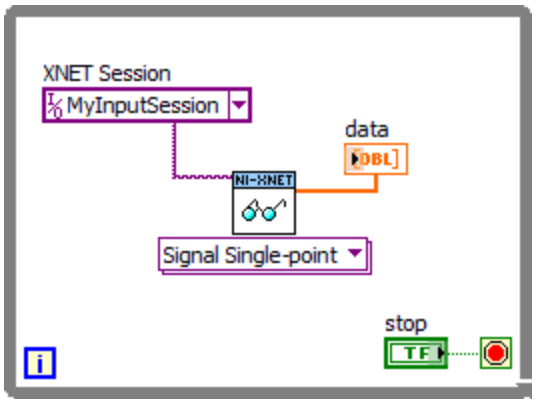
The NI-XNET controls palette includes I/O name controls that you drag to the your VI front panel. These controls enable the VI end user to select NI-XNET objects from the front panel. You view help for these controls in the same manner as on the functions palette.

Related concepts:

- XNET I/O Names

Basic Programming Model

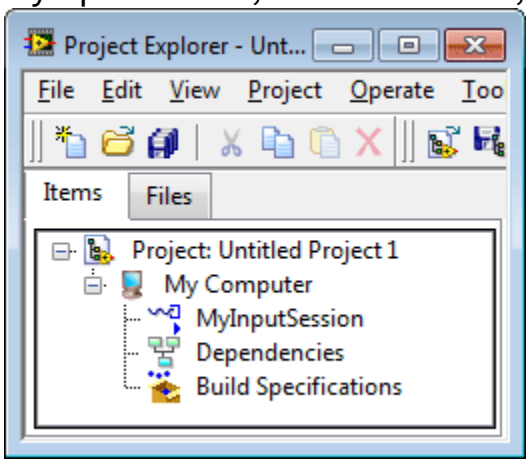
The LabVIEW block diagram in the following figure shows the basic NI-XNET programming model.



Basic Programming Model for NI-XNET for LabVIEW

Complete the following steps to create this block diagram:

1. Create an NI-XNET session in a LabVIEW project. The session name is MyInputSession, as shown below, and the mode is Signal Input Single-Point.



2. Create a new VI in the project and open the block diagram.
3. Drag a while loop to the diagram. Right-click the loop condition (the stop sign) and create a control (stop button).
4. Drag the NI-XNET session from a LabVIEW project to the while loop. This creates the XNET session wired to the corresponding XNET Read VI.
5. Right-click the **data** output from the XNET Read VI and create an indicator.
6. Run the VI. View the received signal values. Stop the VI when you are done.

When you complete the preceding steps, you have created a fully functional NI-XNET application.

You can create sessions for other input or output modes using the same technique. When you drag an output session to the diagram, NI-XNET creates a constant for data

and wires that constant to the XNET Write VI. You can enter constant values to write, or to change the data at run time, right-click the constant and select **Change to Control**.

NI-XNET enables you to create sessions for multiple hardware interfaces. For each interface, you can use multiple input sessions and multiple output sessions simultaneously. The sessions can use different modes. For example, you can use a Signal Input Single-Point session at the same time you use a Frame Input Stream session.

The NI-XNET functions palette includes nodes that extend this programming model to perform tasks such as:

- Creating a session at run time (instead of a LabVIEW project).
- Controlling the configuration and state of a session.
- Browsing and selecting a hardware interface.
- Managing and browsing database files.
- Creating frames or signals at run time (instead of using a database file).

The following topics describe the fundamental concepts used within NI-XNET. Each topic explains how to perform extended programming tasks.

Related concepts:

- Signal Input Single-Point Mode
- Frame Input Stream Mode

Displaying Available Interfaces

Measurement and Automation Explorer (MAX)

Use NI MAX to view your available NI-XNET hardware, including all devices and interfaces.

To view hardware in your local Windows system, select **Devices and Interfaces** under **My System**. Each NI-XNET device is listed by hardware model name followed by port name, for example, NI PCI-8517 "FlexRay1, FlexRay2".

Select each NI-XNET device to view its physical ports. Each port is listed with the current interface name assignment, such as FlexRay1.

In the selected port's window on the right, you can change one property: the interface name. Therefore, you can assign a different interface name than the default. For example, you can change the interface for physical port 2 of a PCI-8517 to FlexRay1 instead of FlexRay2. The blinking LED test panel assists in identifying a specific port when your system contains multiple instances of the same hardware product (for example, a chassis with five CAN devices).

To view hardware in a remote LabVIEW Real-Time system, find the desired system under **Remote Systems** and select **Devices and Interfaces** under that system. The features of NI-XNET devices and interfaces are the same as the local system.

I/O Name

When you create a session at run time, you pass the desired interface to the XNET Create Session VI. The interface uses the XNET Interface I/O name type.

The XNET Interface I/O name has a drop-down list of all available NI-XNET interfaces. This list matches the list of interfaces shown in NI MAX. You select a specific interface from the list for use with the XNET Create Session VI.

By right-clicking the **XNET Create Session VI** interface input, you can create a constant or control for the XNET Interface I/O name. The constant is placed on your block diagram. You typically use a constant when you have only a single NI-XNET device, to use fixed names such as CAN1 and CAN2. The control is placed on your front panel. You typically use a control when you have a large number of NI-XNET devices and want the VI end user to select from available interfaces.

LabVIEW Project

When you create a session in a LabVIEW project, you enter the interface in the session dialog. This dialog has a list of available interfaces, in a manner similar to the XNET Interface I/O name.

If you are creating a session in a LabVIEW project and do not yet have NI-XNET hardware in your system, you can type an interface name such as CAN1 in the dialog.

This enables you to create sessions and program VIs prior to installing the hardware.

System Configuration API

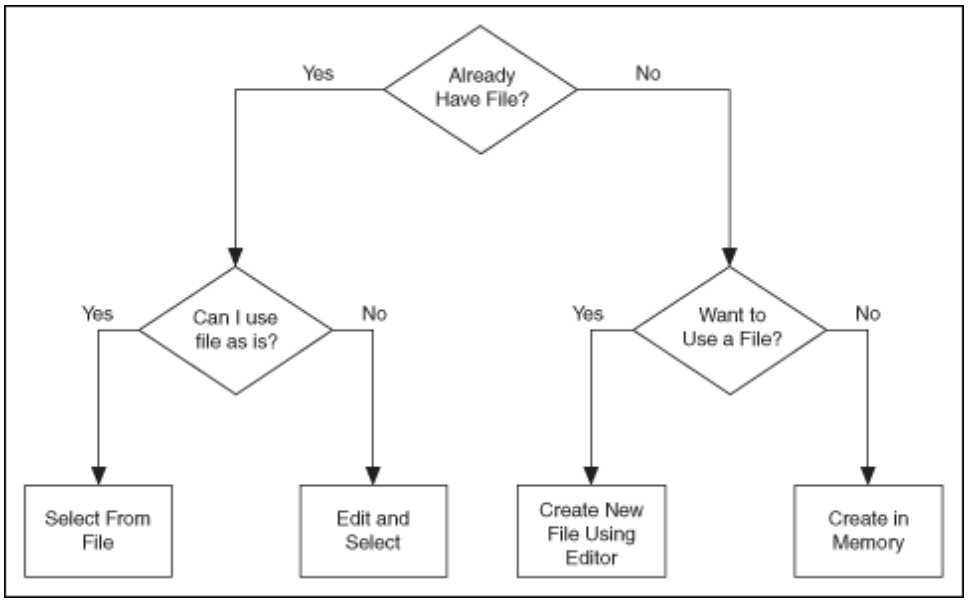
In some cases, you may need to provide features similar to NI MAX within your own application. For example, if you distribute your LabVIEW application to end users who are not familiar with MAX, you may need to display a similar view within the application itself.

The System Configuration API can be used to query for available XNET hardware, including devices, such as PXIe cards, and interfaces (for example, CAN ports). For additional information on the System Configuration API, refer to ***NI System Configuration API Help***, which is available at ni.com/docs.

Database Programming for LabVIEW API

The NI-XNET software provides various methods for creating your application database configuration. The following figure shows a process for deciding the database source. A description of each step in the process follows the flowchart.

Figure 3. Decision Process for Choosing a Database Source



Already Have File?

If you are testing an ECU used within a vehicle, the vehicle maker (or the maker's supplier) already may have provided a database file. This file likely would be in CANdb, FIBEX, AUTOSAR, or LDF format. When you have this file, using NI-XNET is relatively straightforward.

Can I Use File As Is?

Is the file up to date with respect to your ECU(s)?

If you do not know the answer to this question, the best choice is to assume Yes and begin using NI-XNET with the file. If you encounter problems, you can use the techniques discussed in "Edit and Select" to update your application without significant redesign.

Select From File

There are three options for selecting the database objects to use for NI-XNET sessions:

- A LabVIEW project
- I/O Names
- Property Nodes

- **Edit in Memory**: First, you select the frames or signals for the NI-XNET session using one of the options described in ***Select From File***, above.

  Next, you wire the selected objects to the corresponding property node and set properties to change the value. When you edit the object using its property node, this changes the representation in memory, but does not save the change to the file. When you pass the object into the XNET Create Session VI, the changes in memory (not the original file) are used.

- **Edit the File**: The NI-XNET Database Editor is a tool for editing database files for use with NI-XNET. Using this tool, you open an existing file, edit the objects, and save those changes. You can save the changes to the existing file or a new file.

  When you have a file with the changes you need, you select objects in your

application as described in "Select From File."

Edit and Select

There are two options for editing the database objects to use for NI-XNET sessions: edit in memory and edit the file.

Want to Use a File?

If you do not have a usable database file, you can choose to create a file or avoid files altogether for a self-contained application.

Create New File Using Editor

You can use the NI-XNET Database Editor to create a new database file. Once you have a file, you select objects in your application as described in "Select From File."

As a general rule, for FlexRay applications, using a FIBEX file is recommended. FlexRay communication configuration requires a large number of complex properties, and storage in a file makes this easier to manage. The NI-XNET Database Editor has features that facilitate this configuration.

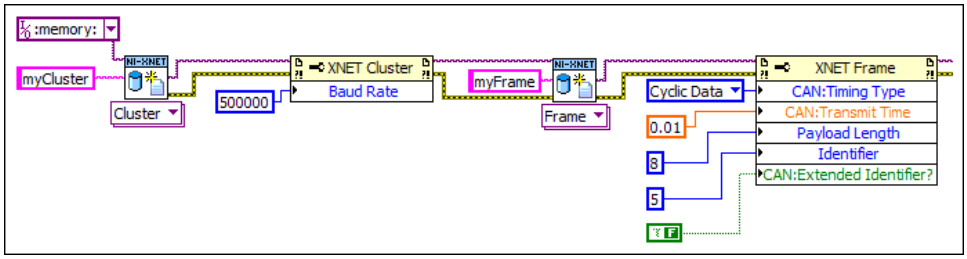
Create in Memory

You can use the XNET Database Create Object VI to create new database objects in memory. Using this technique, you can avoid files entirely and make your application self contained.

You configure each object you create using the property node. Each class of database object contains required properties that you must set (refer to "Required Properties").

Because CAN is a more straightforward protocol, it is easier to create a self-contained application. For example, you can create a session to transmit a CAN frame with only two objects.

Figure 4. Create a Cluster and Frame for CAN



The figure above shows a sample diagram that creates a cluster and frame in memory. The database name is :memory:. This special database name specifies a database that does not originate from a file. The cluster name is myCluster. For CAN, the only property required for the cluster is **Baud Rate**. The cluster is wired to create a frame object named myFrame. The frame has several required properties. The XNET Frame `CAN:Timing Type` property specifies how to transmit the frame, with **Cyclic Data** meaning to transmit every `CAN:Transmit Time` seconds (0.01, or 10 ms). The remaining properties configure the frame to use 8 bytes of payload data and CAN standard ID 5. If the subsequent diagram passed the frame to the XNET Create Session (Frame Output Queued) VI, this would create a session you can use to write data for transmit.

After you create and configure objects in memory, you can use >XNET Database Save VI to save the objects to a file. This enables you to implement a database editor within your application.

Multiple Databases Simultaneously

NI-XNET allows up to 63 database sessions to be open at the same time. You can open any database from a database file or in memory. To open multiple in-memory databases, use the name `:memory[<digit>]:`; for example, `:memory:`, `:memory1:`, `:memory2:`.

XNET I/O Names

LabVIEW I/O names (also known as refnum tags) are provided for various object classes within NI-XNET.

I/O names provide user interface features for easy configuration. You can use an I/O

name as a:

- **Control (or indicator)** : Use an I/O name control to select a specific instance on the front panel. NI-XNET I/O name controls are in the front panel **Modern»I/O»XNET** controls palette.Typically, you use I/O name controls to select an instance during configuration, and the instance is used at run time. For example, prior to running a VI, you can use XNET Signal I/O name controls to select signals to read. When the user runs the VI, the selected signals are passed to the XNET Create Session VI , followed by calls to the XNET Read VI to read and display data for the selected signals.As an alternative, you also can use I/O name controls to select an instance at run time. This applies when the VI always is running for the end user, and the VI uses distinct stages for configuration and I/O. Using the previous example, the user clicks XNET Signal I/O name controls to select signals during the configuration stage. Next, the user clicks a **Go** button to proceed to the I/O stage, in which the XNET Create Session VI and the XNET Read VI are called.You can build a standalone application (executable) that contains NI-XNET I/O name controls on its front panel. While running in an executable, the I/O name drop-down menu is supported, but the right-click menu is not operational.
- **Constant** : Use an I/O name constant to select a specific instance on the block diagram. NI-XNET I/O name constants are in the block diagram **Measurement I/O»XNET** functions palette. You can access I/O name constants only during configuration, prior to running the VI.

If you are using LabVIEW Real-Time (RT), you can right-click the RT target within a LabVIEW Project and select the **Connect** menu item. This connects to the RT target over TCP/IP, which in turn enables the user interface of NI-XNET I/O names to operate remotely. You can select names from the databases on the RT target and menu items to manage database deployments.

At run time, the VIs use I/O names to access features for the selected instance. The I/O name has two simultaneous LabVIEW types:

- **String** : When you wire the I/O name to a LabVIEW string, the string contains the selected instance name. Use this string to store the I/O name is a portable form, such as a text file.You can wire a LabVIEW string directly to an I/O name.
- **Refnum** : At run time, the I/O name contains a numeric reference to the instance for use with NI-XNET property nodes and VIs. The property node for the I/O name provides access to its configuration. The VIs provide methods for the instance, such

as to change state (start/stop), or access data (read/write).

I/O Name Classes

NI-XNET includes the following I/O name classes:

Session

Each session represents a connection between your National Instruments hardware and hardware products on the external network. Your application uses XNET sessions to read and write I/O data.

The session I/O name is primarily for sessions created during configuration using a LabVIEW project. When you create a session at run time with the XNET Create Session VI , the I/O name serves only as a refnum (its string is irrelevant).

Database Classes

To communicate with hardware products on the external network, NI-XNET applications must understand how that hardware communicates in the actual embedded system, such as the vehicle. This embedded communication is described within a standardized file, such as CANdb (`.dbc`), FIBEX (`.xml`), AUTOSAR (`.arxml`), or LIN Description File (`.ldf`). Within NI-XNET, this file is referred to as a database. The database contains many object classes, each of which describes a distinct entity in the embedded system:

- **Database** : Each database is represented as a distinct instance in NI-XNET. Although the I/O name string can be the complete file path, it typically uses a shortened alias.
- **Cluster** : Each database contains one or more clusters, where the cluster represents a collection of hardware products all connected over a shared cabling harness. In other words, each cluster represents a single network. For example, the database may describe a single vehicle, where the vehicle contains one Body CAN cluster, another Powertrain CAN cluster, and one Chassis FlexRay cluster.
- **ECU** : Each Electronic Control Unit (ECU) represents a single hardware product in the embedded system. The cluster contains one or more ECUs, all connected over a network cable. Multiple clusters can contain a single ECU, in which case it behaves as a gateway between the clusters.

- **Frame** : Each frame represents a unique unit of data transfer over the cluster cable. The frame bits contain payload data and an identifier that specifies the data (signal) content. Only one ECU in the cluster transmits each frame, and one or more ECUs receive each frame.
- **Signal** : Each frame contains zero or more values, each of which is called a signal. For example, the first two bytes of a frame payload may represent a temperature, and the third payload byte may represent a pressure. Within the database, each signal specifies its name, position, and length of the raw bits in the frame, and a scaling formula to convert raw bits to/from a physical unit. The physical unit uses a LabVIEW double-precision floating-point numeric type. The signal is the highest level of abstraction for embedded networks. When you use an XNET Session to read/write signal values as physical units, your application does not need to be concerned with protocol (CAN/FlexRay/LIN) and frame encoding details.
- **LIN Schedule** : The LIN protocol is different than CAN or FlexRay, in that it supports multiple schedules that determine when frames transmit. You can change the current schedule at runtime.
- **LIN Schedule Entry** : Each LIN Schedule contains one or more entries, or slots. Each entry in turn contains one or more frames that can transmit during the entry's time slot. A single frame can be located in multiple LIN schedules and within multiple LIN schedule entries.

Additional database classes include PDU and Subframe.

System Classes

These classes describe hardware in your National Instruments system, such as PXI or a desktop PC containing PCI cards.

- **Device** : This represents the National Instruments device for CAN/FlexRay/LIN, such as a PXI or PCI card. Each NI-XNET device contains one or more interfaces.
- **Interface** : This represents a single CAN, FlexRay, or LIN connector (port) on the device. Within NI-XNET, the interface is the object used to communicate with external hardware described in the database. When you create an NI-XNET session, you specify a physical and logical connection between the NI interface and a cluster. Because the cluster represents a single physical cable harness, it does not make sense to connect the NI interface to multiple clusters simultaneously.
- **Terminal** : Each interface contains various terminals. The terminals are for NI-XNET synchronization features, to connect triggers and timebases (clocks) to/from the

interface hardware. The terminal I/O name is for selecting a string input to the XNET Connect Terminals or XNET Disconnect Terminals VI, both of which operate on the session. Unlike the other I/O name classes, the terminal does not provide refnum features such as property nodes.

Related concepts:

- XNET Cluster I/O Name
- XNET Database I/O Name
- XNET ECU I/O Name
- XNET Frame I/O Name
- XNET Interface I/O Name
- XNET Signal I/O Name
- XNET Session I/O Name
- XNET LIN Schedule I/O Name
- XNET LIN Schedule Entry I/O Name
- XNET PDU I/O Name
- XNET Subframe I/O Name
- XNET Device I/O Name
- XNET Terminal I/O Name
- Getting Started with NI-XNET LabVIEW API

XNET Cluster I/O Name

Each database contains one or more clusters, where the cluster represents a collection of hardware products all connected over a shared cabling harness. In other words, each cluster represents a single CAN network or FlexRay network. For example, the database may describe a single vehicle, where the vehicle contains a Body CAN cluster, a Powertrain CAN cluster, and a Chassis FlexRay cluster.

Use the XNET Cluster I/O name to select a cluster, access properties, and invoke methods. For general information about I/O names, such as when to use them, refer to NI-XNET I/O Names.

User Interface

When you select the drop-down arrow on the right side of the I/O name, you see a list

of all clusters known to NI-XNET, followed by a separator (line), then a list of menu items.

Each cluster in the drop-down list uses the syntax specified in String Use. The list of clusters spans all database aliases known to NI-XNET. If you have not added an alias, the list of clusters is empty.

You can select a cluster from the drop-down list or by typing the name. As you type a name, LabVIEW selects the closest match from the list.

Right-clicking the I/O name displays a menu of LabVIEW items and items specific to NI-XNET.

The XNET Cluster I/O name includes the following menu items (in right-click or drop-down menus):

- **Browse For Database File** : If you have an existing CANdb (`.dbc`), FIBEX (`.xml`), AUTOSAR (`.arxml`), LIN Description File (`.ldf`), or NI-CAN (`.ncd`) database file, select this item to add an alias to NI-XNET. Use the file dialog to browse to the database file on your system. When you select **OK** , NI-XNET adds an alias to the file. The alias uses the filename, such as MyDatabase for a file path of `C:\Embedded\Vehicle5\MyDatabase.dbc`. If the alias is not unique, NI-XNET appends a number per LabVIEW conventions (for example, MyDatabase 2). After adding the alias, you can select the objects in that database from any NI-XNET I/O name.
- **New XNET Database** : If you do not have an existing database file, select this item to launch the NI-XNET Database Editor. You can use the NI-XNET Database Editor to create objects for the database and then save to a file. When you save the file, the NI-XNET Database Editor also adds an alias. Therefore, after you save from the editor, the clusters in the database become available in the XNET Cluster I/O name drop-down list.
- **Edit XNET Database** : If you selected a cluster using the I/O name, select this item to launch the NI-XNET Database Editor with that cluster's database file. You can use the editor to make changes to the database file, including the cluster.
- **Manage Database Aliases** : Select this menu item to open a dialog for managing aliases. You can review your list of aliases and associated file paths, remove an alias (without deleting the file), and add new aliases.

If you are using LabVIEW Real-Time (RT), you can right-click the RT target within **LabVIEW Project** and select the **Connect** menu item. This connects to the RT target over TCP/IP, which in turn enables the user interface of NI-XNET I/O names to operate remotely. If you open the **Manage** dialog while connected to an RT target, the dialog provides features for reviewing the list of databases on the RT target, deploying a new database from Windows to the RT target, and undeploying a database (removing an alias and file from RT target).

String Use

Use one of two syntax conventions for the string in the XNET Cluster I/O name:

- <alias>.<cluster>
- <alias>

The first syntax convention is the most complete, in that it contains the name of a database alias, followed by a dot separator, followed by the name of the cluster within that database. Use this syntax with FIBEX files, which contain multiple named clusters.

The second syntax convention uses the database alias only. This is supported for CANdb (`.dbc`), LDF (`.ldf`), and NI-CAN (`.ncd`) files, which always contain a single unnamed cluster.

Lowercase letters (a–z), uppercase letters (A–Z), numbers, underscore (_), and space ( ) are valid characters for <alias>. Period (.) and other special characters are not supported within the <alias> name. Because the <alias> is used as the filename portion of an internal filepath (that is, absolute path and file extension removed), it must use the minimum file conventions for all operating systems. The alias name is not case sensitive.

Lowercase letters (a–z), uppercase letters (A–Z), numbers, and the underscore (_) are valid characters for <cluster>. The space ( ), period (.), and other special characters are not supported within the cluster name. The cluster name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The cluster name is limited to 128 characters. The cluster name is case sensitive.

For FIBEX (`.xml`) and AUTOSAR (`.arxml`) files, the <cluster> name is stored in the database file. For CANdb (`.dbc`), LDF (`.ldf`), or NI-CAN (`.ncd`) files, no <cluster>

name is stored in the file, so NI-XNET uses the name Cluster when a name is required.

You can use the XNET Cluster I/O name string as follows:

- **XNET Create Session (Frame In Stream, Frame Out Stream, Generic) VI** : The stream I/O sessions transfer an arbitrary sequence of frames on the cluster, so only the XNET Cluster is required for configuration (not specific frames). The Generic instance provides advanced features to pass in database object names as strings, including the cluster. Within Create Session, NI-XNET opens the database file, reads information for the cluster, and then closes the database.
- **Open Refnum** : LabVIEW can open the XNET Cluster I/O name automatically. Wire the I/O name to a property node or VI, and the refnum is opened prior to the first use.

Refnum Use

You can use the XNET Cluster I/O name refnum as follows:

- **XNET Cluster Property Node** : The XNET Cluster property node provides information about its contents, such as the list of all XNET Frames. This property node is the most common use case for the XNET Cluster I/O name, because it provides the features needed to query and/or edit the cluster contents in the database file.
- **Create (ECU, Frame)** : If you are creating a new database, call this VI to create a new XNET ECU or Frame within the cluster.

Related concepts:

- XNET Database I/O Name
- XNET I/O Names
- XNET ECU I/O Name
- XNET Frame I/O Name
- XNET PDU I/O Name
- XNET Signal I/O Name

XNET Database I/O Name

To communicate with hardware products on the external CAN/FlexRay/LIN network, NI-XNET applications must understand how that hardware communicates in the actual embedded system, such as the vehicle. This embedded communication is described within a standardized file, such as CANdb (`.dbc`) or NI-CAN (`.ncd`) for CAN, FIBEX (`.xml`), or AUTOSAR (`.arxml`). Within NI-XNET, this file is referred to as a database. The database contains many object classes, each of which describes a distinct entity in the embedded system.

Use the XNET Database I/O name to select a database, access properties, and invoke methods (for example, save). For general information about I/O names, such as when to use them, refer to NI-XNET I/O names.

When using a database file with NI-XNET, you can specify the file path or specify an alias to the file. The alias provides a shorter, easier-to-read name for use within your application. For example, for the file path `C:\Documents and Settings\All Users\Documents\Vehicle5\MyDatabase.dbc` , you can add an alias named MyDatabase. In addition to improving readability, the alias concept isolates your LabVIEW application from the specific filepath. For example, if your application uses the alias MyDatabase , and you change its file path to `C:\Embedded\Vehicle5\MyDatabase.dbc` , your LabVIEW application continues to run without change. The alias concept is used in most NI-XNET features, including deployment of database files to LabVIEW Real-Time targets. For more information about aliases, refer to What Is an Alias?.

User Interface

When you select the drop-down arrow on the right side of the I/O name, you see a list of all database aliases known to NI-XNET, followed by a separator (line), then a list of menu items. If you have not added an alias, the first list is empty.

You can select an alias from the drop-down list or by typing the name. As you type a name, LabVIEW selects the closest match from the list.

Right-clicking the I/O name displays a menu of LabVIEW items and items specific to NI-XNET.

The XNET Database I/O name provides the following menu items in right-click and drop-down menus:

- **Browse For Database File** : If you have an existing CANdb (`.dbc`), FIBEX (`.xml`), AUTOSAR (`.arxml`), LIN Description File (`.ldf`), or NI-CAN (`.ncd`) database file, select this item to add an alias to NI-XNET. Use the file dialog to browse to the database file on your system. When you select **OK** , NI-XNET adds an alias to the file. The alias uses the filename, such as MyDatabase for a file path of `C:\Embedded\Vehicle5\MyDatabase.dbc`. If the alias is not unique, NI-XNET appends a number per LabVIEW conventions (for example, MyDatabase 2). After adding the alias, you can select the objects in that database from any NI-XNET I/O name.
- **New XNET Database** : If you do not have an existing database file, select this item to launch the NI-XNET Database Editor. You can use the NI-XNET Database Editor to create objects for the database and then save to a file. When you save the file, the NI-XNET Database Editor also adds an alias. Therefore, after you save from the editor, the database becomes available in the XNET Database I/O name drop-down list.
- **Edit XNET Database** : If you have selected a database using the I/O name, select this item to launch the NI-XNET Database Editor with that database file. You can use the editor to make changes to the database file.
- **Manage Database Aliases** : Select this menu item to open a dialog to manage aliases. You can review your list of aliases and associated file paths, remove an alias (without deleting the file), and add new aliases.If you are using LabVIEW Real-Time (RT), you can right-click the RT target within a LabVIEW Project and select the **Connect** menu item. This connects to the RT target over TCP/IP, which in turn enables the user interface of NI-XNET I/O names to operate remotely. If you open the Manage dialog while connected to an RT target, the dialog provides features to review the list of databases on the RT target, deploy a new database from Windows to the RT target, and undeploy a database (remove the alias and file from the RT target).

String Use

Use one of two syntax conventions for the XNET Database I/O name string:

- <alias>
- <filepath>

The <alias> is the database file short name, used as an alias to the complete filepath. This syntax is the only option available when you select a database from the drop-

down list or use the menu items.

Lowercase letters (a–z), uppercase letters (A–Z), numbers, underscore (_), and space ( ) are valid characters for <alias>. Period (.) and other special characters are not supported within the <alias> name. Because the <alias> is used as the filename portion of an internal filepath (that is, absolute path and file extension removed), it must use the minimum file conventions for all operating systems. The alias name is not case sensitive.

The <filepath> is the absolute path to the text database file, using the operating system file conventions (such as `C:\Embedded\Vehicle5\MyDatabase.dbc`). You can use the <filepath> syntax to open the database directly, without adding an alias to NI-XNET.

Valid characters for <filepath> include any characters your operating system supports for an absolute file path. Relative file paths are not supported. Because special characters typically are required in an absolute filepath (such as `\` or `:`), NI-XNET uses these characters to distinguish the <alias> syntax from <filepath> syntax.

You can use the XNET Database I/O name string as follows:

- **XNET Create Session (Generic) VI** : The commonly used XNET Create Session VI instances use signal or frame I/O names and not the database directly. The Generic instance provides advanced features to pass in database object names as strings, including the database itself. Within Create Session, NI-XNET opens the database file, reads information, and closes the database.
- **Open Refnum** : LabVIEW can open the XNET Database I/O name automatically. Wire the I/O name to a property node or VI, and the refnum is opened prior to the first use.
- **Remove Alias, Deploy, Undeploy** : These VIs enable you to manage an existing alias at run time, much like the Manage Database Aliases dialog. The XNET Database is passed in as a string, and is not opened as a refnum. These VIs require the <alias> syntax for the XNET Database (not filepath).

Refnum Use

You can use the XNET Database I/O name refnum as follows:

- **XNET Database Property Node** : The XNET Database property node provides information on its contents, such as the list of all XNET Clusters. This property node is the most common use case for the XNET Database I/O name, because it provides the features needed to query and/or edit all database contents from the top-level down to all other objects.
- **XNET Database Create (Cluster) VI** : If you are creating a new database, call this VI to create a new XNET Cluster within the database.
- **XNET Database Save VI** : After you set properties for the database or any of its objects, call this VI to save those changes to the file. The file is saved in the FIBEX format.

Related concepts:

- XNET Cluster I/O Name
- XNET I/O Names
- Creating a Database Alias
- XNET Signal I/O Name
- XNET Frame I/O Name

XNET Device I/O Name

Within NI-XNET, the term device refers to your National Instruments CAN/FlexRay/LIN hardware product, such as a PXI or PCI card.

Each device contains one or more interfaces to communicate on a CAN/FlexRay/LIN network.

User Interface

The XNET Device I/O name is not intended for use on VI front panels or as a diagram constant. This I/O name class is returned as the value of the following properties:

- XNET System Devices
- XNET Interface Device

The value these properties return is used as a refnum only.

String Use

NI-XNET determines the XNET Device I/O name string syntax internally. This syntax may change in future versions, so string display or formation is not recommended.

Refnum Use

You can use the XNET Device I/O name refnum as a device node. The XNET Device property node provides information such as the serial number and list of interfaces contained within the device.

LabVIEW closes the XNET device automatically. This occurs when the last top-level VI using the device goes idle (aborted or stops executing).

Related concepts:

- XNET Interface I/O Name
- XNET I/O Names

XNET ECU I/O Name

Each Electronic Control Unit (ECU) represents a single hardware product in the embedded system. The cluster contains one or more ECUs, all connected by a CAN, FlexRay, or LIN cable.

Use the XNET ECU I/O name to select an ECU, access properties, and invoke methods. For general information about I/O names, such as when to use them, refer to XNET I/O Names.

User Interface

Before using the ECU I/O name, you must use **Select Database** to select a cluster within a known database. Because the NI-XNET hardware interface physically connects to a single cluster in your embedded system, it makes sense to limit the list to ECUs contained in a single cluster.

When you select the drop-down arrow on the right side of the I/O name, you see a list of all ECUs within the selected cluster, followed by a separator (line), then a list of

menu items.

Each ECU in the drop-down list uses the syntax specified in String Use.

You can select an ECU from the drop-down list or by typing the name. As you type a name, LabVIEW selects the closest match from the list.

Right-clicking the I/O name displays a menu of LabVIEW items and items specific to NI-XNET.

The XNET ECU I/O name provides the following menu items in right-click and drop-down menus:

- **Select Database** : In the drop-down list, this menu item opens a dialog to select a cluster. In the right-click menu, this item provides a pull-right menu to select the cluster.You must select a cluster to specify the frame selection scope. The list of clusters uses the same list as the XNET Cluster I/O name. Each cluster name typically is just the database <alias> only, but when a FIBEX file is used, each <alias>.<cluster> name is listed.
- **Browse For Database File** : If you have an existing CANdb (`.dbc`), FIBEX (`.xml`), AUTOSAR (`.arxml`), LIN Description File (`.ldf`), or NI-CAN (`.ncd`) database file, select this item to add an alias to NI-XNET. Use the file dialog to browse to the database file on your system. When you select **OK** , NI-XNET adds an alias to the file. The alias uses the filename, such as MyDatabase for a file path of `C:\Embedded\Vehicle5\MyDatabase.dbc` . If the alias is not unique, NI-XNET appends a number per LabVIEW conventions (for example, MyDatabase 2). After adding the alias, you can select the objects in that database from any NI-XNET I/O name.After adding the alias, it appears in the **Select Database** list, and the first cluster in the database is selected automatically.
- **New XNET Database** : If you do not have an existing database file, select this item to launch the NI-XNET Database Editor. You can use the NI-XNET Database Editor to create objects for the database and then save to a file. When you save the file, the NI-XNET Database Editor also adds an alias. Therefore, after you save from the editor, the clusters in the database become available in the **Select Database** list. You must select the desired cluster when you finish using the NI-XNET Database Editor.
- **Edit XNET Database** : If you have selected a cluster using **Select Database** , select this item to launch the NI-XNET Database Editor with that cluster's database file.

You can use the editor to make changes to the database file, including the ECUs.

- **Manage Database Aliases** : Select this menu item to open a dialog to manage aliases. You can review your list of aliases and associated file paths, remove an alias (without deleting the file), and add new aliases.If you are using LabVIEW Real-Time (RT), you can right-click the RT target within a LabVIEW Project and select the **Connect** menu item. This connects to the RT target over TCP/IP, which in turn enables the user interface of NI-XNET I/O names to operate remotely. If you open the Manage dialog while connected to an RT target, the dialog provides features to review the list of databases on the RT target, deploy a new database from Windows to the RT target, and undeploy a database (remove the alias and file from an RT target).

String Use

Use the following syntax convention for the XNET ECU I/O name string:

<ecu>\n<dbSelection>

The string contains the ECU name, followed by a new line (\n) as a separator, followed by the selected cluster name.

When you drop the I/O name onto your front panel, the control displays only one line by default. This enables the VI end user to focus on selecting the <ecu> , rather than the more complex syntax that includes <dbSelection> .

Lowercase letters (a–z), uppercase letters (A–Z), numbers, and the underscore (_) are valid characters for <ecu>. The space ( ), period (.), and other special characters are not supported within the ECU name. The <ecu> name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The <ecu> name is limited to 128 characters. The ECU name is case sensitive.

For FIBEX (`.xml`), AUTOSAR (`.arxml`), LIN Description File (`.ldf`), and CANdb (`.dbc`) files, the database file stores the <ecu> name. ECU specifications are not provided within NI-CAN (`.ncd`) files.

The <dbSelection> is appended to the ECU name to ensure that the XNET ECU I/O name is unique. LabVIEW requires each I/O name to use a unique name, because each instance is located using its name. By appending the cluster name, NI-XNET ensures

that the entire name is unique in large applications that use multiple NI-XNET interfaces (multiple clusters). The characters for <dbSelection> are the same as the name you selected using **Select Database** , which uses the same syntax convention as the XNET Cluster I/O name. To view the <dbSelection> when the I/O name is displayed, resize its constant/control to show multiple lines.

You can use the XNET ECU I/O name string as follows:

- **Open Refnum** : LabVIEW can open the XNET ECU I/O name automatically. Wire the I/O name to a property node or VI, and the refnum is opened prior to the first use.

Refnum Use

You can use the XNET ECU I/O name refnum as follows:

- **XNET ECU Property Node** : The XNET ECU property node provides the list of all frames the ECU transmits and receives. When you are creating an application to test a single ECU product, these frame lists help you create sessions for input/output of all frames (or signals) to fully test the ECU behavior.

Related concepts:

- XNET Cluster I/O Name
- XNET I/O Names
- XNET Frame I/O Name

XNET Frame I/O Name

Each frame represents a unique unit of data transfer over the cluster cable. The frame bits contain payload data and an identifier that specifies the data (signal) content. Only one ECU in the cluster transmits each frame, and one or more ECUs receive each frame.

For CAN, each frame is identified by its arbitration ID. The XNET Frame Identifier and CAN:Extended Identifier? properties specify this arbitration ID.

For FlexRay, each frame is identified by its location within the FlexRay cycle and channels. The XNET Frame Identifier , FlexRay:Base Cycle , FlexRay:Cycle Repetition ,

FlexRay:Channel Assignment , and FlexRay:In Cycle Repetitions:Enabled? properties specify this location.

Use the XNET Frame I/O name to select a frame, access properties, and invoke methods. For general information about I/O names, such as when to use them, refer to NI-XNET I/O names.

User Interface

Before using the frame I/O name, you must use **Select Database** to select a cluster within a known database. Because the NI-XNET hardware interface physically connects to a single cluster in your embedded system, it makes sense to limit the list to frames contained in a single cluster.

When you select the drop-down arrow on the right side of the I/O name, you see a list of all frames within the selected cluster, followed by a separator (line), then a list of menu items.

Each frame in the drop-down list uses the syntax specified in String Use.

You can select a frame from the drop-down list or by typing the name. As you type a name, LabVIEW selects the closest match from the list.

Right-clicking the I/O name displays a menu of LabVIEW items and items specific to NI-XNET.

The XNET Frame I/O name includes the following menu items in right-click and drop-down menus:

- **Select Database** : In the drop-down list, this menu item opens a dialog to select a cluster. In the right-click menu, this item includes a pull-right menu to select the cluster.You must select a cluster to specify the frame selection scope. The list of clusters uses the same list as the XNET Cluster I/O name. Each cluster name typically is just the database <alias> only, but when a FIBEX file is used, each <alias>.<cluster> name is listed.
- **Browse For Database File** : If you have an existing CANdb (`.dbc`), FIBEX (`.xml`), AUTOSAR (`.arxml`), LIN Description File (`.ldf`), or NI-CAN (`.ncd`) database file, select this item to add an alias to NI-XNET. Use the file dialog to

browse to the database file on your system. When you select **OK** , NI-XNET adds an alias to the file. The alias uses the filename, such as MyDatabase for a file path of `C:\Embedded\Vehicle5\MyDatabase.dbc` . If the alias is not unique, NI-XNET appends a number per LabVIEW conventions (for example, MyDatabase 2). After adding the alias, you can select the objects in that database from any NI-XNET I/O name.After adding the alias, it appears in the **Select Database** list, and the first cluster in the database is selected automatically.

- **New XNET Database** : If you do not have an existing database file, select this item to launch the NI-XNET Database Editor. You can use the NI-XNET Database Editor to create objects for the database and then save to a file. When you save the file, the NI-XNET Database Editor also adds an alias. Therefore, after you save from the editor, the clusters in the database become available in the **Select Database** list. You must select the desired cluster when you finish using the NI-XNET Database Editor.
- **Edit XNET Database** : If you have selected a cluster using **Select Database** , select this item to launch the NI-XNET Database Editor with that cluster's database file. You can use the editor to make changes to the database file, including the frames.
- **Manage Database Aliases** : Select this menu item to open a dialog to manage aliases. You can review your list of aliases and associated file paths, remove an alias (without deleting the file), and add new aliases.If you are using LabVIEW Real-Time (RT), you can right-click the RT target within a LabVIEW Project and select the **Connect** menu item. This connects to the RT target over TCP/IP, which in turn enables the user interface of NI-XNET I/O names to operate remotely. If you open the Manage dialog while connected to an RT target, the dialog provides features to review the list of databases on the RT target, deploy a new database from Windows to the RT target, and undeploy a database (remove the alias and file from the RT target).

String Use

Use the following syntax convention for the XNET Frame I/O name string:

<frame>\n<dbSelection>

The string contains the frame name, followed by a new line (\n) as a separator, followed by the selected cluster name.

When you drop the I/O name onto your front panel, the control displays only one line

by default. This enables the VI end user to focus on selecting the <frame> , rather than the more complex syntax that includes <dbSelection> .

Lowercase letters (a–z), uppercase letters (A–Z), numbers, and the underscore (_) are valid characters for <frame> . The space ( ), period (.), and other special characters are not supported within the <frame> name. The <frame> name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The <frame> name is limited to 128 characters. The frame name is case sensitive.

For all supported database formats, the database file stores the <frame> name.

The <dbSelection> is appended to the frame name to ensure that the XNET Frame I/O name is unique. LabVIEW requires each I/O name to use a unique name, because each instance is located using its name. By appending the cluster name, NI-XNET ensures that the entire name is unique in large applications that use multiple NI-XNET interfaces (multiple clusters). The characters for <dbSelection> are the same as the name you selected using **Select Database** , which uses the same syntax convention as the XNET Cluster I/O name. To view the <dbSelection> when the I/O name is displayed, resize its constant/control to show multiple lines.

You can use the XNET Frame I/O name string as follows:

- **XNET Create Session (Frame In Queued, Frame In Single-Point, Frame Out Queued, Frame Out Single-Point, Generic) VI** : The queued I/O sessions transfer a sequence of values for a single frame in the cluster. The single-point I/O sessions transfer the recent value for a list of frames. The Generic instance provides advanced features to pass in database object names as strings, including one or more frames. For all of these instances, the XNET Frame I/O name is passed in as input, but is used as a string. Within Create Session, NI-XNET opens the database file, reads information for the frames, and closes the database.
- **Open Refnum** : LabVIEW can open the XNET Frame I/O name automatically. Wire the I/O name to a property node or VI, and the refnum is opened prior to the first use.

Refnum Use

You can use the XNET Frame I/O name refnum as follows:

- **XNET Frame Property Node** : The XNET Frame property node provides the information such as the network identification, number of payload bytes, and the list of signals within the frame.
- **XNET Database Create (Signal, Subframe) VI** : If you are creating a new database, call this VI to create a new XNET Signal or Subframe within the frame.

Related concepts:

- Using NI-CAN
- XNET Database I/O Name
- XNET ECU I/O Name
- XNET Cluster I/O Name
- XNET Signal I/O Name
- XNET I/O Names
- XNET Interface I/O Name

XNET Interface I/O Name

The XNET interface represents a single CAN, FlexRay, LIN, or Ethernet connector (port) on the device. Within NI-XNET, the interface is the object used to communicate with external hardware described in the database. When you create an NI-XNET session, you specify a physical and logical connection between the NI interface and a cluster. Because the cluster represents a single physical cable harness, it does not make sense to have the NI interface connected to multiple clusters simultaneously.

The XNET interface I/O name is used to select an interface to pass to the XNET Create Session VI , and to read hardware information properties. For general information about I/O names, such as when you can use them, refer to NI-XNET I/O Names.

User Interface

Select the drop-down arrow on the right side of the I/O name to display a list of all interfaces available in your system. You can select an interface from the drop-down list or by typing the name. As you type a name, LabVIEW selects the closest match from the list.

You can type the name of an interface that does not exist in your system. For example,

you can type `CAN4` even if only CAN1 and CAN2 exist in your system. The check for an actual CAN4 interface does not occur until it is used at runtime (for example, within a session).

If you are using LabVIEW Real-Time (RT), you can right-click the RT target within LabVIEW project and select **Connect**. This connects to the RT target over TCP/IP, which in turn enables the user interface of NI-XNET I/O names to operate remotely. The XNET interface drop-down list shows **(target disconnected)** until you connect the RT target. When the RT target is connected, the drop-down list shows all interfaces on that RT target (for example, a PXI chassis).

When you right-click the I/O name, the menu contains LabVIEW items including **I/O Name Filtering**. Use this menu item to filter the interface names shown in the I/O name. You can show all interfaces, CAN only, FlexRay only, LIN only, or Ethernet only. The selected filtering is saved along with the VI that uses the I/O name.

I/O Name Filtering is available at edit-time only, before you run your VI. This is done under the assumption that if you filter for a specific protocol, the code in the VI block diagram works with that protocol only. Therefore, you do not want to allow the VI end users to select a different protocol at runtime.

String Use

Use one of two syntax conventions for the string in the XNET Interface I/O name:

<protocol><n>

The protocol is CAN for a CAN interface, FlexRay for a FlexRay interface, or ENET for an Ethernet interface. The protocol name is not case sensitive.

The number <n> identifies the specific interface within the scope of the protocol. The numbering starts at 1. For example, if you have a two-port CAN device and a two-port FlexRay device in your system, the interface names will be CAN1 , CAN2 , FlexRay1 , and FlexRay2 .

Although you can change the interface number <n> within MAX, the typical practice is to allow NI-XNET to select the number automatically. NI-XNET always starts at 1 and increments for each new interface found. If you do not change the number in MAX, and

your system always uses a single two-port CAN device, you can write all of your applications to assume CAN1 and CAN2. For as long as that CAN card exists in your system, NI-XNET uses the same interface numbers for that device, even if new CAN cards are added.

You can use the XNET Interface I/O name string as follows:

- **XNET Create Session** VI : All XNET Create Session VI instances use the interface I/O name to specify the interface for the session's I/O. Within the XNET Create Session VI , NI-XNET opens the interface and configures the hardware for the session's I/O communication.

Refnum Use

The XNET interface refnum always is opened and closed automatically. When you wire the I/O name to one of the following nodes, LabVIEW opens a refnum for the interface. The refnum is closed automatically when it is no longer used. The XNET interface refnum features are for hardware information and identification, prior to using the interface within a session. You can use the XNET Frame I/O name refnum as follows:

- **XNET Interface Property Node** : The XNET Interface property node provides information for the hardware, such as the port number next to the connector.
- **Blink** : If no session is in use for the interface, you can use this VI to identify a specific interface within a large system (for example, chassis with multiple CAN devices).

Related concepts:

- XNET Device I/O Name
- XNET I/O Names
- XNET Frame I/O Name
- XNET Terminal I/O Name

XNET Session I/O Name

The XNET Session represents a connection between your National Instruments CAN/FlexRay/LIN hardware and hardware products on the external CAN/FlexRay/LIN

network. Your application uses sessions to read and write I/O data.

Use the session class I/O name primarily for sessions created at edit time using a LabVIEW project. When you create a session at run time with the XNET Create Session VI, the I/O name serves only as a refnum (its string is irrelevant).

Use the XNET Session I/O name to select a session defined in a LabVIEW project, for use with methods such as the XNET Read or XNET Write VIs. For general information about I/O names, such as when to use them, refer to NI-XNET I/O Names.

User Interface

When you select the drop-down arrow on the right side of the I/O name, you see a list of all available sessions.

If you are using a VI within a LabVIEW project, the available sessions are listed under the VI target (**RT** or **My Computer**). If you are using a VI within a built application (`.exe`), the available sessions are in the NI-XNET configuration file (`nixnetSession.txt`) the LabVIEW build generates.

You can select a session from the drop-down list or by typing the name. As you type a name, LabVIEW selects the closest match from the list.

If you are using LabVIEW Real-Time (RT), you can right-click the RT target within a LabVIEW project and select the **Connect** menu item. This connects to the RT target over TCP/IP, which in turn enables the user interface of NI-XNET I/O names to operate remotely. The XNET session drop-down list shows **(target disconnected)** until you connect the RT target. When the RT target is connected, the drop-down list shows all sessions on that RT target (for example, PXI chassis).

When you right-click the I/O name, the menu contains LabVIEW items and the following items:

- **Edit XNET Session** : This item opens the Properties dialog for the selected session. You can change the session properties and select **OK** to save those changes in the project. This menu item is available at edit time only, before you run your VI.
- **New XNET Session** : This launches the wizard to create a new XNET Session. The new session is created under the same target as the current VI. This menu item is

available at edit time only, before you run your VI.

String Use

Use a session name from the drop-down list.

LabVIEW conventions for names in a project allow any character, including special characters such as space ( ) and slash (/).

The session name is case sensitive.

The XNET Session I/O name string is not used directly, in that it always is opened automatically for use as a refnum.

Refnum Use

The XNET Session refnum always is opened and closed automatically. When you wire the I/O name to a node, LabVIEW opens a refnum for the session. The refnum is closed automatically when your top-level VIs are no longer executing (idle). You also can close the refnum by calling the XNET Clear VI.

The XNET Session refnum features represent the core NI-XNET functionality, in that you use the session to read and write data on the embedded network using the following property node and VIs:

- XNET Session Node: Use the XNET Session property node to change the session configuration.
- XNET Read VI: Read data for an input session and read state information for the session interface.
- XNET Write VI: Write data for an output session.
- XNET Start, XNET Stop, and XNET Flush VIs: Control the session and buffer states.
- XNET Wait and XNET Create Timing Source VIs: Handle notification of events that occur in the session.
- XNET Connect Terminals and XNET Disconnect Terminals VIs: Connect/disconnect synchronization terminals.
- XNET Clear VI : Close the session refnum, including stopping all I/O. If this VI is not called, LabVIEW closes the refnum automatically when your top-level VIs are no longer executing (idle).

Related concepts:

- XNET I/O Names
- XNET Signal I/O Name

XNET Signal I/O Name

Each frame contains zero or more values, each of which is called a signal. For example, the first two bytes of a frame payload may represent a temperature, and the third payload byte may represent a pressure. Within the database, each signal specifies its name, position, and length of the raw bits in the frame, and a scaling formula to convert raw bits to/from a physical unit. The physical unit uses a LabVIEW double-precision floating-point numeric type. The signal is the highest level of abstraction for embedded networks. When you use an XNET Session to read/write signal values as physical units, your application does not need to be concerned with protocol (CAN/FlexRay/LIN) details and frame encoding.

Use the XNET Signal I/O name to select a signal, access properties, and invoke methods. For general information about I/O names, such as when to use them, refer to NI-XNET I/O Names.

User Interface

Before using the signal I/O name, you must use **Select Database** to select a cluster within a known database. Because the NI-XNET hardware interface physically connects to a single cluster in your embedded system, it makes sense to limit the list to signals contained in a single cluster.

When you select the drop-down arrow on the right side of the I/O name, you see a list of all signals within the selected cluster, followed by a separator (line), then a list of menu items.

Each signal in the drop-down list uses the syntax specified in String Use.

You can select a signal from the drop-down list or by typing the name. As you type a name, LabVIEW selects the closest match from the list.

Right-clicking the I/O name displays a menu of LabVIEW items and items specific to NI-

XNET.

The XNET Signal I/O name provides the following menu items in right-click and drop-down menus:

- **Select Database**: In the drop-down list, this menu item opens a dialog to select a cluster. In the right-click menu, this item provides a pull-right menu to select the cluster. You must select a cluster to specify the signal selection scope. The list of clusters uses the same list as the XNET Cluster I/O name. Each cluster name typically is just the database <alias> only, but when a FIBEX file is used, each <alias>.<cluster> name is listed.
- **Browse For Database File**: If you have an existing CANdb (`.dbc`), FIBEX (`.xml`), AUTOSAR (`.arxml`), LIN Description File (`.ldf`), or NI-CAN (`.ncd`) database file, select this item to add an alias to NI-XNET. Use the file dialog to browse to the database file on your system. When you select **OK**, NI-XNET adds an alias to the file. The alias uses the filename, such as MyDatabase for a file path of `C:\Embedded\Vehicle5\MyDatabase.dbc`. If the alias is not unique, NI-XNET appends a number per LabVIEW conventions (for example, MyDatabase 2). After adding the alias, you can select the objects in that database from any NI-XNET I/O name.After adding the alias, it appears in the **Select Database** list, and the first cluster in the database is selected automatically.
- **New XNET Database**: If you do not have an existing database file, select this item to launch the NI-XNET Database Editor. You can use the NI-XNET Database Editor to create objects for the database and then save to a file. When you save the file, the NI-XNET Database Editor also adds an alias. Therefore, after you save from the editor, the clusters in the database become available in the **Select Database** list. You must select the desired cluster when you finish using the NI-XNET Database Editor.
- **Edit XNET Database** : If you have selected a cluster using **Select Database**, select this item to launch the NI-XNET Database Editor with that cluster's database file. You can use the editor to make changes to the database file, including the signals.
- **Manage Database Aliases**: Select this menu item to open a dialog to manage aliases. You can review your list of aliases and associated file paths, remove an alias (without deleting the file), and add new aliases.If you are using LabVIEW Real-Time (RT), you can right-click the RT target within a LabVIEW Project and select the **Connect** menu item. This connects to the RT target over TCP/IP, which in turn enables the user interface of NI-XNET I/O names to operate remotely. If you open

the Manage dialog while connected to an RT target, the dialog provides features to review the list of databases on the RT target, deploy a new database from Windows to the RT target, and undeploy a database (remove the alias and file from the RT target).

String Use

Use one of two syntax conventions for the XNET Signal I/O name string:

- <signal>\n<dbSelection>
- <frame>.<signal>\n<dbSelection>

Use the first syntax convention when the signal name is unique within the cluster (not used in multiple frames). This is the recommended design for signal names, because it provides a clear and simple syntax. The string contains the name of the signal, followed by a new line (\n) as a separator, followed by the selected cluster name.

Use the second syntax convention when the signal name is used in multiple frames. The string contains the name of frame, followed by a dot separator, followed by the text of the first syntax convention (signal name and selected cluster).

When you drop the I/O name onto your front panel, the control displays only one line by default. This enables the VI end user to focus on selecting the <signal>, rather than the more complex syntax that includes <dbSelection>.

Lowercase letters (a–z), uppercase letters (A–Z), numbers, and the underscore (_) are valid characters for <signal>. The space ( ), period (.), and other special characters are not supported within the signal name. The <signal> name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The <signal> name is limited to 128 characters. The signal name is case sensitive.

For all supported database formats, the <signal> name is stored in the database file.

The <dbSelection> is appended to the signal name to ensure that the XNET Signal I/O name is unique. LabVIEW requires each I/O name to use a unique name, because each instance is located using its name. By appending the cluster name, NI-XNET ensures that the entire name is unique in large applications that use multiple NI-XNET interfaces (multiple clusters). The characters for <dbSelection> are the same as the

name you selected using **Select Database**, which uses the same syntax convention as the XNET Cluster I/O name. To view the <dbSelection> when the I/O name is displayed, resize its constant/control to show multiple lines.

You can use the XNET Signal I/O name string as follows:

- **XNET Create Session (Signal In Single-Point, Signal In Waveform, Signal In XY, Signal Out Single-Point, Signal Out Waveform, Signal Out XY, Generic) VI** : The single-point I/O sessions transfer the recent value for a list of signals. The waveform I/O sessions transfer signal data as LabVIEW waveforms. The XY I/O sessions transfer a sequence of values for each signal in a list. The Generic instance provides advanced features to pass in database object names as strings, including one or more signals. For all these instances, the XNET Signal I/O name is passed in as an input, but is used as a string. Within the XNET Create Session VI, NI-XNET opens the database file, reads information for the signals, and closes the database.
- **Open Refnum** : LabVIEW can open the XNET Signal I/O name automatically. Wire the I/O name to a property node or VI, and the refnum is opened prior to the first use.

Refnum Use

You can use the XNET Signal I/O name refnum as follows:

- **XNET Signal Property Node** : The XNET Signal property node provides information such as the signal position and size in the payload, scaling formula to physical units, and so on.

Related concepts:

- Using NI-CAN
- XNET Database I/O Name
- XNET Frame I/O Name
- XNET I/O Names
- XNET Session I/O Name
- XNET Cluster I/O Name

XNET Subframe I/O Name

Within your embedded network, some frames may use a feature called data multiplexing (also known as mode-dependent messages). The frame specifies a single signal called the data multiplexer. A specific range of bits within the multiplexed frame is designated to contain subframes. Each subframe contains a distinct set of signals, referred to as dynamic signals. When a frame is transmitted on the network, the data multiplexer signal value selects the subframe. For example, if the data multiplexer is 0, a subframe with dynamic signals A and B may exist in the last bytes; if the data multiplexer is 1, a subframe with dynamic signals C and D may exist in the same last bytes.

Use the XNET Subframe I/O name to access properties for a specific subframe.

User Interface

The XNET Subframe I/O name is not intended for use on VI front panels or as a diagram constant. This I/O name class is returned as the value of the following properties:

- XNET Frame Mux:Subframes
- XNET Signal Mux:Subframe

String Use

NI-XNET determines the XNET Subframe I/O name string syntax internally. This syntax may change in future versions, so string display or formation is not recommended.

You can use the XNET Frame I/O name string as follows:

- **Open Refnum**: LabVIEW can open the XNET Subframe I/O name automatically. Wire the I/O name to a property node or VI, and the refnum is opened prior to the first use.

Refnum Use

You can use the XNET Frame I/O name refnum as follows:

- XNET Subframe Property Node: The XNET Subframe property node provides the information such as the data multiplexer value for the subframe and the list of dynamic signals within the subframe.
- XNET Database Create (Dynamic Signal) VI: If you are creating a new database, call

this VI to create a new XNET Signal within the frame. This instance creates a dynamic signal contained within the subframe. To create a static signal that exists in all frame values, call the XNET Database Create (Signal) VI using the parent XNET Frame (not the subframe).

Related concepts:

- XNET I/O Names

XNET Terminal I/O Name

Each interface contains various terminals. The terminals are for NI-XNET synchronization features, to connect triggers and timebases (clocks) to/from the interface hardware.

Use the XNET Terminal I/O name to select a string input to the XNET Connect Terminals or XNET Disconnect Terminals VIs, both of which operate on the session. For general information about I/O names, such as when to use them, refer to NI-XNET I/O Names.

User Interface

When you select the drop-down arrow on the right side of the I/O name, you see a list of all terminals any NI-XNET interface uses.

You can select a terminal from the drop-down list or by typing the name. As you type a name, LabVIEW selects the closest match from the list.

The list of terminals is not specific to a particular interface. For example, if you have only a CAN device in your system, the drop-down list still contains terminals for FlexRay interfaces.

String Use

Use a terminal name from the drop-down list.

For a description of each name, refer to the XNET Connect Terminals VI.

Lowercase letters (a–z), uppercase letters (A–Z), numbers, and the underscore (_) are

valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The terminal name is not case sensitive.

The terminal name scope always is local to the XNET interface used within the session that you pass to the XNET Connect Terminals VI. One of the terminals (source or destination) is on the trigger bus (PXI backplane or PCI RTSI cable), and the other is within the XNET interface.

You can use the XNET Interface I/O name term as follows:

- **XNET Connect Terminals VI** : Connect a source terminal to a destination terminal on the interface.
- **XNET Disconnect Terminals VI** : Disconnect a pair of terminals on the interface.

Refnum Use

The XNET Terminal does not provide refnum features such as property nodes.

Related concepts:

- XNET I/O Names
- XNET Interface I/O Name

XNET LIN Schedule I/O Name

The LIN protocol is different than CAN or FlexRay, in that it supports multiple schedules that determine when frames transmit. You can change the current schedule at runtime. Within a database file, a cluster for LIN contains one or more LIN schedules. Each LIN schedule contains one or more LIN schedule entries.

Use the XNET LIN Schedule I/O name to select a schedule, access properties, and invoke methods. For general information about I/O names, such as when to use them, refer to XNET I/O Names.

User Interface

Before using the LIN Schedule I/O name, you must use **Select Database** to select a cluster within a known database. Because the NI-XNET hardware interface physically

connects to a single cluster in your embedded system, it makes sense to limit the list to schedules contained in a single cluster.

When you select the drop-down arrow on the right side of the I/O name, you see a list of all LIN schedules within the selected cluster, followed by a separator (line), then a list of menu items.

Each schedule in the drop-down list uses the syntax specified in String Use.

You can select a schedule from the drop-down list or by typing the name. As you type a name, LabVIEW selects the closest match from the list.

Right-clicking the I/O name displays a menu of LabVIEW items and items specific to NI-XNET.

The XNET LIN Schedule I/O name provides the following menu items in right-click and drop-down menus:

- **Select Database**: In the drop-down list, this menu item opens a dialog to select a cluster. In the right-click menu, this item provides a pull-right menu to select the cluster. You must select a cluster to specify the LIN schedule selection scope. The list of clusters uses the same list as the XNET Cluster I/O Name. Each cluster name typically is just the database <alias> only, but when a FIBEX file is used, each <alias>.<cluster> name is listed.
- **Browse For Database File**: If you have an existing CANdb (`.dbc`), FIBEX (`.xml`), AUTOSAR (`.arxml`), LIN Description File (`.ldf`), or NI-CAN (`.ncd`) database file, select this item to add an alias to NI-XNET. Use the file dialog to browse to the database file on your system. When you select **OK**, NI-XNET adds an alias to the file. The alias uses the filename, such as MyDatabase for a file path of `C:\Embedded\Vehicle5\MyDatabase.dbc`. If the alias is not unique, NI-XNET appends a number per LabVIEW conventions (for example, MyDatabase 2). After adding the alias, you can select the objects in that database from any NI-XNET I/O name.After adding the alias, it appears in the **Select Database** list, and the first cluster in the database is selected automatically.
- **Manage Database Aliases**: Select this menu item to open a dialog to manage aliases. You can review your list of aliases and associated file paths, remove an alias (without deleting the file), and add new aliases.If you are using LabVIEW Real-Time (RT), you can right-click the RT target within a LabVIEW Project and select the

Connect menu item. This connects to the RT target over TCP/IP, which in turn enables the user interface of NI-XNET I/O names to operate remotely. If you open the Manage dialog while connected to an RT target, the dialog provides features to review the list of databases on the RT target, deploy a new database from Windows to the RT target, and undeploy a database (remove the alias and file from the RT target).

String Use

Use the following syntax convention for the XNET LIN Schedule I/O name string:

<schedule>\n<dbSelection>

The string contains the LIN schedule name, followed by a new line (\n) as a separator, followed by the selected cluster name.

When you drop the I/O name onto your front panel, the control displays only one line by default. This enables the VI end user to focus on selecting the <schedule> , rather than the more complex syntax that includes <dbSelection>.

Lowercase letters (a–z), uppercase letters (A–Z), numbers, and the underscore (_) are valid characters for <schedule>. The space ( ), period (.), and other special characters are not supported within the schedule name. The <schedule> name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The <schedule> name is limited to 128 characters. The schedule name is case sensitive.

For LIN Description Files (`.ldf`), the database file stores the <schedule> name. The NI-CAN (`.ncd`) and CANdb (`.dbc`) file formats do not support LIN. The current version of NI-XNET does not support LIN with FIBEX (`.xml`) and AUTOSAR (`.arxml`).

The <dbSelection> is appended to the schedule name to ensure that the XNET LIN Schedule I/O name is unique. LabVIEW requires each I/O name to use a unique name, because each instance is located using its name. By appending the cluster name, NI-XNET ensures that the entire name is unique in large applications that use multiple NI-XNET interfaces (multiple clusters). The characters for <dbSelection> are the same as the name you selected using **Select Database**, which uses the same syntax convention as the XNET Cluster I/O Name. To view the <dbSelection> when the I/O name is displayed, resize its constant/control to show multiple lines.

You can use the XNET LIN Schedule I/O name string as follows:

- **Open Refnum**: LabVIEW can open the XNET LIN Schedule I/O name automatically. Wire the I/O name to a property node or VI, and the refnum is opened prior to the first use.
- **Write (LIN Schedule Change)**: While running your session, you can change the currently running LIN schedule. You wire the XNET LIN Schedule I/O name to the XNET Write (State LIN Schedule Change) VI as a string to specify the schedule to execute.

Refnum Use

You can use the XNET LIN Schedule I/O name refnum as follows:

- **XNET LIN Schedule Property Node** : The LIN schedule property node provides the list of all schedule entries, plus other aspects of the schedule such as run mode.

Related concepts:

- XNET I/O Names

XNET LIN Schedule Entry I/O Name

Each LIN Schedule contains one or more entries, or slots. Each entry in turn contains one or more frames that can transmit during the entry's time slot. A single frame can be located in multiple LIN schedules and within multiple LIN schedule entries.

Use the XNET LIN Schedule Entry I/O name to access properties for a specific schedule entry.

User Interface

The XNET LIN Schedule Entry I/O name is not intended for use on VI front panels or as a diagram constant. This I/O name class is returned as the value of the XNET LIN Schedule Entries property.

String Use

NI-XNET determines the XNET LIN Schedule Entry I/O name string syntax internally. This syntax may change in future versions, so string display or formation is not recommended.

You can use the XNET LIN Schedule Entry I/O name string as follows:

- Open Refnum : LabVIEW can open the XNET LIN Schedule Entry I/O name automatically. Wire the I/O name to a property node or VI, and the refnum is opened prior to the first use.

Refnum Use

You can use the XNET LIN Schedule Entry I/O name refnum as follows:

- XNET LIN Schedule Entry Property Node : The XNET LIN Schedule Entry property node provides the information such as the entry type, list of frames transmitted, and so on.
- XNET Database Create (LIN Schedule Entry) VI : If you are creating a new database, call this VI to create a new XNET LIN Schedule Entry within the LIN schedule.

Related concepts:

- XNET I/O Names

XNET PDU I/O Name

Many FlexRay networks use the concept of a Protocol Data Unit (PDU) to implement configurations similar to CAN. The PDU is a container of signals. You can use a single PDU within multiple frames for faster timing. A single frame can contain multiple PDUs, each updated independently. For more information, refer to Protocol Data Units (PDUs) in NI-XNET.

Use the XNET PDU I/O name to select a PDU, access properties, and invoke methods. For general information about I/O names, such as when to use them, refer to XNET I/O Names.

User Interface

Before using the PDU I/O name, you must use **Select Database** to select a cluster within a known database. Because the NI-XNET hardware interface physically connects to a single cluster in your embedded system, it makes sense to limit the list to PDUs contained in a single cluster.

When you select the drop-down arrow on the right side of the I/O name, you see a list of all PDUs within the selected cluster, followed by a separator (line), then a list of menu items.

Each PDU in the drop-down list uses the syntax specified in String Use.

You can select a PDU from the drop-down list or by typing the name. As you type a name, LabVIEW selects the closest match from the list.

Right-clicking the I/O name displays a menu of LabVIEW items and items specific to NI-XNET.

The XNET PDU I/O name includes the following menu items in right-click and drop-down menus:

- **Select Database:** In the drop-down list, this menu item opens a dialog to select a cluster. In the right-click menu, this item provides a pull-right menu to select the cluster.You must select a cluster to specify the PDU selection scope. The list of clusters uses the same list as the XNET Cluster I/O name. Each cluster name typically is just the database <alias> only, but when a FIBEX file is used, each <alias>.<cluster> name is listed.
- **Browse For Database File:** If you have an existing CANdb (`.dbc`), FIBEX (`.xml`), AUTOSAR (`.arxml`), LIN Description File (`.ldf`), or NI-CAN (`.ncd`) database file, select this item to add an alias to NI-XNET. Use the file dialog to browse to the database file on your system. When you select **OK**, NI-XNET adds an alias to the file. The alias uses the filename, such as MyDatabase for a file path of `C:\Embedded\Vehicle5\MyDatabase.dbc`. If the alias is not unique, NI-XNET appends a number per LabVIEW conventions (for example, MyDatabase 2). After adding the alias, you can select the objects in that database from any NI-XNET I/O name.After adding the alias, it appears in the **Select Database** list, and the first cluster in the database is selected automatically.

- **New XNET Database:** If you do not have an existing database file, select this item to launch the NI-XNET Database Editor. You can use the NI-XNET Database Editor to create objects for the database and then save to a file. When you save the file, the NI-XNET Database Editor also adds an alias. Therefore, after you save from the editor, the clusters in the database become available in the **Select Database** list. You must select the desired cluster when you finish using the NI-XNET Database Editor.
- **Edit XNET Database:** If you have selected a cluster using **Select Database**, select this item to launch the NI-XNET Database Editor with that cluster's database file. You can use the editor to make changes to the database file, including the signals.
- **Manage Database Aliases:** Select this menu item to open a dialog to manage aliases. You can review your list of aliases and associated file paths, remove an alias (without deleting the file), and add new aliases.If you are using LabVIEW Real-Time (RT), you can right-click the RT target within a LabVIEW Project and select the **Connect** menu item. This connects to the RT target over TCP/IP, which in turn enables the user interface of NI-XNET I/O names to operate remotely. If you open the **Manage** dialog while connected to an RT target, the dialog provides features to review the list of databases on the RT target, deploy a new database from Windows to the RT target, and undeploy a database (remove the alias and file from the RT target).

String Use

Use the following syntax convention for the XNET PDU I/O name string:

<pdu>\n<dbSelection>

The string contains the PDU name, followed by a new line (\n) as a separator, followed by the selected cluster name.

When you drop the I/O name onto your front panel, the control displays only one line by default. This enables the VI end user to focus on selecting the <pdu>, rather than the more complex syntax that includes <dbSelection>.

Lowercase letters (a–z), uppercase letters (A–Z), numbers, and the underscore (_) are valid characters for <pdu>. The space ( ), period (.), and other special characters are not supported within the <pdu> name. The <pdu> name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The <pdu> name is limited

to 128 characters. The PDU name is case sensitive.

For all supported database formats, the database file stores the <pdu> name.

The <dbSelection> is appended to the PDU name to ensure that the XNET PDU I/O name is unique. LabVIEW requires each I/O name to use a unique name, because each instance is located using its name. By appending the cluster name, NI-XNET ensures that the entire name is unique in large applications that use multiple NI-XNET interfaces (multiple clusters). The characters for <dbSelection> are the same as the name you selected using **Select Database**, which uses the same syntax convention as the XNET Cluster I/O name. To view the <dbSelection> when the I/O name is displayed, resize its constant/control to show multiple lines.

You can use the XNET PDU I/O name string as follows:

- **XNET Create Session (Frame In PDU Queued, Frame In PDU Single-Point, Frame Out PDU Queued, Frame Out PDU Single-Point, Generic) VI:** These modes operate on PDUs in a manner equivalent to frames. The queued I/O sessions transfer a sequence of values for a single PDU in the cluster. The single-point I/O sessions transfer the recent value for a list of PDUs. The Generic instance provides advanced features to pass in database object names as strings, including one or more PDUs. For all instances, the XNET PDU I/O name is passed in as input, but is used as a string. Within Create Session, NI-XNET opens the database file, reads information for the PDUs, and closes the database.
- **Open Refnum:** LabVIEW can open the XNET PDU I/O name automatically. Wire the I/O name to a property node or VI, and the refnum is opened prior to the first use.

Refnum Use

You can use the XNET PDU I/O name refnum as follows:

- **XNET PDU Property Node:** The PDU property node provides information such as the PDU position and size in the frame, contained signals, and so on.

Related concepts:

- XNET I/O Names
- PDUs

- XNET Cluster I/O Name

Using NI-CAN

NI-CAN is the legacy application programming interface (API) for National Instruments CAN hardware. Generally speaking, NI-CAN is associated with the legacy CAN hardware, and NI-XNET is associated with the new NI-XNET hardware.

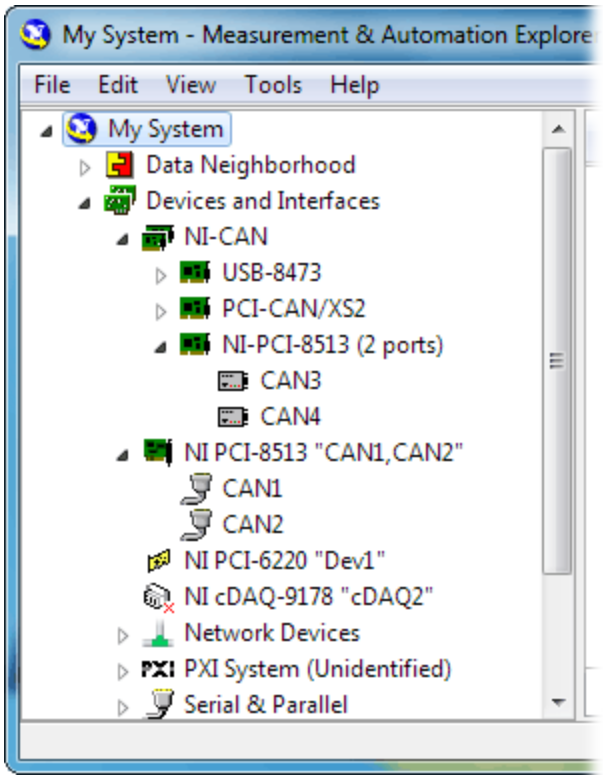
If you are starting a new application, you typically use NI-XNET (not NI-CAN).

Compatibility

If you have an existing application that uses NI-CAN, a compatibility library is provided so that you can reuse that code with a new NI-XNET CAN product. Because the features of the compatibility library apply to the NI-CAN API and not NI-XNET, it is described in the NI-CAN documentation. For more information, refer to the ***NI-CAN Manual***.

NI-XNET CAN Products in MAX

When the compatibility library is installed, NI-XNET CAN products also are visible in the **NI-CAN**branch under **Devices and Interfaces**. Here you can configure the devices for use with the NI-CAN API. This configuration is independent from the configuration of the same device for NI-XNET under the root of **Devices and Interfaces**. The following figure shows the same NI-XNET device, the NI PCI-8513, configured for use with the NI-XNET API (interfaces CAN1 and CAN2) and with the NI-CAN API (interfaces CAN3 and CAN4).



Transition

If you have an existing application that uses NI-CAN and intend to use only new NI-XNET hardware from now on, you may want to transition your code to NI-XNET.

NI-XNET unifies many concepts of the earlier NI-CAN API, but the key features are similar.

The following table lists NI-CAN terms and analogous NI-XNET terms.

NI-CAN Term	NI-XNET Term	Comment
CANdb file	Database	NI-XNET supports more database file formats than the NI-CAN Channel API, including the FIBEX, AUTOSAR, and LDF formats.
Message	Frame	The term Frame is the industry convention for the bits that transfer on the bus. This term is used in standards such as CAN.
Channel	Signal	The term Signal is the industry convention. This term is used in standards such as FIBEX and AUTOSAR.
Channel API	Session	Unlike NI-CAN, NI-XNET supports simultaneous use of channel (signal) I/O

NI-CAN Term	NI-XNET Term	Comment
Task	(Signal I/O)	and frame I/O.
Frame API CAN Object (Queue Length Zero)	Session (Frame I/O Single-Point)	The NI-CAN CAN Object provided both input (read) and output (write) in one object. NI-XNET provides a different object for each direction, for better control. If the NI-CAN queue length for a direction is zero, that is analogous to NI-XNET Frame I/O Single-Point.
Frame API CAN Object (Queue Length Nonzero)	Session (Frame I/O Queued)	If the NI-CAN queue length for a direction is nonzero, that is analogous to NI-XNET Frame I/O Queued.
Frame API Network Interface Object	Session (Frame I/O Stream)	The NI-CAN Network Interface Object provided both input (read) and output (write) in one object. NI-XNET provides a different object for each direction, for better control.
Interface	Interface	NI-CAN started interface names at CAN0, but NI-XNET starts at CAN1 (or FlexRay1).

Related concepts:

- Databases
- XNET Frame I/O Name
- XNET Signal I/O Name
- Sessions
- Interfaces

CAN Timing Type and Session Mode

For each XNET Frame CAN:Timing Type property value, this topic describes how the frame behaves for each XNET session mode.

An input session receives the CAN data frame from the network, and an output session transmits the CAN data frame. The CAN data frame data (payload) is mapped to/from signal values.

You use CAN remote frames to request the associated CAN data frame from a remote ECU. When Timing Type is Cyclic Remote or Event Remote, an input session transmits the CAN remote frame, and an output session receives the CAN remote frame.

Related concepts:

- Session Modes
- CAN Overview

Cyclic Data

The data frame transmits in a cyclic (periodic) manner. The XNET Frame CAN:Transmit Time property defines the time between cycles.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, and Frame Input Queued Modes

You specify the CAN frame (or its signals) when you create the session. When the CAN data frame is received, a subsequent call to the XNET Read VI returns its data. For information about how the data is represented for each mode, refer to Session Modes.

If the CAN remote frame is received, it is ignored (with no effect on the XNET Read VI).

Frame Input Stream Mode

You specify the CAN cluster when you create the session, but not the specific CAN frame. When the CAN data frame is received, a subsequent call to the XNET Read VI returns its data.

If the CAN remote frame is received, a subsequent call to the XNET Read VI for the stream returns it.

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

You specify the CAN frame (or its signals) when you create the session. When you write data using the XNET Write VI, the CAN data frame is transmitted onto the network. For information about how the data is represented for each mode, refer to Session Modes.

When the session and its associated interface are started, the first cycle occurs, and the CAN data frame transmits. After that first transmit, the CAN data frame transmits once every cycle, regardless of whether the XNET Read VI is called. If no new data is available for transmit, the next cycle transmits using the previous CAN data frame (repeats the payload).

If you pass the CAN remote frame to the XNET Read VI, it is ignored.

Frame Output Stream Mode

You specify the CAN cluster when you create the session, but not the specific CAN frame. When you write the CAN data frame using the XNET Read VI function, it is transmitted onto the network.

The stream I/O modes do not use the database-specified timing for frames. Therefore, CAN data and CAN remote frames transmit only when you pass them to the XNET Read VI function, and do not transmit cyclically afterward.

When using a stream output timing of immediate mode, data is transmitted onto the network as soon as possible.

When using a stream output timing of either Replay Exclusive or Replay Inclusive, data is transmitted onto the network based on the timestamps in the frame.

Event Data

The data frame transmits in an event-driven manner. For output sessions, the event is the XNET Read VI. The XNET Frame CAN:Transmit Time property defines the minimum interval.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, and Frame Input Queued Modes

The behavior is the same as Cyclic Data.

Frame Input Stream Mode

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the

database-specified timing for all frames, you can read either CAN data or CAN remote frames.

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

The behavior is the same as Cyclic Data, except that the CAN data frame does not continue to transmit cyclically after the data from the XNET Read VI has transmitted. Because the database-specified timing for the frame is event based, after the CAN data frames for the XNET Read VI have transmitted, the CAN data frame does not transmit again until a subsequent call to the XNET Read VI.

Frame Output Stream Mode

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can write either CAN data or CAN remote frames.

Cyclic Remote

The CAN remote frame transmits in a cyclic (periodic) manner, followed by the associated CAN data frame as a response.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, and Frame Input Queued Modes

You specify the CAN frame (or its signals) when you create the session. When the CAN data frame is received, a subsequent call to the XNET Read VI returns its data. For information about how the data is represented for each mode, refer to Session Modes.

When the session and its associated interface are started, the first cycle occurs, and the CAN remote frame transmits. This CAN remote frame requests data from the remote ECU, which soon responds with the associated CAN data frame (same identifier). After that first transmit, the CAN remote frame transmits once every cycle. You do not call the XNET Write VI for the session.

The CAN remote frame cyclic transmit is independent of the corresponding CAN data frame reception. When NI-XNET transmits a CAN remote frame, it transmits a CAN

remote frame again CAN:Transmit Time later, even if no CAN data frame is received.

Frame Input Stream Mode

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can read either CAN data or CAN remote frames.

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

You specify the CAN frame (or its signals) when you create the session. When you write data using the XNET Write VI, the CAN data frame is transmitted onto the network when the associated CAN remote frame is received (same identifier). For information about how the data is represented for each mode, refer to Session Modes.

Although the session receives the CAN remote frame, you do not call XNET Read VI to read that frame. NI-XNET detects the received CAN remote frame, and immediately transmits the next CAN data frame. Your application uses the XNET Write VI to provide the CAN data frames used for transmit. When you call the XNET Write VI, the CAN data frame does not transmit immediately, but instead waits for the associated CAN remote frame to be received.

Frame Output Stream Modes

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can write either CAN data or CAN remote frames.

Event Remote

The CAN remote frame transmits in an event-driven manner, followed by the associated CAN data frame as a response. For input sessions, the event is the XNET Write VI.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, and Frame Input Queued Modes

You specify the CAN frame (or its signals) when you create the session. When the CAN data frame is received, its data is returned from a subsequent call to the XNET Read VI. For information about how the data is represented for each mode, refer to Session Modes.

This CAN Timing Type and mode combination is somewhat advanced, in that you must call both the XNET Read VI and the XNET Write VI. You must call the XNET Write VI to provide the event that triggers the CAN remote frame transmit. When you call the XNET Write VI, the data is ignored, and one CAN remote frame transmits as soon as possible. Each call to the XNET Write VI transmits only one CAN remote frame, even if you provide multiple signal or frame values. When the remote ECU receives the CAN remote frame, it responds with a CAN data frame, which is received and read using the XNET Read VI.

Frame Input Stream Modes

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can read either CAN data or CAN remote frames.

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

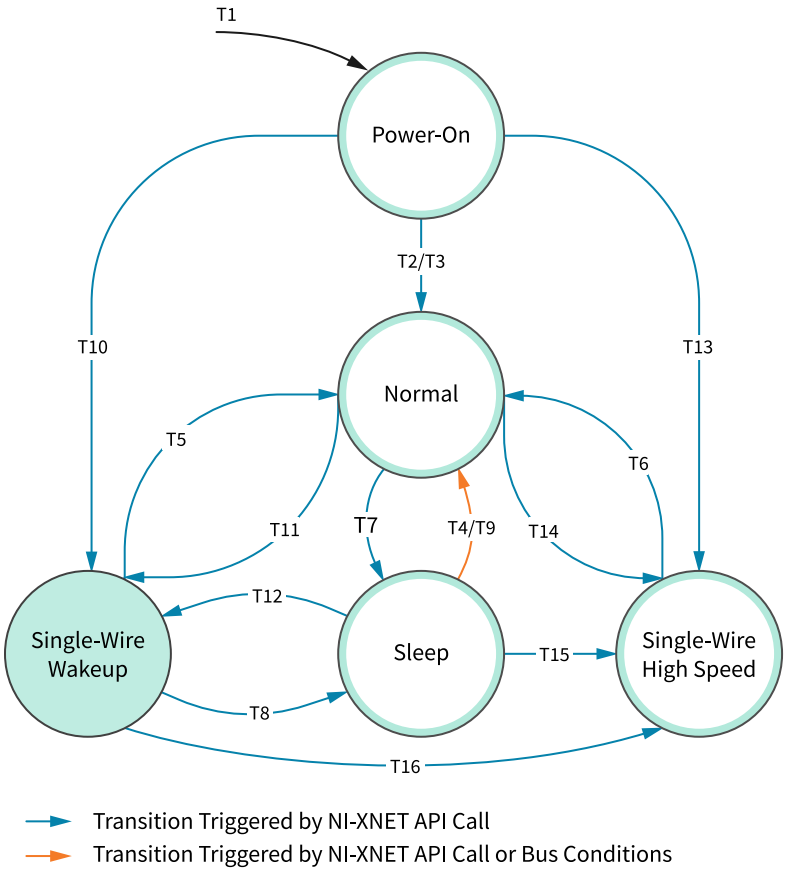
The behavior is the same as Cyclic Remote. When you write data using the XNET Write VI, the CAN data frame transmits onto the network when the associated CAN remote frame is received (same identifier). Unlike Cyclic Data, the remote ECU sends the associated CAN remote frame in an event-driven manner, but the behavior is the same regarding the XNET Write VI and the CAN data frame transmit.

Frame Output Stream Mode

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can write either CAN data or CAN remote frames.

CAN Transceiver State Machine

The CAN hardware internally runs a state machine for controlling the transceiver state. The transceiver can either be an internal transceiver or an external transceiver. On hardware that contains software selectable transceivers, you can configure the selected transceiver bysetting the Interface:CAN:Transceiver Type property. If you choose an external transceiver,you can configure its behaviors by setting the Interface:CAN:External Transceiver Config property. Both bus conditions as well as the Interface:CAN:Transceiver State property can affect the current transceiver state. The following state machine shows the different states of the transceiver state machine and how the various states transition.



T#	Condition	From	To
1	Power-on/close last session	Any	Power-on
2	Interface is started	Power-on	Normal
3	Interface:CAN:Transceiver State with value Normal	Power-on	Normal

T#	Condition	From	To
4	Interface:CAN:Transceiver State with value Normal	Sleep	Normal
5	Interface:CAN:Transceiver State with value Normal	SW Wakeup	Normal
6	Interface:CAN:Transceiver State with value Normal	SW High Speed	Normal
7	Interface:CAN:Transceiver State with value Sleep	Normal	Sleep
8	Interface:CAN:Transceiver State with value Sleep	SW Wakeup	Sleep
9	Wakeup Pattern received on the bus	Sleep	Normal
10	Interface:CAN:Transceiver State with value SWWakeup	Power-on	SW Wakeup
11	Interface:CAN:Transceiver State with value SWWakeup	Normal	SW Wakeup
12	Interface:CAN:Transceiver State with value SWWakeup	Sleep	SW Wakeup
13	Interface:CAN:Transceiver State with value SWHighSpeed	Power-on	SWHigh Speed
14	Interface:CAN:Transceiver State with value SWHighSpeed	Normal	SWHigh Speed
15	Interface:CAN:Transceiver State with value SWHighSpeed	Sleep	SWHigh Speed
16	Interface:CAN:Transceiver State with value SWHighSpeed	SW Wakeup	SWHigh Speed

Using FlexRay

This topic summarizes some useful NI-XNET features specific to the FlexRay protocol.

Starting Communication

FlexRay is a Time Division Multiple Access (TDMA) protocol, which means that all hardware products on the network share a synchronized clock. Slots of time for that clock determine when each frame transmits.

To start communication on FlexRay, the first step is to start the synchronized network clock. In the FlexRay database, two or more hardware products are designated to transmit a special startup frame. These products (nodes) are called coldstart nodes. Each coldstart node uses the startup frame to contribute its local clock as part of the shared network clock.

Because at least two coldstart nodes are required to start FlexRay communication,

your NI-XNET FlexRay interface may need to act as a coldstart node, and therefore transmit a special startup frame. The properties of each startup frame (including the time slot used) are specified in the FlexRay database.

The following scenarios apply to FlexRay startup frames:

- **Port to port** : When you get started with your NI-XNET FlexRay hardware, you can connect two FlexRay interfaces (ports) to run simple programs, such as the NI-XNET examples. Because this is a cluster with two nodes, each NI-XNET interface must transmit a different startup frame.
- **Connect to existing cluster** : If you connect your NI-XNET FlexRay interface to an existing cluster (for example, a FlexRay network within a vehicle), that cluster already must contain coldstart nodes. In this scenario, the NI-XNET interface should not transmit a startup frame.
- **Test single ECU that is coldstart** : If you connect to a single ECU (and nothing else), and that ECU is a coldstart node, the NI-XNET interface must transmit a startup frame. The NI-XNET interface must transmit a startup frame that is different than the startup frame the ECU transmits.
- **Test single ECU that is not coldstart** : If you connect to a single ECU (and nothing else), and that ECU is not a coldstart node, you must connect two NI-XNET interfaces. The ECU cannot communicate without two coldstart nodes (two clocks). According to the FlexRay specification, a single FlexRay interface can transmit only one startup frame. Therefore, you need to connect two NI-XNET FlexRay interfaces to the ECU, and each NI-XNET interface must transmit a different startup frame.

NI-XNET has two options to transmit a startup frame:

- **Key Slot Identifier** : The NI-XNET Session Node includes a property called Interface:FlexRay:Key Slot Identifier. This property specifies the static slot that the session interface uses to transmit a startup frame. The value of this property is zero (0) by default, meaning that no startup frame transmits. If you set this property, the value specifies the static slot (identifier) to transmit as a coldstart node. The startup frame transmits automatically when the interface starts, and its payload is null (no data). The session can be input or output, and the startup frame is not required in the session's list of frames/signals.
- **Output Startup Frame** : If you create an NI-XNET output session, and the session's list of frames/signals uses a startup frame, the NI-XNET interface acts as a coldstart

node.

To find startup frames in the database, look for a frame with the FlexRay:Startup? property true. You can use that frame name for an output session or use its identifier as the key slot. When selecting a startup frame, avoid selecting one that the ECUs you connect to already transmit.

Understanding FlexRay Frame Timing

When you use an NI-XNET database for FlexRay, the properties of each FlexRay frame specify the FlexRay data transfer timing. To understand how the FlexRay frame timing properties apply to NI-XNET sessions, refer to FlexRay Timing Type and Session Mode.

In LabVIEW Real-Time, NI-XNET provides a timing source you can use to synchronize your LabVIEW VI with the timing of frames. For more information, refer to Using LabVIEW Real-Time.

Protocol Data Unit (PDU)

Many FlexRay networks use a Protocol Data Unit (PDU) to implement configurations similar to CAN. The PDU is a signal container. You can use a single PDU within multiple frames for faster timing. A single frame can contain multiple PDUs, each updated independently. For more information, refer to Protocol Data Units (PDUs) in NI-XNET.

Related concepts:

- FlexRay Timing Type and Session Mode
- Using LabVIEW Real-Time
- PDUs

FlexRay Startup/Wakeup

Use the FlexRay Startup mechanism to take an idle interface and properly integrate into a FlexRay cluster.

If your cluster does not support the wakeup mechanism, this process is straightforward. After creating your FlexRay session, call `nxStart`, which causes the

interface to transition from **Default Config** to **Ready**, where it attempts to integrate with the FlexRay cluster. If your node is a coldstart node, it initiates integration; otherwise, it attempts to integrate with a running FlexRay cluster. Once integration has occurred, the interface transitions to **Normal Active**, where it typically remains while it is communicating with other FlexRay nodes. When you call `nxStop`, the interface transitions back to **Default Config** (via **Halt**) to be ready to start the process again.

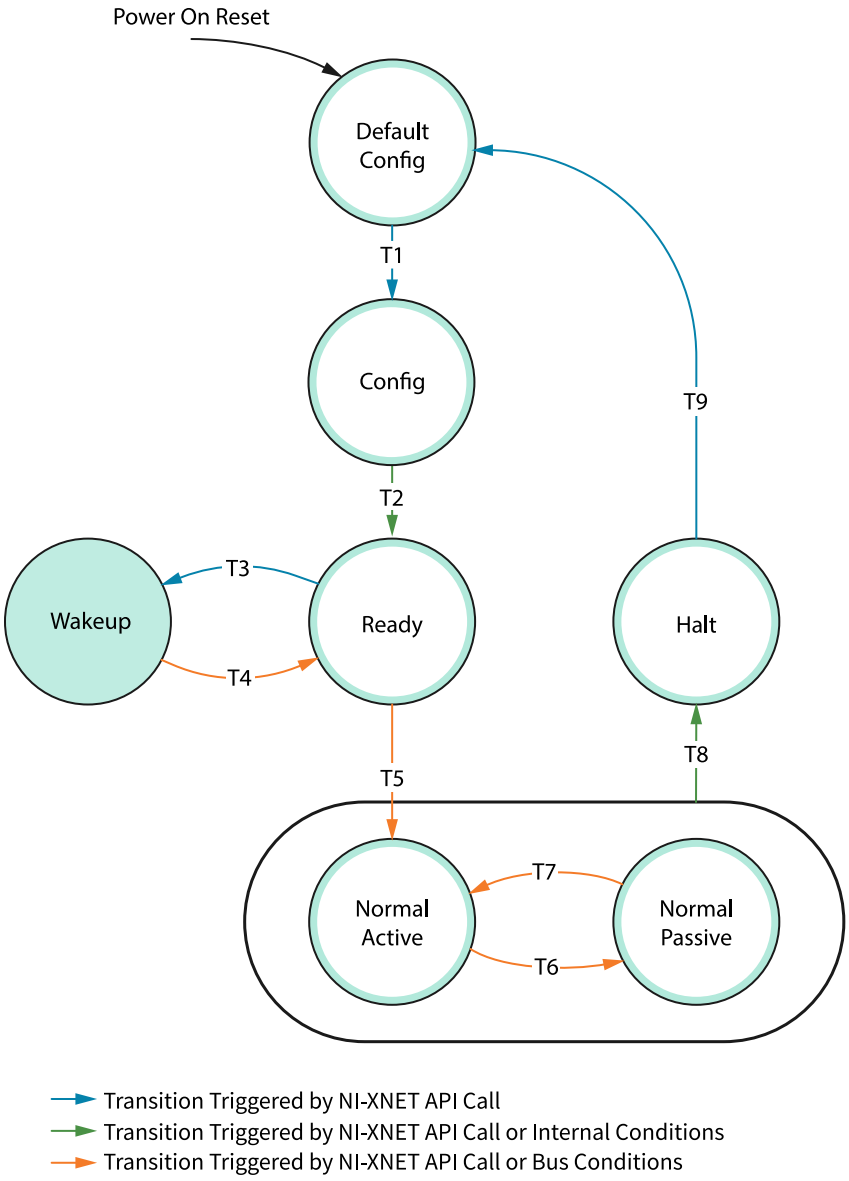
If your cluster supports the wakeup mechanism, the process becomes a bit more complex. The route the XNET hardware takes depends on whether the interface is currently awake or asleep. By default, XNET hardware starts in the awake state, and the startup process is exactly the same as if your cluster does not support wakeup. However, to use the wakeup mechanism your cluster is configured for, before calling `nxStart`, you need to put the interface to sleep. You can do this in one of two ways. First, you can set the Interface:FlexRay:Sleep property to `nxFlexRaySleep_LocalSleep`. This performs the one-time action of putting the interface to sleep. Alternately,you can set the Interface:FlexRay:Auto Asleep When Stopped? property to true. This puts the interface to sleep immediately. It also puts the interface to sleep automatically every time the interface is stopped, so the startup process is the same between your first start and subsequent starts.

If your interface is asleep when the `nxStart` API call is invoked, the interface progresses to **Ready**, where it waits for all connected channels to be awake before attempting to integrate with the cluster. After all connected channels are awake, the integration process occurs exactly like a cluster that does not support wakeup.

If you want your interface to wake up a sleeping network, you must configure your FlexRay interface to wake up the bus. You can do this in two ways. The first way is to set the Interface:FlexRay:Sleep property to `nxFlexRaySleep_RemoteWake` after you put your FlexRay interface to sleep. When you invoke the `nxStart` API call, the interface progresses though the **Ready** state and into the **Wakeup** state. In **Wakeup**, the interface generates the wakeup pattern on the FlexRay channel configured by the Interface:FlexRay:Wakeup Channel property and transitions back to **Ready**. If you have a multichannel bus, a separate node on the bus wakes up the other channel.

After all connected channels are awake, the integration process occurs exactly like a cluster that does not support wakeup. The second way is to invoke the `nxStart` API call to start the interface. The interface progresses to **Ready**, where it waits for all connected channels to be awake before attempting to integrate with the cluster.

During this time, if you set the Interface:FlexRay:Sleep property to `nxFlexRaySleep_RemoteWake`, the interface transitions into Wakeup, where it generates the wakeup pattern on the FlexRay channel configured by the Interface:FlexRay:Wakeup Channel property and transitions back to **Ready**. If you have a multichannel bus, a separate node on the bus wakes up the other channel. After all connected channels are awake, the integration process occurs exactly like a cluster that does not support wakeup.



T#	Condition	From	To
1	Start trigger received[1]	Default Config	Config[2]

T#	Condition	From	To
2	Startup process initiated	Config	Ready
3	Remote Wakeup initiated (Interface:FlexRay:Sleep property set to `nxFlexRaySleep_RemoteWake`)	Ready	Wakeup
4	Wakeup channel awake	Wakeup	Ready
5	All connected channels are awake and integration is successful [3].	Ready	Normal Active
6	Clock Correction Failed counter reached Maximum Without Clock Correction Passive. Value	Normal Active	Normal Passive
7	Number of valid correction terms reached the passive to active limit	Normal Passive	Normal Active
8	1. Clock Correction Failed counter reached Maximum Without Clock Correction Fatal Value. 2. Interface stopped (`nxStop`).		
9	Interface stopped (`nxStop`)	Halt	Default Config

[1] If you are not using synchronization, the `nxStart` API call internally generates the Start Trigger.

[2] In NI-XNET, this is a transitory state under normal situations. The Config state is nontransitory only if the startup procedure fails to continue.

[3] Any of the following conditions can satisfy all channels awake: the wakeup pattern was transmitted or received on all connected channels, a local wakeup is requested, or the interface is not asleep.

FlexRay Timing Type and Session Mode

For each XNET Frame FlexRay:Timing Type property value, this topic describes how the frame behaves for each XNET session mode.

An input session receives the FlexRay data frame from the network, and an output session transmits the FlexRay data frame. The FlexRay data frame data (payload) is mapped to/from signal values.

You use FlexRay null frames in the static segment to indicate that no new payload exists for the frame. In the dynamic segment, if no new payload exists for the frame, it simply does not transmit (no frame).

For NI-XNET input sessions, the Timing Type does not directly impact the representation of data from the appropriate `nxRead` function.

For NI-XNET output sessions, the Timing Type determines whether to transmit a data frame when no new payload data is available.

Cyclic Data

The data frame transmits in a cyclic (periodic) manner.

If the frame is in the static segment, the rate can be once per cycle (FlexRay:Cycle Repetition 1), once every N cycles (FlexRay:Cycle Repetition N), or multiple times per cycle (FlexRay:In Cycle Repetitions:Enabled?).

If the frame is in the dynamic segment, the rate is once per cycle.

If no new payload data is available when it is time to transmit, the payload data from the previous transmit is repeated.

Signal Input Single-Point, Signal Input Waveform, and Signal Input XY Modes

You specify the FlexRay signals when you create the session, and a specific FlexRay data frame contains each signal. When the FlexRay data frame is received, a subsequent call to the appropriate `nxRead` function returns its data. For information about how the data is represented for each mode, refer to Session Modes.

If a FlexRay null frame is received, it is ignored (no effect on the `nxRead` function). FlexRay null frames are not used to map signal values.

Frame Input Queued and Frame Input Single-Point Modes

You specify the FlexRay frame(s) when you create the session. When the FlexRay data frame is received, a subsequent call to the appropriate `nxRead` function returns its data. For information about how the data is represented for each mode, refer to Session Modes.

If a FlexRay null frame is received, it is ignored (not returned).

Frame Input Stream Mode

You specify the FlexRay cluster when you create the session, but not the specific FlexRay frames. When any FlexRay data frame is received, a subsequent call to the appropriate `nxRead` function returns it.

If the XNET Session Interface:FlexRay:Null Frames To Input Stream? property is true, and FlexRay null frames are received, a subsequent call to `nxRead` for the stream returns them. If Null Frames To Input Stream? is false (default), FlexRay null frames are ignored (not returned). You can determine whether each frame value is data or null by evaluating the `type` element (refer to the appropriate `nxRead` function).

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

You specify the FlexRay frame (or its signals) when you create the session. When you write data using the appropriate `nxWrite`function, the FlexRay data frame is transmitted onto the network. For information about how the data is represented for each mode, refer to Session Modes.

When the session and its associated interface are started, the FlexRay data frame transmits according to its rate. After that first transmit, the FlexRay data frame transmits according to its rate, regardless of whether the appropriate `nxWrite` function is called. If no new data is available for transmit, the next cycle transmits using the previous FlexRay data frame (repeats the payload).

If the frame is contained in the static segment, a FlexRay data frame transmits at all times. The FlexRay null frame is not transmitted. If you pass the FlexRay null frame to the appropriate `nxWrite` function, it is ignored.

If the frame is contained in the dynamic segment, a FlexRay data frame transmits every cycle. The dynamic frame minislot is always used.

Frame Output Stream Mode

This session mode is not supported for FlexRay.

Event Data

The data frame transmits in an event-driven manner. The event is the appropriate `nxWrite` function.

Because FlexRay is a time-driven protocol, the minimum interval between events is specified based on the FlexRay cycle. This minimum interval is configured in the same manner as a Cyclic frame.

If the frame is in the static segment, the interval can be once per cycle (FlexRay:Cycle Repetition 1), once every N cycles (FlexRay:Cycle Repetition N), or multiple times per cycle (FlexRay:In Cycle Repetitions:Enabled?).

If the frame is in the dynamic segment, the interval is once per cycle.

If no new event (payload data) is available when it is time to transmit, no frame transmits. In the static segment, this lack of new data is represented as a null frame.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, Frame Input Queued, and Frame Input Stream Modes

The behavior is the same as Cyclic Data.

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

The behavior is similar to Cyclic Data, except that the FlexRay data frame does not continue to transmit cyclically after the data from the appropriate `nxWrite` function

has transmitted. Because the database-specified timing for the frame is event based, after the FlexRay data frames for the appropriate `nxWrite` function have transmitted, the FlexRay data frame does not transmit again until a subsequent call to the appropriate `nxWrite` function.

If the frame is contained in the static segment, a FlexRay null frame transmits when no new data is available (no new call to the appropriate `nxWrite` function). If you pass the FlexRay null frame to the appropriate `nxWrite` function, it is ignored.

If the frame is contained in the dynamic segment, the frame does not transmit when no new data is available. The dynamic frame minislot is used only when new data is provided to the appropriate `nxWrite` function.

Frame Output Stream Mode

This session mode is not supported for FlexRay.

Related concepts:

- Session Modes
- Using FlexRay

Using LIN

This section summarizes some useful NI-XNET features specific to the LIN protocol.

Changing the LIN Schedule

LIN networks (clusters) always include 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, a single ECU in the network (slave) responds by transmitting the payload for the requested ID. The master ECU also can respond to a specific header, and thus the master can transmit payload data for the slave ECUs to receive.

Unlike some other scheduled protocols such as FlexRay, LIN allows the master ECU to change the schedule of frame headers. For example, the master can initially use a

"normal" schedule that requests IDs 1, 2, 3, 4, and then the master can change to a "diagnostic" schedule that requests IDs 60 and 61.

With NI-XNET, you change the LIN schedule using the XNET Write (State LIN Schedule Change) VI. When you want the NI-XNET interface to act as a master on the network, you must call this XNET Write VI at least once, to specify the schedule to run. When you write a schedule change, this automatically configures NI-XNET as master (the XNET Session Interface:LIN:Master? property is set to true). As a LIN master, NI-XNET handles all real-time scheduling of frame headers for you, using the LIN interface hardware onboard processor.

If you do not write a schedule change, NI-XNET leaves the interface at its default configuration of slave. As a LIN slave, you still can write signal or frame values to an output session, but NI-XNET waits for each frame's header to arrive before transmitting payload data.

Understanding LIN Frame Timing

Because LIN is a scheduled network, the headers that the master transmits determine the timing of all frames. To understand how and when each frame transmits, you must examine the entries in each schedule. Each entry transfers one frame (or possibly multiple frames). For more information, refer to the XNET LIN Schedule Entry Type property.

Because it is possible to use a single frame in multiple schedules and schedule entries, the overall timing for an individual frame can be complex. Nevertheless, each LIN schedule entry generally fits the concepts of cyclic and event timing that are common for other protocols such as CAN and FlexRay. For more information about how these concepts apply to LIN, refer to Cyclic and Event Timing.

LIN Diagnostics

Refer to the XNET Write (State LIN Diagnostic Schedule Change) VI for details.

Special Considerations for Using Stream Output Mode with LIN

Refer to the Interface:Output Stream Timing property for details.

Related concepts:

- Cyclic and Event Timing

LIN Frame Timing and Session Mode

This section describes the LIN behavior for each XNET session mode. As context for describing LIN frame transfer on the network, this section uses the timing concepts described in the LIN section of Cyclic and Event Timing.

An input session receives the LIN data frame (payload) from the network, and an output session transmits the LIN data frame. The LIN data frame payload is mapped to/from signal values.

For NI-XNET input sessions, the timing of each LIN schedule entry does not directly impact the representation of data from the XNET Read VI.

For NI-XNET output sessions, the timing of each LIN schedule entry determines whether to transmit a data frame when no new payload data is available.

You can configure the NI-XNET LIN interface to run as the LIN master by requesting a schedule (XNET Write (State LIN Schedule Change) VI). If the NI-XNET LIN interface runs as a LIN slave (default), a remote ECU on the network must execute schedules as LIN master for these modes to operate.

Cyclic

The LIN data frame transmits in a cyclic (periodic) manner.

This implies that the LIN master is running a continuous schedule, and the LIN data frame is contained within an unconditional schedule entry.

If no new payload data is available when it is time to transmit, the payload data from the previous transmit is repeated.

Signal Input Single-Point, Signal Input Waveform, and Signal Input XY Modes

You specify the signals when you create the session, and a specific LIN data frame

contains each signal. When the LIN data frame is received, a subsequent call to the XNET Read VI returns its signal data. For information about how the data is represented for each mode, refer to Session Modes.

Frame Input Queued and Frame Input Single-Point Modes

You specify the LIN frame(s) when you create the session. When the LIN data frame is received, a subsequent call to the XNET Read VI returns its data. For information about how the data is represented for each mode, refer to Session Modes.

Frame Input Stream Mode

You specify the LIN cluster when you create the session, but not the specific LIN frames. When any LIN data frame is received, a subsequent call to the XNET Read VI returns it.

Signal Output Single-Point, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

You specify the LIN frame (or its signals) when you create the session. When you write data using the XNET Write VI, the LIN data frame is transmitted onto the network. For information about how the data is represented for each mode, refer to Session Modes.

When the session and its associated interface are started, the LIN data frame transmits according to its schedule entry. Assuming that the LIN frame is contained in only one entry of the continuous schedule, the time between frame transmissions is the same as the time to execute the entire schedule (all entries). After that first transmit, the LIN data frame transmits according to its schedule entry, regardless of whether the XNET Write VI is called. If no new data is available for transmit, the next cycle transmits using the previous LIN data frame (repeats the payload).

Signal Output Waveform Mode

If the NI-XNET interface runs as a LIN master, NI-XNET executes schedules, and therefore controls the timing of LIN frames. When running as a LIN master, this session mode is supported, and NI-XNET resamples the waveform data such that it transmits at the scheduled frame rates.

If the NI-XNET interface runs as a LIN slave (default), this session mode is not supported. When running as a LIN slave, NI-XNET does not know which schedule the LIN master is executing. Because the LIN schedule is not known, the frame transfer rates also are not known, which makes it impossible to resample the waveform data.

Frame Output Stream Mode

This mode is available only when the LIN interface is master. You specify the LIN cluster when you create the session, but not the specific LIN frame.

The stream I/O modes do not use the database-specified timing for frames. Therefore, LIN data frames transmit only when you pass them to the XNET Write VI and do not transmit cyclically afterward.

When using a stream output timing of immediate mode, data is transmitted onto the network as soon as possible. Specifically, if the data array is empty, only the header part of the frame is transmitted (with the expectation that a slave transmits the response). If the data array is not empty, the header + response parts of the frame (the full frame) is transmitted. You can use this mode in conjunction with the scheduler, in which case each frame written to stream output is handled as a run-once schedule with lowest priority and having a single one-frame entry. A run-continuous schedule is interrupted to transmit the frame. A run-once schedule is not interrupted, and the frame is transmitted only when there are no pending run-once schedules with higher-than-lowest priority.

When using a stream output timing of either Replay Exclusive or Replay Inclusive, data is transmitted onto the network based on the timestamps in the frame.

Refer to the Interface:Output Stream Timing property for more details about using this mode with LIN.

Event

The LIN data frame transmits in an event-driven manner. The event is the XNET Write VI.

If no new event (payload data) is available when it is time to transmit, no frame transmits. This means that the LIN master transmits the frame header, but no payload

data follows this header.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, Frame Input Queued, and Frame Input Stream Modes

The behavior is the same as Cyclic.

Signal Output Single-Point, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

The behavior is similar to Cyclic, except that the LIN data frame does not continue to transmit after the data from the XNET Write VI has transmitted.

If the frame is contained in a sporadic schedule entry, and there are values for multiple frames pending for that entry, NI-XNET selects a single frame to transmit in each entry. NI-XNET selects the frame using the order in the XNET LIN Schedule Entry Frames property. For example, if the Frames property contains three frames, and you write data for the first and third, NI-XNET transmits the first frame (index 0) in the next occurrence of the sporadic entry, and then transmits the third frame (index 2) when that sporadic entry executes again.

If the frame is contained in an event-triggered schedule entry, a collision may occur if another ECU transmits in the same schedule entry. If the NI-XNET LIN interface runs as a LIN master, it automatically uses the XNET LIN Schedule Entry Collision Resolving Schedule property to resolve this collision.

Signal Output Waveform Mode

The behavior is the same as Cyclic.

If the NI-XNET interface runs as a LIN master, NI-XNET executes schedules, and therefore controls the timing of LIN frames. An event-driven LIN frame can transmit at most once per execution of its schedule entry.

If the NI-XNET interface runs as a LIN slave (default), this session mode is not supported.

Frame Output Stream Mode

When using a stream output timing of immediate mode, if the frame for transmit is defined as an event-triggered frame in the database, and a collision occurs during transmit, the interface automatically executes the collision resolving schedule defined for the frame, exactly as if the frame were transmitted in a scheduled event-triggered slot.

When using a stream output timing of either Replay Exclusive or Replay Inclusive, if the frame for transmit is determined to be defined as an event-triggered frame in the database, the frame is transmitted with a header ID equal to the unconditional frame ID contained in data byte 0. The data is transmitted without modification. In other words, the frame is transmitted as an unconditional frame associated with the event-triggered frame.

Refer to the Interface:Output Stream Timing property for more details about using this mode with LIN.

Using Ethernet

This topic summarizes some of the NI-XNET features specific to the Ethernet protocol.

Hardware Design

Each port has three data paths: XNET monitor, XNET endpoint, and OS stack. All three data paths can be used simultaneously.

Monitor Path

The monitor path reads and inspects all Ethernet frames that are received or transmitted on the port. For Ethernet interfaces, the suffix "/monitor" indicates the use of a monitor path when it is appended to the interface name (e.g., ENET1/monitor). When a port is configured for Direct mode, the monitor path reads frames that are received on the interface as well as frames that are echoed from transmit by the interface. When Tap is enabled, the monitor path reads the Ethernet frames that are received from one Tap partner before being transmitted by the other Tap partner.

Endpoint Path

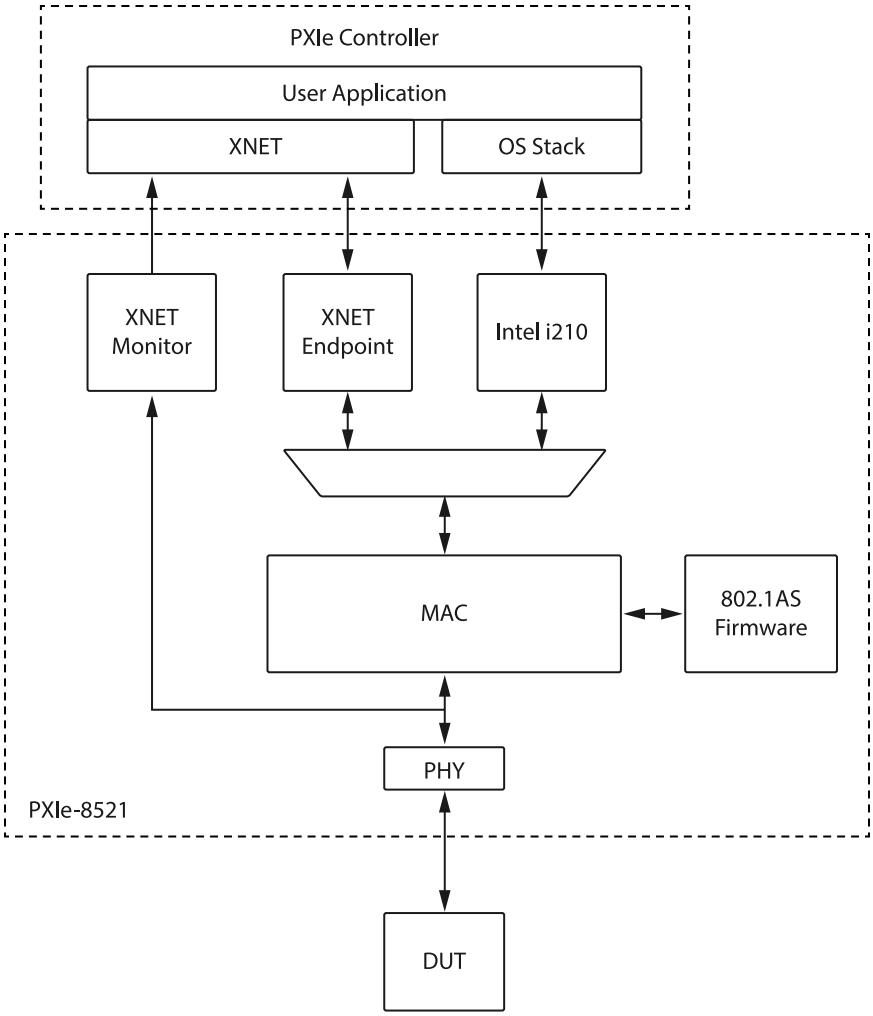
An Ethernet interface that is configured to use Direct mode functions as an endpoint. An endpoint path is the connection between an endpoint and the channel to which it is connected. The endpoint path transmits and receives Ethernet frames on the port. The endpoint path is typically used if you need it to act as an AVB endpoint. NI-XNET represents an endpoint path as the interface name itself, with no suffix.

OS Stack

The OS stack is the host computer, including operating system, application software, subroutines, and communication protocols. The OS stack path transmits and receives data using standard network sockets via the operating system's network stack. The OS stack is typically used with applications designed to use traditional TCP- or UDP-based protocols for its network communication.

The OS detects a separate network interface for each physical XNET port present on the system. The labeling of these ports is OS-specific, but you can determine which label matches a specific XNET interface by examining the properties reported in MAX, including the name, MAC address, and IP address. For information about configuring IP addressing, refer to your operating system instructions. For NI LabVIEW Real-Time systems, this configuration is exposed in the Network Settings of the target.

The following figure represents a block diagram of a single port, showing all three data paths.



For Ethernet interfaces, only XNET Create Session Frame Input Stream and Frame Output Stream are supported; other Create Session modes are not supported. All other modes, including Conversion Mode, return an invalid mode error, indicating that the selected session mode is not supported by the protocol of the interface.

Port Modes

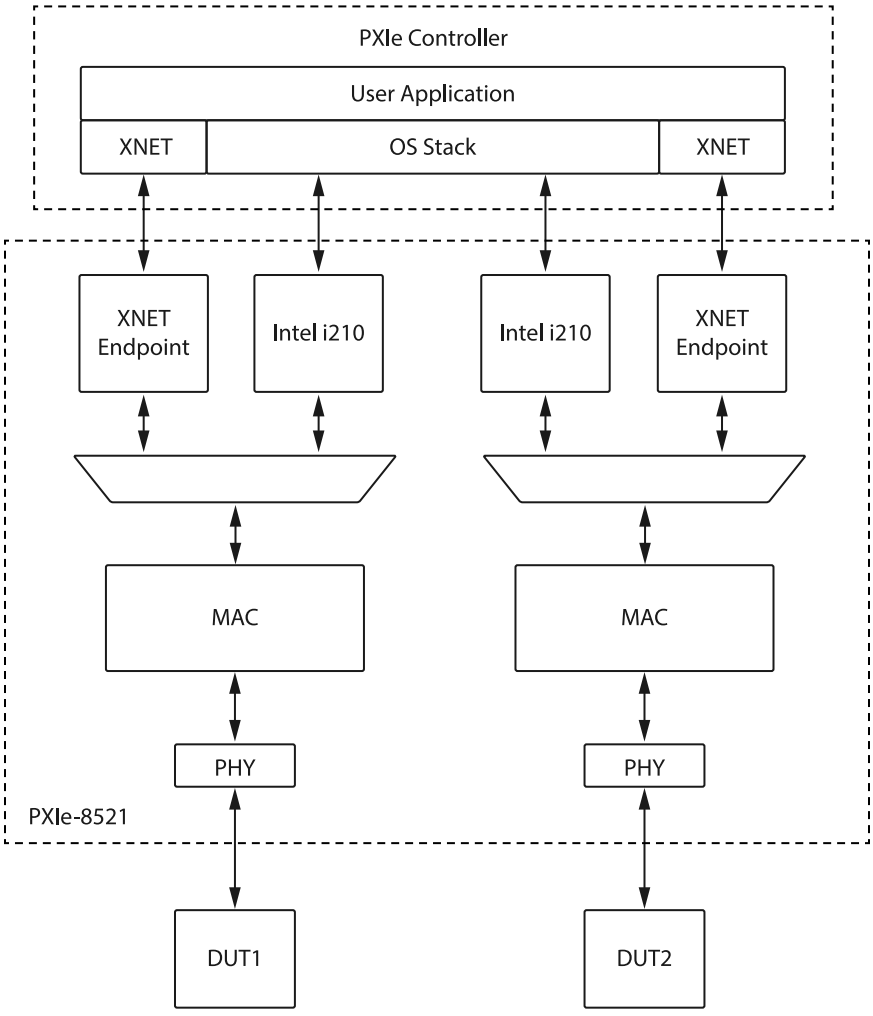
An Ethernet interface includes ports that can be configured as independent network interfaces. On Automotive Ethernet hardware, these ports can function in either Direct mode or Tap mode. Regardless of mode, traffic on each port can be monitored. When monitoring is enabled, all traffic that is transmitted or received on that port is captured.

Note Port mode cannot be changed while an XNET session is started on the

port. When the port mode is changed, port connectivity is lost to configure the change.

Direct Mode

In Direct mode, ports are directly connected and function as endpoints; Ethernet frames received and transmitted on the port have no relationship to any other ports on the device. Input and output sessions are supported in Direct mode. The following diagram shows a design with two ports. In this example, the ports are configured in Direct mode, and each port can run independently.

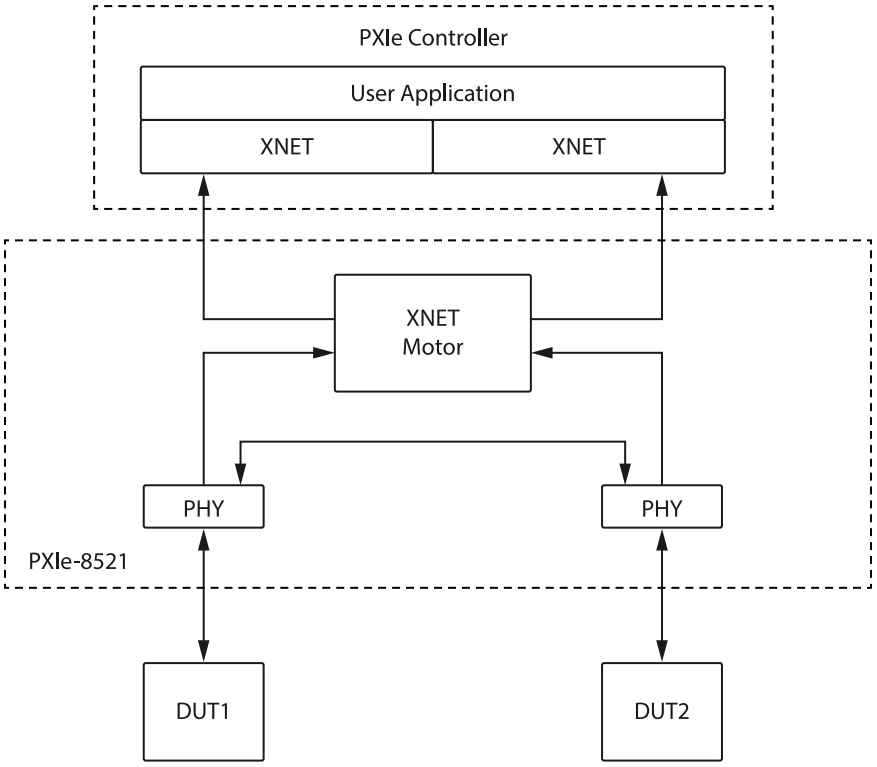


Tap Mode

In Tap mode, a pair of adjacent ports, called Tap partners, are connected to form a Tap that allows the interface to monitor traffic. For example, on a 4-port device, physical

ports 1 and 2 might be Tap partners, while ports 3 and 4 are Tap partners. A frame received on one Tap partner is immediately transmitted out the other Tap partner, to mimic behavior of an Ethernet cable. When you set Tap mode on one port, its Tap partner is automatically set to Tap mode as well.

The following diagram shows a design using a single Tap. As the connected ports are configured in Tap mode, traffic received on one Tap partner is transmitted to the other; monitored frames are also transmitted to the user application at the controller.



When an input session is created using an XNET interface for either Tap partner, and the monitor suffix is used with the XNET interface, the session reads frames going through the Tap partners. Output sessions are not supported in Tap mode.

When Port Mode is set to Tap for the interface, only the monitor names are shown; otherwise, both the monitor and endpoint names are shown.

Related concepts:

- Interfaces

Using LabVIEW Real-Time

The LabVIEW Real-Time (RT) module combines LabVIEW graphical programming with the power of a real-time operating system, enabling you to build real-time applications. NI-XNET provides features and performance specifically designed for LabVIEW RT.

High Priority Loops

Many real-time applications contain at least one loop that must execute at the highest priority. This high-priority loop typically contains code to read inputs, execute a control algorithm, and then write outputs. The high-priority loop executes at a fast period, such as 500 µs (2 kHz). To ensure that the loop diagram executes within the period, the average execution time (cost) of read and write VIs must be low. The execution time also must be consistent from one loop iteration to another (low jitter).

Within NI-XNET, the session modes for single-point I/O are designed for use within high-priority loops. This applies to all four single-point modes: input, output, signal, or frame. The XNET Read and XNET Write VIs provide fast and consistent execution time, and they avoid access to shared resources such as the memory manager.

The session modes other than single-point all use queues to store data. Although you can use the queued session modes within a high priority loop, those modes use a variable amount of data for each read/write. This requires a variable amount of time to process the data, which can introduce jitter to the loop. When using the queued modes, measure the performance of your code within the loop to ensure that it meets your requirements even when bus traffic is variable.

When the XNET Read and XNET Write VIs execute for the very first loop iteration, they often perform tasks such as auto-start of the session, allocation of internal memory, and so on. These tasks result in high cost for the first iteration compared to any subsequent iteration. When you measure performance of the XNET Read and XNET Write VIs, discard the first iteration from the measurement.

For another VI or property node (not the XNET Read or XNET Write VI), you must assume it is not designed for use within high priority loops. The property nodes are designed for configuration purposes. VIs that change state (for example, the XNET Start

VI) require time for hardware/software configuration. Nevertheless, there are exceptions for which certain properties and VIs support high-priority use. Refer to the help for the specific features you want to use within a high priority loop. This help may specify an exception.

XNET I/O Names

You can use a LabVIEW project to program RT targets. When you open a VI front panel on an RT target, that front panel accesses the target remotely (over TCP/IP).

When you use an XNET I/O name on a VI front panel on LabVIEW RT, the remote access provides the user interface features of that I/O name. For example, the drop-down list of an XNET Interface provides all CAN, FlexRay, and LIN interfaces on the RT target (for example, a PXI chassis).

For the remote access to operate properly, you must connect the LabVIEW RT target using a LabVIEW project. To connect the target, right-click the target in a LabVIEW project and select **Connect**. The target shows a green LED in project, and the user interface of I/O names is operational.

If the RT target is disconnected in a LabVIEW project, each I/O name displays the text (target disconnected) in its drop-down list.

Deploying Databases

When you create an NI-XNET application for LabVIEW RT, you must assign an alias to your database file. When you deploy to the RT target, the text database file is compressed to an optimized binary format, and that binary file is transferred to the target.

When you create NI-XNET sessions using a LabVIEW project, you assign the alias within the session dialog (for example, Browse for Database File). When you drag the session to a VI under the RT target, then run that VI, NI-XNET automatically deploys the database file to the target.

When you create NI-XNET sessions at run time, you must explicitly deploy the database to the RT target. There are two options for this deployment:

- **I/O names** : If you are using I/O names for database objects, you can click on an I/O name and select **Manage Database Deployment**. This opens a dialog you can use to assign new aliases and deploy them to the RT target.
- **File Management VIs** : To manage database deployment from a VI running on the host (Windows computer), use VIs in the NI-XNET **File Management** palette. This palette includes VIs to add an alias and deploy the database to the RT target.

To delete the database file from the RT target after execution of a test, you perform this undeploy using either option described above.

Memory Use for Databases

When you access properties of a database object (for example, cluster, frame, signal) on the diagram of your VI, NI-XNET opens the database on disk and maintains a binary image in memory. Use XNET Database Close.vi to close the database prior to performing memory-sensitive tasks, such as a control loop on LabVIEW Real-Time.

When you pass database objects as input to XNET Create Session.vi, NI-XNET internally opens the database, reads the information required to create the session, then closes the database. Therefore, there is no need to explicitly close the database after creating sessions.

FlexRay Timing Source

FlexRay is a deterministic protocol, which means it enables ECUs to synchronize code execution and data exchange. When you use LabVIEW to test an ECU that uses these deterministic features, you typically need to synchronize the LabVIEW VI to the FlexRay communication cycle. For example, to validate that the ECU transmits a different value each FlexRay cycle, you must read that frame every FlexRay cycle.

NI-XNET provides the XNET Create Timing Source (FlexRay Cycle) VI to create a LabVIEW timing source. You wire this timing source to a LabVIEW timed loop to execute LabVIEW code synchronized to the FlexRay cycle. Because the length of time for each FlexRay cycle is a few milliseconds, LabVIEW RT provides the required real-time execution.

Creating a Built Real-Time Application

NI-XNET supports creation of a real-time application, which you can set to run automatically when you power on the RT target. Create the real-time application by right-clicking **Build Specifications** under the RT target, then selecting **New»Real-Time Application**.

If you created NI-XNET sessions in a LabVIEW project, those sessions are deployed to the RT target in the same manner as running a VI.

Deployment of databases for a real-time application is the same as running a VI.

Related concepts:

- Creating a Built Application
- Using FlexRay

System Configuration API

NI-XNET supports the NI System Configuration API, which provides programmatic access to many operations in NI MAX. This enables you to perform these operations within your application. The NI System Configuration API uses product experts to gather information about devices on local and remote systems. You can create a filter to gather information for a single type of product, such as filtering for NI-XNET devices only. The NI-XNET expert programmatic name is `xnet`.

Although XNET System API property nodes (XNET System Node, XNET Device Node, and XNET Interface Node) are provided for compatibility with previous versions of NI-XNET, the NI System Configuration API is recommended for its several advantages:

- Discovery and configuration of all NI hardware products, not just XNET hardware
- Consistency with features available in NI MAX
- Ability to save configuration changes for use in multiple LabVIEW applications
- Remote discovery and configuration for LabVIEW Real-Time (RT) targets

When you physically connect an NI-XNET interface to your network, some properties must be configured to enable communication. Some of these properties, such as an

Ethernet PHY state or CAN termination, are specific to your interface but are not necessarily maintained in or provided by a database. These properties can be written using the NI System Configuration API rather than an NI-XNET session.

When you write a property using the NI System Configuration API, you must invoke the Save Changes VI in order for the change to take effect. (See Save Changes (VI) in the ***NI System Configuration API Help***.)

The NI-XNET expert returns a flat list of "hardware resources" to System Configuration API for each NI-XNET device (i.e., hardware model) and NI-XNET interface (e.g., port) that is connected to the system. Use the following properties in System Configuration API to convert the resource list into the typical hierarchy displayed in NI MAX.

- `Device`: IsDevice=T, Devices&Chassis:ProvidesLinkName=<unique-device-name>
- `Interface`: IsDevice=F, Device&Chassis:ConnectsToLinkName=<unique-device-name>, ExpertInfo:UserAlias=<XNET Interface>

For example, a single, 2-port USB-8506 on Windows returns three hardware resources:

- `Device`: ProvidesLinkName="NI USB-8506(SerialNumber01BE2C4C)"
- `Interface (Port 1)`: ConnectsToLinkName="NI USB-8506(SerialNumber01BE2C4C)" and UserAlias="LIN1"
- `Interface (Port 2)`: ConnectsToLinkName="NI USB-8506(SerialNumber01BE2C4C)" and UserAlias="LIN2"

In the NI System Configuration API, for an interface (IsDevice=F), the NI-XNET expert returns the UserAlias using the value of the XNET interface name. Therefore, you can use the UserAlias for the interface input to XNET Create Session.

Note Unlike the XNET Interface I/O control, the System Configuration API does not provide the "/monitor" suffix for the interface name it returns; you will need to concatenate the interface name and suffix for the monitor path.

The System Configuration API includes the following hardware properties under the category "XNET":

- Device > Number of Ports

- Interface > Blink
- Interface > Port Number
- Interface > Protocol
- Interface > CAN > Transceiver Capability
- Interface > CAN > Termination Capability
- Interface > Dongle > Dongle ID
- Interface > Dongle > Dongle State
- Interface > Ethernet > PHY State Configured
- Interface > Ethernet > Port Mode
- Interface > Ethernet > Link Speed
- Interface > Ethernet > Link Speed Configured
- Interface > Ethernet > Interrupt Moderation
- Interface > Ethernet > Jumbo Frames
- Interface > Ethernet > Sleep Capability Configured
- Interface > Ethernet > Sleep Capability
- Interface > Ethernet > PHY Power Mode
- Interface > Ethernet > MAC Address
- Interface > Ethernet > IP4Address
- Interface > Ethernet > OS Network Adapter Name
- Interface > Ethernet > OS Network Adapter Description

Automotive Ethernet Socket API

The XNET Automotive Ethernet Socket API enables you to create BSD-like network sockets for TCP and UDP communication using the TCP and UDP Socket VIs in the IP Stack subpalette. This implementation is independent of the limitations of the IP stack native to your operating system.

XNET IP Stack

An XNET IP stack is an implementation of the TCP/IP protocol suite. The IP stack provides tools to create everything required for TCP and UDP communication, independent from the limitations of the IP stack native to your operating system (OS). A test application typically uses a single XNET IP Stack for each XNET Interface (physical port), but more complex configurations are possible.

For example, suppose that you are testing eight identical instances of an ECU, each instance connected to a distinct XNET Interface (e.g., two 4-port Automotive Ethernet Interface Modules). For each of the eight repeated test setups, you could use the same static IP address for each XNET Interface, and communicate with the same static IP address in the ECU. This configuration is difficult to achieve using the native Windows or Linux IP stack, because the OS assumes that each interface uses a different unicast IP address.

As another example, to fully test a physical ECU, suppose you need to simulate six real ECUs that are part of a single in-vehicle network. (This is sometimes called "restbus simulation.") The XNET IP stack enables you to configure six distinct virtual interfaces in the IP stack to represent multiple simulated ECUs. These virtual interfaces can all run on the IP stack associated with a single XNET Interface (physical port) that is connected to your real ECU under test.

After you configure the IP stacks as needed for your test, you can use the Automotive Ethernet Socket API for TCP and/or UDP communication. The Socket API is analogous to LabVIEW's built-in TCP/UDP palettes for the OS stack, which you can find on the Functions Palette under **Data Communication** » **Protocols**. The alignment of these socket APIs is intended to reduce the learning curve and to facilitate re-use of code between stacks.

For a given XNET Interface, TCP and UDP traffic switch from the OS stack to XNET IP Stack when you call XNET IP Stack Create.vi the first time for that XNET Interface. Communication changes back to the OS stack when you call XNET IP Stack Clear.vi the last time for that XNET Interface. When you are viewing traffic on the XNET Interface (e.g., Wireshark on ENET2), you might notice that some protocols run in the OS stack (e.g., Windows running DHCPv6), but those protocols cease after XNET IP Stack Create.vi.

Supported Features

Beginning with NI-XNET 20.5, the Automotive Ethernet Socket API supports IPv4 and IPv6 addresses. The XNET IP Stack supports the following protocols:

- Transmission Control Protocol (TCP)
- Universal Datagram Protocol (UDP)

- Address Resolution Protocol (ARP)
- Internet Control Message Protocol, v4 (ICMPv4)
- Internet Control Message Protocol, v6 (ICMPv6)
- Internet Group Management Protocol (IGMP)

Each XNET IP stack that you create supports one NI-XNET interface. The NI-XNET interface can be used simultaneously with one or more XNET IP Stacks and with the XNET Session palette in LabVIEW. Note that more than one stack can use the same XNET interface.

The NI-XNET interface contains one or more MACs (simulated hardware ports), each with a distinct MAC address. For a given XNET interface, each MAC address must be unique across all stacks.

Each MAC contains one or more virtual interfaces (VLANs), each with a distinct VLAN ID. The VLAN ID is either untagged or a 12-bit tagged ID. Each MAC supports jumbo frames if the XNET interface operates at gigabit speed (e.g., 1000BASE-T1).

Each virtual interface contains one IPv4 address (unicast) and one IPv4 gateway address. All virtual interfaces can use a static IPv4 address. Within each stack, one virtual interface can use link-local addressing (also known as Auto IP).

Creating a Built Application

NI-XNET supports creation of a built application using a LabVIEW project.

For a LabVIEW Real-Time (RT) target, the built application typically is used as a startup application. For information about creating a built application for LabVIEW RT, refer to Using LabVIEW Real-Time.

For a Windows target (My Computer), the built application is an executable (`.exe`). You typically distribute the executable to multiple end users, which means you copy to multiple computers (targets).

This topic describes creating a built application for Windows that uses NI-XNET.

Create the executable by right-clicking **Build Specifications** under **My Computer**, then

select **New»Application (EXE)**.

Sessions

If you created NI-XNET sessions under **My Computer**, the configuration for those sessions is generated to the following text file:

`nixnetSession.txt`

This text file is in the same folder as the executable (`.exe`).

You must include this text file as part of your distribution. Copy this text file along with the `.exe`, always to the same folder.

If you create sessions at run time using the XNET Create Session VI, those sessions are standalone (no text file required).

Databases

If your application uses the in-memory database (:memory:), that database is standalone (no file or alias required). For more information about the in-memory database, refer to the Create In Memory section of Database Programming.

If your application accesses a database file using a filepath (not alias), you must ensure that the file exists at the same filepath on every computer. Because LabVIEW uses absolute filepaths (for example, `c:\MyDatabases\Database5.dbc`), this implies that every computer that runs the executable must use the same file system layout.

If your application accesses a database file using an alias, you must add the alias using the XNET Database Add Alias VI. You can use this VI as part of an installation process or call it within the executable itself. Using an alias provides more flexibility than a filepath. For example, your application can check for the required file at a likely filepath and add the alias if found, or otherwise pop up a dialog for the end user to browse to the correct filepath (then add an alias).

Related concepts:

- Using LabVIEW Real-Time
- Database Programming for the C API

Error Handling

In NI-XNET, the term error refers to a problem that occurs within the execution of a node in the block diagram (VI or property node). The term fault refers to a problem that occurs asynchronously to execution of an NI-XNET node. For example, an invalid parameter to an XNET Read VI is an error, but a communication problem on the network is a fault. For more information about faults, refer to Fault Handling.

LabVIEW uses error clusters to pass error information through each VI.

NI-XNET uses the **error in** and **error out** clusters in each VI and property node. The elements of these clusters are:

	**status** is true if error occurred or false if success or warning occurred.
	**code** is a number that identifies the error or warning. A value of 0 means success. A negative value means error: The VI did not perform the intended operation. A positive value means warning: The VI performed the intended operation, but something occurred that may require your attention. For a description of the code, right-click the error cluster and select **Explain Error** or **Explain Warning**. You also can wire the error cluster to the LabVIEW Simple Error Handler VI to obtain the description at runtime.
	**source** identifies the VI where the error or warning occurred.

For most NI-XNET VIs, if **error in** indicates an error, the VI passes the error information to **error out** and does not perform the intended operation. In other words, NI-XNET VIs do not execute under error conditions. The exceptions to this behavior are the XNET Clear VI and XNET Database Close VI. These VIs always perform the intended operation of closing or otherwise cleaning up, even when **error in** indicates an error.

If **error in** indicates success or warning, the NI-XNET VI executes and returns the result of its operation to **error out**.

The **error in** cluster is an optional input to a VI, with a default value of no error (**status** false and **code** 0).

Related concepts:

- Fault Handling

Fault Handling

In NI-XNET, the term error refers to a problem that occurs within the execution of a node in the block diagram (VI or property node). The term fault refers to a problem that occurs asynchronously to execution of an NI-XNET node. For example, passing an invalid session to a VI is an error, but a communication problem on the network is a fault. For more information about errors, refer to Error Handling.

Examples of faults include:

- The CAN, FlexRay, and LIN protocol standards each specify mechanisms to detect communication problems on the network. These problems are reflected in the communication state and other information.
- If you pass invalid data to the XNET Write VI , in some cases the problem cannot be detected until the data is about to be transmitted. Because the transmission occurs after the XNET Write VI returns, this is reported as a fault.

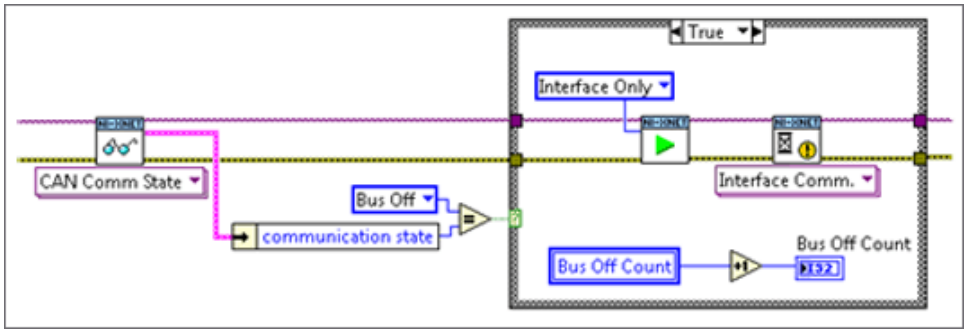
NI-XNET reports faults from a special XNET Read VI instance for the communication state. For CAN, this is the XNET Read (State CAN Comm) VI , for FlexRay this is the XNET Read (State FlexRay Comm) VI , and for LIN this is the XNET Read (State LIN Comm) VI.

The information returned from these VIs is not limited to faults. Each VI provides information about the current state of communication on the network. Because the XNET Read VI executes quickly, it often is useful within the primary loop of your application to ascertain the current network state.

Each XNET Read VI returns a cluster with various indicators. Most of the indicators provide state information that the protocol specifies, including faults (communication stopped). Faults specific to NI-XNET are reported in **fault?** and **fault code**. **fault?** is similar to the **status** of a LabVIEW error cluster, and **fault code** is similar to the **code** of a LabVIEW error cluster.

To detect a fault without stopping the execution of your VIs, you read and interpret the

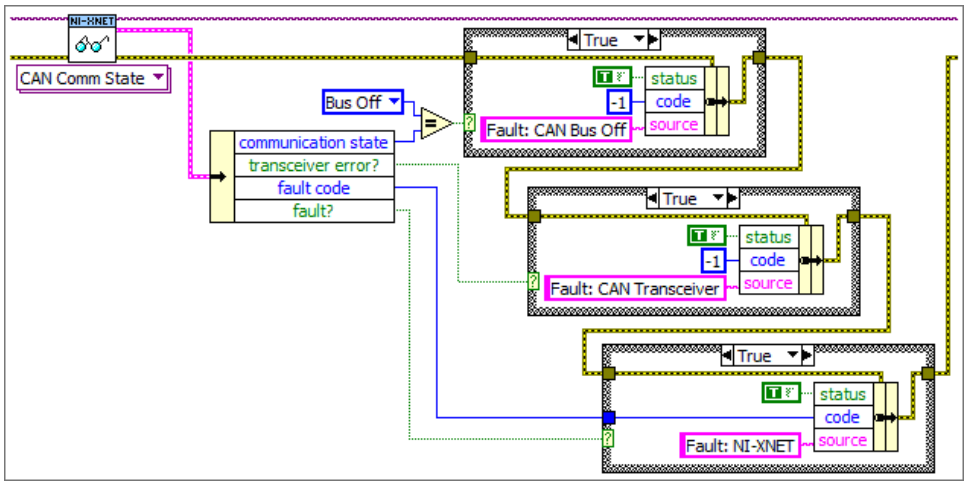
communication state separately from the LabVIEW error cluster flow. For example, you may want to intentionally introduce noise into CAN cables to test how your ECU behaves when the CAN bus off state occurs. The following figure shows an example block diagram for this method.



Restart on CAN Bus Off State

The block diagram detects the CAN bus off state, which means that communication stopped due to numerous problems on the bus. When CAN bus off state is detected, the block diagram increments a count and restarts the NI-XNET interface. It then waits for the interface to be reintegrated with the bus before continuing. This process of reintegrating into a CAN bus after going bus off is known as bus off recovery. Because the CAN bus off fault was not propagated as an error, the test continues to execute.

To detect a fault and propagate it to an error to break the LabVIEW flow, use a diagram similar to the following example.



Propagating CAN Faults to an Error

The block diagram in the figure above first checks for CAN bus off state, which indicates that communication stopped due to a serious problem in the CAN protocol state machine (data link layer). Next, the block diagram checks for a fault in the CAN transceiver (physical layer). Finally, the block diagram checks for a fault in NI-XNET. If any of these three faults are detected, it overwrites the previous error in the standard LabVIEW error cluster. If a fault or error occurs, execution of subsequent VIs ceases, effectively stopping the LabVIEW application execution.

Related concepts:

- Error Handling

Handling Timestamps

When the hardware is in a replay mode, the first frame received from the application is considered the start time, and all subsequent frames are transmitted at the appropriate delta from the start time. For example, if the first frame has a timestamp of `12:01.123` , and the second frame has a timestamp of `12:01.456`, the second frame is transmitted 333 ms after the first frame.

If a frame's time is identical or goes backwards relative to the first timestamp, this is treated as a new start time, and the frame is transmitted immediately on the bus. Subsequent frames are compared to this new start time to determine the transmission time. For example, assume that the application sends the hardware four frames with the following timestamps: `12:01.123`, `12:01.456`, `12:01.100`, and `12:02.100`. In this scenario, the first frame transmits immediately, the second frame transmits 333 ms after the first, the third transmits immediately after the second, and the fourth transmits one second after the third. Using this behavior, you can replay a logfile of frames repeatedly, and each new replay of the file begins with new timing.

A frame with a timestamp that goes backward relative to the previous timestamp, but is still forward relative to the start time, is transmitted immediately. For example, assume that the application sends the hardware four frames with the following timestamps: `12:01.123` , `12:01.456`, `12:01.400`, and `12:02.100` . In this scenario, the first frame transmits immediately, the second frame transmits 333 ms

after the first, the third transmits immediately after the second, and the fourth transmits 544 ms after the third.

The first frame is always transmitted immediately, even if the absolute time of the first frame is in the future relative to the current time (this is the current CAN/LIN behavior). If you want to synchronize the start of multiple replay streams, you must use a Future Time Trigger frame. See Special Frames and Synchronized Replay for more information

Related concepts:

- Special Frames
- Synchronized Replay

TDMS

This topic describes how NI-XNET frame data is stored within National Instruments Technical Data Management Streaming (`.TDMS`) files. The National Instruments TDMS file format provides efficient and flexible storage on NI platforms. The TDMS file format enables storage of a wide variety of measurement types in a single binary file, including CAN, FlexRay, LIN, analog, digital, and so on.

This topic specifies the method used to store NI-XNET raw frame data within TDMS. Although you also can store NI-XNET signal waveforms within TDMS, raw frame data is the most efficient and complete way to store NI-XNET data. Raw frame data can be easily converted to/from protocol-specific frames or signal waveforms for display and analysis.

TDMS is recommended for new applications that access NI-XNET data within files. For examples that demonstrate use of TDMS with NI-XNET, refer to the **NI-XNET Logging and Replay** category in the NI Example Finder (for example, **Hardware Input and Output : CAN : NI-XNET : Logging and Replay**).

Previous versions of NI-XNET and NI-CAN used a file format called NCL to store raw frame data. If you have an existing application that uses NCL, you can continue to use that file format. Examples for NCL continue to be installed with NI-XNET (`examples\nixnet` folder in your LabVIEW directory), but they no longer appear in the NI Example Finder. If you need to store multiple sources of data in a single file (for

example, multiple CAN interfaces, or CAN with analog input), you should consider transitioning your application from NCL to TDMS. Because both file formats use the same raw frame data, the changes required for this transition are relatively small.

Within the TDMS file, a sequence of raw frames is stored in a distinct TDMS channel for each NI-XNET interface (for example, CAN port). From the TDMS perspective, the frame data is an array of U8 values. The U8 array represents one or more raw frames.

The version of TDMS used with this specification must be 2.0 or higher.

Channel Name and Group Name

The name of the TDMS channel can use any conventions that you desire, but it should be sufficient to identify the network that is stored. For example, if you log data from two CAN interfaces, you might name the first TDMS channel `Powertrain network` and the second TDMS channel `Body network` . If you have an NI-XNET database that contains distinct clusters for each network, the Name (Short) property often provides a useful description of the network, and can be used directly as the TDMS channel name.

The name of the TDMS group can use any conventions that you desire. The group name is required for NI-XNET frame data, but if you do not use multiple groups in the TDMS file, you can select a simple group name (for example, `My Group`).

Channel Data

The data you read and write to the TDMS channel must be an array of U8 values. No other TDMS data types are supported.

The channel data contains one or more frames encoded using the Raw Frame Format. The raw frame format encodes all information received on the network, along with precise timestamps. The protocols supported include CAN, FlexRay, and LIN.

The TDMS Channel Properties specify additional requirements for encoding of the raw frame data. The property `NI_network_frame_byte_order` is particularly important, as this specifies the byte order used for the Timestamp and Identifier elements within each raw frame.

Channel Properties

Special properties are used on each TDMS channel to distinguish the data from a plain array of U8 samples. Properties are also provided to assist in interpreting the data, such as conversion to signals (physical units).

All properties for NI-XNET frame data use the prefix `NI_network_` . This prefix ensures that the properties do not conflict with names used by your application. The following table lists the channel properties.

Table 18. Channel Properties

Name	Data Type	Permissions	Description
NI_network_protocol	I32	Required	Specifies the network protocol used for all frames in this channel. The property value is an enumeration: 0 — CAN 1 — FlexRay 2 — LIN If this property does not exist, the data shall not be interpreted as raw frames, but as plain U8 samples.
NI_network_frame_version	I32	Required	Specifies the raw frame encoding version. The encoding of this number is specific to each protocol listed in `NI_network_protocol`. For CAN, FlexRay, and LIN, the version encoding is the Upgrade Version in lowest order byte, and Major Version in next order byte. The two upper order bytes are 0.

Name	Data Type	Permissions	Description
			The Major Version indicates a change that breaks compatibility with the previous version. The value for this specification is 2. The Upgrade Version indicates a change that retains compatibility with Upgrade Version 0. The value for this specification is 0. If this property does not exist, the data is not interpreted as raw frames, but as plain U8 samples.
NI_network_frame_byte_order		Required	Specifies the byte order for multibyte elements within each frame's data. For example, the frame's Identifier is a 32-bit value, and Timestamp is a 64-bit value. Refer to the Raw Frame Format for details. This property does not specify byte order for TDMS properties or other TDMS channels. This property does not specify byte order for signals within the frame's Payload (that is, covered by specifications such as CANdb, LDF, AUTOSAR, and FIBEX). The property value is an enumeration: 0 — Little-endian (that is, least significant byte in lowest offset, Intel byte order). 1 — Big-endian (that is, most

<table>
<tr><th>Name</th><th>Data Type</th><th>Permissions</th><th>Description</th></tr>
<tr><td></td><td></td><td></td><td>
<table>
<tr><td></td><td>significant byte in lowest offset, Motorola byte order).</td></tr>
<tr><td>2</td><td>If this property does not exist, the data is not interpreted as raw frames, but as plain U8 samples.</td></tr>
</table>
</td></tr>
<tr><td>NI_network_content</td><td>  </td><td>Optional</td><td>Provides information that describes the content of the payload of frames on this network. This typically is information to map and scale physical-unit values from each frame's payload. The encoding of this string is specific to each protocol listed in <code>NI_network_protocol</code>.

For CAN, FlexRay, and LIN, the string encoding is:

<code><alias>.<cluster></code>

The <alias> specifies an alias to a network database file (content specification). This alias provides a short name, used to refer to a database file across multiple systems. When you register an alias with tools, you typically use the database filename on the local system, without the preceding path or file extension. For example, the path <code>c:\MyDatabases\CANdb\</code></td></tr>
</table>

Name	Data Type	Permissions	Description
			`Powertrain.dbc` would use an alias of `Powertrain`. The <cluster> refers to a specific cluster (network) within the database. A database file can specify multiple networks within a vehicle. This portion of the string is optional (you can use <alias> without "." or <cluster>). If the cluster does not exist, it is assumed that only one network is specified within the database. When you use NI-XNET, this string uses the same syntax as the XNET Cluster I/O Name. The registered alias refers to a file on Windows (DBC, LDF, AUTOSAR, or FIBEX text file), or on LabVIEW Real-Time (compressed binary file). When you use tools that do not explicitly contain NI-XNET (for example, NI DIAdem), support for this property may have limitations. For example, DBC files may be supported, but not LDF or FIBEX. This property is optional. For applications that read the log file, if this property does not exist, the effect will depend on the goal: Display of frame values: no effect—the network content is not needed. Display of signal values: application opens a dialog to ask the customer to browse to the file.

Related concepts:

- Raw Frame Format

Timescales

NI-XNET uses time for a variety of features, including timestamping of received frames, timestamping of trigger signals, waveform sampling, and timestamped transmission. Timescale refers to the concept of a clock that measures the progression of time. NI-XNET uses three distinct timescales:

- **Local time** is the clock on the XNET hardware product, which in some cases is used to synchronize with other National Instruments products.
- **Network time** is, for XNET Ethernet products, the time on the network of your ECUs, such as when IEEE Std 802.1AS is used to synchronize time among ECUs.
- **Host time** is the clock of the operating system where LabVIEW is running (e.g., Windows or Linux).

Note When Intf.Enet.OutStrmTimescale or nxPropSession_IntfEnetOutStrmTimescale is used with XNET Write, the `Interface:Ethernet:Output Stream Timescale` property determines which timestamp field is used with the write.

Local Time

An XNET PXI product, by default, uses the PXI backplane clock (PXI_Clk10, PXI_Clk100), for synchronization with other products in the PXI chassis. If the PXI backplane clock is not available (e.g., turned off), the product uses its local oscillator.

An XNET PCI or USB product, by default, uses its local oscillator, and trigger signals can be used to achieve synchronization.

An XNET C Series module, by default, uses the time provided by the C Series chassis. If time is not available from the C Series chassis, the XNET C Series module uses its local oscillator.

For some XNET products, the default source of local time can be changed with XNET Connect Terminals and/or terminal properties. With XNET Connect Terminals, use the destination terminal of the Master Timebase to change the local clock.

Most clocks that are used for local time provide frequency (with an oscillator), but not date/time information. When a session is created, XNET initializes the date/time information for the local clock using host time.

In DAQmx terminology, XNET local time is analogous to DAQmx I/O device time.

Network Time

Many in-vehicle Ethernet networks use a protocol such as IEEE Std 802.1AS to synchronize time among ECUs. XNET Ethernet products participate in the time synchronization protocol in the ECU network. This network time is used to timestamp received Ethernet frames (in addition to the timestamp from the local time).

When an XNET Ethernet port acts as the grandmaster in the ECU network (i.e., Port State is Master), local time is used for the grandmaster clock, and date/time information in the ECU network is initialized from host time.

When an XNET Ethernet port acts as a slave in the ECU network, local time and network time will eventually drift relative to one another. The date/time information for network time is obtained from the ECU that acts as the grandmaster.

Host Time

Most computers and controllers maintain date/time information for the timescale provided by the operating system. This host time can obtain the date/time using a Real Time Clock (RTC), or a Network Time Protocol (NTP) server. Many implementations of host time are traceable to a global timescale, such as Coordinated Universal Time (UTC) and International Atomic Time (TAI).

Although host time provides accurate date/time information, the accuracy and resolution of its clock can often be in tens of milliseconds. In contrast, the XNET hardware for local time and network time provides resolution in nanoseconds. Although local time and network time use host time to initialize their date/time information, local and network times do not use the same physical clock as host time.

Therefore, both local time and network time will eventually drift relative to host time.

Many National Instruments products initialize their date/time information from host time as described above. This initialization occurs at the moment that the hardware is initialized. Because each hardware product initializes at a different moment, the date/time information for each local clock might not be identical for a given point in time. For example, if you connect a shared start trigger to two DAQmx PXI cards and two XNET PXI cards, each of the four cards might report a slightly different timestamp for the pulse of that start trigger (e.g., t0 in a waveform). The cards are tightly synchronized in reality, but the differing timestamps give the appearance of inaccuracy. This issue can be corrected using techniques such as the LabVIEW Align Waveform Timestamps VI and NI-XNET Adjust Local Time property.

Related concepts:

- Synchronized Replay

Getting Started with NI-XNET C API

This section helps you get started using NI-XNET for C. It includes basic information about using LabWindows/CVI and Microsoft Visual C, and C examples.

LabWindows/CVI

To view the NI-XNET function panels, select **Library»NI-XNET**. This opens a dialog containing the NI-XNET classes. You also can use the Library Tree to access all the function panels quickly. To use the NI-XNET Library Tree, go to **View** and make sure that **Library Tree** is selected. In the Library Tree, expand **Libraries** and scroll down to **NI-XNET**.

You can access the help for each class or function panel by right-clicking the function panel and selecting **Class Help...** or **Function Help...**.

NI-XNET includes LabWindows/CVI examples that demonstrate a wide variety of use cases. The examples build on the basic concepts to demonstrate more in-depth use cases.

To view the NI-XNET examples, select **Find Examples...** from the LabWindows/CVI **Help** menu. When you browse examples by task, NI-XNET examples are under **Hardware Input and Output**. The examples are grouped by protocol in CAN, FlexRay, and LIN folders. Although you can write NI-XNET applications for either protocol, and each folder contains shared examples, this organization helps you find examples for your specific hardware product.

Open an example project by double-clicking its name. To run the example, select values using the front panel controls, then read the instructions on the front panel to run the examples. Suggested examples to get started with NI-XNET include:

CAN (**Hardware Input and Output»CAN»NI-XNET»Basic**):

- CAN Signal Input Single Point with CAN Signal Output Single Point
- CAN Signal Input Waveform with CAN Signal Output Waveform

- CAN Frame Input Stream with any output example.

FlexRay (**Hardware Input and Output»FlexRay»Basic**):

- FlexRay Signal Input Single Point with FlexRay Signal Output Single Point.
- FlexRay Signal Input Waveform with FlexRay Signal Output Waveform.
- FlexRay Frame Input Stream with any output example.

LIN (**Hardware Input and Output»LIN»NI-XNET»Basic**):

- LIN Signal Input Single Point with LIN Signal Output Single Point
- LIN Signal Input Waveform with LIN Signal Output Waveform
- LIN Frame Input Stream with any output example

Visual C++

Refer to the ***Microsoft Visual Studio Support*** section in the NI-XNET readme file for the versions of Microsoft Visual C/C++ that your NI-XNET software supports.

The `NIEXTCCOMPILERSUPP` environment variable is provided as an alias to the C language header file and library location. You can use this variable when compiling and linking an application.

For compiling applications that use the NI-XNET API, you must include the `nixnet.h` header file in the code.

For C applications (files with a .c extension), include the header file by adding a `#include` to the beginning of the code, such as:

```
#include "nixnet.h"
```

In your project options for compiling, you must include this statement to add a search directory to find the header file:

```
/I "$(NIEXTCCOMPILERSUPP)include"
```

For linking applications, you must add the nixnet.lib file and the following statement to your linker project options to search for the library:

```
/libpath:"$(NIEXTCCOMPILERSUPP)\lib32\msvc"
```

The reference for each NI-XNET API function is in ***NI-XNET API for C Reference***.

For compiling applications that use the Automotive Ethernet Socket API for C, you must include the `nxsocket.h` header file in the code. To include the header file, add `#include` to the beginning of the code. For example:

```
#include "nxsocket.h"
```

In your project options for compiling, you must include the following statement to add a search directory to find the header file:

```
/I "$(NIEXTCCOMPILERSUPP)include"
```

For linking applications, you must add the nixntipstack.lib file and the following statement to your linker project options to search for the library (for a 32-bit OS, replace ***lib64*** with ***lib32***):

```
/libpath:"$(NIEXTCCOMPILERSUPP)\lib64\msvc"
```

The reference for each NI-XNET IP Stack API function is in IP Stack Functions. The reference for each NI-XNET Socket API function is in Socket Functions, and each socket option is referenced in Socket Options.

Related concepts:

- How Do I Create a Session?
- IP Stack
- Sockets

Displaying Available Interfaces

Measurement and Automation Explorer (MAX)

Use NI MAX to view your available NI-XNET hardware, including all devices and interfaces.

To view hardware in your local Windows system, select **Devices and Interfaces** under **My System**. Each NI-XNET device is listed by hardware model name followed by port name, for example, NI PCI-8517 "FlexRay1, FlexRay2".

Select each NI-XNET device to view its physical ports. Each port is listed with the current interface name assignment, such as FlexRay1.

In the selected port's window on the right, you can change one property: the interface name. Therefore, you can assign a different interface name than the default. For example, you can change the interface for physical port 2 of a PCI-8517 to FlexRay1 instead of FlexRay2. The blinking LED test panel assists in identifying a specific port when your system contains multiple instances of the same hardware product (for example, a chassis with five CAN devices).

To view hardware in a remote LabVIEW Real-Time system, find the desired system under **Remote Systems** and select **Devices and Interfaces** under that system. The features of NI-XNET devices and interfaces are the same as the local system.

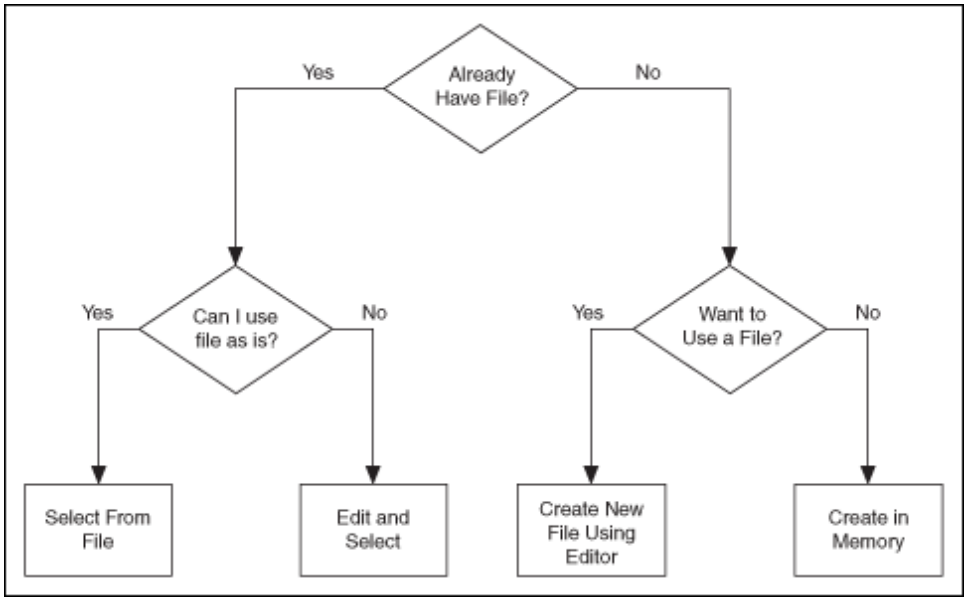
Related concepts:

- How Do I Create a Session?

Database Programming for the C API

The NI-XNET software provides various methods for creating your application database configuration. The following figure shows a process for deciding the database source. A description of each step in the process follows the flowchart.

Figure 5. Decision Process for Choosing a Database Source



Already Have File?

If you are testing an ECU used within a vehicle, the vehicle maker (or the maker's supplier) already may have provided a database file. This file likely would be in CANdb, FIBEX, AUTOSAR, or LDF format. When you have this file, using NI-XNET is relatively straightforward.

Can I Use File As Is?

Is the file up to date with respect to your ECU(s)?

If you do not know the answer to this question, the best choice is to assume Yes and begin using NI-XNET with the file. If you encounter problems, you can use the techniques discussed in "Edit and Select" to update your application without significant redesign.

Select From File

You can simply pass the names of objects from the database to the `List` parameter and the database name (alias or filepath) itself to the `DatabaseName` parameter of `nxCreateSession`. This uses the selected objects from the database in the session created.

Edit and Select

There are two options for editing the database objects to use for NI-XNET sessions: edit in memory and edit the file.

- **Edit in Memory**: Use `nxdbFindObject` and `nxdbSetProperty` to change properties of selected objects. This changes the representation in memory, but does not save the change to the file. When you pass the object into `nxCreateSession`, the changes in memory (not the original file) are used.
- **Edit the File**: The NI-XNET Database Editor is a tool for editing database files for use with NI-XNET. Using this tool, you open an existing file, edit the objects, and save those changes. You can save the changes to the existing file or a new file.

  When you have a file with the changes you need, you select objects in your application as described in "Select From File."

Want to Use a File?

If you do not have a usable database file, you can choose to create a file or avoid files altogether for a self-contained application.

Create New File Using Editor

You can use the NI-XNET Database Editor to create a new database file. Once you have a file, you select objects in your application as described in "Select From File."

As a general rule, for FlexRay applications, using a FIBEX file is recommended. FlexRay communication configuration requires a large number of complex properties, and storage in a file makes this easier to manage. The NI-XNET Database Editor has features that facilitate this configuration.

Create in Memory

You can use `nxdbCreateObject` to create new database objects in memory. Using this technique, you can avoid files entirely and make your application self contained.

You configure each object you create using the property node. Each class of database object contains required properties that you must set (refer to "Required Properties").

The database name is :memory:. This special database name specifies a database that does not originate from a file.

After you create and configure objects in memory, you can use `nxdbSaveDatabase` to save the objects to a file. This enables you to implement a database editor within your application.

Multiple Databases Simultaneously

NI-XNET allows up to 63 database sessions to be open at the same time. You can open any database from a database file or in memory. To open multiple in-memory databases, use the name `:memory[<digit>]:`; for example, `:memory:`, `:memory1:`, `:memory2:`.

Related concepts:

- Creating a Built Application
- How Do I Create a Session?
- CAN Overview

Using NI-CAN

NI-CAN is the legacy application programming interface (API) for National Instruments CAN hardware. Generally speaking, NI-CAN is associated with the legacy CAN hardware, and NI-XNET is associated with the new NI-XNET hardware.

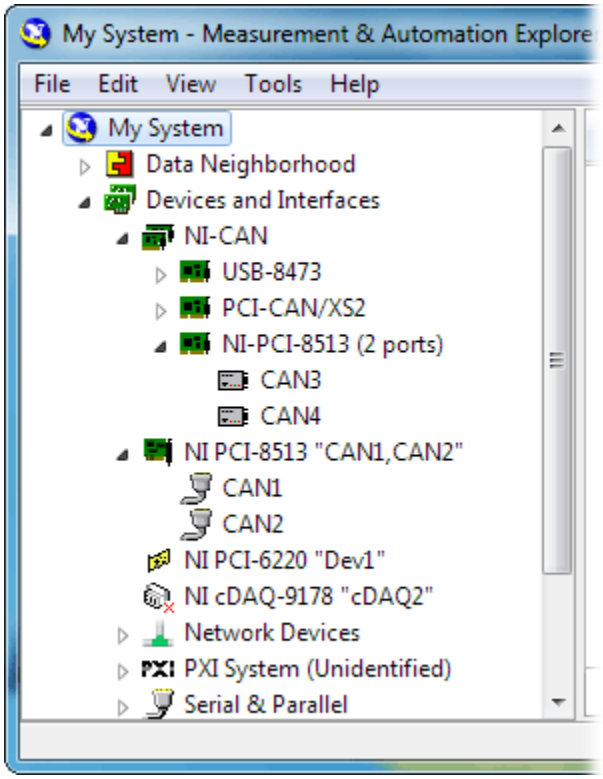
If you are starting a new application, you typically use NI-XNET (not NI-CAN).

Compatibility

If you have an existing application that uses NI-CAN, a compatibility library is provided so that you can reuse that code with a new NI-XNET CAN product. Because the features of the compatibility library apply to the NI-CAN API and not NI-XNET, it is described in the NI-CAN documentation. For more information, refer to the ***NI-CAN Manual***.

NI-XNET CAN Products in MAX

When the compatibility library is installed, NI-XNET CAN products also are visible in the **NI-CAN**branch under **Devices and Interfaces**. Here you can configure the devices for use with the NI-CAN API. This configuration is independent from the configuration of the same device for NI-XNET under the root of **Devices and Interfaces**. The following figure shows the same NI-XNET device, the NI PCI-8513, configured for use with the NI-XNET API (interfaces CAN1 and CAN2) and with the NI-CAN API (interfaces CAN3 and CAN4).



Transition

If you have an existing application that uses NI-CAN and intend to use only new NI-XNET hardware from now on, you may want to transition your code to NI-XNET.

NI-XNET unifies many concepts of the earlier NI-CAN API, but the key features are similar.

The following table lists NI-CAN terms and analogous NI-XNET terms.

NI-CAN Term	NI-XNET Term	Comment
CANdb file	Database	NI-XNET supports more database file formats than the NI-CAN Channel API, including the FIBEX, AUTOSAR, and LDF formats.
Message	Frame	The term Frame is the industry convention for the bits that transfer on the bus. This term is used in standards such as CAN.
Channel	Signal	The term Signal is the industry convention. This term is used in standards such as FIBEX and AUTOSAR.
Channel API Task	Session (Signal I/O)	Unlike NI-CAN, NI-XNET supports simultaneous use of channel (signal) I/O and frame I/O.
Frame API CAN Object (Queue Length Zero)	Session (Frame I/O Single-Point)	The NI-CAN CAN Object provided both input (read) and output (write) in one object. NI-XNET provides a different object for each direction, for better control. If the NI-CAN queue length for a direction is zero, that is analogous to NI-XNET Frame I/O Single-Point.
Frame API CAN Object (Queue Length Nonzero)	Session (Frame I/O Queued)	If the NI-CAN queue length for a direction is nonzero, that is analogous to NI-XNET Frame I/O Queued.
Frame API Network Interface Object	Session (Frame I/O Stream)	The NI-CAN Network Interface Object provided both input (read) and output (write) in one object. NI-XNET provides a different object for each direction, for better control.
Interface	Interface	NI-CAN started interface names at CAN0, but NI-XNET starts at CAN1 (or FlexRay1).

CAN Timing Type and Session Mode

For each XNET Frame CAN:Timing Type property value, this topic describes how the frame behaves for each XNET session mode.

An input session receives the CAN data frame from the network, and an output session transmits the CAN data frame. The CAN data frame data (payload) is mapped to/from signal values.

You use CAN remote frames to request the associated CAN data frame from a remote ECU. When Timing Type is Cyclic Remote or Event Remote, an input session transmits the CAN remote frame, and an output session receives the CAN remote frame.

Cyclic Data

The data frame transmits in a cyclic (periodic) manner. The XNET Frame CAN:Transmit Time property defines the time between cycles.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, and Frame Input Queued Modes

You specify the CAN frame (or its signals) when you create the session. When the CAN data frame is received, a subsequent call to the appropriate `nxRead` function returns its data. For information about how the data is represented for each mode, refer to Session Modes.

If the CAN remote frame is received, it is ignored (with no effect on the appropriate `nxRead` function).

Frame Input Stream Mode

You specify the CAN cluster when you create the session, but not the specific CAN frame. When the CAN data frame is received, a subsequent call to the appropriate `nxRead` function returns its data.

If the CAN remote frame is received, a subsequent call to the appropriate `nxRead` function for the stream returns it.

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

You specify the CAN frame (or its signals) when you create the session. When you write data using the appropriate `nxWrite` function, the CAN data frame is transmitted onto the network. For information about how the data is represented for each mode, refer to Session Modes.

When the session and its associated interface are started, the first cycle occurs, and the

CAN data frame transmits. After that first transmit, the CAN data frame transmits once every cycle, regardless of whether the appropriate `nxWrite` function is called. If no new data is available for transmit, the next cycle transmits using the previous CAN data frame (repeats the payload).

If you pass the CAN remote frame to the appropriate `nxWrite` function, it is ignored.

Frame Output Stream Mode

You specify the CAN cluster when you create the session, but not the specific CAN frame. When you write the CAN data frame using the `nxWrite` function, it is transmitted onto the network.

The stream I/O modes do not use the database-specified timing for frames. Therefore, CAN data and CAN remote frames transmit only when you pass them to the `nxWrite` function, and do not transmit cyclically afterward.

When using a stream output timing of immediate mode, data is transmitted onto the network as soon as possible.

When using a stream output timing of either Replay Exclusive or Replay Inclusive, data is transmitted onto the network based on the timestamps in the frame.

Event Data

The data frame transmits in an event-driven manner. For output sessions, the event is the appropriate `nxWrite` function. The XNET Frame CAN:Transmit Time property defines the minimum interval.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, and Frame Input Queued Modes

The behavior is the same as Cyclic Data.

Frame Input Stream Mode

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can read either CAN data or CAN remote

frames.

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

The behavior is the same as Cyclic Data, except that the CAN data frame does not continue to transmit cyclically after the data from the appropriate `nxWrite` function has transmitted. Because the database-specified timing for the frame is event based, after the CAN data frames for the appropriate `nxWrite` function have transmitted, the CAN data frame does not transmit again until a subsequent call to the appropriate `nxWrite` function.

Frame Output Stream Mode

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can write either CAN data or CAN remote frames.

Cyclic Remote

The CAN remote frame transmits in a cyclic (periodic) manner, followed by the associated CAN data frame as a response.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, and Frame Input Queued Modes

You specify the CAN frame (or its signals) when you create the session. When the CAN data frame is received, a subsequent call to the appropriate `nxRead` function returns its data. For information about how the data is represented for each mode, refer to Session Modes.

When the session and its associated interface are started, the first cycle occurs, and the CAN remote frame transmits. This CAN remote frame requests data from the remote ECU, which soon responds with the associated CAN data frame (same identifier). After that first transmit, the CAN remote frame transmits once every cycle. You do not call the appropriate `nxWrite` function for the session.

The CAN remote frame cyclic transmit is independent of the corresponding CAN data

frame reception. When NI-XNET transmits a CAN remote frame, it transmits a CAN remote frame again CAN:Transmit Time later, even if no CAN data frame is received.

Frame Input Stream Mode

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can read either CAN data or CAN remote frames.

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

You specify the CAN frame (or its signals) when you create the session. When you write data using the appropriate `nxWrite` function, the CAN data frame is transmitted onto the network when the associated CAN remote frame is received (same identifier). For information about how the data is represented for each mode, refer to Session Modes.

Although the session receives the CAN remote frame, you do not call `nxRead` to read that frame. NI-XNET detects the received CAN remote frame, and immediately transmits the next CAN data frame. Your application uses the appropriate `nxWrite` function to provide the CAN data frames used for transmit. When you call the appropriate `nxWrite` function, the CAN data frame does not transmit immediately, but instead waits for the associated CAN remote frame to be received.

Frame Output Stream Modes

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can write either CAN data or CAN remote frames.

Event Remote

The CAN remote frame transmits in an event-driven manner, followed by the associated CAN data frame as a response. For input sessions, the event is the appropriate `nxWrite` function.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, and Frame Input Queued Modes

You specify the CAN frame (or its signals) when you create the session. When the CAN data frame is received, its data is returned from a subsequent call to the appropriate `nxRead` function. For information about how the data is represented for each mode, refer to Session Modes.

This CAN Timing Type and mode combination is somewhat advanced, in that you must call both the appropriate `nxRead` and `nxWrite` functions. You must call the appropriate `nxWrite` function to provide the event that triggers the CAN remote frame transmit. When you call the appropriate `nxWrite` function, the data is ignored, and one CAN remote frame transmits as soon as possible. Each call to the appropriate `nxWrite` function transmits only one CAN remote frame, even if you provide multiple signal or frame values. When the remote ECU receives the CAN remote frame, it responds with a CAN data frame, which is received and read using the appropriate `nxRead` function.

Frame Input Stream Modes

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can read either CAN data or CAN remote frames.

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

The behavior is the same as Cyclic Remote. When you write data using the appropriate `nxWrite` function, the CAN data frame transmits onto the network when the associated CAN remote frame is received (same identifier). Unlike Cyclic Data, the remote ECU sends the associated CAN remote frame in an event-driven manner, but the behavior is the same regarding the appropriate `nxWrite` function and the CAN data frame transmit.

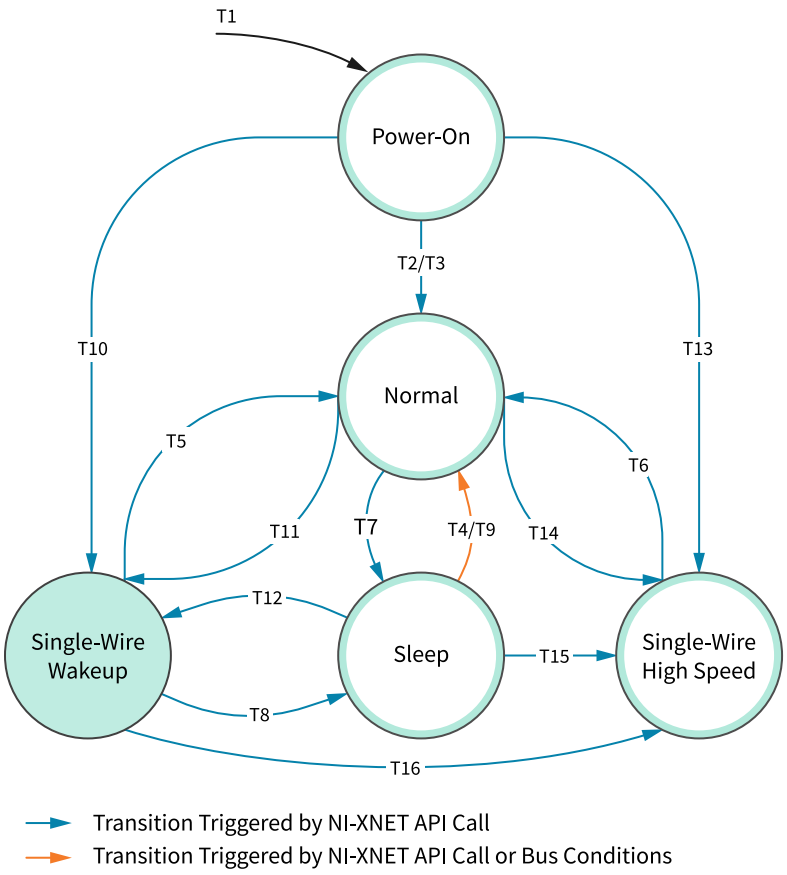
Frame Output Stream Mode

The behavior is the same as Cyclic Data. Because the stream I/O modes ignore the database-specified timing for all frames, you can write either CAN data or CAN remote

frames.

CAN Transceiver State Machine

The CAN hardware internally runs a state machine for controlling the transceiver state. The transceiver can either be an internal transceiver or an external transceiver. On hardware that contains software selectable transceivers, you can configure the selected transceiver bysetting the Interface:CAN:Transceiver Type property. If you choose an external transceiver,you can configure its behaviors by setting the Interface:CAN:External Transceiver Config property. Both bus conditions as well as the Interface:CAN:Transceiver State property can affect the current transceiver state. The following state machine shows the different states of the transceiver state machine and how the various states transition.



T#	Condition	From	To
1	Power-on/close last session	Any	Power-on

T#	Condition	From	To
2	Interface is started	Power-on	Normal
3	Interface:CAN:Transceiver State with value Normal	Power-on	Normal
4	Interface:CAN:Transceiver State with value Normal	Sleep	Normal
5	Interface:CAN:Transceiver State with value Normal	SW Wakeup	Normal
6	Interface:CAN:Transceiver State with value Normal	SW High Speed	Normal
7	Interface:CAN:Transceiver State with value Sleep	Normal	Sleep
8	Interface:CAN:Transceiver State with value Sleep	SW Wakeup	Sleep
9	Wakeup Pattern received on the bus	Sleep	Normal
10	Interface:CAN:Transceiver State with value SWWakeup	Power-on	SW Wakeup
11	Interface:CAN:Transceiver State with value SWWakeup	Normal	SW Wakeup
12	Interface:CAN:Transceiver State with value SWWakeup	Sleep	SW Wakeup
13	Interface:CAN:Transceiver State with value SWHighSpeed	Power-on	SWHigh Speed
14	Interface:CAN:Transceiver State with value SWHighSpeed	Normal	SWHigh Speed
15	Interface:CAN:Transceiver State with value SWHighSpeed	Sleep	SWHigh Speed
16	Interface:CAN:Transceiver State with value SWHighSpeed	SW Wakeup	SWHigh Speed

Using FlexRay

This topic summarizes some useful NI-XNET features specific to the FlexRay protocol.

Starting Communication

FlexRay is a Time Division Multiple Access (TDMA) protocol, which means that all hardware products on the network share a synchronized clock. Slots of time for that clock determine when each frame transmits.

To start communication on FlexRay, the first step is to start the synchronized network clock. In the FlexRay database, two or more hardware products are designated to transmit a special startup frame. These products (nodes) are called coldstart nodes. Each coldstart node uses the startup frame to contribute its local clock as part of the

shared network clock.

Because at least two coldstart nodes are required to start FlexRay communication, your NI-XNET FlexRay interface may need to act as a coldstart node, and therefore transmit a special startup frame. The properties of each startup frame (including the time slot used) are specified in the FlexRay database.

The following scenarios apply to FlexRay startup frames:

- **Port to port** : When you get started with your NI-XNET FlexRay hardware, you can connect two FlexRay interfaces (ports) to run simple programs, such as the NI-XNET examples. Because this is a cluster with two nodes, each NI-XNET interface must transmit a different startup frame.
- **Connect to existing cluster** : If you connect your NI-XNET FlexRay interface to an existing cluster (for example, a FlexRay network within a vehicle), that cluster already must contain coldstart nodes. In this scenario, the NI-XNET interface should not transmit a startup frame.
- **Test single ECU that is coldstart** : If you connect to a single ECU (and nothing else), and that ECU is a coldstart node, the NI-XNET interface must transmit a startup frame. The NI-XNET interface must transmit a startup frame that is different than the startup frame the ECU transmits.
- **Test single ECU that is not coldstart** : If you connect to a single ECU (and nothing else), and that ECU is not a coldstart node, you must connect two NI-XNET interfaces. The ECU cannot communicate without two coldstart nodes (two clocks). According to the FlexRay specification, a single FlexRay interface can transmit only one startup frame. Therefore, you need to connect two NI-XNET FlexRay interfaces to the ECU, and each NI-XNET interface must transmit a different startup frame.

NI-XNET has two options to transmit a startup frame:

- **Key Slot Identifier** : The NI-XNET Session Node includes a property called Interface:FlexRay:Key Slot Identifier. This property specifies the static slot that the session interface uses to transmit a startup frame. The value of this property is zero (0) by default, meaning that no startup frame transmits. If you set this property, the value specifies the static slot (identifier) to transmit as a coldstart node. The startup frame transmits automatically when the interface starts, and its payload is null (no data). The session can be input or output, and the startup frame is not

required in the session's list of frames/signals.
- **Output Startup Frame** : If you create an NI-XNET output session, and the session's list of frames/signals uses a startup frame, the NI-XNET interface acts as a coldstart node.

To find startup frames in the database, look for a frame with the FlexRay:Startup? property true. You can use that frame name for an output session or use its identifier as the key slot. When selecting a startup frame, avoid selecting one that the ECUs you connect to already transmit.

Understanding FlexRay Frame Timing

When you use an NI-XNET database for FlexRay, the properties of each FlexRay frame specify the FlexRay data transfer timing. To understand how the FlexRay frame timing properties apply to NI-XNET sessions, refer to FlexRay Timing Type and Session Mode.

In LabVIEW Real-Time, NI-XNET provides a timing source you can use to synchronize your LabVIEW VI with the timing of frames. For more information, refer to Using LabVIEW Real-Time.

Protocol Data Unit (PDU)

Many FlexRay networks use a Protocol Data Unit (PDU) to implement configurations similar to CAN. The PDU is a signal container. You can use a single PDU within multiple frames for faster timing. A single frame can contain multiple PDUs, each updated independently. For more information, refer to Protocol Data Units (PDUs) in NI-XNET.

FlexRay Startup/Wakeup

Use the FlexRay Startup mechanism to take an idle interface and properly integrate into a FlexRay cluster.

If your cluster does not support the wakeup mechanism, this process is straightforward. After creating your FlexRay session, call `nxStart`, which causes the interface to transition from **Default Config** to **Ready**, where it attempts to integrate with the FlexRay cluster. If your node is a coldstart node, it initiates integration; otherwise, it attempts to integrate with a running FlexRay cluster. Once integration has

occurred, the interface transitions to **Normal Active**, where it typically remains while it is communicating with other FlexRay nodes. When you call `nxStop`, the interface transitions back to **Default Config** (via **Halt**) to be ready to start the process again.

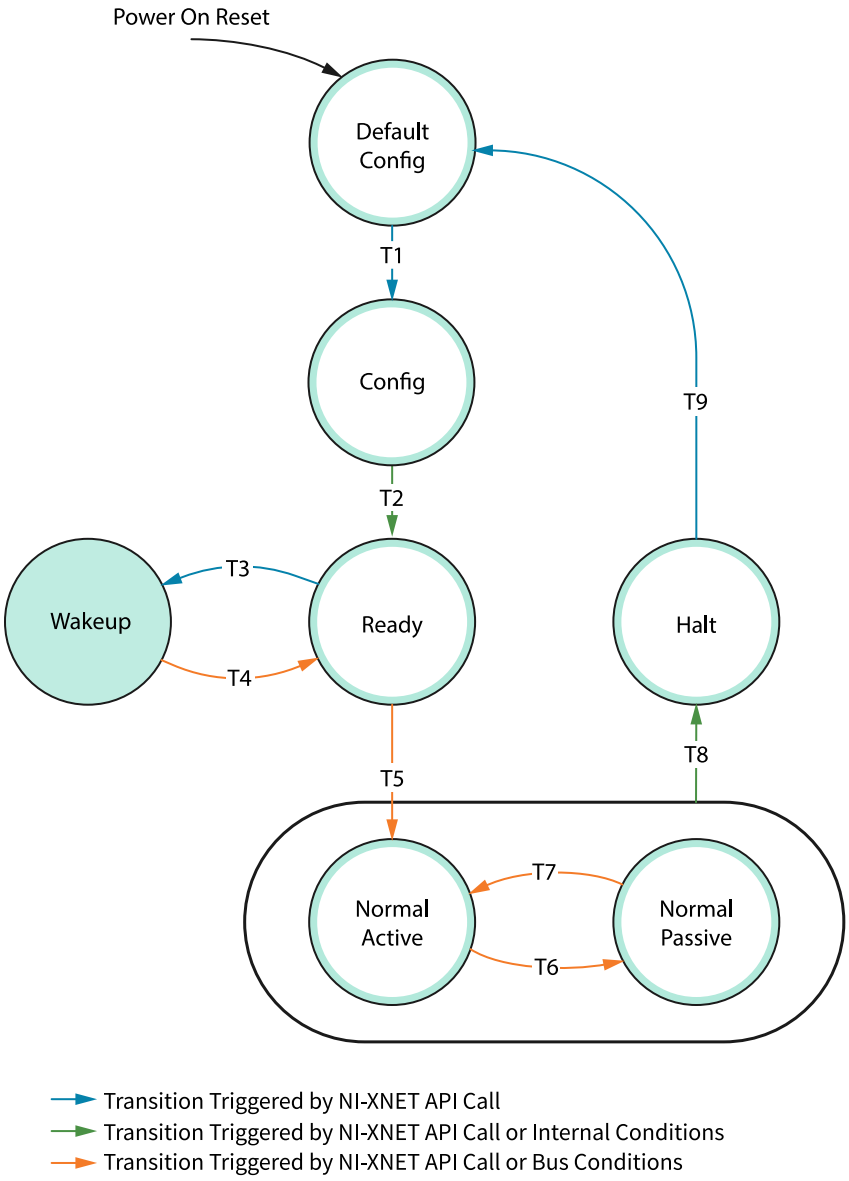
If your cluster supports the wakeup mechanism, the process becomes a bit more complex. The route the XNET hardware takes depends on whether the interface is currently awake or asleep. By default, XNET hardware starts in the awake state, and the startup process is exactly the same as if your cluster does not support wakeup. However, to use the wakeup mechanism your cluster is configured for, before calling `nxStart`, you need to put the interface to sleep. You can do this in one of two ways. First, you can set the Interface:FlexRay:Sleep property to `nxFlexRaySleep_LocalSleep`. This performs the one-time action of putting the interface to sleep. Alternately,you can set the Interface:FlexRay:Auto Asleep When Stopped? property to true. This puts the interface to sleep immediately. It also puts the interface to sleep automatically every time the interface is stopped, so the startup process is the same between your first start and subsequent starts.

If your interface is asleep when the `nxStart` API call is invoked, the interface progresses to **Ready**, where it waits for all connected channels to be awake before attempting to integrate with the cluster. After all connected channels are awake, the integration process occurs exactly like a cluster that does not support wakeup.

If you want your interface to wake up a sleeping network, you must configure your FlexRay interface to wake up the bus. You can do this in two ways. The first way is to set the Interface:FlexRay:Sleep property to `nxFlexRaySleep_RemoteWake` after you put your FlexRay interface to sleep. When you invoke the `nxStart` API call, the interface progresses though the **Ready** state and into the **Wakeup** state. In **Wakeup**, the interface generates the wakeup pattern on the FlexRay channel configured by the Interface:FlexRay:Wakeup Channel property and transitions back to **Ready**. If you have a multichannel bus, a separate node on the bus wakes up the other channel.

After all connected channels are awake, the integration process occurs exactly like a cluster that does not support wakeup. The second way is to invoke the `nxStart` API call to start the interface. The interface progresses to **Ready**, where it waits for all connected channels to be awake before attempting to integrate with the cluster. During this time, if you set the Interface:FlexRay:Sleep property to `nxFlexRaySleep_RemoteWake`, the interface transitions into Wakeup, where it generates the wakeup pattern on the FlexRay channel configured by the

Interface:FlexRay:Wakeup Channel property and transitions back to **Ready**. If you have a multichannel bus, a separate node on the bus wakes up the other channel. After all connected channels are awake, the integration process occurs exactly like a cluster that does not support wakeup.



T#	Condition	From	To
1	Start trigger received[1]	Default Config	Config[2]
2	Startup process initiated	Config	Ready
3	Remote Wakeup initiated	Ready	Wakeup

T#	Condition	From	To
	(Interface:FlexRay:Sleep property set to `nxFlexRaySleep_RemoteWake`)		
4	Wakeup channel awake	Wakeup	Ready
5	All connected channels are awake and integration is successful [3].	Ready	Normal Active
6	Clock Correction Failed counter reached Maximum Without Clock Correction Passive. Value	Normal Active	Normal Passive
7	Number of valid correction terms reached the passive to active limit	Normal Passive	Normal Active
8	1. Clock Correction Failed counter reached Maximum Without Clock Correction Fatal Value. 2. Interface stopped (`nxStop`).		
9	Interface stopped (`nxStop`)	Halt	Default Config

[1] If you are not using synchronization, the `nxStart` API call internally generates the Start Trigger.

[2] In NI-XNET, this is a transitory state under normal situations. The Config state is nontransitory only if the startup procedure fails to continue.

[3] Any of the following conditions can satisfy all channels awake: the wakeup pattern was transmitted or received on all connected channels, a local wakeup is requested, or the interface is not asleep.

FlexRay Timing Type and Session Mode

For each XNET Frame FlexRay:Timing Type property value, this topic describes how the frame behaves for each XNET session mode.

An input session receives the FlexRay data frame from the network, and an output session transmits the FlexRay data frame. The FlexRay data frame data (payload) is mapped to/from signal values.

You use FlexRay null frames in the static segment to indicate that no new payload exists for the frame. In the dynamic segment, if no new payload exists for the frame, it simply does not transmit (no frame).

For NI-XNET input sessions, the Timing Type does not directly impact the representation of data from the appropriate `nxRead` function.

For NI-XNET output sessions, the Timing Type determines whether to transmit a data frame when no new payload data is available.

Cyclic Data

The data frame transmits in a cyclic (periodic) manner.

If the frame is in the static segment, the rate can be once per cycle (FlexRay:Cycle Repetition 1), once every N cycles (FlexRay:Cycle Repetition N), or multiple times per cycle (FlexRay:In Cycle Repetitions:Enabled?).

If the frame is in the dynamic segment, the rate is once per cycle.

If no new payload data is available when it is time to transmit, the payload data from the previous transmit is repeated.

Signal Input Single-Point, Signal Input Waveform, and Signal Input XY Modes

You specify the FlexRay signals when you create the session, and a specific FlexRay data frame contains each signal. When the FlexRay data frame is received, a subsequent call to the appropriate `nxRead` function returns its data. For information about how the data is represented for each mode, refer to Session Modes.

If a FlexRay null frame is received, it is ignored (no effect on the `nxRead` function). FlexRay null frames are not used to map signal values.

Frame Input Queued and Frame Input Single-Point Modes

You specify the FlexRay frame(s) when you create the session. When the FlexRay data frame is received, a subsequent call to the appropriate `nxRead` function returns its data. For information about how the data is represented for each mode, refer to Session Modes.

If a FlexRay null frame is received, it is ignored (not returned).

Frame Input Stream Mode

You specify the FlexRay cluster when you create the session, but not the specific FlexRay frames. When any FlexRay data frame is received, a subsequent call to the appropriate `nxRead` function returns it.

If the XNET Session Interface:FlexRay:Null Frames To Input Stream? property is true, and FlexRay null frames are received, a subsequent call to `nxRead` for the stream returns them. If Null Frames To Input Stream? is false (default), FlexRay null frames are ignored (not returned). You can determine whether each frame value is data or null by evaluating the `type` element (refer to the appropriate `nxRead` function).

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

You specify the FlexRay frame (or its signals) when you create the session. When you write data using the appropriate `nxWrite`function, the FlexRay data frame is transmitted onto the network. For information about how the data is represented for each mode, refer to Session Modes.

When the session and its associated interface are started, the FlexRay data frame transmits according to its rate. After that first transmit, the FlexRay data frame transmits according to its rate, regardless of whether the appropriate `nxWrite` function is called. If no new data is available for transmit, the next cycle transmits using the previous FlexRay data frame (repeats the payload).

If the frame is contained in the static segment, a FlexRay data frame transmits at all times. The FlexRay null frame is not transmitted. If you pass the FlexRay null frame to the appropriate `nxWrite` function, it is ignored.

If the frame is contained in the dynamic segment, a FlexRay data frame transmits every cycle. The dynamic frame minislot is always used.

Frame Output Stream Mode

This session mode is not supported for FlexRay.

Event Data

The data frame transmits in an event-driven manner. The event is the appropriate `nxWrite` function.

Because FlexRay is a time-driven protocol, the minimum interval between events is specified based on the FlexRay cycle. This minimum interval is configured in the same manner as a Cyclic frame.

If the frame is in the static segment, the interval can be once per cycle (FlexRay:Cycle Repetition 1), once every N cycles (FlexRay:Cycle Repetition N), or multiple times per cycle (FlexRay:In Cycle Repetitions:Enabled?).

If the frame is in the dynamic segment, the interval is once per cycle.

If no new event (payload data) is available when it is time to transmit, no frame transmits. In the static segment, this lack of new data is represented as a null frame.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, Frame Input Queued, and Frame Input Stream Modes

The behavior is the same as Cyclic Data.

Signal Output Single-Point, Signal Output Waveform, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

The behavior is similar to Cyclic Data, except that the FlexRay data frame does not continue to transmit cyclically after the data from the appropriate `nxWrite` function

has transmitted. Because the database-specified timing for the frame is event based, after the FlexRay data frames for the appropriate `nxWrite` function have transmitted, the FlexRay data frame does not transmit again until a subsequent call to the appropriate `nxWrite` function.

If the frame is contained in the static segment, a FlexRay null frame transmits when no new data is available (no new call to the appropriate `nxWrite` function). If you pass the FlexRay null frame to the appropriate `nxWrite` function, it is ignored.

If the frame is contained in the dynamic segment, the frame does not transmit when no new data is available. The dynamic frame minislot is used only when new data is provided to the appropriate `nxWrite` function.

Frame Output Stream Mode

This session mode is not supported for FlexRay.

Using LIN

This section summarizes some useful NI-XNET features specific to the LIN protocol.

Changing the LIN Schedule

LIN networks (clusters) always include 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, a single ECU in the network (slave) responds by transmitting the payload for the requested ID. The master ECU also can respond to a specific header, and thus the master can transmit payload data for the slave ECUs to receive.

Unlike some other scheduled protocols such as FlexRay, LIN allows the master ECU to change the schedule of frame headers. For example, the master can initially use a "normal" schedule that requests IDs 1, 2, 3, 4, and then the master can change to a "diagnostic" schedule that requests IDs 60 and 61.

With NI-XNET, you change the LIN schedule using the `nxWriteState (u32 StateID=nxState_LINScheduleChange)`. When you want the NI-XNET

interface to act as a master on the network, you must call this function at least once, to specify the schedule to run. When you write a schedule change, this automatically configures NI-XNET as master (the XNET Session Interface:LIN:Master? property is set to true). As a LIN master, NI-XNET handles all real-time scheduling of frame headers for you, using the LIN interface hardware onboard processor.

If you do not write a schedule change, NI-XNET leaves the interface at its default configuration of slave. As a LIN slave, you still can write signal or frame values to an output session, but NI-XNET waits for each frame's header to arrive before transmitting payload data.

Understanding LIN Frame Timing

Because LIN is a scheduled network, the headers that the master transmits determine the timing of all frames. To understand how and when each frame transmits, you must examine the entries in each schedule. Each entry transfers one frame (or possibly multiple frames). For more information, refer to the XNET LIN Schedule Entry Type property.

Because it is possible to use a single frame in multiple schedules and schedule entries, the overall timing for an individual frame can be complex. Nevertheless, each LIN schedule entry generally fits the concepts of cyclic and event timing that are common for other protocols such as CAN and FlexRay. For more information about how these concepts apply to LIN, refer to Cyclic and Event Timing.

LIN Diagnostics

Refer to `nxWriteState (nxState_LINDiagnosticScheduleChange)` for details.

Special Considerations for Using Stream Output Mode with LIN

Refer to the `Interface:Output Stream Timing` Session property for details.

LIN Frame Timing and Session Mode

This section describes the LIN behavior for each XNET session mode. As context for

describing LIN frame transfer on the network, this section uses the timing concepts described in the LIN section of Cyclic and Event Timing.

An input session receives the LIN data frame (payload) from the network, and an output session transmits the LIN data frame. The LIN data frame payload is mapped to/from signal values.

For NI-XNET input sessions, the timing of each LIN schedule entry does not directly impact the representation of data from the appropriate `nxRead` function.

For NI-XNET output sessions, the timing of each LIN schedule entry determines whether to transmit a data frame when no new payload data is available.

You can configure the NI-XNET LIN interface to run as the LIN master by requesting a schedule (`nxWriteState` function). If the NI-XNET LIN interface runs as a LIN slave (default), a remote ECU on the network must execute schedules as LIN master for these modes to operate.

Cyclic

The LIN data frame transmits in a cyclic (periodic) manner.

This implies that the LIN master is running a continuous schedule, and the LIN data frame is contained within an unconditional schedule entry.

If no new payload data is available when it is time to transmit, the payload data from the previous transmit is repeated.

Signal Input Single-Point, Signal Input Waveform, and Signal Input XY Modes

You specify the signals when you create the session, and a specific LIN data frame contains each signal. When the LIN data frame is received, a subsequent call to the appropriate `nxRead` function returns its signal data. For information about how the data is represented for each mode, refer to Session Modes.

Frame Input Queued and Frame Input Single-Point Modes

You specify the LIN frame(s) when you create the session. When the LIN data frame is received, a subsequent call to the appropriate `nxRead` function returns its data. For

information about how the data is represented for each mode, refer to Session Modes.

Frame Input Stream Mode

You specify the LIN cluster when you create the session, but not the specific LIN frames. When any LIN data frame is received, a subsequent call to the appropriate `nxRead` function returns it.

Signal Output Single-Point, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

You specify the LIN frame (or its signals) when you create the session. When you write data using the appropriate `nxWrite` function, the LIN data frame is transmitted onto the network. For information about how the data is represented for each mode, refer to Session Modes.

When the session and its associated interface are started, the LIN data frame transmits according to its schedule entry. Assuming that the LIN frame is contained in only one entry of the continuous schedule, the time between frame transmissions is the same as the time to execute the entire schedule (all entries). After that first transmit, the LIN data frame transmits according to its schedule entry, regardless of whether the appropriate `nxWrite` function is called. If no new data is available for transmit, the next cycle transmits using the previous LIN data frame (repeats the payload).

Signal Output Waveform Mode

If the NI-XNET interface runs as a LIN master, NI-XNET executes schedules, and therefore controls the timing of LIN frames. When running as a LIN master, this session mode is supported, and NI-XNET resamples the waveform data such that it transmits at the scheduled frame rates.

If the NI-XNET interface runs as a LIN slave (default), this session mode is not supported. When running as a LIN slave, NI-XNET does not know which schedule the LIN master is executing. Because the LIN schedule is not known, the frame transfer rates also are not known, which makes it impossible to resample the waveform data.

Frame Output Stream Mode

This mode is available only when the LIN interface is master. You specify the LIN cluster when you create the session, but not the specific LIN frame.

The stream I/O modes do not use the database-specified timing for frames. Therefore, LIN data frames transmit only when you pass them to the `nxWrite` function and do not transmit cyclically afterward.

When using a stream output timing of immediate mode, data is transmitted onto the network as soon as possible. Specifically, if the data array is empty, only the header part of the frame is transmitted (with the expectation that a slave transmits the response). If the data array is not empty, the header + response parts of the frame (the full frame) is transmitted. You can use this mode in conjunction with the scheduler, in which case each frame written to stream output is handled as a run-once schedule with lowest priority and having a single one-frame entry. A run-continuous schedule is interrupted to transmit the frame. A run-once schedule is not interrupted, and the frame is transmitted only when there are no pending run-once schedules with higher-than-lowest priority.

When using a stream output timing of either Replay Exclusive or Replay Inclusive, data is transmitted onto the network based on the timestamps in the frame.

Refer to the Interface:Output Stream Timing property for more details about using this mode with LIN.

Event

The LIN data frame transmits in an event-driven manner. The event is the appropriate `nxWrite` function.

If no new event (payload data) is available when it is time to transmit, no frame transmits. This means that the LIN master transmits the frame header, but no payload data follows this header.

Signal Input Single-Point, Signal Input Waveform, Signal Input XY, Frame Input Single-Point, Frame Input Queued, and Frame Input Stream Modes

The behavior is the same as Cyclic.

Signal Output Single-Point, Signal Output XY, Frame Output Single-Point, and Frame Output Queued Modes

The behavior is similar to Cyclic , except that the LIN data frame does not continue to transmit after the data from the appropriate `nxWrite` function has transmitted.

If the frame is contained in a sporadic schedule entry, and there are values for multiple frames pending for that entry, NI-XNET selects a single frame to transmit in each entry. NI-XNET selects the frame using the order in the XNET LIN Schedule Entry Frames property. For example, if the Frames property contains three frames, and you write data for the first and third, NI-XNET transmits the first frame (index 0) in the next occurrence of the sporadic entry, and then transmits the third frame (index 2) when that sporadic entry executes again.

If the frame is contained in an event-triggered schedule entry, a collision may occur if another ECU transmits in the same schedule entry. If the NI-XNET LIN interface runs as a LIN master, it automatically uses the XNET LIN Schedule Entry Collision Resolving Schedule property to resolve this collision.

Signal Output Waveform Mode

The behavior is the same as Cyclic.

If the NI-XNET interface runs as a LIN master, NI-XNET executes schedules, and therefore controls the timing of LIN frames. An event-driven LIN frame can transmit at most once per execution of its schedule entry.

If the NI-XNET interface runs as a LIN slave (default), this session mode is not supported.

Frame Output Stream Mode

When using a stream output timing of immediate mode, if the frame for transmit is

defined as an event-triggered frame in the database, and a collision occurs during transmit, the interface automatically executes the collision resolving schedule defined for the frame, exactly as if the frame were transmitted in a scheduled event-triggered slot.

When using a stream output timing of either Replay Exclusive or Replay Inclusive, if the frame for transmit is determined to be defined as an event-triggered frame in the database, the frame is transmitted with a header ID equal to the unconditional frame ID contained in data byte 0. The data is transmitted without modification. In other words, the frame is transmitted as an unconditional frame associated with the event-triggered frame.

Refer to the Interface:Output Stream Timing property for more details about using this mode with LIN.

Related concepts:

- Cyclic and Event Timing
- Session Modes
- Signal Output Waveform Mode

Automotive Ethernet Socket API for C

The Automotive Ethernet Stack functions provide tools to create everything required for TCP and UDP communication, independent from the limitations of the IP stack native to your operating system (OS). A test application typically uses a single IP Stack for each XNET Interface (physical port), but more complex configurations are possible.

For example, suppose that you are testing eight identical instances of an ECU, each instance connected to a distinct XNET Interface (e.g., two 4-port Automotive Ethernet Interface Modules). For each of the eight repeated test setups, you could use the same static IP address for each XNET Interface, and communicate with the same static IP address in the ECU. This configuration is difficult to achieve using the native Windows or Linux IP stack, because the operating system assumes that each interface uses a different unicast IP address.

As another example, to fully test a physical ECU, suppose you need to simulate six real

ECUs that are part of a single in-vehicle network. (This is sometimes called "restbus simulation.") The IP stack enables you to configure six distinct virtual interfaces in the IP stack to represent multiple simulated ECUs. These virtual interfaces can all run on the IP stack associated with a single XNET Interface (physical port) that is connected to your real ECU under test.

For a given XNET Interface, TCP and UDP traffic switch from the OS stack to IP Stack when you call nxIpStackCreate the first time for that XNET Interface. Communication changes back to OS stack when you call nxIpStackClear the last time for that XNET Interface. When you are viewing traffic on the XNET Interface (e.g., Wireshark on ENET2), you might notice that some protocols run in the OS stack (e.g., Windows running DHCPv6), but those protocols cease after you call nxIpStackCreate.

IP Stack functions are specific to NI-XNET.

IP Stack

Use the IP Stack functions to configure IP stacks as needed, and then use the Socket API functions for TCP and/or UDP communication. The header file for IP Stack and Socket functions is named `nxsocket.h` (for more information, refer to Getting Started).

Find details for the following stack functions in the Automotive Ethernet Socket API for C Reference.

- nxIpStackCreate
- nxIpStackOpen
- nxIpStackClear
- nxIpStackGetInfo
- nxIpStackFreeInfo
- nxIpStackGetAllStacksInfoStr
- nxIpStackFreeAllStacksInfoStr
- nxIpStackWaitForInterface

Related concepts:

- Getting Started with NI-XNET C API

Sockets

After you configure the IP stacks as needed for your test, use the Socket API functions for TCP and/or UDP communication. The Socket API functions are designed to match the well-known Berkeley Sockets API (also known as BSD Sockets). The C APIs for Ethernet communication on Windows and Linux align with the Berkeley Sockets API. The alignment of these socket APIs is intended to reduce the learning curve and to facilitate re-use of code between stacks.

The header file for IP Stack and Sockets functions is named `nxsocket.h` (for more information, refer to Getting Started).

Within `nxsocket.h`, the Berkeley Sockets functions, constants, and types use an nx prefix in order to avoid naming collisions with analogous Windows and Linux APIs. Other than this prefix, the Socket API uses the same naming as other Berkeley Sockets APIs. Consistent naming makes it easy to find the wealth of documentation and code on the Internet. For example, if you remove the prefix from the function `nxlisten`, and search the Internet for "socket listen", you can find descriptions and hints for using the `listen` function.

Find details for the following socket functions in the Automotive Ethernet Socket API for C Reference.

- nxaccept
- nxbind
- nxclose
- nxconnect
- nxfreeaddrinfo
- nxgetaddrinfo
- nxgetlasterrornum
- nxgetlasterrorstr
- nxgetnameinfo
- nxgetpeername
- nxgetsockname
- nxgetsockopt
- nxinet_addr
- nxinet_aton

- nxinet_ntoa
- nxinet_ntop
- nxinet_pton
- nxlisten
- nxrecv
- nxrecvfrom
- nxselect
- nxsend
- nxsendto
- nxsetsockopt
- nxshutdown
- nxsocket
- nxstrerr_r

Related concepts:

- Getting Started with NI-XNET C API

Tools

NI-XNET includes several tools, which enable you to analyze network traffic, troubleshoot issues, configure NI-XNET hardware, and verify configuration.

- **Bus Monitor**—Displays statistics for CAN, FlexRay, or LIN frames. This is a basic tool for analyzing CAN, FlexRay, or LIN network traffic.
- **IP Stack Info**—Returns configuration information about active IP stacks on a local or remote target system. Use this tool to troubleshoot issues and verify that IP stacks are created correctly.
- **NI I/O Trace**—Monitors, records, and displays function calls to APIs such as the NI-XNET API. This tool helps in debugging application programming problems.
- **NI-XNET Database Editor**—Enables you to view and manage automotive databases for CAN, CAN/FD, FlexRay, and LIN networks.
- **Port Configuration Utility**—Provides basic functionality to configure NI-XNET devices. You can update firmware and perform a self-test on NI-XNET devices, rename ports, and perform a port blink. The Port Configuration Utility can configure hardware on Windows hosts only.

Bus Monitor

The NI-XNET Bus Monitor is a universal analysis tool for displaying and logging CAN, FlexRay, or LIN network data. You can display network information as either last recent data or historical data view. To identify more detailed frame information, you can assign a network database to the Bus Monitor. If a received frame is found in the database, you can display the message name and comment information in the Monitor view or ID Log view. In addition to the network data, the Bus Monitor can provide statistical information. For offline data analysis, you can stream all received network data to disk in two log file formats.

In the Bus Monitor in the CAN protocol mode, you can interactively transmit an event frame or a periodic frame onto the network. In this mode, you can quickly verify the correct setup of your CAN network and debug your communication with the device under test.

NI-XNET errors that appear while doing a CAN, FlexRay, or LIN measurement within the Bus Monitor are shown in the main user interface.

You can launch the NI-XNET Bus Monitor in three distinct protocol modes: CAN, FlexRay, or LIN, from NI MAX or the Windows **Start** menu.

- In NI MAX, right-click an NI-XNET interface and select **Bus Monitor** from the context menu.
- From the Windows Start menu, select **Start** » **National Instruments** and select **NI-XNET Bus Monitor**.

You cannot switch from one protocol mode to the other during run time. You can run the Bus Monitor in multiple instances on different ports, and can verify the network communication on several CAN, FlexRay, or LIN bus topologies in parallel.

Note The NI-XNET Bus Monitor utility does not support Automotive Ethernet hardware.

IP Stack Info

The NI-XNET IP Stack Info tool returns configuration information about active IP stacks on a target system. The target can be either a local or a remote system. This utility can be launched on Windows hosts only; however, it can also display the information of remote Linux RT targets that have NI-XNET installed.

After installing NI-XNET, you can find the XNET IP Stack Information tool in the following directory: `\Program Files (x86)\National Instruments\NI-XNET\ipStackInfo`.

The utility can also be launched from NI MAX:

1. In MAX, expand **Devices and Interfaces** in the configuration tree.
2. Select any Automotive Ethernet interface on your system.
3. Click **IP Stack Info** above the interface settings at the right of the window.

The XNET IP Stack Info tool displays the information formatted in a hierarchical tree

view, which enables you to expand/close items to show/hide configuration information for different levels of IP stack characteristics.

NI I/O Trace

NI I/O Trace is an application that monitors, records, and displays API calls made by NI applications. Use NI I/O Trace to quickly locate and analyze any erroneous API calls that your application makes, and to verify that the communication with your instrument is correct.

You can launch NI I/O Trace from NI MAX or the Windows **Start** menu.

- In NI MAX, expand the **Software** node of the NI MAX Configuration tree, right-click **NI I/O Trace**, and select **Launch NI I/O Trace**.
- From the Windows Start menu, select **Start** » **National Instruments** and select **NI IO Trace**.

Database Editor

The NI-XNET Database Editor enables you to view and manage automotive databases for CAN, CAN/FD, FlexRay, and LIN networks. This utility provides support for modern automotive industry standards such as AUTOSAR.

To launch the NI-XNET Database Editor, select **Start** » **National Instruments** » **NI-XNET Database Editor**.

Port Configuration Utility

The NI-XNET Port Configuration Utility is a tool included with the NI-XNET run-time installer that provides basic functionality to configure NI-XNET devices. You can update firmware and perform a self-test on NI-XNET devices, rename ports, and perform a port blink. This utility can configure hardware on Windows hosts only.

After installing NI-XNET, you can find the Port Configuration Utility in the following directory: `\Program Files (x86)\National Instruments\NI-XNET\`

`portConfig` . After installing NI-XNET, you can find the Port Configuration Utility in the following directory: `\Program Files (x86)\National Instruments\NI-XNET\portConfig`.