COMPREHENSIVE SERVICES

We offer competitive repair and calibration services, as well as easily accessible documentation and free downloadable resources.

SELL YOUR SURPLUS

We buy new, used, decommissioned, and surplus parts from every NI series. We work out the best solution to suit your individual needs.

Sell For Cash Get Credit Receive a Trade-In Deal

OBSOLETE NI HARDWARE IN STOCK & READY TO SHIP

We stock **New, New Surplus, Refurbished,** and **Reconditioned** NI Hardware.

APEX WAVES

Bridging the gap *between the manufacturer and your legacy test system.*

1-800-915-6216

www.apexwaves.com

sales@apexwaves.com

All trademarks, brands, and brand names are the property of their respective owners.

Request a Quote CLICK HERE ***PCI-8516***

XNET

NI-XNET Hardware and Software Manual

July 2015
372840K-01

NATIONAL INSTRUMENTS™

Worldwide Technical Support and Product Information

ni.com

Worldwide Offices

Visit ni.com/niglobal to access the branch office Websites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

For further support information, refer to the *NI Services* appendix. To comment on National Instruments documentation, refer to the National Instruments website at ni.com/info and enter the Info Code feedback.

© 2009–2015 National Instruments. All rights reserved.

Legal Information

Limited Warranty

This document is provided ‘as is’ and is subject to being changed, without notice, in future editions. For the latest version, refer to `ni.com/manuals`. NI reviews this document carefully for technical accuracy; however, NI MAKES NO EXPRESS OR IMPLIED WARRANTIES AS TO THE ACCURACY OF THE INFORMATION CONTAINED HEREIN AND SHALL NOT BE LIABLE FOR ANY ERRORS.

NI warrants that its hardware products will be free of defects in materials and workmanship that cause the product to fail to substantially conform to the applicable NI published specifications for one (1) year from the date of invoice.

For a period of ninety (90) days from the date of invoice, NI warrants that (i) its software products will perform substantially in accordance with the applicable documentation provided with the software and (ii) the software media will be free from defects in materials and workmanship.

If NI receives notice of a defect or non-conformance during the applicable warranty period, NI will, in its discretion: (i) repair or replace the affected product, or (ii) refund the fees paid for the affected product. Repaired or replaced Hardware will be warranted for the remainder of the original warranty period or ninety (90) days, whichever is longer. If NI elects to repair or replace the product, NI may use new or refurbished parts or products that are equivalent to new in performance and reliability and are at least functionally equivalent to the original part or product.

You must obtain an RMA number from NI before returning any product to NI. NI reserves the right to charge a fee for examining and testing Hardware not covered by the Limited Warranty.

This Limited Warranty does not apply if the defect of the product resulted from improper or inadequate maintenance, installation, repair, or calibration (performed by a party other than NI); unauthorized modification; improper environment; use of an improper hardware or software key; improper use or operation outside of the specification for the product; improper voltages; accident, abuse, or neglect; or a hazard such as lightning, flood, or other act of nature.

THE REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND THE CUSTOMER’S SOLE REMEDIES, AND SHALL APPLY EVEN IF SUCH REMEDIES FAIL OF THEIR ESSENTIAL PURPOSE.

EXCEPT AS EXPRESSLY SET FORTH HEREIN, PRODUCTS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND AND NI DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, WITH RESPECT TO THE PRODUCTS, INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE OR NON-INFRINGEMENT, AND ANY WARRANTIES THAT MAY ARISE FROM USAGE OF TRADE OR COURSE OF DEALING. NI DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF OR THE RESULTS OF THE USE OF THE PRODUCTS IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NI DOES NOT WARRANT THAT THE OPERATION OF THE PRODUCTS WILL BE UNINTERRUPTED OR ERROR FREE.

In the event that you and NI have a separate signed written agreement with warranty terms covering the products, then the warranty terms in the separate agreement shall control.

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

End-User License Agreements and Third-Party Legal Notices

You can find end-user license agreements (EULAs) and third-party legal notices in the following locations:

- Notices are located in the `<National Instruments>\_Legal Information` and `<National Instruments>` directories.
- EULAs are located in the `<National Instruments>\Shared\MDF\Legal\license` directory.
- Review `<National Instruments>\_Legal Information.txt` for information on including legal information in installers built with NI products.

U.S. Government Restricted Rights

If you are an agency, department, or other entity of the United States Government (“Government”), the use, duplication, reproduction, release, modification, disclosure or transfer of the technical data included in this manual is governed by the Restricted Rights provisions under Federal Acquisition Regulation 52.227-14 for civilian agencies and Defense Federal Acquisition Regulation Supplement Section 252.227-7014 and 252.227-7015 for military agencies.

Trademarks

Refer to the *NI Trademarks and Logo Guidelines* at `ni.com/trademarks` for more information on National Instruments trademarks.

ARM, Keil, and µVision are trademarks or registered of ARM Ltd or its subsidiaries.

LEGO, the LEGO logo, WEDO, and MINDSTORMS are trademarks of the LEGO Group.

TETRIX by Pitsco is a trademark of Pitsco, Inc.

FIELDBUS FOUNDATION™ and FOUNDATION™ are trademarks of the Fieldbus Foundation.

EtherCAT® is a registered trademark of and licensed by Beckhoff Automation GmbH.

CANopen® is a registered Community Trademark of CAN in Automation e.V.

DeviceNet™ and EtherNet/IP™ are trademarks of ODVA.

Go!, SensorDAQ, and Vernier are registered trademarks of Vernier Software & Technology. Vernier Software & Technology and `vernier.com` are trademarks or trade dress.

Xilinx is the registered trademark of Xilinx, Inc.

Taptite and Trilobular are registered trademarks of Research Engineering & Manufacturing Inc.

FireWire® is the registered trademark of Apple Inc.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Handle Graphics®, MATLAB®, Real-Time Workshop®, Simulink®, Stateflow®, and xPC TargetBox® are registered trademarks, and TargetBox™ and Target Language Compiler™ are trademarks of The MathWorks, Inc.

Tektronix®, Tek, and Tektronix, Enabling Technology are registered trademarks of Tektronix, Inc.

The Bluetooth® word mark is a registered trademark owned by the Bluetooth SIG, Inc.

The ExpressCard™ word mark and logos are owned by PCMCIA and any use of such marks by National Instruments is under license.

The mark LabWindows is used under a license from Microsoft Corporation. Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

Patents

For patents covering National Instruments products/technology, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your media, or the *National Instruments Patent Notice* at `ni.com/patents`.

Export Compliance Information

Refer to the *Export Compliance Information* at `ni.com/legal/export-compliance` for the National Instruments global trade compliance policy and how to obtain relevant HTS codes, ECCNs, and other import/export data.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

YOU ARE ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY AND RELIABILITY OF THE PRODUCTS WHENEVER THE PRODUCTS ARE INCORPORATED IN YOUR SYSTEM OR APPLICATION, INCLUDING THE APPROPRIATE DESIGN, PROCESS, AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

PRODUCTS ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING IN THE OPERATION OF NUCLEAR FACILITIES; AIRCRAFT NAVIGATION; AIR TRAFFIC CONTROL SYSTEMS; LIFE SAVING OR LIFE SUSTAINING SYSTEMS OR SUCH OTHER MEDICAL DEVICES; OR ANY OTHER APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, “HIGH-RISK USES”). FURTHER, PRUDENT STEPS MUST BE TAKEN TO PROTECT AGAINST FAILURES, INCLUDING PROVIDING BACK-UP AND SHUT-DOWN MECHANISMS. NI EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES.

Compliance

Electromagnetic Compatibility Information

This hardware has been tested and found to comply with the applicable regulatory requirements and limits for electromagnetic compatibility (EMC) as indicated in the hardware's Declaration of Conformity (DoC)[1]. These requirements and limits are designed to provide reasonable protection against harmful interference when the hardware is operated in the intended electromagnetic environment. In special cases, for example when either highly sensitive or noisy hardware is being used in close proximity, additional mitigation measures may have to be employed to minimize the potential for electromagnetic interference.

While this hardware is compliant with the applicable regulatory EMC requirements, there is no guarantee that interference will not occur in a particular installation. To minimize the potential for the hardware to cause interference to radio and television reception or to experience unacceptable performance degradation, install and use this hardware in strict accordance with the instructions in the hardware documentation and the DoC[1].

If this hardware does cause interference with licensed radio communications services or other nearby electronics, which can be determined by turning the hardware off and on, you are encouraged to try to correct the interference by one or more of the following measures:

- Reorient the antenna of the receiver (the device suffering interference).
- Relocate the transmitter (the device generating interference) with respect to the receiver.
- Plug the transmitter into a different outlet so that the transmitter and the receiver are on different branch circuits.

Some hardware may require the use of a metal, shielded enclosure (windowless version) to meet the EMC requirements for special EMC environments such as, for marine use or in heavy industrial areas. Refer to the hardware's user documentation and the DoC[1] for product installation requirements.

When the hardware is connected to a test object or to test leads, the system may become more sensitive to disturbances or may cause interference in the local electromagnetic environment.

Operation of this hardware in a residential area is likely to cause harmful interference. Users are required to correct the interference at their own expense or cease operation of the hardware.

Changes or modifications not expressly approved by National Instruments could void the user's right to operate the hardware under the local regulatory rules.

[1] The Declaration of Conformity (DoC) contains important EMC compliance information and instructions for the user or installer. To obtain the DoC for this product, visit `ni.com/certification`, search by model number or product line, and click the appropriate link in the Certification column.

Contents

About This Manual

Related Documentation xxxiii

Chapter 1
Introduction

Chapter 2
Installation and Configuration

Safety Information 2-1
Measurement & Automation Explorer (MAX) 2-3
Verifying NI-XNET Hardware Installation 2-4
XNET C Series Modules Firmware Update 2-5
Configuring NI-XNET Interfaces 2-7
LabVIEW Real-Time (RT) Configuration 2-7
Getting Started with CompactRIO 2-8
Tools 2-12
System Configuration API 2-13

Chapter 3
NI-XNET Hardware Overview

Overview 3-1
NI-XNET FlexRay Hardware 3-1
- FlexRay Physical Layer 3-1
  - Transceiver 3-1
  - Bus Power Requirements 3-1
  - Cabling Requirements for FlexRay 3-1
  - Cable Lengths and Number of Devices 3-2
  - Termination 3-2
- Pinout 3-2

NI-XNET CAN Hardware 3-3
- NI-XNET Transceiver Cables 3-3
- XS Software Selectable Physical Layer 3-3
- High-Speed Physical Layer 3-4
  - Transceiver 3-4
  - Bus Power Requirements 3-4
  - Cabling Requirements for High-Speed CAN 3-5
  - Cable Lengths 3-5

Number of Devices ... 3-5
Cable Termination ... 3-5
Cabling Example ... 3-7
Low-Speed/Fault-Tolerant Physical Layer ... 3-7
Transceiver ... 3-7
Bus Power Requirements ... 3-8
Cabling Requirements for Low-Speed/ Fault-Tolerant CAN ... 3-8
Number of Devices ... 3-9
Termination ... 3-9
Determining the Necessary Termination Resistance for the Board ... 3-10
Single Wire CAN Physical Layer ... 3-11
Transceiver ... 3-11
Bus Power Requirements ... 3-12
Cabling Requirements for Single Wire CAN ... 3-12
Cable Length ... 3-12
Number of Devices ... 3-12
Termination (Bus Loading) ... 3-12
External CAN Transceiver ... 3-12
Pinouts ... 3-13
PXI-8511/8512/8513 and PCI-8511/8512/8513 ... 3-13
C Series NI 9861/9862 ... 3-14
NI-XNET LIN Hardware ... 3-14
LIN Physical Layer ... 3-14
Transceiver ... 3-15
Bus Power Requirements ... 3-15
Cabling Requirements for LIN ... 3-15
Cable Lengths ... 3-15
Number of Devices ... 3-16
Termination ... 3-16
Pinout ... 3-16
PXI-8516 and PCI-8516 ... 3-16
C Series NI 9866 and NI-XNET LIN Transceiver Cable ... 3-17
Isolation ... 3-17
LEDs ... 3-18
Synchronization ... 3-20
PXI NI-XNET and PCI NI-XNET ... 3-20
C Series and NI-XNET Transceiver Cables ... 3-20

Chapter 4
NI-XNET API for LabVIEW

Getting Started 4-1
- LabVIEW Project 4-1
- Examples 4-1
- Palettes 4-2

Basic Programming Model 4-3

Interfaces 4-4
- What Is an Interface? 4-4
- How Do I View Available Interfaces? 4-5
  - Measurement and Automation Explorer (MAX) 4-5
  - I/O Name 4-6
  - LabVIEW Project 4-6
  - System Node 4-6

Databases 4-7
- What Is a Database? 4-7
- What Is an Alias? 4-8
- Database Programming 4-9
  - Already Have File? 4-9
  - Can Use File As Is? 4-9
  - Select From File 4-10
  - Edit and Select 4-11
  - Want to Use a File? 4-12
  - Create New File Using the Database Editor 4-12
  - Create in Memory 4-12
  - Multiple Databases Simultaneously 4-13

Sessions 4-13
- What Is a Session? 4-13
- Session Modes 4-14
  - Frame Input Queued Mode 4-15
  - Frame Input Single-Point Mode 4-18
  - Frame Input Stream Mode 4-19
  - Frame Output Queued Mode 4-22
  - Frame Output Single-Point Mode 4-24
  - Frame Output Stream Mode 4-27
  - Signal Input Single-Point Mode 4-29
  - Signal Input Waveform Mode 4-32
  - Signal Input XY Mode 4-35
  - Signal Output Single-Point Mode 4-37
  - Signal Output Waveform Mode 4-38
  - Signal Output XY Mode 4-41
  - Conversion Mode 4-45

How Do I Create a Session? 4-47
LabVIEW Project 4-48
XNET Create Session.vi 4-48
Using CAN 4-48
Understanding CAN Frame Timing 4-48
Configuring Frame I/O Stream Sessions 4-49
Using FlexRay 4-50
Starting Communication 4-50
Understanding FlexRay Frame Timing 4-51
Protocol Data Unit (PDU) 4-51
Using LIN 4-51
Changing the LIN Schedule 4-51
Understanding LIN Frame Timing 4-52
LIN Diagnostics 4-52
Special Considerations for Using Stream Output Mode with LIN 4-52
Using LabVIEW Real-Time 4-53
High Priority Loops 4-53
XNET I/O Names 4-54
Deploying Databases 4-54
Memory Use for Databases 4-54
FlexRay Timing Source 4-55
Creating a Built Real-Time Application 4-55
J1939 Sessions 4-55
Compatibility Issue 4-56
J1939 Basics 4-57
Node Addresses in NI-XNET 4-58
Address Claiming Procedure 4-59
Transmitting Frames 4-59
Transmitting Frames without Granted Node Address 4-59
Mixing J1939 and CAN Messages 4-59
Transport Protocol (TP) 4-60
NI-XNET Sessions 4-60
Not Supported in the Current NI-XNET Version 4-60
Signal Ranges 4-60
NI-XNET API for LabVIEW Reference 4-61
XNET Session Constant 4-61
XNET Create Session.vi 4-62
XNET Create Session (Conversion).vi 4-63
XNET Create Session (Frame Input Queued).vi 4-64
XNET Create Session (Frame Input Single-Point).vi 4-65
XNET Create Session (Frame Input Stream).vi 4-66
XNET Create Session (PDU Input Queued).vi 4-68
XNET Create Session (PDU Input Single Point).vi 4-68
XNET Create Session (Frame Output Queued).vi 4-69

XNET Create Session (Frame Output Single-Point).vi 4-70
XNET Create Session (Frame Output Stream).vi 4-71
XNET Create Session (PDU Output Queued).vi 4-73
XNET Create Session (PDU Output Single-Point).vi 4-73
XNET Create Session (Generic).vi 4-74
XNET Create Session (Signal Input Single-Point).vi 4-76
XNET Create Session (Signal Input Waveform).vi 4-77
XNET Create Session (Signal Input XY).vi 4-78
XNET Create Session (Signal Output Single-Point).vi 4-79
XNET Create Session (Signal Output Waveform).vi 4-80
XNET Create Session (Signal Output XY).vi 4-81
XNET Session Property Node 4-82
Interface Properties 4-83
CAN Interface Properties 4-83
Interface:CAN:External Transceiver Config 4-84
Interface:CAN:FD Baud Rate 4-87
Interface:CAN:I/O Mode 4-89
Interface:CAN:Listen Only? 4-90
Interface:CAN:Pending Transmit Order 4-91
Interface:CAN:Single Shot Transmit? 4-93
Interface:CAN:Termination 4-94
Interface:CAN:Transceiver State 4-96
Interface:CAN:Transceiver Type 4-99
Interface:CAN:Transmit I/O Mode 4-101
FlexRay Interface Properties 4-102
Interface:FlexRay:Accepted Startup Range 4-102
Interface:FlexRay:Allow Halt Due To Clock? 4-103
Interface:FlexRay:Allow Passive to Active 4-104
Interface:FlexRay:Auto Asleep When Stopped 4-105
Interface:FlexRay:Cluster Drift Damping 4-106
Interface:FlexRay:Coldstart? 4-107
Interface:FlexRay:Connected Channels 4-108
Interface:FlexRay:Decoding Correction 4-109
Interface:FlexRay:Delay Compensation Ch A 4-110
Interface:FlexRay:Delay Compensation Ch B 4-111
Interface:FlexRay:Key Slot Identifier 4-112
Interface:FlexRay:Latest Tx 4-114
Interface:FlexRay:Listen Timeout 4-115
Interface:FlexRay:Macro Initial Offset Ch A 4-116
Interface:FlexRay:Macro Initial Offset Ch B 4-117
Interface:FlexRay:Max Drift 4-118
Interface:FlexRay:Micro Initial Offset Ch A 4-119
Interface:FlexRay:Micro Initial Offset Ch B 4-120
Interface:FlexRay:Microtick 4-121

Interface:FlexRay:Null Frames To Input Stream? 4-122
Interface:FlexRay:Offset Correction 4-123
Interface:FlexRay:Offset Correction Out 4-124
Interface:FlexRay:Rate Correction 4-125
Interface:FlexRay:Rate Correction Out 4-126
Interface:FlexRay:Samples Per Microtick 4-127
Interface:FlexRay:Single Slot Enabled? 4-128
Interface:FlexRay:Sleep 4-129
Interface:FlexRay:Statistics Enabled? 4-131
Interface:FlexRay:Symbol Frames To Input Stream? 4-132
Interface:FlexRay:Sync Frames Channel A Even 4-133
Interface:FlexRay:Sync Frames Channel A Odd 4-134
Interface:FlexRay:Sync Frames Channel B Even 4-135
Interface:FlexRay:Sync Frames Channel B Odd 4-136
Interface:FlexRay:Sync Frame Status 4-137
Interface:FlexRay:Termination 4-138
Interface:FlexRay:Wakeup Channel 4-139
Interface:FlexRay:Wakeup Pattern 4-140
LIN Interface Properties 4-141
Interface:LIN:Break Length 4-141
Interface:LIN:DiagP2min 4-142
Interface:LIN:DiagSTmin 4-143
Interface:LIN:Master? 4-144
Interface:LIN:Output Stream Slave Response List By NAD 4-145
Interface:LIN:Schedules 4-146
Interface:LIN:Sleep 4-147
Interface:LIN:Start Allowed without Bus Power? 4-150
Interface:LIN:Termination 4-151
Source Terminal Interface Properties 4-152
Interface:Source Terminal:Start Trigger 4-152
Interface:Baud Rate 4-153
Interface:Echo Transmit? 4-156
Interface:I/O Name 4-157
Interface:Output Stream List 4-158
Interface:Output Stream List By ID 4-159
Interface:Output Stream Timing 4-160

Interface:Start Trigger Frames to Input Stream? 4-164
Interface:Bus Error Frames to Input Stream? 4-164
Session:Application Protocol 4-165
SAE J1939:ECU 4-166
SAE J1939:ECU Busy 4-167
SAE J1939:Hold Time Th 4-168
SAE J1939:Maximum Repeat CTS 4-169
SAE J1939:Node Address 4-170
SAE J1939:NodeName 4-171
SAE J1939:Number of Packets Received 4-172
SAE J1939:Number of Packets Response 4-173
SAE J1939:Response Time Tr_GD 4-174
SAE J1939:Response Time Tr_SD 4-175
SAE J1939:Timeout T1 4-176
SAE J1939:Timeout T2 4-177
SAE J1939:Timeout T3 4-178
SAE J1939:Timeout T4 4-179
Frame Properties 4-180
CAN Frame Properties 4-180
Frame:CAN:Start Time Offset 4-180
Frame:CAN:Transmit Time 4-181
Frame:Active 4-182
Frame:LIN:Transmit N Corrupted Checksums 4-183
Frame:Skip N Cyclic Frames 4-184
Auto Start? 4-185
Cluster 4-186
Database 4-187
List of Frames 4-188
List of Signals 4-189
Mode 4-190
Number in List 4-190
Number of Values Pending 4-191
Number of Values Unused 4-192
Payload Length Maximum 4-193
Protocol 4-194
Queue Size 4-195
Resample Rate 4-201
XNET Read.vi 4-202
XNET Read (Frame CAN).vi 4-204
XNET Read (Frame FlexRay).vi 4-208
XNET Read (Frame LIN).vi 4-213
XNET Read (Frame Raw).vi 4-218
XNET Read (Signal Single-Point).vi 4-221
XNET Read (Signal Waveform).vi 4-222

XNET Read (Signal XY).vi ... 4-224
XNET Read (State CAN Comm).vi ... 4-227
XNET Read (State FlexRay Comm).vi ... 4-231
XNET Read (State LIN Comm).vi ... 4-235
XNET Read (State FlexRay Cycle Macrotick).vi ... 4-240
XNET Read (State FlexRay Statistics).vi ... 4-242
XNET Read (State Time Comm).vi ... 4-244
XNET Read (State Time Current).vi ... 4-245
XNET Read (State Time Start).vi ... 4-246
XNET Read (State Session Info).vi ... 4-248
XNET Write.vi ... 4-249
XNET Write (Signal Single-Point).vi ... 4-251
XNET Write (Signal Waveform).vi ... 4-252
XNET Write (Signal XY).vi ... 4-254
XNET Write (Frame CAN).vi ... 4-256
XNET Write (Frame FlexRay).vi ... 4-260
XNET Write (Frame LIN).vi ... 4-264
XNET Write (Frame Raw).vi ... 4-268
XNET Write (State FlexRay Symbol).vi ... 4-271
XNET Write (State LIN Schedule Change).vi ... 4-272
XNET Write (State LIN Diagnostic Schedule Change).vi ... 4-275
Database Subpalette ... 4-278
XNET Database Property Node ... 4-278
Clusters ... 4-279
ShowInvalidFromOpen? ... 4-280
XNET Database Constant ... 4-281
XNET Cluster Property Node ... 4-281
FlexRay Properties ... 4-282
FlexRay:Action Point Offset ... 4-282
FlexRay:CAS Rx Low Max ... 4-283
FlexRay:Channels ... 4-284
FlexRay:Cluster Drift Damping ... 4-285
FlexRay:Cold Start Attempts ... 4-286
FlexRay:Cycle ... 4-287
FlexRay:Dynamic Segment Start ... 4-288
FlexRay:Dynamic Slot Idle Phase ... 4-289
FlexRay:Latest Guaranteed Dynamic Slot ... 4-290
FlexRay:Latest Usable Dynamic Slot ... 4-291
FlexRay:Listen Noise ... 4-292
FlexRay:Macro Per Cycle ... 4-293
FlexRay:Macrotick ... 4-294
FlexRay:Max Without Clock Correction Fatal ... 4-295
FlexRay:Max Without Clock Correction Passive ... 4-296

FlexRay:Minislot Action Point Offset 4-297
FlexRay:Minislot 4-298
FlexRay:Network Management Vector Length ... 4-299
FlexRay:NIT Start 4-300
FlexRay:NIT 4-301
FlexRay:Number of Minislots 4-302
FlexRay:Number of Static Slots 4-303
FlexRay:Offset Correction Start 4-304
FlexRay:Payload Length Dynamic Maximum 4-305
FlexRay:Payload Length Maximum 4-306
FlexRay:Payload Length Static 4-307
FlexRay:Static Slot 4-308
FlexRay:Symbol Window Start 4-309
FlexRay:Symbol Window 4-310
FlexRay:Sync Node Max 4-311
FlexRay:TSS Transmitter 4-312
FlexRay:Use Wakeup 4-313
FlexRay:Wakeup Symbol Rx Idle 4-314
FlexRay:Wakeup Symbol Rx Low 4-315
FlexRay:Wakeup Symbol Rx Window 4-316
FlexRay:Wakeup Symbol Tx Idle 4-317
FlexRay:Wakeup Symbol Tx Low 4-318
Application Protocol 4-319
Baud Rate 4-320
CAN:FD Baud Rate 4-321
CAN:I/O Mode 4-322
Comment 4-323
Configuration Status 4-323
Database 4-324
ECUs 4-324
Frames 4-325
LIN:Schedules 4-326
LIN:Tick 4-327
Name (Short) 4-328
PDUs 4-330
PDUs Required? 4-331
Protocol 4-333
Signals 4-333
XNET Cluster Constant 4-334
XNET ECU Property Node 4-334
Cluster 4-335
FlexRay:Coldstart? 4-335
FlexRay:Connected Channels 4-336
FlexRay:Startup Frame 4-336

FlexRay:Wakeup Channels ... 4-337
FlexRay:Wakeup Pattern ... 4-337
Comment ... 4-339
Configuration Status ... 4-339
Frames Received ... 4-340
Frames Transmitted ... 4-340
LIN:Master? ... 4-341
LIN:Protocol Version ... 4-341
LIN:Initial NAD ... 4-342
LIN:Configured NAD ... 4-342
LIN:Supplier ID ... 4-343
LIN:Function ID ... 4-343
LIN:P2min ... 4-344
LIN:STmin ... 4-344
Name (Short) ... 4-345
XNET ECU Constant ... 4-347
XNET Frame Property Node ... 4-347
CAN:Extended Identifier? ... 4-347
CAN:Timing Type ... 4-348
CAN:Transmit Time ... 4-350
Application Protocol ... 4-351
Cluster ... 4-352
Comment ... 4-352
Configuration Status ... 4-353
Default Payload ... 4-354
FlexRay:Base Cycle ... 4-356
FlexRay:Channel Assignment ... 4-358
FlexRay:Cycle Repetition ... 4-359
FlexRay:Payload Preamble? ... 4-361
FlexRay:Startup? ... 4-362
FlexRay:Sync? ... 4-363
FlexRay:Timing Type ... 4-364
FlexRay:In Cycle Repetitions:Channel Assignments ... 4-365
FlexRay:In Cycle Repetitions:Enabled? ... 4-366
FlexRay:In Cycle Repetitions:Identifiers ... 4-367
Identifier ... 4-368
LIN:Checksum ... 4-370
Mux:Data Multiplexer Signal ... 4-371
Mux:Is Data Multiplexed? ... 4-371
Mux:Static Signals ... 4-372
Mux:Subframes ... 4-372
Name (Short) ... 4-373
Payload Length ... 4-375

PDU_Mapping....4-376
Signals....4-377
XNET Frame Constant....4-378
XNET PDU Property Node....4-378
Cluster....4-379
Comment....4-379
Configuration Status....4-380
Frames....4-381
Mux:Data Multiplexer Signal....4-381
Mux:Is Data Multiplexed?....4-382
Mux:Static Signals....4-382
Mux:Subframes....4-383
Name (Short)....4-384
Payload Length....4-385
Signals....4-386
XNET PDU Constant....4-386
XNET Subframe Property Node....4-387
Dynamic Signals....4-388
Frame....4-388
Multiplexer Value....4-389
Name (Short)....4-390
PDU....4-392
XNET Signal Property Node....4-393
Byte Order....4-394
Comment....4-396
Configuration Status....4-397
Data Type....4-398
Default Value....4-399
Mux:Dynamic?....4-400
Frame....4-401
Maximum Value....4-401
Minimum Value....4-402
Mux:Multiplexer Value....4-402
Mux:Data Multiplexer?....4-403
Name (Short)....4-404
Number of Bits....4-406
PDU....4-407
Scaling Factor....4-408
Scaling Offset....4-408
Start Bit....4-409
Mux:Subframe....4-411
Unit....4-411
XNET Signal Constant....4-412
XNET Database Open.vi....4-412

XNET Database Close.vi ... 4-413
XNET Database Close (Cluster).vi ... 4-414
XNET Database Close (Database).vi ... 4-415
XNET Database Close (ECU).vi ... 4-416
XNET Database Close (Frame).vi ... 4-417
XNET Database Close (PDU).vi ... 4-418
XNET Database Close (Signal).vi ... 4-419
XNET Database Close (Subframe).vi ... 4-420
XNET Database Close (LIN Schedule).vi ... 4-421
XNET Database Close (LIN Schedule Entry).vi ... 4-422
XNET Database Create Object.vi ... 4-423
XNET Database Create (Cluster).vi ... 4-424
XNET Database Create (Dynamic Signal).vi ... 4-426
XNET Database Create (ECU).vi ... 4-428
XNET Database Create (Frame).vi ... 4-429
XNET Database Create (PDU).vi ... 4-430
XNET Database Create (Signal).vi ... 4-431
XNET Database Create (Subframe).vi ... 4-432
XNET Database Create (LIN Schedule).vi ... 4-434
XNET Database Create (LIN Schedule Entry).vi ... 4-435
XNET Database Delete Object.vi ... 4-437
XNET Database Delete (Cluster).vi ... 4-438
XNET Database Delete (ECU).vi ... 4-439
XNET Database Delete (Frame).vi ... 4-440
XNET Database Delete (PDU).vi ... 4-441
XNET Database Delete (Signal).vi ... 4-442
XNET Database Delete (Subframe).vi ... 4-443
XNET Database Delete (LIN Schedule).vi ... 4-444
XNET Database Delete (LIN Schedule Entry).vi ... 4-445
XNET Database Merge.vi ... 4-446
XNET Database Merge (Frame).vi ... 4-447
XNET Database Merge (PDU).vi ... 4-449
XNET Database Merge (ECU).vi ... 4-451
XNET Database Merge (LIN Schedule).vi ... 4-453
XNET Database Merge (Cluster).vi ... 4-455
XNET Database Save.vi ... 4-457
XNET Database Export.vi ... 4-458
File Management Subpalette ... 4-459
XNET Database Add Alias.vi ... 4-459
XNET Database Remove Alias.vi ... 4-461
XNET Database Get List.vi ... 4-462
XNET Database Deploy.vi ... 4-464
XNET Database Undeploy.vi ... 4-466

XNET LIN Schedule Property Node 4-467
Cluster 4-467
Comment 4-468
Configuration Status 4-469
Entries 4-470
Name (Short) 4-471
Priority 4-472
Run Mode 4-473
XNET LIN Schedule Entry Property Node 4-474
Collision Resolving Schedule 4-475
Delay 4-476
Event Identifier 4-476
Frames 4-477
Name (Short) 4-478
Node Configuration:Free Format:Data Bytes 4-479
Schedule 4-480
Type 4-481
XNET Database Get DBC Attribute.vi 4-482
Notify Subpalette 4-484
XNET Wait.vi 4-484
XNET Wait (Transmit Complete).vi 4-485
XNET Wait (Interface Communicating).vi 4-486
XNET Wait (CAN Remote Wakeup).vi 4-488
XNET Wait (LIN Remote Wakeup).vi 4-489
XNET Create Timing Source.vi 4-490
XNET Create Timing Source (FlexRay Cycle).vi 4-490
Advanced Subpalette 4-499
XNET Start.vi 4-499
XNET Stop.vi 4-502
XNET Clear.vi 4-504
XNET Flush.vi 4-505
XNET Connect Terminals.vi 4-506
XNET Disconnect Terminals.vi 4-513
XNET Terminal Constant 4-514
XNET System Property Node 4-514
Devices 4-515
Interfaces (FlexRay) 4-515
Interfaces (All) 4-516
Interfaces (CAN) 4-516
Interfaces (LIN) 4-517
Version:Build 4-518
Version:Major 4-519
Version:Minor 4-520

Version:Phase 4-521
Version:Update 4-522
XNET Device Property Node 4-523
Form Factor 4-523
Interfaces 4-524
Number of Ports 4-525
Product Name 4-525
Product Number 4-526
Serial Number 4-526
Slot Number 4-527
XNET Interface Property Node 4-527
CAN.Termination Capability 4-528
CAN.Transceiver Capability 4-529
Device 4-530
Name 4-530
Number 4-531
Port Number 4-532
Protocol 4-533
XNET Interface Constant 4-534
XNET Blink.vi 4-534
XNET System Close.vi 4-536
XNET String to IO Name.vi 4-537
XNET Convert.vi 4-538
XNET Convert (Frame CAN to Signal).vi 4-539
XNET Convert (Frame FlexRay to Signal).vi 4-542
XNET Convert (Frame LIN to Signal).vi 4-545
XNET Convert (Frame Raw to Signal).vi 4-547
XNET Convert (Signal to Frame CAN).vi 4-549
XNET Convert (Signal to Frame FlexRay).vi 4-551
XNET Convert (Signal to Frame LIN).vi 4-554
XNET Convert (Signal to Frame Raw).vi 4-556
Controls Palette 4-558
XNET Session Control 4-558
Database Controls 4-558
System Controls 4-559
Additional Topics 4-560
Overall 4-560
Creating a Built Application 4-560
Cyclic and Event Timing 4-561
Error Handling 4-562
Fault Handling 4-563
Multiplexed Signals 4-565
Raw Frame Format 4-567
Special Frames 4-572

Required Properties....4-577
State Models....4-579
TDMS....4-587
CAN....4-592
NI-CAN....4-592
CAN Timing Type and Session Mode....4-594
CAN Transceiver State Machine....4-598
FlexRay....4-600
FlexRay Timing Type and Session Mode....4-600
Protocol Data Units (PDUs) in NI-XNET....4-603
FlexRay Startup/Wakeup....4-606
LIN....4-609
LIN Frame Timing and Session Mode....4-609
XNET I/O Names....4-613
I/O Name Classes....4-614
XNET Cluster I/O Name....4-615
XNET Database I/O Name....4-618
XNET Device I/O Name....4-621
XNET ECU I/O Name....4-621
XNET Frame I/O Name....4-624
XNET Interface I/O Name....4-627
XNET Session I/O Name....4-628
XNET Signal I/O Name....4-630
XNET Subframe I/O Name....4-633
XNET Terminal I/O Name....4-634
XNET LIN Schedule I/O Name....4-635
XNET LIN Schedule Entry I/O Name....4-637
XNET PDU I/O Name....4-638

Chapter 5
NI-XNET API for C

Getting Started....5-1
LabWindows/CVI....5-1
Examples....5-1
Visual C++....5-2
Examples....5-3
Interfaces....5-3
What Is an Interface?....5-3
How Do I View Available Interfaces?....5-4
Measurement and Automation Explorer (MAX)....5-4
Databases....5-4
What Is a Database?....5-4
What Is an Alias?....5-5

Database Programming 5-6
Already Have File? 5-6
Can I Use File as Is? 5-6
Select From File 5-7
Edit and Select 5-7
Want to Use a File? 5-7
Create New File Using the Database Editor 5-7
Create in Memory 5-7
Sessions 5-8
What Is a Session? 5-8
Session Modes 5-9
Frame Input Queued Mode 5-10
Frame Input Single-Point Mode 5-12
Frame Input Stream Mode 5-13
Frame Output Queued Mode 5-16
Frame Output Single-Point Mode 5-18
Frame Output Stream Mode 5-21
Signal Input Single-Point Mode 5-24
Signal Input Waveform Mode 5-26
Signal Input XY Mode 5-28
Signal Output Single-Point Mode 5-30
Signal Output Waveform Mode 5-31
Signal Output XY Mode 5-34
Conversion Mode 5-38
J1939 Sessions 5-41
Compatibility Issue 5-41
J1939 Basics 5-42
Node Addresses in NI-XNET 5-43
Address Claiming Procedure 5-44
Transmitting Frames 5-45
Transmitting Frames without Granted Node Address 5-45
Mixing J1939 and CAN Messages 5-45
Transport Protocol (TP) 5-45
NI-XNET Sessions 5-46
Not Supported in the Current NI-XNET Version 5-46
Signal Ranges 5-46
NI-XNET API for C Reference 5-47
Functions 5-47
nxBlink 5-47
nxClear 5-49
nxConnectTerminals 5-50
nxConvertFramesToSignalsSinglePoint 5-57
nxConvertSignalsToFramesSinglePoint 5-59
nxCreateSession 5-61

nxCreateSessionByRef....5-66
nxdbAddAlias....5-68
nxdbCloseDatabase....5-70
nxdbCreateObject....5-71
nxdbDeleteObject....5-73
nxdbDeploy....5-74
nxdbFindObject....5-76
nxdbGetDatabaseList....5-78
nxdbGetDatabaseListSizes....5-80
nxdbGetDBCAttribute....5-82
nxdbGetDBCAttributeSize....5-84
nxdbGetProperty....5-85
nxdbGetPropertySize....5-86
nxdbMerge....5-87
nxdbOpenDatabase....5-90
nxdbRemoveAlias....5-91
nxdbSaveDatabase....5-92
nxdbSetProperty....5-94
nxdbUndeploy....5-95
nxDisconnectTerminals....5-96
nxFlush....5-98
nxGetProperty....5-99
nxGetPropertySize....5-101
nxGetSubProperty....5-102
nxGetSubPropertySize....5-103
nxReadFrame....5-104
nxReadSignalSinglePoint....5-107
nxReadSignalWaveform....5-109
nxReadSignalXY....5-111
nxReadState....5-113
nxSetProperty....5-125
nxSetSubProperty....5-126
nxStart....5-127
nxStatusToString....5-129
nxStop....5-130
nxSystemClose....5-132
nxSystemOpen....5-133
nxWait....5-134
nxWriteFrame....5-136
nxWriteSignalSinglePoint....5-139
nxWriteSignalWaveform....5-140
nxWriteSignalXY....5-142
nxWriteState....5-144

Properties 5-147
XNET Cluster Properties 5-147
Baud Rate 5-147
CAN:FD Baud Rate 5-148
CAN:I/O Mode 5-149
Comment 5-150
Configuration Status 5-150
Database 5-151
ECUs 5-151
FlexRay:Action Point Offset 5-152
FlexRay:CAS Rx Low Max 5-153
FlexRay:Channels 5-154
FlexRay:Cluster Drift Damping 5-155
FlexRay:Cold Start Attempts 5-156
FlexRay:Cycle 5-157
FlexRay:Dynamic Segment Start 5-158
FlexRay:Dynamic Slot Idle Phase 5-159
FlexRay:Latest Guaranteed Dynamic Slot 5-160
FlexRay:Latest Usable Dynamic Slot 5-161
FlexRay:Listen Noise 5-162
FlexRay:Macro Per Cycle 5-163
FlexRay:Macrotick 5-164
FlexRay:Max Without Clock Correction Fatal 5-165
FlexRay:Max Without Clock Correction Passive 5-166
FlexRay:Minislot 5-167
FlexRay:Minislot Action Point Offset 5-168
FlexRay:Network Management Vector Length 5-169
FlexRay:NIT 5-170
FlexRay:NIT Start 5-171
FlexRay:Number of Minislots 5-172
FlexRay:Number of Static Slots 5-173
FlexRay:Offset Correction Start 5-174
FlexRay:Payload Length Dynamic Maximum 5-175
FlexRay:Payload Length Maximum 5-176
FlexRay:Payload Length Static 5-177
FlexRay:Static Slot 5-178
FlexRay:Symbol Window 5-179
FlexRay:Symbol Window Start 5-180
FlexRay:Sync Node Max 5-181
FlexRay:TSS Transmitter 5-182
FlexRay:Use Wakeup 5-183
FlexRay:Wakeup Symbol Rx Idle 5-184
FlexRay:Wakeup Symbol Rx Low 5-185
FlexRay:Wakeup Symbol Rx Window 5-186

FlexRay:Wakeup Symbol Tx Idle 5-187
FlexRay:Wakeup Symbol Tx Low 5-188
Frames 5-189
Name (Short) 5-190
PDUs 5-191
PDUs Required? 5-192
Protocol 5-194
Schedules 5-194
Signals 5-195
Tick 5-196
Application Protocol 5-197
XNET Database Properties 5-198
Clusters 5-198
ShowInvalidFromOpen? 5-199
XNET Device Properties 5-200
Form Factor 5-200
Interfaces 5-201
Number of Ports 5-201
Product Name 5-202
Product Number 5-202
Serial Number 5-203
Slot Number 5-203
XNET ECU Properties 5-204
Cluster 5-204
Comment 5-204
Configuration Status 5-205
FlexRay:Coldstart? 5-206
FlexRay:Connected Channels 5-206
FlexRay:Startup Frame 5-207
FlexRay:Wakeup Channels 5-207
FlexRay:Wakeup Pattern 5-208
Frames Received 5-208
Frames Transmitted 5-209
LIN Master 5-209
LIN Version 5-210
LIN:Initial NAD 5-210
LIN:Configured NAD 5-211
LIN:Supplier ID 5-211
LIN:Function ID 5-212
LIN:P2min 5-212
LIN:STmin 5-213
Name (Short) 5-214

XNET Frame Properties 5-215
CAN:Extended Identifier? 5-215
CAN:Timing Type 5-216
CAN:Transmit Time 5-218
Cluster 5-219
Comment 5-219
Configuration Status 5-220
Default Payload 5-221
FlexRay:Base Cycle 5-223
FlexRay:Channel Assignment 5-225
FlexRay:Cycle Repetition 5-226
FlexRay:In Cycle Repetitions:Channel Assignments 5-228
FlexRay:In Cycle Repetitions:Enabled? 5-229
FlexRay:In Cycle Repetitions:Identifiers 5-230
FlexRay:Payload Preamble? 5-231
FlexRay:Startup? 5-232
FlexRay:Sync? 5-233
FlexRay:Timing Type 5-234
Identifier 5-235
LIN:Checksum 5-237
Mux:Data Multiplexer Signal 5-238
Mux:Is Data Multiplexed? 5-238
Mux:Static Signals 5-239
Mux:Subframes 5-239
Name (Short) 5-240
Payload Length 5-241
PDU References 5-242
PDU Start Bits 5-243
PDU Update Bits 5-244
Signals 5-245
Application Protocol 5-246
XNET Interface Properties 5-247
CAN.Termination Capability 5-247
CAN.Transceiver Capability 5-248
Device 5-249
Name 5-249
Number 5-250
Port Number 5-251
Protocol 5-252
XNET LIN Schedule Properties 5-253
Cluster 5-253
Comment 5-253
Configuration Status 5-254
Entries 5-255

Name....5-255
Priority....5-256
Run Mode....5-257
XNET LIN Schedule Entry Properties....5-258
Collision Resolving Schedule....5-258
Delay....5-258
Event Identifier....5-259
Frames....5-260
Name....5-261
Name Unique to Cluster....5-262
Node Configuration:Free Format:Data Bytes....5-263
Schedule....5-264
Type....5-265
XNET PDU Properties....5-266
Cluster....5-266
Comment....5-266
Configuration Status....5-267
Frames....5-268
Mux:Data Multiplexer Signal....5-268
Mux:Is Data Multiplexed?....5-269
Mux:Static Signals....5-269
Mux:Subframes....5-270
Name (Short)....5-270
Payload Length....5-271
Signals....5-272
XNET Session Properties....5-273
Interface Properties....5-273
CAN Interface Properties....5-273
Interface:CAN:External Transceiver Config....5-273
Interface:CAN:FD Baud Rate....5-276
Interface:CAN:I/O Mode....5-278
Interface:CAN:Listen Only?....5-279
Interface:CAN:Pending Transmit Order....5-280
Interface:CAN:Single Shot Transmit?....5-282
Interface:CAN:Termination....5-283
Interface:CAN:Transceiver State....5-285
Interface:CAN:Transceiver Type....5-288
Interface:CAN:Transmit I/O Mode....5-290
FlexRay Interface Properties....5-291
Interface:FlexRay:Accepted Startup Range....5-291

Interface:FlexRay:Allow Halt Due To Clock? 5-292
Interface:FlexRay:Allow Passive to Active 5-293
Interface:FlexRay: AutoAsleepWhenStopped 5-294
Interface:FlexRay:Cluster Drift Damping 5-295
Interface:FlexRay:Coldstart? 5-296
Interface:FlexRay:Connected Channels 5-297
Interface:FlexRay:Decoding Correction 5-298
Interface:FlexRay:Delay Compensation Ch A 5-299
Interface:FlexRay:Delay Compensation Ch B 5-300
Interface:FlexRay:Key Slot Identifier 5-301
Interface:FlexRay:Latest Tx 5-303
Interface:FlexRay:Listen Timeout 5-304
Interface:FlexRay:Macro Initial Offset Ch A 5-305
Interface:FlexRay:Macro Initial Offset Ch B 5-306
Interface:FlexRay:Max Drift 5-307
Interface:FlexRay:Micro Initial Offset Ch A 5-308
Interface:FlexRay:Micro Initial Offset Ch B 5-309
Interface:FlexRay:Microtick 5-310
Interface:FlexRay:Null Frames To Input Stream? 5-311
Interface:FlexRay:Offset Correction 5-312
Interface:FlexRay:Offset Correction Out 5-313
Interface:FlexRay:Rate Correction 5-314
Interface:FlexRay:Rate Correction Out 5-315
Interface:FlexRay:Samples Per Microtick 5-316
Interface:FlexRay:Single Slot Enabled? 5-317
Interface:FlexRay:Sleep 5-318
Interface:FlexRay:Statistics Enabled? 5-320

Interface:FlexRay:Symbol Frames To |Input Stream? ... 5-321
Interface:FlexRay:Sync Frame Status ... 5-322
Interface:FlexRay:Sync Frames Channel A Even ... 5-323
Interface:FlexRay:Sync Frames Channel A Odd ... 5-324
Interface:FlexRay:Sync Frames Channel B Even ... 5-325
Interface:FlexRay:Sync Frames Channel B Odd ... 5-326
Interface:FlexRay:Termination ... 5-327
Interface:FlexRay:Wakeup Channel ... 5-328
Interface:FlexRay:Wakeup Pattern ... 5-329
LIN Interface Properties ... 5-330
Interface:LIN:Break Length ... 5-330
Interface:LIN:DiagP2min ... 5-331
Interface:LIN:DiagSTmin ... 5-332
Interface:LIN:Master? ... 5-333
Interface:LIN:Output Stream Slave Response List By NAD ... 5-334
Interface:LIN:Schedule Names ... 5-335
Interface:LIN:Sleep ... 5-336
Interface:LIN:Start Allowed without Bus Power? ... 5-339
Interface:LIN:Termination ... 5-340
Source Terminal Interface Properties ... 5-341
Interface:Source Terminal:Start Trigger ... 5-341
Interface:Baud Rate ... 5-342
Interface:Echo Transmit? ... 5-345
Interface:Output Stream List ... 5-346
Interface:Output Stream List By ID ... 5-347
Interface:Output Stream Timing ... 5-348
Interface:Start Trigger Frames to Input Stream? ... 5-352
Interface:Bus Error Frames to Input Stream? ... 5-353
Session:Application Protocol ... 5-353
SAE J1939:ECU ... 5-354
SAE J1939:ECU Busy ... 5-355
SAE J1939:Hold Time Th ... 5-356
SAE J1939:Maximum Repeat CTS ... 5-357
SAE J1939:Node Address ... 5-358
SAE J1939:NodeName ... 5-359

SAE J1939:Number of Packets Received 5-360
SAE J1939:Number of Packets Response..... 5-361
SAE J1939:Response Time Tr_GD 5-362
SAE J1939:Response Time Tr_SD 5-363
SAE J1939:Timeout T1..... 5-364
SAE J1939:Timeout T2..... 5-365
SAE J1939:Timeout T3..... 5-366
SAE J1939:Timeout T4..... 5-367
Frame Properties 5-368
CAN Frame Properties..... 5-368
Frame:CAN:Start Time Offset 5-368
Frame:CAN:Transmit Time 5-369
Frame:LIN:Transmit N Corrupted Checksums ... 5-370
Frame:Skip N Cyclic Frames..... 5-371
Auto Start?..... 5-372
ClusterName..... 5-373
DatabaseName..... 5-373
List..... 5-374
Mode..... 5-374
Number in List 5-375
Number of Values Pending 5-375
Number of Values Unused 5-377
Payload Length Maximum 5-378
Protocol 5-378
Queue Size..... 5-380
Resample Rate..... 5-386
XNET Signal Properties 5-387
Byte Order 5-387
Comment 5-389
Configuration Status..... 5-390
Data Type 5-391
Default Value 5-392
Frame..... 5-393
Maximum Value..... 5-393
Minimum Value 5-394
Mux:Data Multiplexer?..... 5-395
Mux:Dynamic?..... 5-396
Mux:Multiplexer Value..... 5-397
Mux:Subframe..... 5-397
Name (Short)..... 5-398
Name Unique to Cluster..... 5-399
Number of Bits..... 5-400
PDU..... 5-401
Scaling Factor..... 5-401

Scaling Offset 5-402
Start Bit 5-403
Unit 5-405
XNET Subframe Properties 5-405
Dynamic Signals 5-405
Frame 5-406
Multiplexer Value 5-407
Name (Short) 5-408
Name Unique to Cluster 5-409
PDU 5-409
XNET System Properties 5-410
Devices 5-410
Interfaces (All) 5-411
Interfaces (CAN) 5-411
Interfaces (FlexRay) 5-412
Interfaces (LIN) 5-412
Version:Build 5-413
Version:Major 5-414
Version:Minor 5-415
Version:Phase 5-416
Version:Update 5-417
Additional Topics 5-418
Overall 5-418
Cyclic and Event Timing 5-418
Multiplexed Signals 5-419
Raw Frame Format 5-421
Special Frames 5-429
Required Properties 5-432
State Models 5-434
CAN 5-441
NI-CAN 5-441
CAN Timing Type and Session Mode 5-443
CAN Transceiver State Machine 5-447
FlexRay 5-449
FlexRay Timing Type and Session Mode 5-449
Protocol Data Units (PDUs) in NI-XNET 5-452
FlexRay Startup/Wakeup 5-455
LIN 5-457
LIN Frame Timing and Session Mode 5-457

Chapter 6
Troubleshooting and Common Questions

Appendix A
Summary of the CAN Standard

Appendix B
Summary of the FlexRay Standard

Appendix C
Summary of the LIN Standard

Appendix D
Specifications

Appendix E
LabVIEW Project Provider

Appendix F
Bus Monitor

Appendix G
Database Editor

Appendix H
NI Services

Index

About This Manual

This manual describes how to install and configure the NI-XNET hardware and software and summarizes the CAN, FlexRay, and LIN standards. It also includes the NI-XNET LabVIEW and C API reference.

Related Documentation

The following documents contain information that you may find helpful as you read this manual:

- *NI-XNET Hardware and Software Help*
- *NI-XNET Tools and Utilities Help*
- *NI-XNET Hardware and Software Installation Guide*

1 Introduction

Welcome to NI-XNET, the National Instruments software for CAN, FlexRay, and LIN products.

NI-XNET is designed to meet the following goals:

- **Ease of use**: NI-XNET features provide fundamental concepts so that you can get started with programming.
- **Consistency**: NI-XNET uses common industry concepts for embedded networks such as CAN. These concepts help to abstract the differences between protocols, so you can focus on your application.
- **Completeness**: NI-XNET provides a broad spectrum of features, from easy-to-use signal I/O, down to more advanced streaming of raw frames. You can use these features simultaneously on the same interface: input along with output and signal I/O along with frame I/O.
- **Performance**: Read and Write functions are designed to execute quickly, without loss of data. Performance for LabVIEW Real-Time (RT) applications is a key focus of NI-XNET software and hardware architecture.

If you are new to the CAN protocol, refer to Appendix A, *Summary of the CAN Standard*, for an introduction. If you are new to the FlexRay protocol, refer to Appendix B, *Summary of the FlexRay Standard*, for an introduction. If you are new to the LIN protocol, refer to Appendix C, *Summary of the LIN Standard*, for an introduction.

Chapter 3, *NI-XNET Hardware Overview*, summarizes the features of National Instruments hardware for CAN, FlexRay, and LIN.

If you use LabVIEW for programming, refer to *Getting Started* in Chapter 4, *NI-XNET API for LabVIEW*, for a description of NI-XNET software concepts and programming models.

If you use C, C++, or another language for programming, refer to *Getting Started* in Chapter 5, *NI-XNET API for C*, for a description of NI-XNET software concepts and programming models.

2

Installation and Configuration

This chapter explains how to install and configure NI-XNET hardware.

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 covers and filler panels installed during operation of the module.

Do *not* operate the module in an explosive atmosphere or where there may be 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 nonconductive 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, nonconductive 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 the module. Do not install wiring while the module is live with electrical signals.

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 are connected 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 a 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.

[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 may be connected to the MAINS for measuring purposes.

- 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 fixed installations, 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 devices and on ripple control units.

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 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 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 MAX Configuration tree **Devices and Interfaces** branch lists NI-XNET hardware (along with other local computer system hardware), as shown in Figure 2-1.

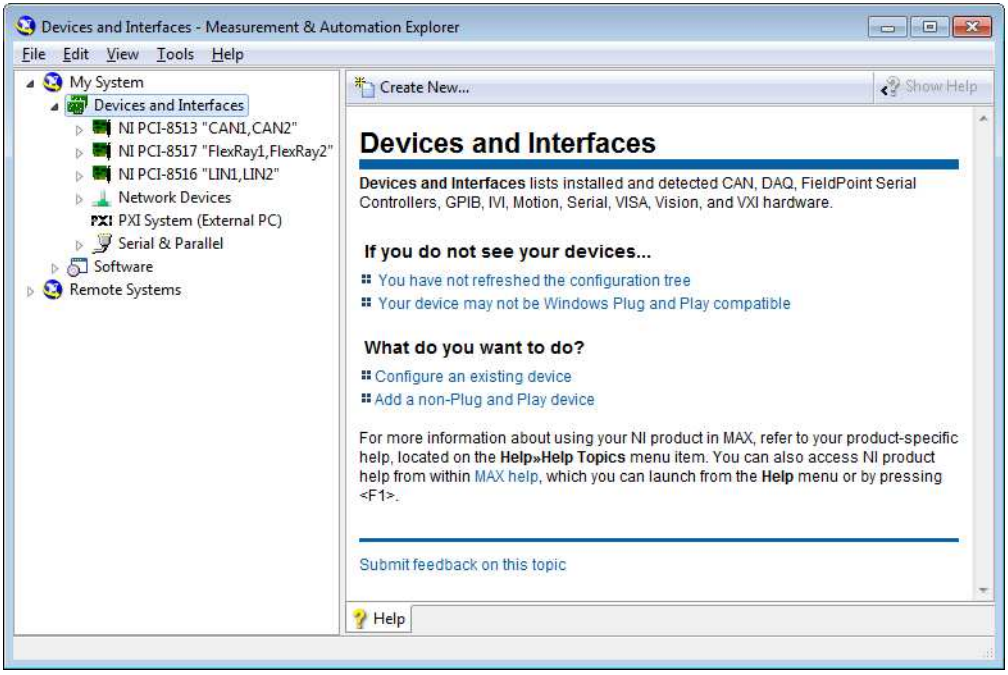


Figure 2-1. NI-XNET Hardware Listed in MAX

If the NI-XNET hardware is not listed here, 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 Chapter 6, *Troubleshooting and Common Questions*, for information about resolving hardware installation problems.

XNET C Series Modules Firmware Update

For C Series modules, the module firmware is not updated automatically when opening an XNET session. Therefore, the right pane in MAX has a second tab that shows the module firmware status.

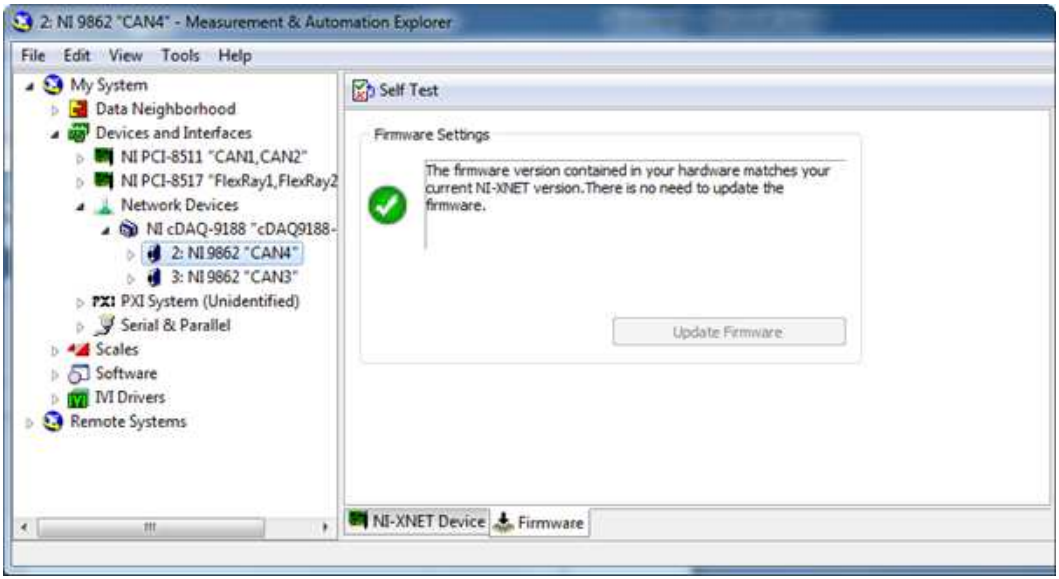


Figure 2-2. Module and XNET Firmware Match

If the module firmware matches the firmware that the current XNET version requires, the right pane in MAX is marked with a check mark, and the **Update Firmware** button is disabled, as shown in Figure 2-2. In case of a version mismatch, the right pane in MAX and the module in the MAX tree view are marked with an exclamation point (!), as shown in Figure 2-3. In this case, the text in the right pane says the firmware must be updated, and the **Update Firmware** button is enabled.

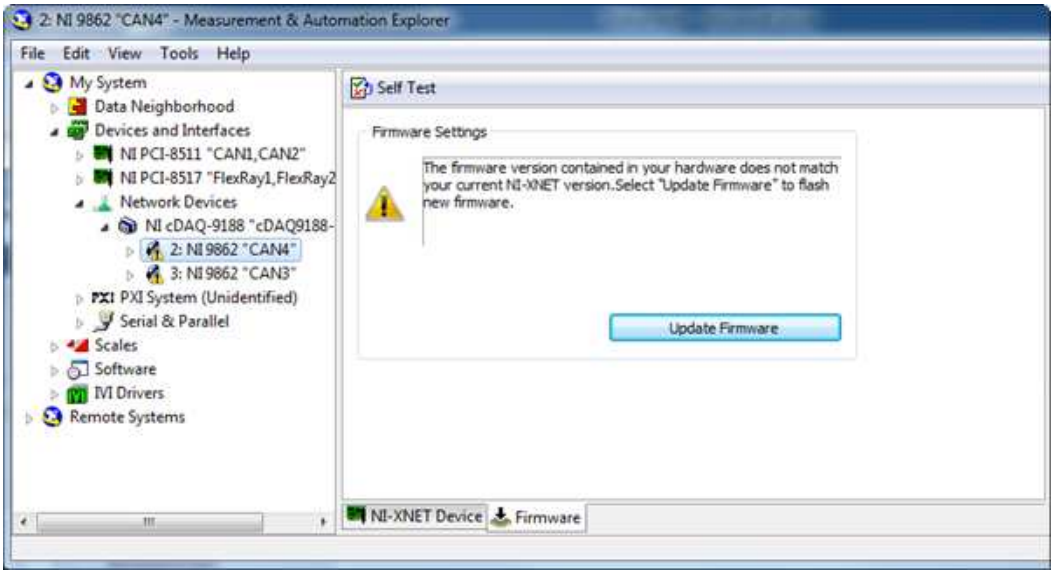


Figure 2-3. Firmware Update Needed

If MAX indicates a firmware version mismatch, you must update the module firmware before using the module.

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 **Interface Name** box in the middle pane, as shown in Figure 2-4.

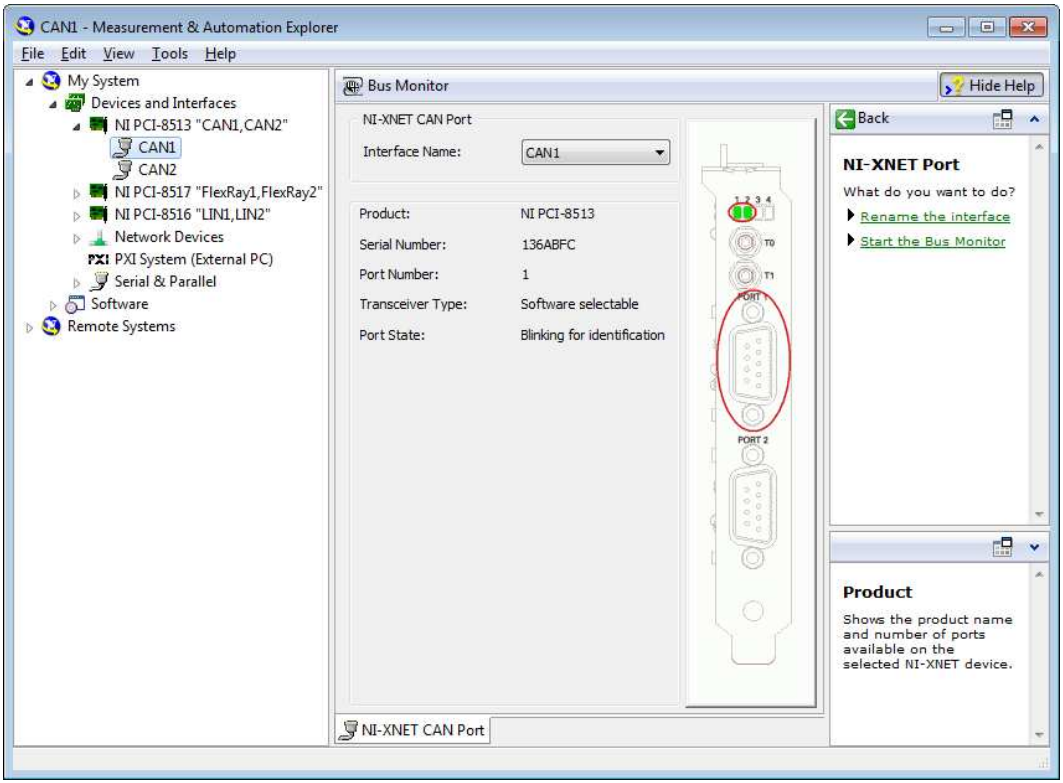


Figure 2-4. Renaming an Interface

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 a National Instruments PXI controller, you can install a PXI-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 a XNET device, using data from received CAN or FlexRay messages to generate outgoing CAN or FlexRay 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 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.

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 2-1. 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 2-1 prior to using NI-XNET.

Table 2-2. 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 2-1.

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 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 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 2-1, 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 *Sessions* in Chapter 4, *NI-XNET API for LabVIEW*.

Tools

NI-XNET includes two tools you can launch from MAX:

- **Bus Monitor**—Displays statistics for CAN, FlexRay, or LIN frames. This is a basic tool for analyzing CAN, FlexRay, or LIN network traffic. Launch this tool by right-clicking an NI-XNET interface and selecting **Bus Monitor** from the context menu.
- **NI I/O Trace**—Monitors function calls to the NI-XNET APIs. This tool helps in debugging application programming problems. To launch

this tool, open the **Software** branch of the MAX Configuration tree, right-click **NI I/O Trace**, and select **Launch NI I/O Trace**.

System Configuration API

NI-XNET supports the National Instruments System Configuration API, which provides programmatic access to many operations in MAX. This enables you to perform these operations within your application.

The System Configuration API gathers information using various product experts. You can create a filter to gather information for one type of product, such as filtering for NI-XNET devices only. The NI-XNET expert programmatic name is `xnet`.

3

NI-XNET Hardware Overview

Overview

NI-XNET is a suite of products that provide connectivity to CAN, FlexRay, and LIN networks.

NI-XNET FlexRay Hardware

FlexRay Physical Layer

The FlexRay physical layer circuitry interfaces the FlexRay protocol controller to the physical bus wires.

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 data sheet.

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. Refer to *Pinout* for the PXI and PCI NI-XNET FlexRay interface pinout.

Cabling Requirements for FlexRay

Cables may be shielded or unshielded and should meet the physical medium requirements described in Table 3-1.

Table 3-1. FlexRay Cable Characteristics

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

Cable Lengths and Number of Devices

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.

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.

Pinout

Table 3-2 describes the FlexRay DB9 pinout.

Table 3-2. FlexRay DB9 Pinout

Pin	Signal	Signal
1	NC	No connection
2	FlexRayA	BM FlexRay channel A bus minus

Table 3-2. FlexRay DB9 Pinout (Continued)

Pin	Signal	Signal
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 CAN Hardware

NI-XNET Transceiver Cables

Hardware supporting NI-XNET Transceiver Cables allows you to select each port individually by plugging in the appropriate Transceiver Cable. Each Transceiver Cable implements the interface physical layer of the interface.

NI-XNET Transceiver Cables are designed to interface to NI-XNET host ports.

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.

High-Speed Physical Layer

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

Transceiver

NI-XNET CAN High-Speed hardware uses either the NXP TJA1041 or NXP TJA 1043 High-Speed CAN transceiver.

The NI-XNET CAN HS/FD Transceiver Cable uses the TJA1043 transceiver. All other PXI and PCI NI-XNET High-Speed CAN interfaces use the TJA1041.

Both the TJA1041 and TJA 1043 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 TJA1041 or TJA 1043 data sheet.

Bus Power Requirements

The High-Speed physical layer on PXI, PCI, 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. Refer to *Pinouts* for the PXI and PCI NI-XNET CAN interface pinout.

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

Cabling Requirements for High-Speed CAN

Cables should meet the physical medium requirements specified in ISO 11898, shown in Table 3-3.

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

Table 3-3. 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
Length-related resistance	70 mΩ /m nominal
Specific line delay	5 ns/m nominal

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 Figure 3-1 for an example of where termination resistors should be placed in a system with more than two devices.

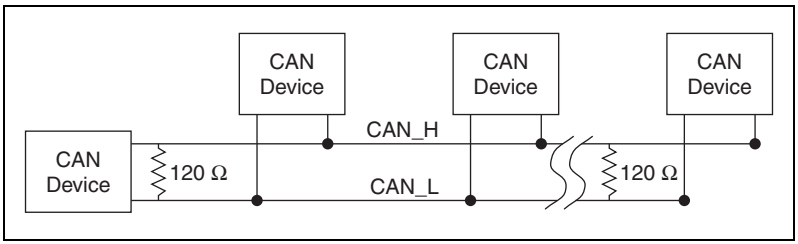


Figure 3-1. Termination Resistor Placement

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 PXI-8512, PCI-8512, PCI-8513 (in high-speed mode), or PXI-8513 (in high-speed mode), 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 for more information.

Cabling Example

Figure 3-2 shows an example of a cable to connect two CAN devices. For the internal power configuration, no V+ connection is required.

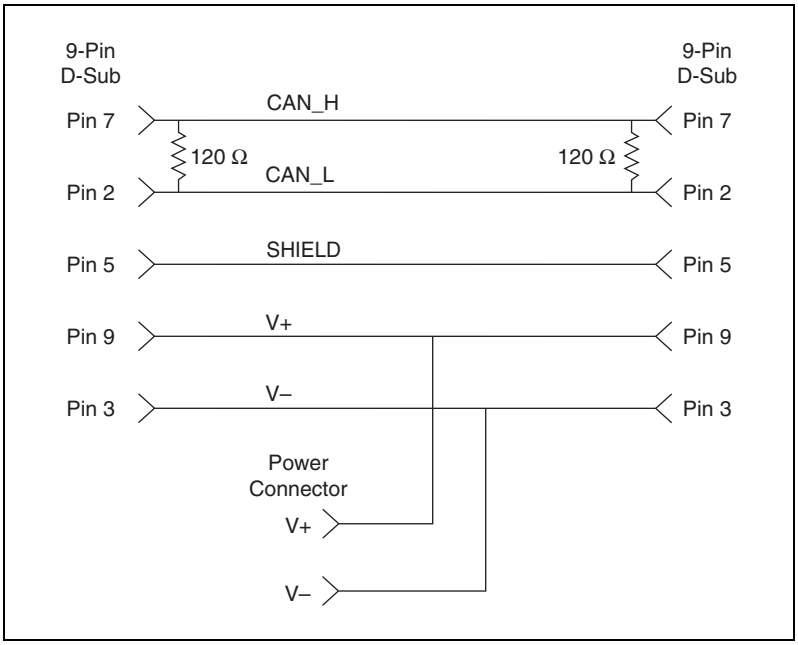


Figure 3-2. Cable Connecting Two CAN Devices

Low-Speed/Fault-Tolerant Physical Layer

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

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 TJA 1055T 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 TJA 1055T 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 TJA1054 and TJA 1055T data sheet.

Bus Power Requirements

The Low-Speed/Fault-Tolerant physical layer on PXI, PCI, 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. Refer to *Pinouts* for the PXI and PCI NI-XNET CAN interface pinout.

The Low-Speed/Fault-Tolerant physical layer on the C Series NI 9861 requires external power supply of +9 to +30 V to operate. Connect the external power supply to the Vsup pin on the module. The COM pins are for reference ground. Refer to *Pinouts* for the C Series NI-XNET CAN module pinout.

Cabling Requirements for Low-Speed/ Fault-Tolerant CAN

Cables should meet the physical medium requirements shown in Table 3-4. Belden cable (3084A) meets all of those requirements and should be suitable for most applications.

Table 3-4. 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

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 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.

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.

Figure 3-3 shows termination resistor placement in a Low-Speed CAN network.

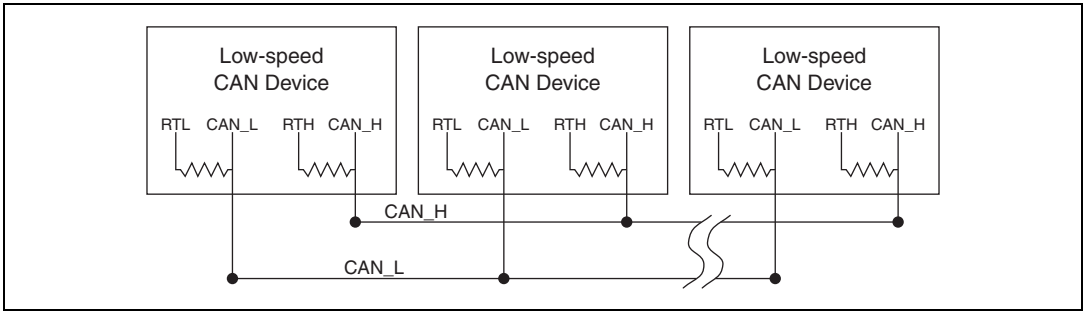


Figure 3-3. Termination Resistor Placement for Low-Speed CAN

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.

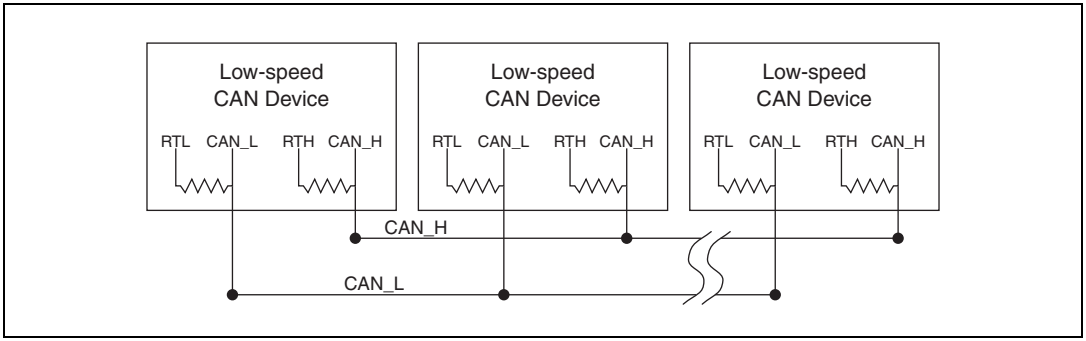
Determining the Necessary Termination Resistance for the Board

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 CAN_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 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 100–500 Ω (each) for a properly terminated low-speed network. You can determine the overall network termination as follows:

$$\frac{1}{R_{RTHoverall}} = \frac{1}{R_{RTHnode1}} + \frac{1}{R_{RTHnode2}} + \frac{1}{R_{RTHnode3}} + \frac{1}{R_{RTHnoden}}$$

NXP also recommends an individual device RTH and RTL termination of 500 Ω–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 100–500 Ω upon connection of the card:



where R_{RTH} *overall* should be 100–500 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.

Single Wire CAN Physical Layer

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

Transceiver

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.

Bus Power Requirements

The Single Wire physical layer on PXI and PCI NI-XNET interfaces requires external power supply of +8 to +18 V (+12 V recommended) to operate. Connect the external power supply to the Ext_Vbat pin on the module. The COM pins are used for reference ground. Refer to *Pinouts* for the PXI and PCI NI-XNET CAN module pinout.

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.

Cable Length

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

Number of Devices

As stated previously, 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.

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.

Pinouts

PXI-8511/8512/8513 and PCI-8511/8512/8513

Table 3-5 describes the CAN DB-9 pinout on PXI and PCI NI-XNET CAN interfaces.

Table 3-5. PXI and PCI NI-XNET CAN DB-9 Pinout

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 hardware only)

Table 3-6 describes the CAN DB-9 pinout on PXI and PCI NI-XNET External CAN transceivers.

Table 3-6. PXI and PCI NI-XNET External CAN Transceiver DB-9 Pinout

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

Table 3-6. PXI and PCI NI-XNET External CAN Transceiver DB-9 Pinout

D-SUB Pin	Signal	Description
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 NI 9861/9862

Table 3-7 describes the CAN DB-9 pinout on C Series NI-XNET CAN interfaces.

Table 3-7. C Series NI-XNET CAN DB-9 Pinout

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	VSUP	External power supply (+9 V to +30 V) required

NI-XNET LIN Hardware

LIN Physical Layer

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.

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 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 specifications, refer to their respective data sheets.

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. Refer to *Pinout* for the PXI and PCI NI-XNET LIN interface pinout.

Table 3-8. NI-XNET LIN Hardware Bus Power Requirements

Characteristic	Specification
Voltage	+8 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.

Cable Lengths

The maximum allowable cable length is 40 m, per the LIN specification.

Number of Devices

The maximum number of devices on a LIN bus is 16, per the LIN specification.

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.

Pinout

PXI-8516 and PCI-8516

Table 3-9 describes the LIN DB-9 pinout on PXI and PCI NI-XNET LIN interfaces.

Table 3-9. PXI and PCI NI-XNET LIN DB-9 Pinout

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

C Series NI 9866 and NI-XNET LIN Transceiver Cable

Table 3-10 describes the LIN DB-9 pinout on C Series and NI-XNET Transceiver Cable NI-XNET LIN interfaces.

Table 3-10. C Series NI-XNET LIN DB-9 Pinout

D-SUB Pin	Signal	Signal
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 to +18 V) required

Isolation

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

LEDs

NI-XNET one and two-port boards include two LEDs per port to help you monitor hardware and bus status. LED 1 primarily indicates whether the hardware is currently in use. LED 2 primarily indicates the activity information of the connected bus. Each LED can display two colors (red or green), which display in the following four patterns:

Pattern	Meaning
Off	No LED illumination
Solid	LED fully illuminated
Blink	Blinks at a constant rate of several times per second
Activity	Blinks in a pseudo-random pattern

The following LED indications are protocol independent:

Condition/State	LED 1	LED 2
Port identification	Blinks green	Blinks green
NI-XNET catastrophic error	Blinks red	Blinks red
No open session on hardware	Off	Off
Open session on hardware, port is properly powered, and hardware is not communicating	Solid green	Off
Open session on hardware, port is missing power	Solid red	Off

The following LED conditions are specific to CAN:

Condition/State	LED 1	LED 2
Hardware is communicating, and controller is in Error Active state	Solid green	Activity green (returns to idle/off one second after last TX or RX)
Hardware is communicating, and controller is in Error Passive state	Solid green	Activity red (returns to idle/off one second after last TX or RX)
Hardware is running, and controller transitioned to bus off	Solid green	Solid red

The following LED conditions are specific to FlexRay:

Condition/State	LED 1	LED 2
Hardware is integrated with a FlexRay cluster, and controller is in Normal Active state	Solid green	Activity green (continues while integrated)
Hardware is integrated with a FlexRay cluster, and controller is in Normal Passive state	Solid green	Activity red (continues while integrated)
Hardware was integrated with a FlexRay cluster and transitioned to Halt state	Solid green	Solid red

The following LED conditions are specific to LIN:

Condition/State	LED 1	LED 2
Hardware is communicating	Solid green	Activity green (returns to idle/off one second after last TX or RX)

Synchronization

PXI NI-XNET and PCI NI-XNET

The PXI chassis features a dedicated synchronization bus integrated into the backplane. NI-XNET products support use of this bus to synchronize with other National Instruments 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 hardware, the RTSI bus interface is a connector at the top of the card. You can synchronize multiple National Instruments PCI cards by connecting a RTSI ribbon cable between the cards that need to share timing and triggering signals.

CAN/XS 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 and NI-XNET Transceiver Cables

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 Interface:Source Terminal:Start Trigger property in Chapter 4, *NI-XNET API for LabVIEW*. For information about performing this routing in C/C++, refer to the Interface:Source Terminal:Start Trigger property in Chapter 5, *NI-XNET API for C*.

4

NI-XNET API for LabVIEW

This chapter explains how to use the NI-XNET API for LabVIEW and describes the NI-XNET LabVIEW VIs and properties.

Getting Started

This section helps you get started using NI-XNET for LabVIEW. It includes information about using NI-XNET within a LabVIEW project, NI-XNET examples, and using the NI-XNET palettes to create your own VI.

LabVIEW Project

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.

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 **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 to find examples for your specific hardware product.

A few examples are suggested to get started with NI-XNET.

For CAN (at **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**.
- **CAN Frame Input Stream.vi** (at **Hardware Input and Output»CAN»NI-XNET»Intro to Sessions»Frame Sessions**) with any output example.

For FlexRay (at **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**.
- **FlexRay Frame Input Stream.vi** (at **Hardware Input and Output»FlexRay»Intro to Sessions»Frame Sessions**)with any output example.

For LIN (at **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**.
- **LIN Frame Input Stream.vi** (at **Hardware Input and Output»LIN»NI-XNET»Intro to Sessions»Frame Sessions**) with any output example.

Open an example VI 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.

Palettes

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.

Basic Programming Model

The LabVIEW block diagram in the following figure shows the basic NI-XNET programming model.

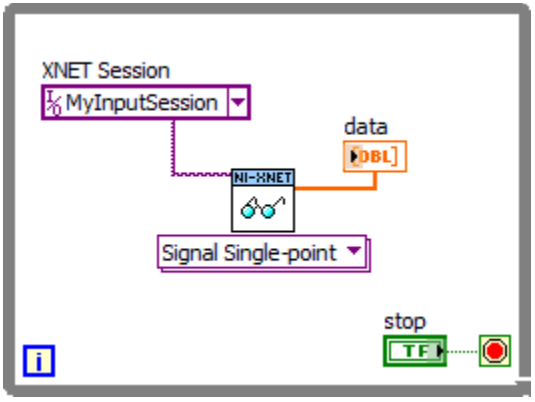
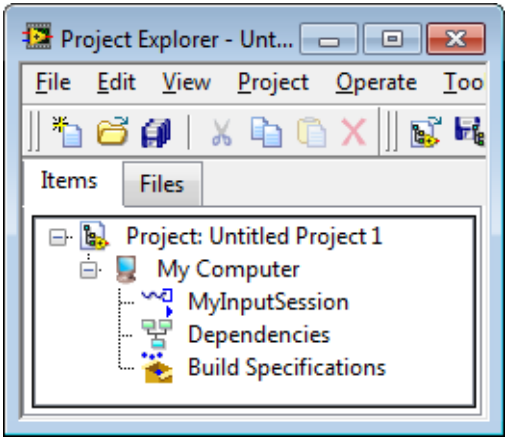


Figure 4-1. Basic Programming Model for NI-XNET for LabVIEW

Complete the following steps to create this block diagram:

1. Create an NI-XNET session in a LabVIEW project. The session name is MyInputSession, 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 block 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 **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 **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 Mode session at the same time you use a Frame Input Stream Mode 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 sections describe the fundamental concepts used within NI-XNET. Each section explains how to perform extended programming tasks.

Interfaces

What Is an Interface?

The interface represents a single CAN, FlexRay, or LIN 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 either *CAN* for a CAN interface, *FlexRay* for a FlexRay interface, or *LIN* for a LIN 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, and a two-port LIN device in your system, the interface names are *CAN1*, *CAN2*, *FlexRay1*, *FlexRay2*, *LIN1*, and *LIN2*, respectively. Devices that use a transceiver cable get only an interface name when the 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 written to use *CAN1* work with either hardware configuration without change.

How Do I View Available Interfaces?

Measurement and Automation Explorer (MAX)

Use 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 with the hardware product name, such as *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 **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 MAX. You select a specific interface from the list for use with **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 Node

In some cases, you may need to provide features similar to 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.

Within the NI-XNET functions palette **Advanced** subpalette, NI-XNET provides property nodes to query for available hardware.

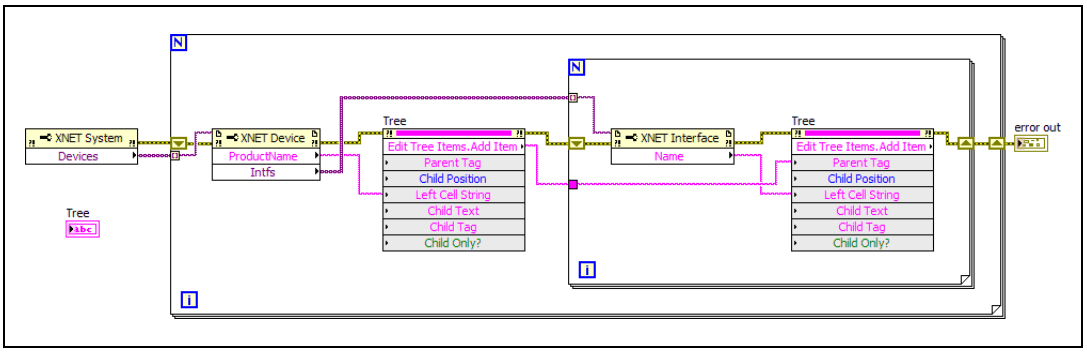


Figure 4-2. Advanced System Example Using Property Nodes

The block diagram in the figure above shows how to populate a LabVIEW tree control with NI-XNET devices and interfaces, in a manner similar to MAX. First, you get the list of devices from the XNET System node. For each XNET Device, you get its name and add that name to the tree. For each XNET interface (port) in the device, you get its name and add that name to the tree (with the device as the parent).

If you use this tree control to select an interface for session creation, you can pass the interface name from the tree directly to **XNET Create Session.vi**. Although **XNET Create Session.vi** uses the XNET Interface I/O name as an input, LabVIEW can cast a string to that I/O name automatically.

Databases

What Is a Database?

For the 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`) for CAN, FIBEX (`.xml`) for FlexRay, or LIN Description File (`.ldf`) for LIN. 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 LabVIEW double-precision floating-point numeric type.

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

What Is an 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 LabVIEW 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 LabVIEW application continues to run without change.

The alias concept is used in most NI-XNET features for the database classes. The XNET I/O Names for database classes include features for adding a new alias, viewing existing aliases, deleting an alias, and so on. You also can perform these tasks at run time, using the VIs available in the NI-XNET functions palette **Database»File Mgt** subpalette.

After you create an alias, it exists until you explicitly delete it. If you exit and relaunch LabVIEW, the aliases from the previous use remain. 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.

An alias is required for deploying databases to LabVIEW Real-Time (RT) targets. When you deploy to a LabVIEW RT target, the large text file is compressed to an optimized binary format, and that binary file is transferred to the target. For more information about databases with LabVIEW RT, refer to *Using LabVIEW Real-Time*.

Database Programming

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.

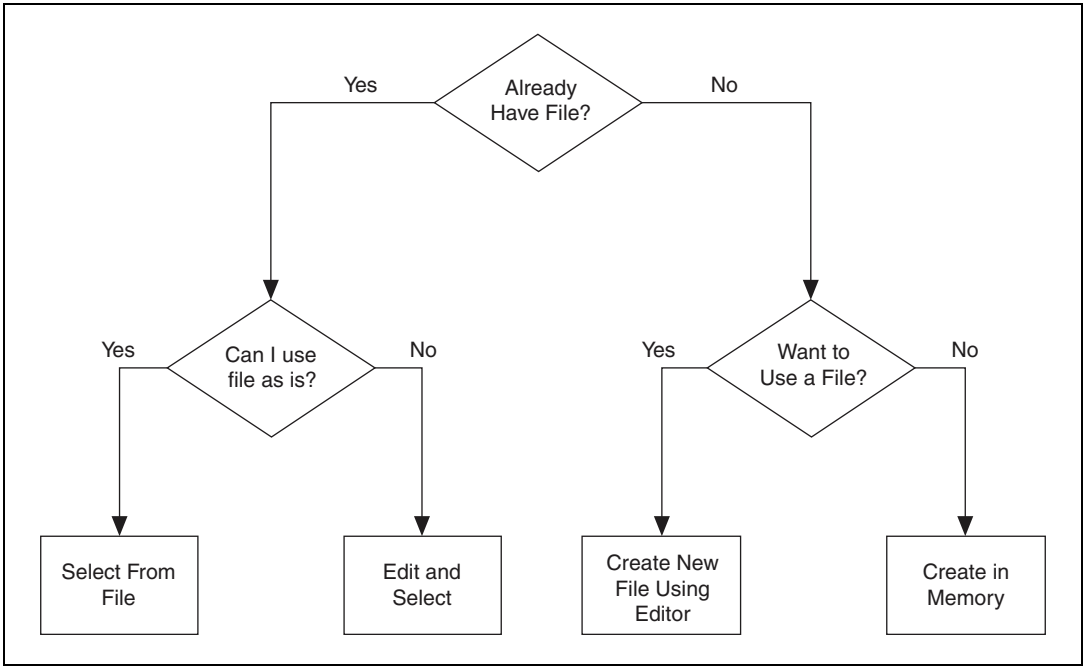


Figure 4-3. Decision Process for Choosing Database Source

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, or LDF format. When you have this file, using NI-XNET is relatively straightforward.

Can 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, and property nodes.

LabVIEW Project

The NI-XNET session in a LabVIEW project assumes that you have a database file. The configuration dialog includes controls to browse to your database file, select a cluster to use, and select a list of frames or signals. For example, if you create a Signal Input Single-Point session, you enter the database and cluster to use, then select one or more signals to read.

I/O Names

If you create sessions at run time, you need to wire objects from the database file to **XNET Create Session.vi**. The easiest way to do this is to use I/O names for the objects that you need.

For example, assume that you want to create a Signal Input Single-Point session and want the VI end user to select signals from the front panel. First, drag **XNET Create Session.vi** from the NI-XNET functions palette. Change the VI selector to Signal Input Single-Point. Right-click the **signal list** input and select **Create»Control**. This creates an array of XNET Signal I/O names on your front panel.

Right-click the **signal list** control and select **Browse for Database File** to find the database file. For a CANdb file, you can click the drop-down list for each array element to select from available signals in the file. For a FIBEX or LDF file, right-click **signal list** and **Select Database** to select a specific cluster within the file, then click the drop-down list to select signals. After you browse to the file and select a cluster, that information is saved with the VI, so you need to select only signal names from that point onward.

Most NI-XNET examples use I/O names to select objects (frames or signals). As a default, the NI-XNET example VIs use an example database file installed with NI-XNET. You can change this file to a different file using the previous steps.

Property Nodes

If you create a session at run time, you may want to implement your own front panel controls to select objects from the database, rather than use I/O names. Although the programming is more advanced than I/O names, you can do this using property nodes for the database classes. These property nodes are found in the NI-XNET functions palette **Database** subpalette.

For example, assume you want a tree control on the VI front panel. The tree shows the frames and signals within a selected cluster. The VI user selects signals from this tree control. The tree control block diagram uses a programming model similar to the *Advanced System Example Using Property Nodes*.

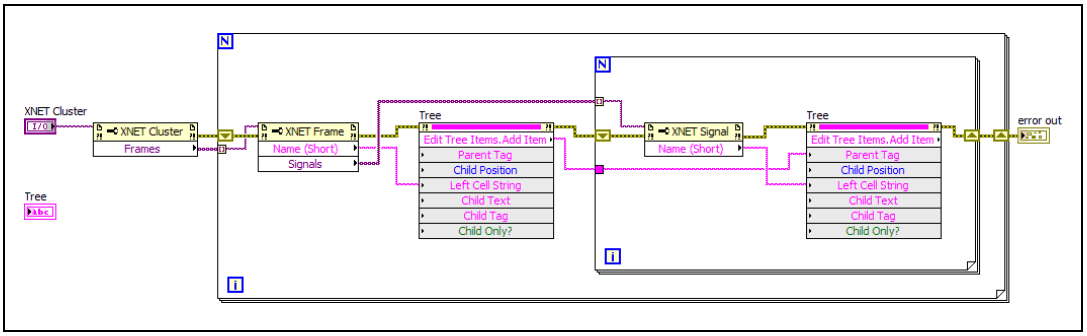


Figure 4-4. Advanced Database Example Using Property Nodes

The block diagram in the figure above shows how to populate a LabVIEW tree control with the frames and signals for a specific cluster. Because a cluster represents a single network connected to your NI-XNET interface, you do not need to show multiple clusters. First, you get the list of frames from the XNET Cluster node. For each XNET Frame, you get its name and add that name to the tree. For each XNET Signal in the frame, you get its name and add that name to the tree (with the frame as the parent).

If you use this tree control to select signals for session creation, you can use names from the tree to form the signal names that you wire to **XNET Create Session.vi**. For information about signal name syntax, refer to *XNET Signal I/O Name*.

Edit and Select

There are two options for editing the database objects for the NI-XNET session: edit in memory and edit the file.

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*.

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 **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*.

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 the Database 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 **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.

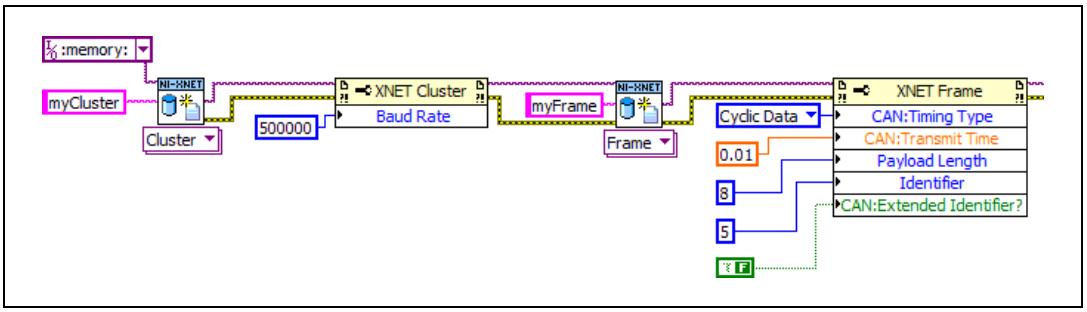


Figure 4-5. Create Cluster and Frame for CAN

Figure 4-5 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 **XNET Create Session (Frame Output Queued).vi**, this would create a session you can use to write data for transmit.

For additional information on in-memory configurations for CAN, refer to *Using CAN*.

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 opening up to seven distinct databases 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:*.

Sessions

What Is a Session?

The NI-XNET session represents a connection between your National Instruments CAN/FlexRay/LIN hardware and hardware products on the external network. As discussed in *Basic Programming Model*, your application uses sessions to read and write I/O data.

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.

In addition to read/write of I/O data, you can use the session to interact with the network in other ways. For example, **XNET Read.vi** 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.

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

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 **XNET Read.vi** 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**.

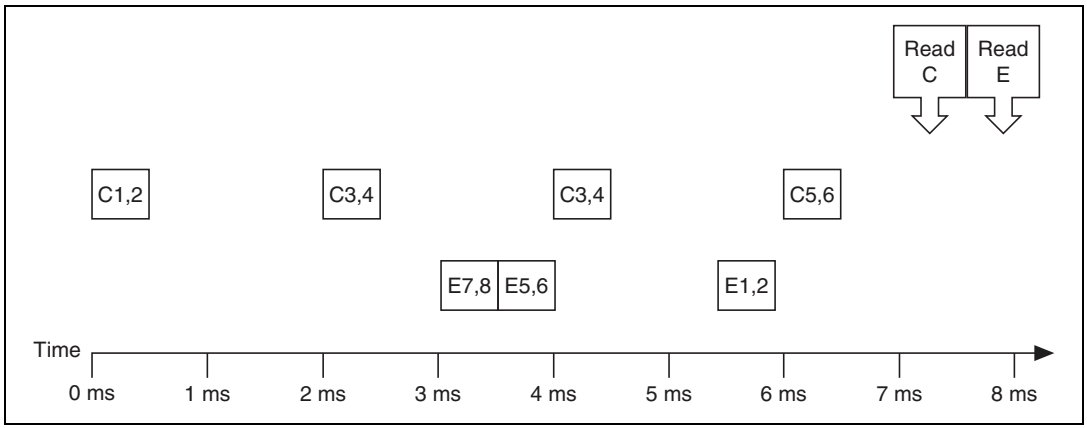
If the session uses a CAN interface, **XNET Read (Frame CAN).vi** is the recommended way to read data for this mode. This VI returns an array of frames, where each frame is a LabVIEW cluster specific to the CAN protocol. If the session uses a FlexRay or LIN interface, the read selection for that protocol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

Example

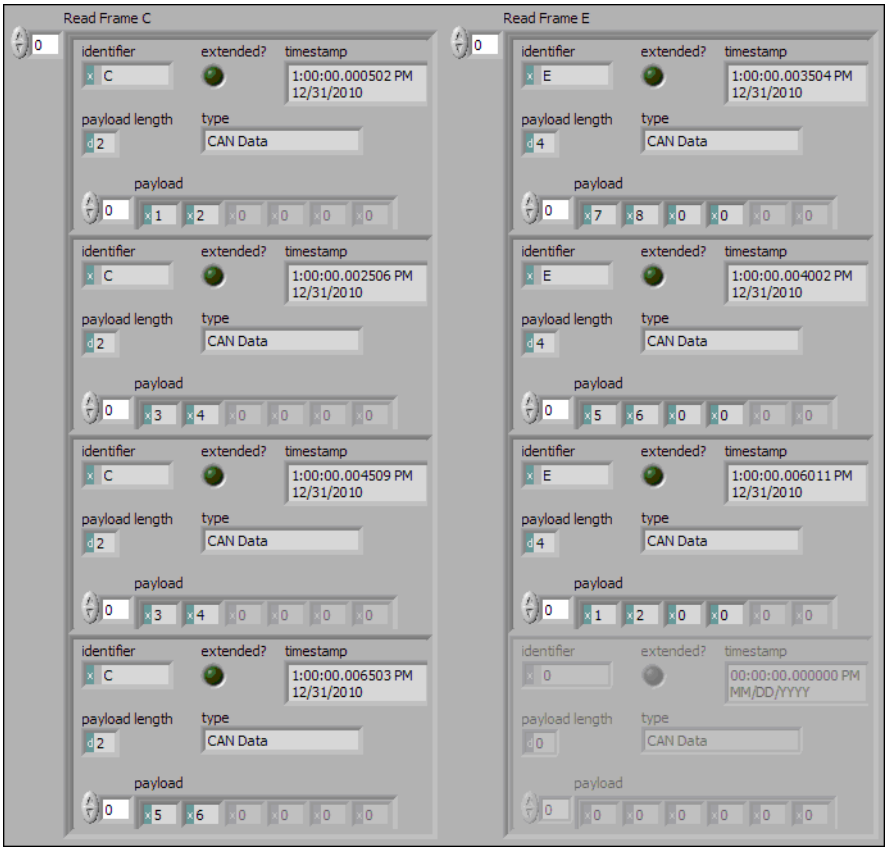
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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 (Frame CAN).vi** (one for C and one for E).



The following figure shows the data returned from the two calls to **XNET Read (Frame CAN).vi** (two different sessions).



The first call to **XNET Read (Frame CAN).vi** returned an array of values for frame C, and the second call to **XNET Read (Frame CAN).vi** returns an array for frame E. Each frame is a LabVIEW cluster with CAN-specific elements. 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 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.

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 **XNET Read.vi**, 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.

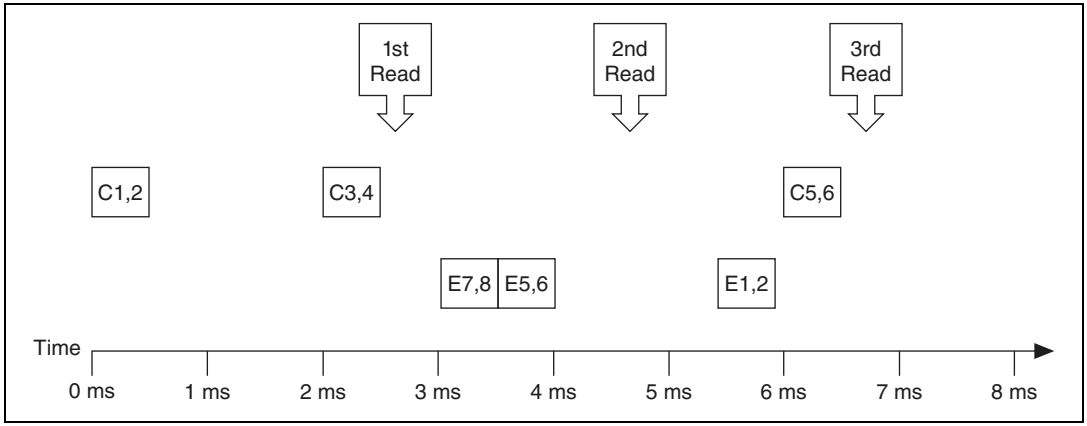
If the session uses a CAN interface, **XNET Read (Frame CAN).vi** is the recommended way to read data for this mode. This instance returns an array of frames, where each frame is a LabVIEW cluster specific to the CAN protocol. If the session uses a FlexRay or LIN interface, the read selection for that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

Example

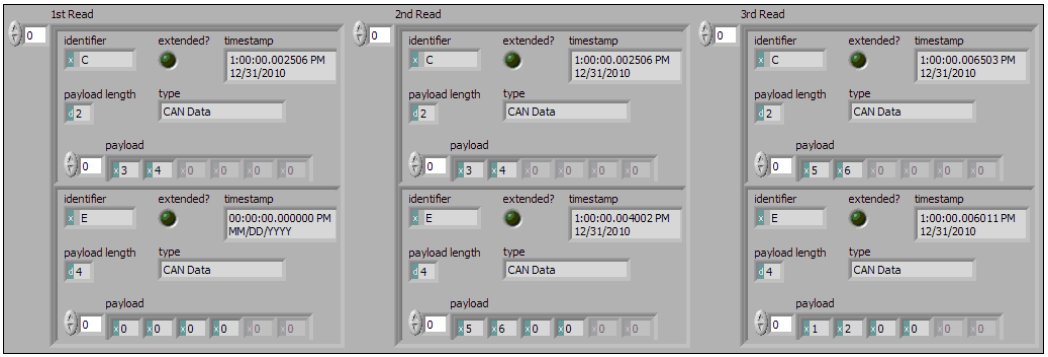
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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

The example uses CAN. The following figure shows a timeline of a frame transfer on the CAN network, followed by a single call to **XNET Read (Frame CAN).vi**. 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 **XNET Read (Frame CAN).vi**. The session contains frame data for two frames: C and E.



In the data returned from the first call to **XNET Read (Frame CAN).vi**, 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 **XNET Read (Frame CAN).vi**, 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 **XNET Read (Frame CAN).vi**, payload values 3 and 4 are returned again for frame C, because no new frame has been received since the previous call to **XNET Read (Frame CAN).vi**. The timestamp for frame C is the same as the first call to **XNET Read (Frame CAN).vi**

In the data returned from the third call to **XNET Read (Frame CAN).vi**, both frame C and frame E are received, so both elements return new values.

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.

If the session uses a CAN interface, **XNET Read (Frame CAN).vi** is the recommended way to read data for this mode. This instance 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, you can use **XNET Read (Frame Raw).vi**, which returns frames in an optimized, protocol-independent format.

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.

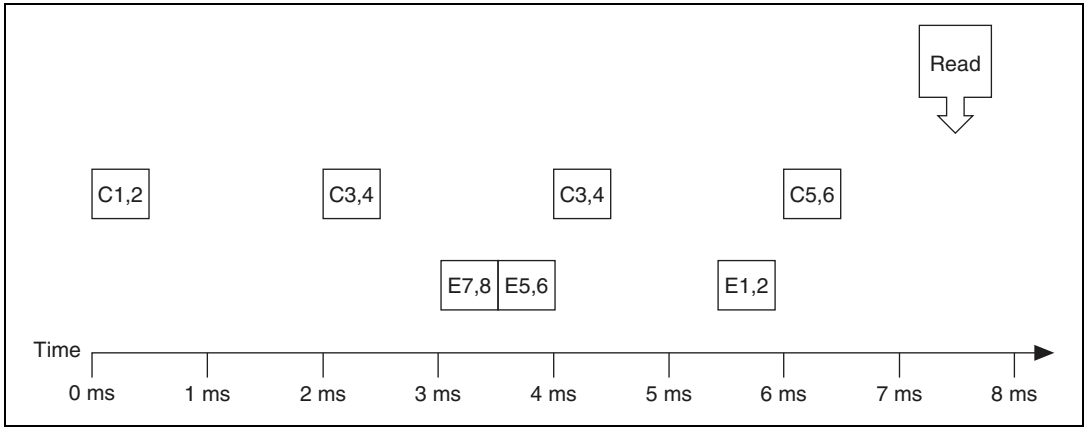
While using one or more Frame Input Stream sessions, you can use other sessions with different input modes. Received frames are copied to Frame Input Stream sessions in addition to any other applicable input session. For example, if you create a Frame Input Single-Point session for FrameA, then create a Frame Input Stream session, when FrameA is received, its data is returned from the call to **XNET Read.vi** 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 **XNET Read.vi**.

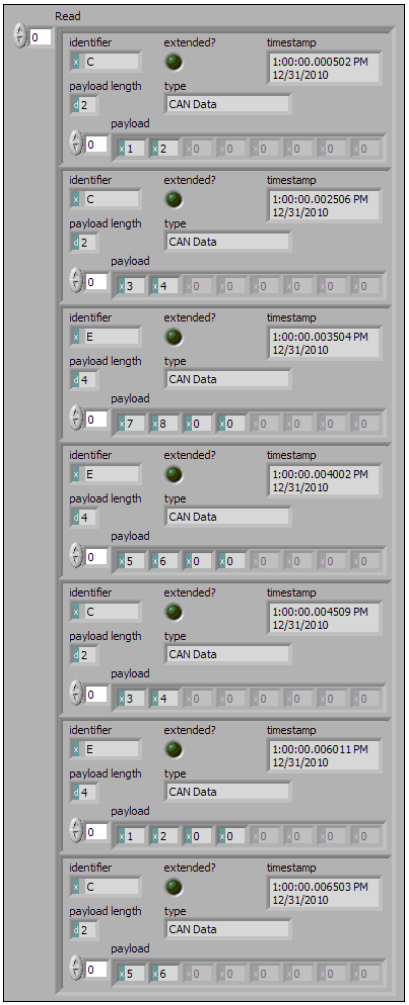
Example

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 **XNET Read (Frame CAN).vi**. Each frame contains its name (C or E), followed by the value of its two signals.



The following figure shows the data returned from **XNET Read (Frame CAN).vi**.



Frame C and frame E are returned in a single array of frames. Each frame is a LabVIEW cluster with CAN-specific elements. 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.

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.

If the session uses a CAN interface, **XNET Write (Frame CAN).vi** is the recommended way to write data for this mode. This instance provides an array of frame values, where each value is a LabVIEW cluster specific to the CAN protocol. If the session uses a FlexRay or LIN interface, the write selection for that protocol is recommended. For more advanced applications, you can use **XNET Write (Frame Raw).vi**, which provides frame values in an optimized, protocol-independent format.

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

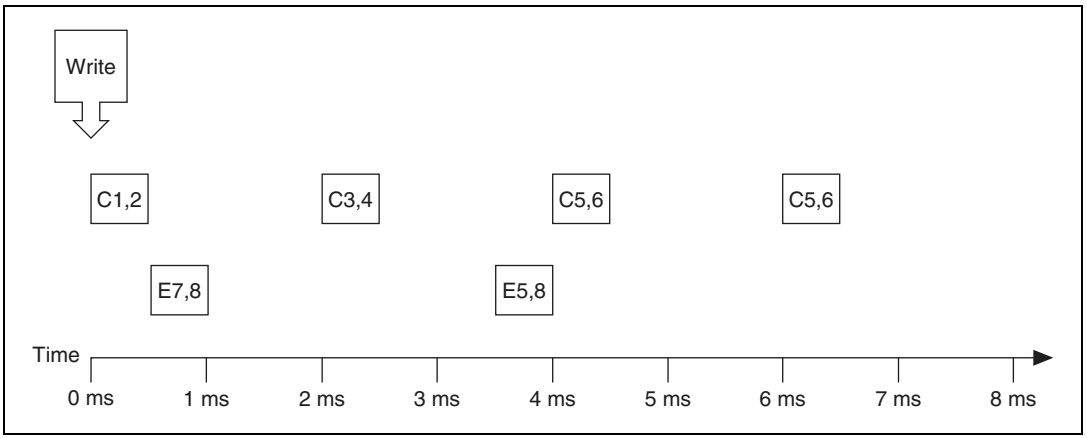
For this mode, NI-XNET transmits each frame according to its properties in the database. Therefore, when you call **XNET Write.vi**, the number of payload bytes in each frame value must match that frame's Payload Length property. The other frame value elements are ignored, so you can leave them uninitialized. 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

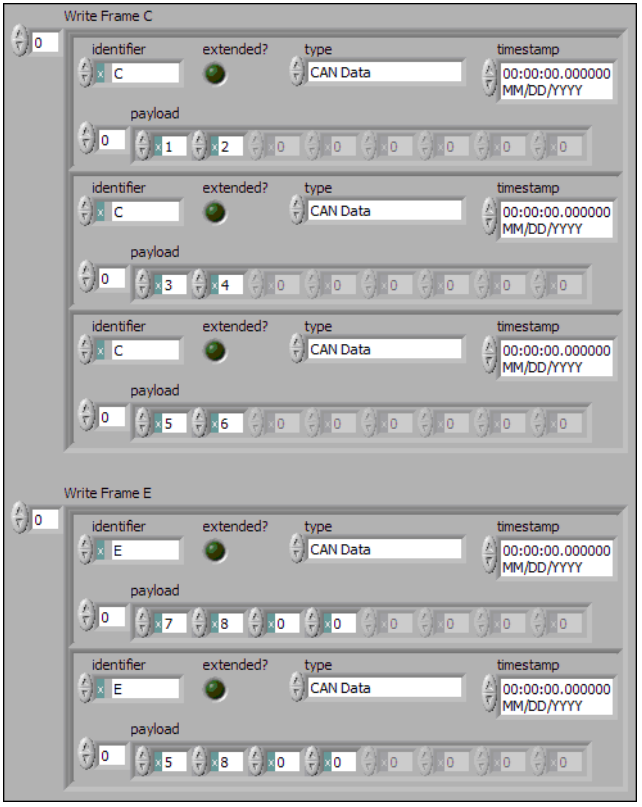
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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 **XNET Write (Frame CAN).vi**, one for frame C, followed immediately by another call for frame E.



The following figure shows the data provided to each call to **XNET Write (Frame CAN).vi**. 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 **XNET Write (Frame CAN).vi**. 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 **XNET Write (Frame CAN).vi** is called again, the new frame value is used.

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 **XNET Write.vi** is called twice before the next transmit, the transmitted frame uses the value from the second call to **XNET Write.vi**.

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

If the session uses a CAN interface, **XNET Write (Frame CAN).vi** is the recommended way to write data for this mode. This instance provides an array of frame values, where each value is a LabVIEW cluster specific to the CAN protocol. If the session uses a FlexRay or LIN interface, the write selection for that protocol is recommended. For more advanced applications, you can use **XNET Write (Frame Raw).vi**, which provides frame values in an optimized, protocol-independent format.

For this mode, NI-XNET transmits each frame according to its properties in the database. Therefore, when you call **XNET Write.vi**, the number of payload bytes in each frame value must match that frame's Payload Length property. The other frame value elements are

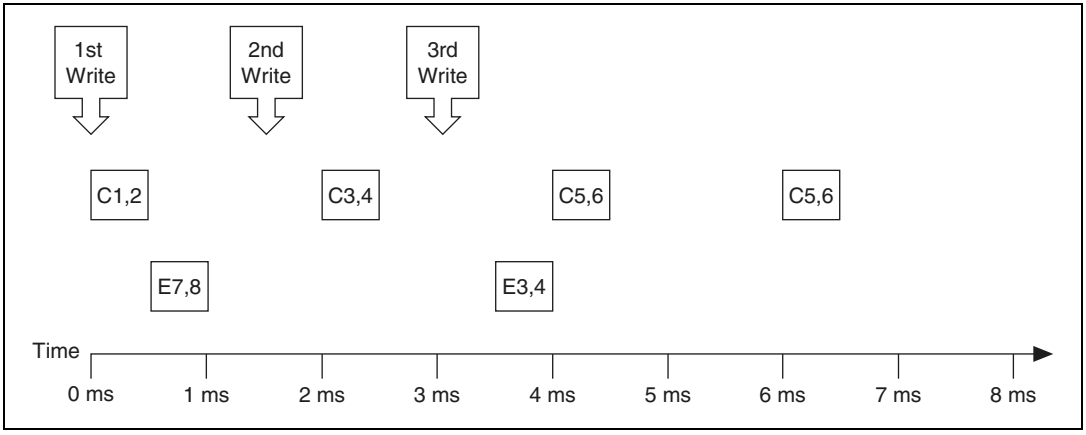
ignored, so you can leave them uninitialized. For CAN interfaces, if the number of payload bytes you write is smaller than the Payload Length configured in the database, the requested number of bytes transmits. If the number of payload bytes is larger than the Payload Length configured in the database, 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.

Example

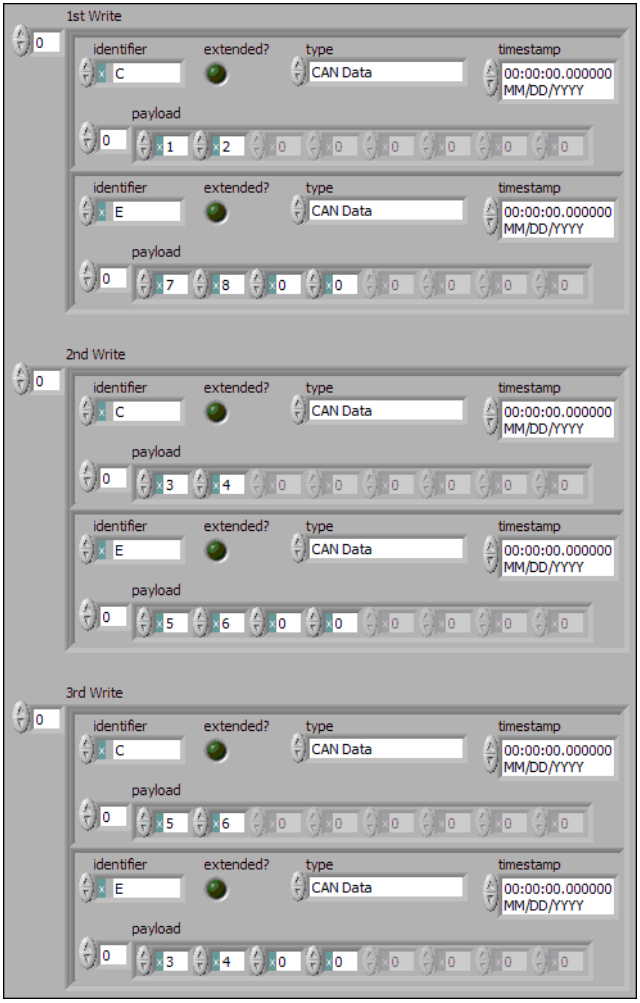
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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 **XNET Write (Frame CAN).vi**.



The following figure shows the data provided to each of the three calls to **XNET Write (Frame CAN).vi**. 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 **XNET Write (Frame CAN).vi**. Frame C transmits followed by frame E, both using frame values from the first call to **XNET Write (Frame CAN).vi**.

After the second call to **XNET Write (Frame CAN).vi**, 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 **XNET Write (Frame CAN).vi** 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 **XNET Write (Frame CAN).vi** is not used.

Frame C transmits the third time using the value from the third call to **XNET Write (Frame CAN).vi** (bytes 5, 6). Because frame C is cyclic, it transmits again using the same value (bytes 5, 6).

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 wired to **XNET Write.vi** 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 **XNET Write.vi**. The timing of each frame as specified in the database is ignored. For example, if you provide four frame values to **XNET Write.vi**, 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 the payload provided to **XNET Write.vi** is always used.

XNET Write (Frame CAN).vi is the recommended way to write data for this mode for CAN. This instance provides an array of frame values, where each value is a LabVIEW cluster specific to the CAN protocol. **XNET Write (Frame LIN).vi** is the recommended way to write data for this mode for LIN. This instance provides an array of frame values, where each value is a LabVIEW cluster specific to the LIN protocol. For more advanced applications, you can use **XNET Write (Frame Raw).vi**, which provides frame values in an optimized format.

Similar to CAN, LIN frame values transmit on the network based entirely on the time when you call **XNET Write.vi**. The timing of each frame as specified in the database is ignored. The LIN frame payload length in the database is ignored. 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 nonempty payload, the header + response for the frame is transmitted. If a frame for transmit is defined in the database (in-memory or otherwise), it is transmitted using its database checksum type. If the frame for transmit is not defined in the database, it is transmitted using enhanced checksum.

XNET Write (Frame LIN).vi is the recommended way to write data for this mode for LIN. This instance provides an array of frame values, where each value is a LabVIEW cluster specific to the LIN protocol. For more advanced applications, you can use **XNET Write (Frame Raw).vi**, which provides frame values in an optimized format.

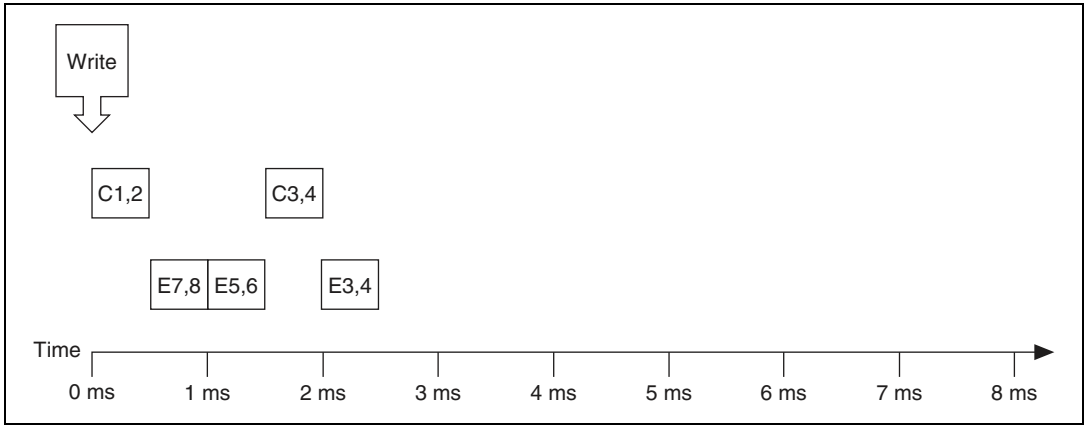
The frame values for this mode are stored in a queue, such that every value provided is transmitted.

Example

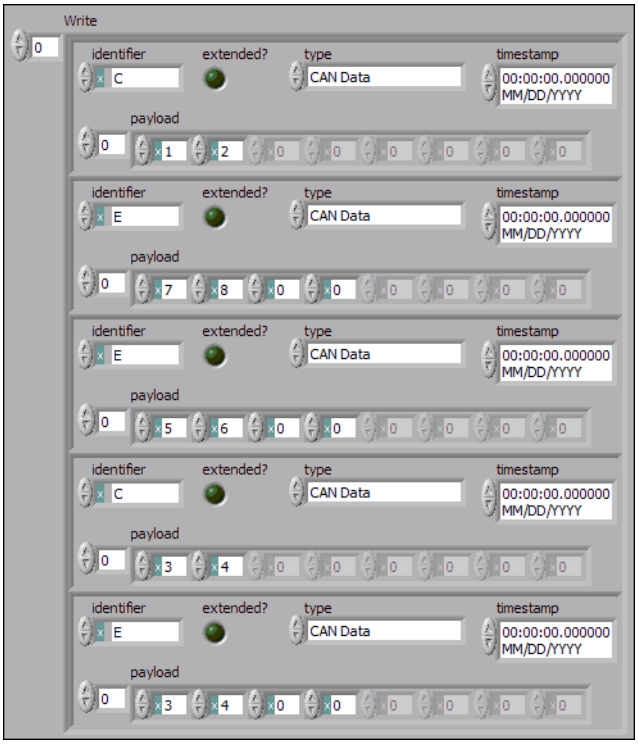
In this example CAN database, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven CAN frames, refer to *Cyclic and Event Timing*.

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 **XNET Write (Frame CAN).vi**.



The following figure shows the data provided to the single call to **XNET Write (Frame CAN).vi**. 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 **XNET Write (Frame CAN).vi**. 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.

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 **XNET Read.vi**, that call to **XNET Read.vi** returns signals for the second frame.

Use **XNET Read (Signal Single-Point).vi** for this mode. For more advanced applications, you can use **XNET Read (Signal XY).vi**, which returns a timestamp for each signal value. You can use the additional timestamps to determine whether each value is new since the last read.

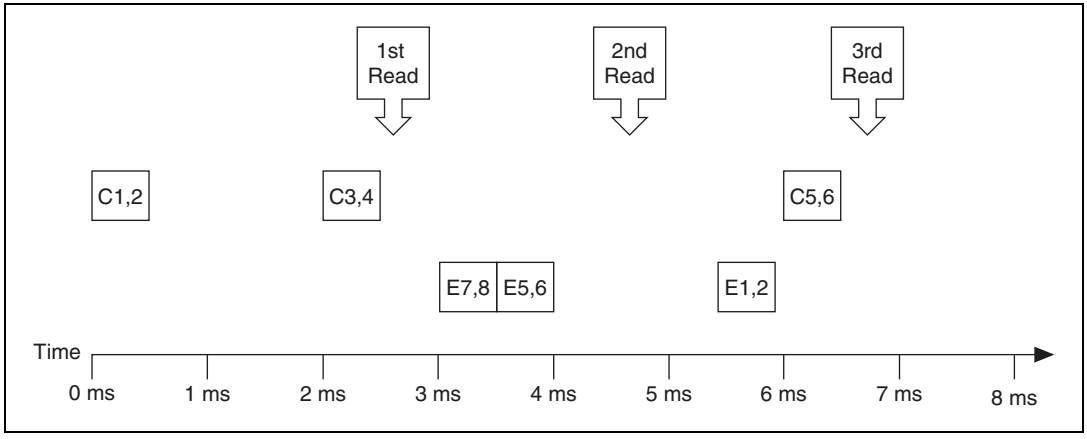
You also can specify a trigger signal for a frame. This signal name is *:trigger:.<frame name>*, and once it is specified in the **XNET Create Session.vi** signal list, it returns a value of 0.0 if the frame did not arrive since the last Read (or Start), and 1.0 if at least one frame of this ID arrived.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

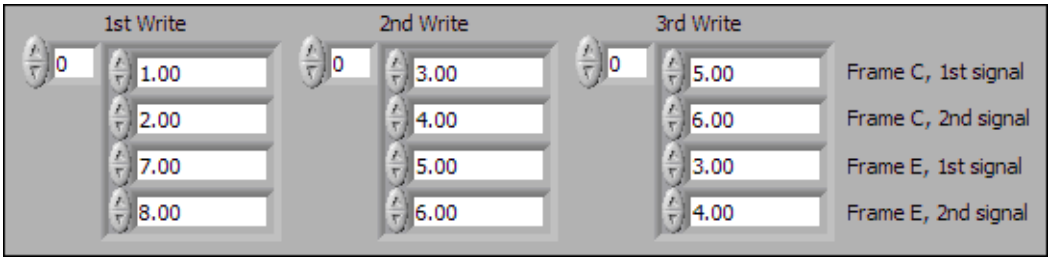
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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 **XNET Read (Signal Single-Point).vi**.



The following figure shows the data returned from each of the three calls to **XNET Read (Signal Single-Point).vi**. The session contains all four signals.



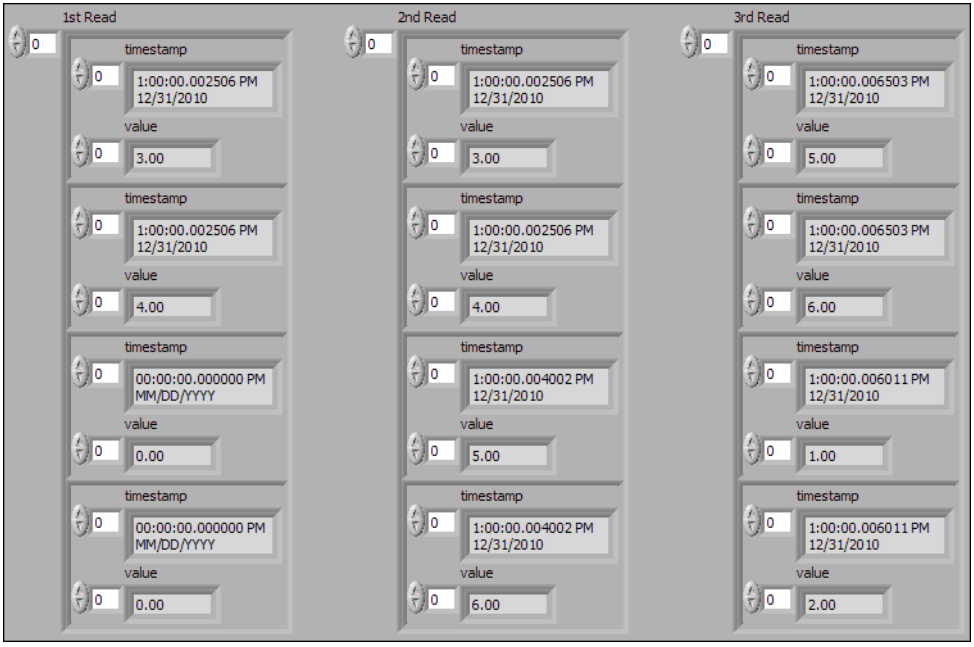
In the data returned from the first call to **XNET Read (Signal Single-Point).vi**, values 3 and 4 are returned for the signals of frame C. The values of the first reception of frame C (1 and 2) were lost, because this mode returns the most recent values.

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 **XNET Read (Signal Single-Point).vi**, 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 **XNET Read (Signal Single-Point).vi**, 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 **XNET Read (Signal Single-Point).vi**. New values are returned for frame E (5 and 6).

In the data returned from the third call to **XNET Read (Signal Single-Point).vi**, both frame C and frame E are received, so all signals return new values.

The following figure shows the data for the same frame timing, but using **XNET Read (Signal XY).vi**. The signal values are the same, but an additional timestamp is provided for each signal.



For the first call to **XNET Read (Signal XY).vi**, notice that the timestamps for frame E (last two signals) are invalid (all zero). This indicates that frame E has not been received since the session started, and therefore the signal values are the default.

For the second call to **XNET Read (Signal XY).vi**, notice that the timestamps for frame C (first two signals) are the same as the first call to **XNET Read (Signal XY).vi**. This indicates that frame C has not been received since the previous read, and therefore the signal values are repeated.

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 **XNET Read (Signal Waveform).vi** for this mode. You can wire the data **XNET Read (Signal Waveform).vi** returns directly to a LabVIEW Waveform Graph or Waveform Chart. The data consists of an array of waveforms, one for each signal specified for the session. Each waveform contains t0 (timestamp of first sample), dt (time between samples in seconds), and an array of resampled values for the signal.

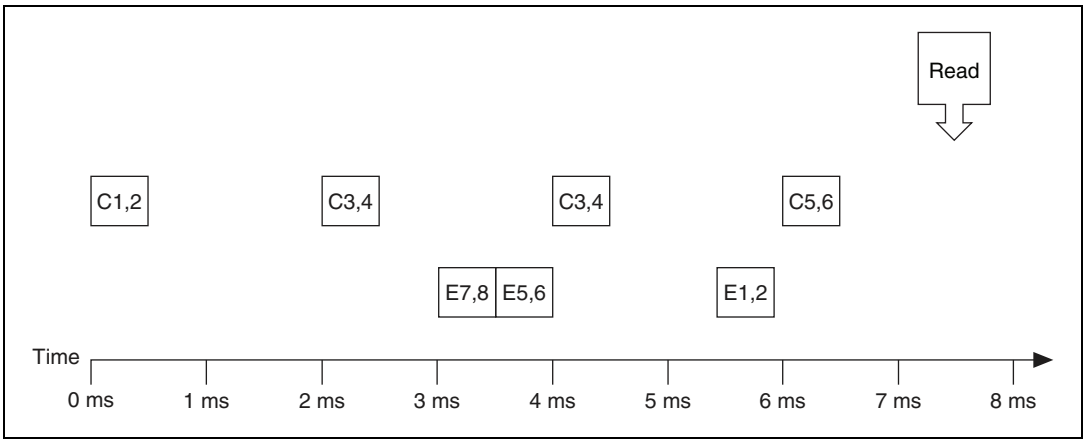
You specify the resample rate using the XNET Session Resample Rate property.

Example

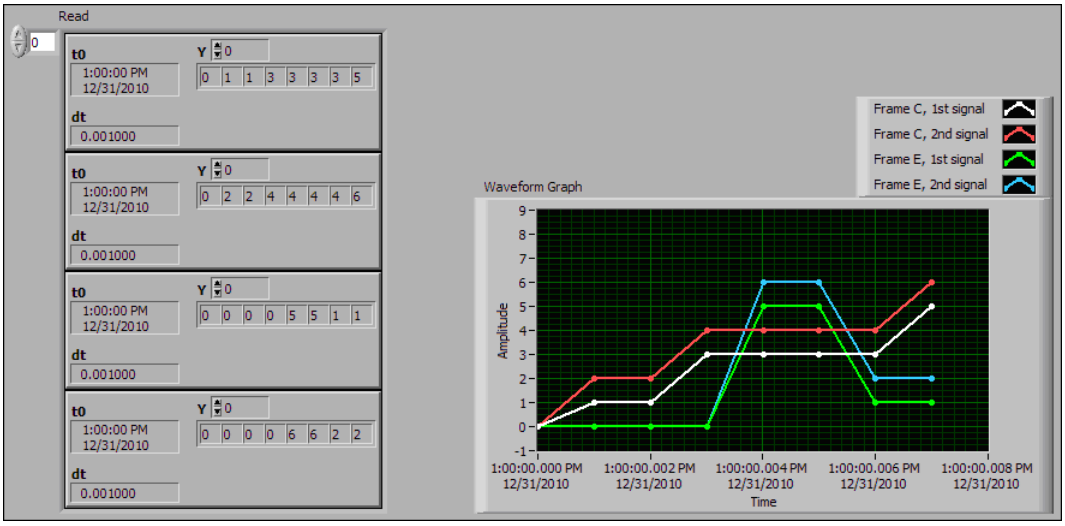
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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 **XNET Read (Signal Waveform).vi**. Each frame contains its name (C or E), followed by the value of its two signals.



The following figure shows the data returned from **XNET Read (Signal Waveform).vi**. The session contains all four signals and uses the default resample rate of 1000.0.



In the data returned from **XNET Read (Signal Waveform).vi**, t0 provides an absolute timestamp for the first sample. Assuming this is the first call to **XNET Read (Signal Waveform).vi** 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 **XNET Read (Signal Waveform).vi** 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 corresponding LabVIEW indicator (XY Graph) does not provide the same features as the indicator for Signal Input Waveform (Waveform Graph). For example, the Waveform Graph can plot consecutive calls to **XNET Read.vi** in a history, whereas XY Graph can plot only values from a single call to **XNET Read.vi**.

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 on the front panel (without significant validation). Use Signal Input XY Mode when you need to analyze CAN/FlexRay/LIN data on the diagram, for validation purposes.

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 **XNET Read (Signal XY).vi** for this mode. You can wire the data **XNET Read (Signal XY).vi** returns directly to a LabVIEW XY Graph.

The data consists of an array of LabVIEW clusters, one for each signal specified for the session. Each cluster contains two arrays, one for timestamp and one for value. For each signal, the timestamp and value array size is always the same, such that it represents a single array of timestamp/value pairs.

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 cluster (signal) to another.

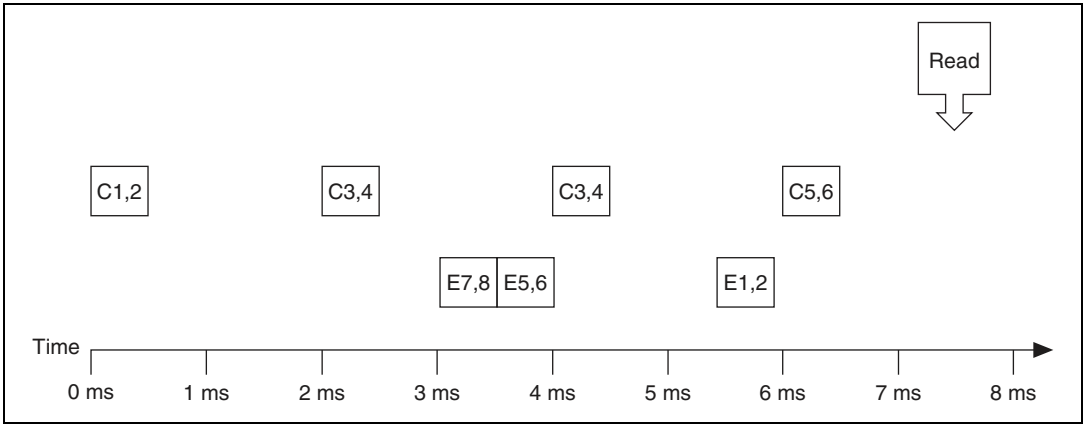
The received frames for this mode are stored in queues to avoid signal data loss.

Example

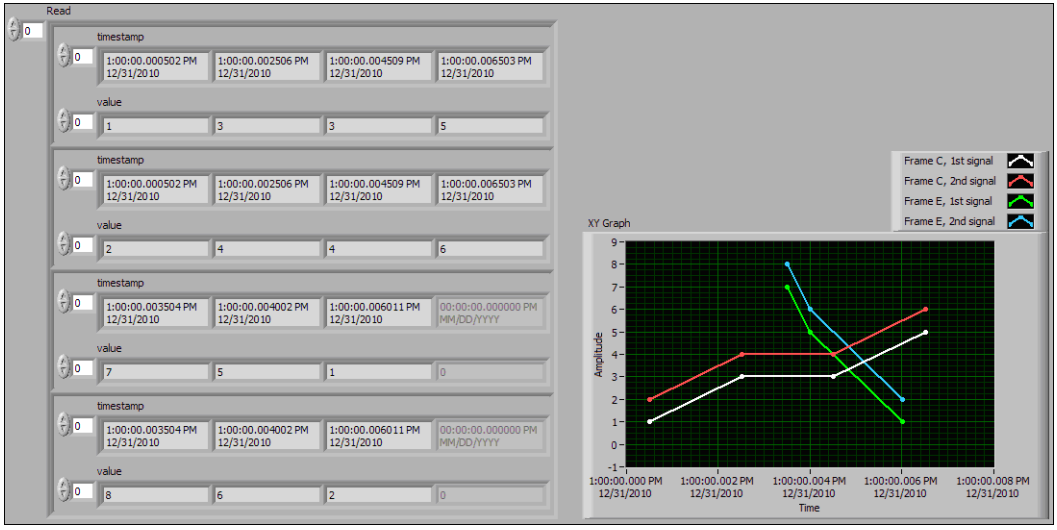
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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 **XNET Read (Signal XY).vi**. Each frame contains its name (C or E), followed by the value of its two signals.



The following figure shows the data returned from **XNET Read (Signal XY).vi**. The session contains all four signals.



Frame C was received four times, resulting in arrays of size 4 in the first two clusters. Frame E was received 3 times, resulting in arrays of size 3 in the first two clusters. 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 **XNET Read (Signal XY).vi**. This display is an accurate representation of signal changes on the network.

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 **XNET Write.vi** is called twice before the next transmit, the transmitted frame uses signal values from the second call to **XNET Write.vi**.

Use **XNET Write (Signal Single-Point).vi** for this mode.

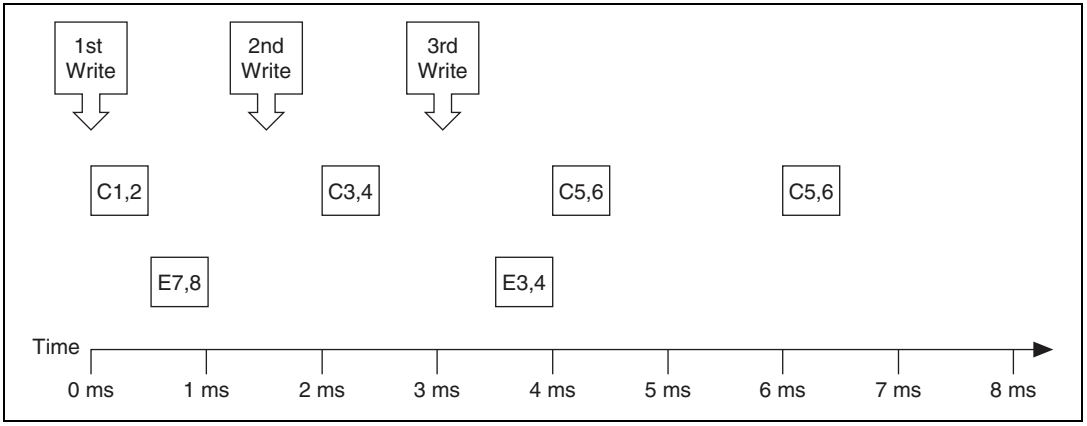
You also can specify a trigger signal for a frame. This signal name is *:trigger:.<frame name>*, and once it is specified in the **XNET Create Session.vi** signal list, you can write a value of 0.0 to suppress writing of that frame, or any value not equal to 0.0 to write the frame. You can specify multiple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

Example

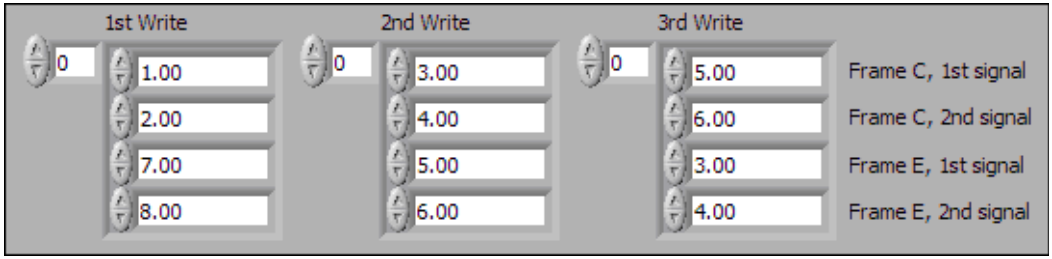
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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 **XNET Write (Signal Single-Point).vi**.



The following figure shows the data provided to each of the three calls to **XNET Write (Signal Single-Point).vi**. The session contains all four signals.



Assuming the Auto Start? property uses the default of true, the session starts within the first call to **XNET Write (Signal Single-Point).vi**. Frame C transmits followed by frame E, both using signal values from the first call to **XNET Write (Signal Single-Point).vi**.

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 **XNET Write (Signal Single-Point).vi**, 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 **XNET Write (Signal Single-Point).vi** 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 **XNET Write (Signal Single-Point).vi** are not used.

Frame C transmits the third time using values from the third call to **XNET Write (Signal Single-Point).vi** (5 and 6). Because frame C is cyclic, it transmits again using the same values (5 and 6).

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 **XNET Write (Signal Waveform).vi** for this mode. You can wire the data provided to **XNET Write (Signal Waveform).vi** directly from a LabVIEW Waveform Graph or

Waveform Chart. The data consists of an array of waveforms, one for each signal specified for the session. Each waveform contains an array of resampled values for the signal.

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

The frames for this mode are stored in queues.

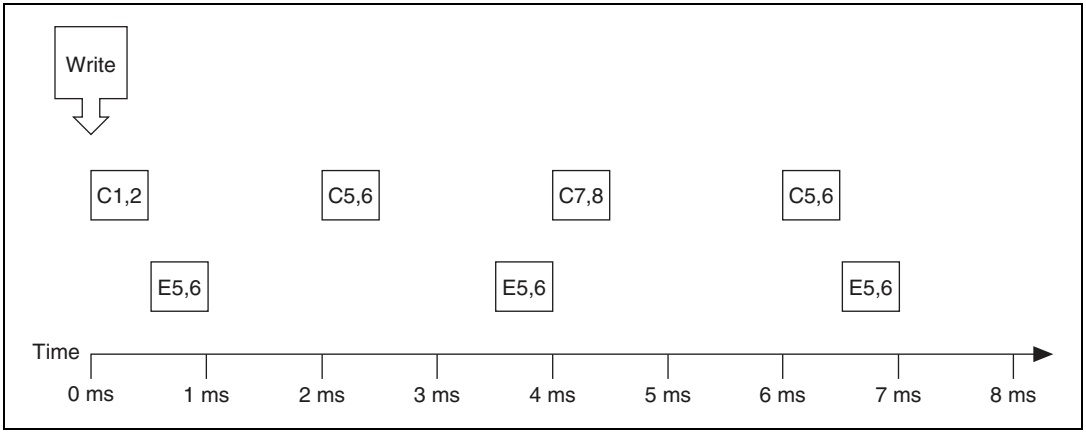
This mode is not supported for a LIN interface operating as slave. For more information, refer to *LIN Frame Timing and Session Mode*.

Example

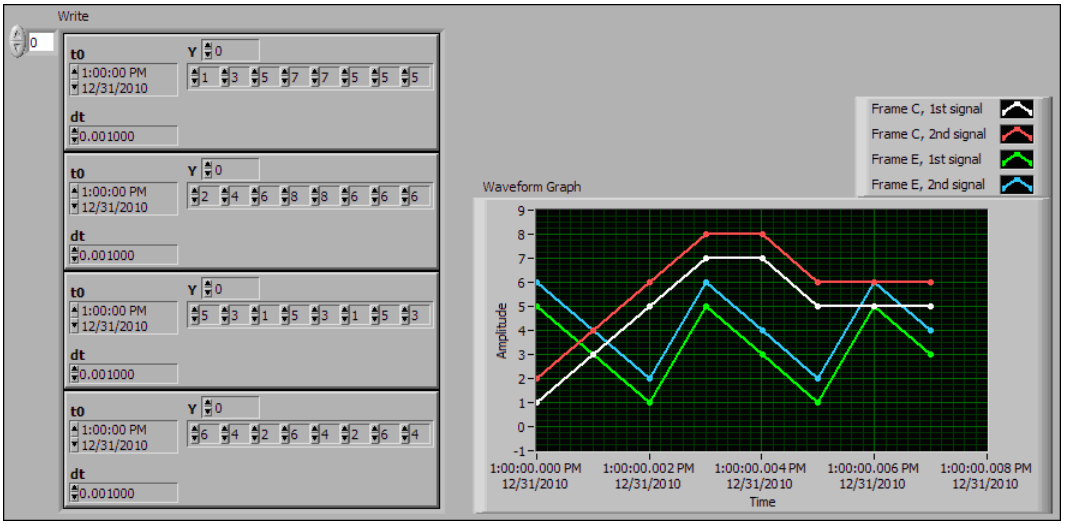
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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 **XNET Write (Signal Waveform).vi**.



The following figure shows the data provided to the call to **XNET Write (Signal Waveform).vi**. 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 **XNET Write (Signal Waveform).vi**. 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 **XNET Write (Signal Waveform).vi**. Transmit of frames starts as soon as the XNET session starts. The frame properties in the database determine 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 **XNET Write (Signal Waveform).vi**. 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 **XNET Write (Signal XY).vi** are mapped directly to transmitted frames, and no resampling occurs. Unless your application requires correlation of output data with DAQmx waveforms, Signal Output XY Mode is the recommended mode for writing a sequence of signal values.

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 **XNET Write (Signal XY).vi** for this mode. The data consists of an array of LabVIEW clusters, one for each signal specified for the session. Each cluster contains two arrays, one for timestamp and one for value. 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.

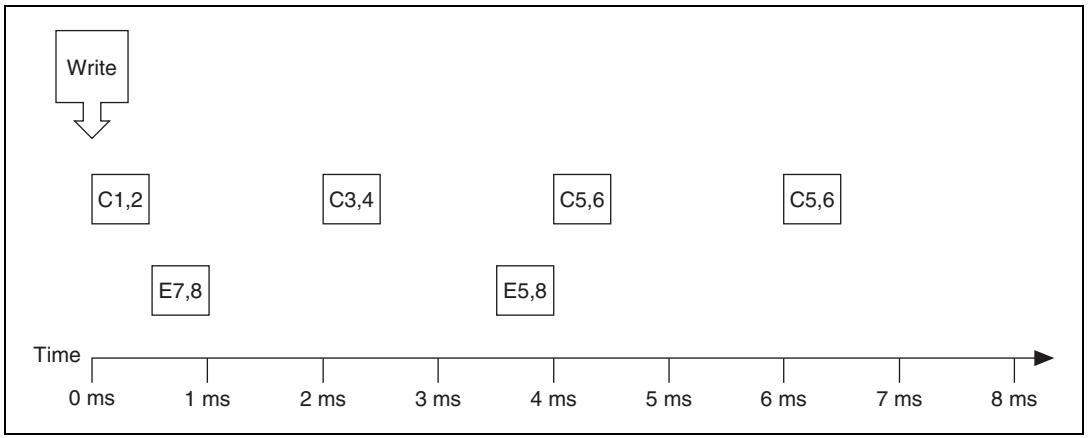
The frames for this mode are stored in queues, such that every signal provided is transmitted in a frame.

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 **XNET Write (Signal XY).vi**.



The following figure shows the data provided to **XNET Write (Signal XY).vi**. The session contains all four signals.



Assuming the Auto Start? property uses the default of true, the session starts within a call to **XNET Write (Signal XY).vi**. 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.

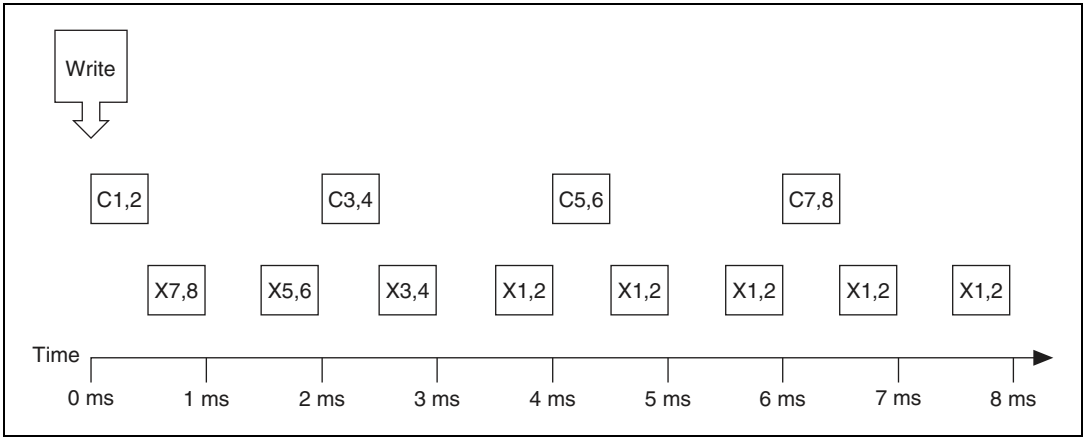
At 4.0 ms in the timeline, signal values 5 and 6 are taken from index 2 of the first two Y arrays and used for transmit of frame C.

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.

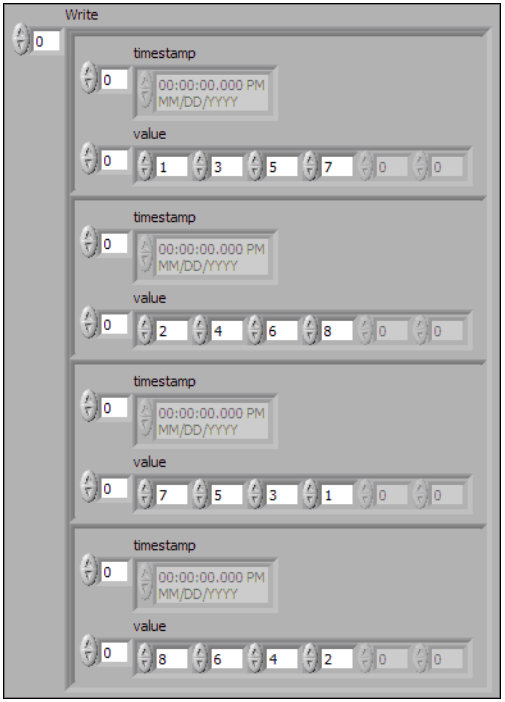
Because frame C is a cyclic frame, it transmits repeatedly. Although there are no more signal values for frame C, the values of the previous frame are used again at 6.0 ms in the timeline and every 2.0 ms thereafter. If **XNET Write (Signal XY).vi** is called again, the new signal values are used.

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 **XNET Write (Signal XY).vi**.



The following figure shows the data provided to **XNET Write (Signal XY).vi**. 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 **XNET Write (Signal XY).vi** 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 **XNET Write (Signal XY).vi**. When you do this, each call to

XNET Write (Signal XY).vi can generate a single event, and the overall timeline is not a major consideration.

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 **XNET Convert.vi**. Neither **XNET Read.vi** nor **XNET Write.vi** work with this mode; they return an error because hardware I/O is not permitted.

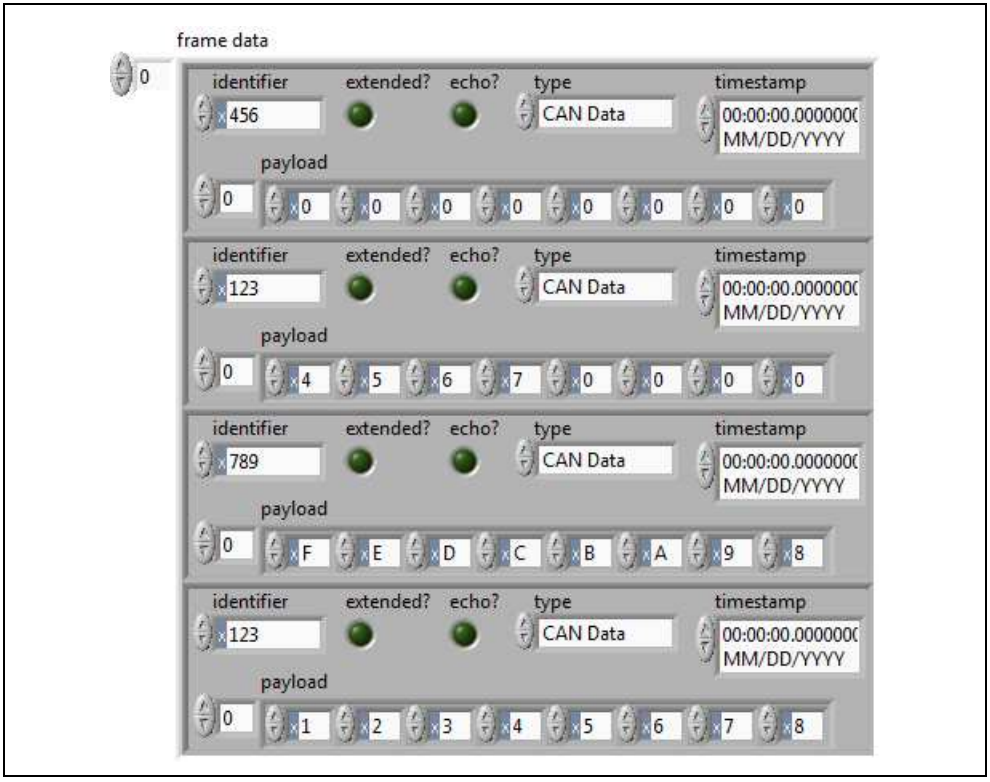
Conversion works similar to Single-Point mode. You specify a set of signals that can span multiple frames. Signal to frame conversion reads a set of values for the signals specified and writes them to the respective frame(s). Frame to signal conversion parses a set of frames and returns the latest signal value read from a corresponding frame.

Frames can be in any NI-XNET frame representations (CAN, FlexRay, LIN, or Raw). You select the conversion direction and the frame type by choosing the appropriate instance of **XNET Convert.vi**.

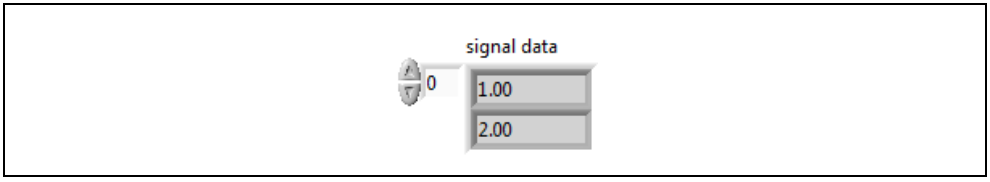
Example 1: Conversion of CAN Frames to Signals

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 **XNET Convert (Frame CAN to Signal).vi** with the following input:



results in the following signal values being returned:

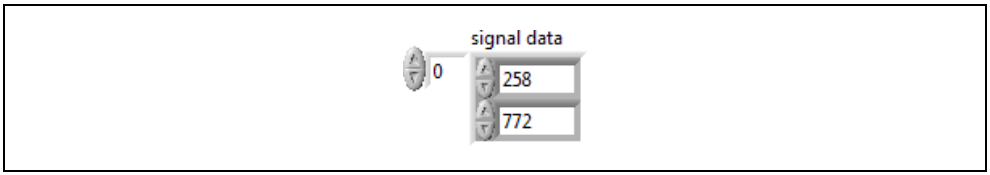


Explanation: The data are taken from frame 4. Frames 1 and 3 are ignored because they have a wrong (unmatched) ID. Frame 2 is ignored because its data are overwritten later with the values from frame 4, because frames are processed in the order of input.

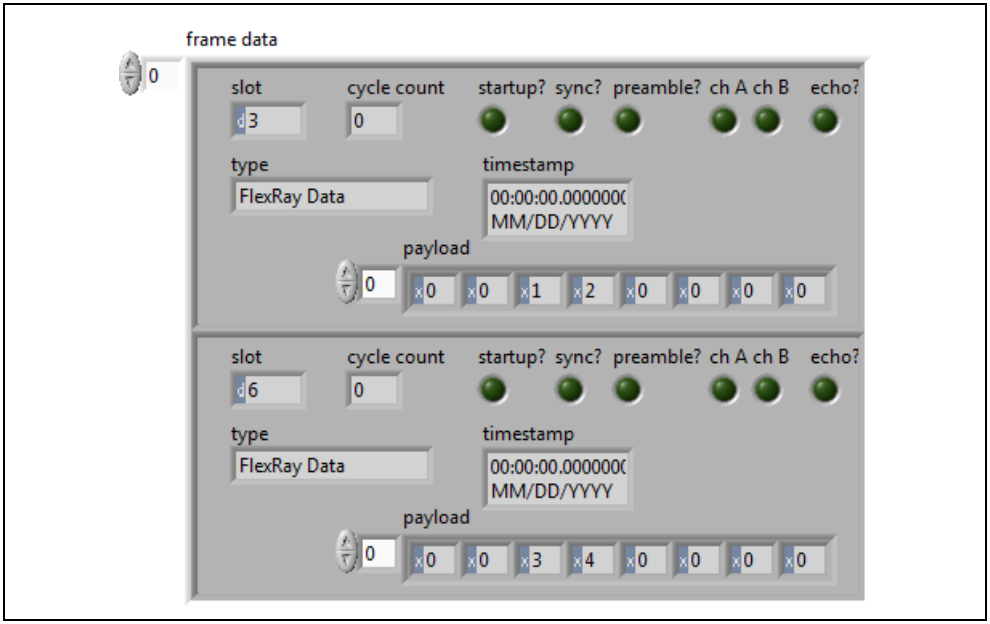
Example 2: Conversion of Signals to FlexRay Frames

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 **XNET Convert (Signal to Frame FlexRay).vi** with the following input:



causes the following frames to be generated:



Explanation: The first signal is converted to the byte sequence 0x01, 0x02 (1 × 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 × 256 + 4) and placed at byte 2 of the frame with slot ID 6. All other data are filled with the default values (0).

How Do I Create a Session?

There are two methods for creating a session: a LabVIEW project and **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 **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), **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.

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*.

Using CAN

This section summarizes some useful NI-XNET features specific to the CAN protocol.

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 **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.

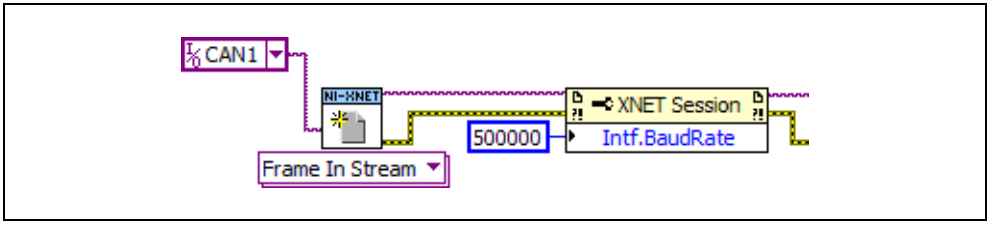


Figure 4-6. Configure CAN Frame Input Stream

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 Figure 4-7. 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:FD Baud Rate property.

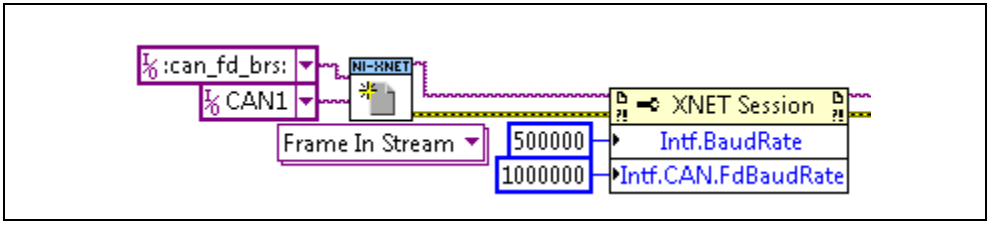


Figure 4-7. Configure CAN Frame Input Stream for a CAN FD Session

Using FlexRay

This section 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 property 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 property is zero 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*.

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 **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 **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.

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. **XNET Read.vi** and **XNET Write.vi** 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 **XNET Read.vi** and **XNET Write.vi** 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 **XNET Read.vi** and **XNET Write.vi**, discard the first iteration from the measurement.

For another VI or property node (not **XNET Read.vi** 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, **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:

- **XNET 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 Subpalette 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 **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.

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 *J1939* instead of *None*. This property is read only, as you cannot change the application protocol while the session is running.

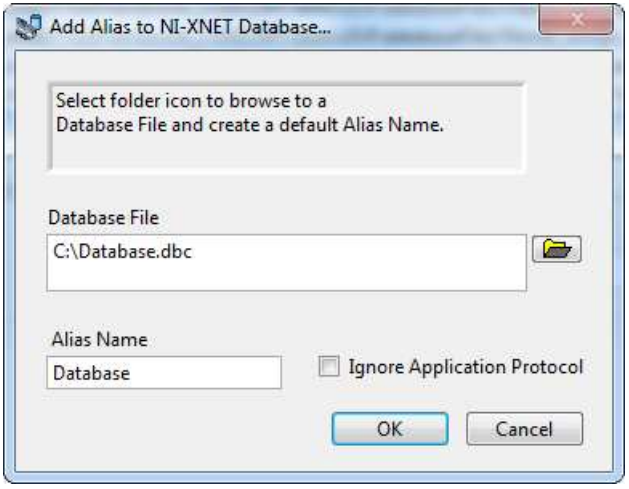
FIBEX databases do not define support for J1939 in the standard. If you save a J1939 database to FIBEX in the NI-XNET Database Editor or with **XNET Database Save.vi**, the J1939 properties are saved in a FIBEX extension defined by National Instruments in the FIBEX XML file.

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. 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.



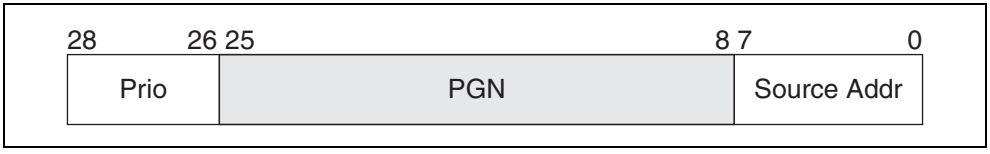
5. To have NI-XNET interpret the alias as an alias for a J1939 database, leave **Ignore Application Protocol** unchecked. To have NI-XNET interpret the alias as an alias for a plain CAN database, check **Ignore Application Protocol**.
6. Click **OK** to complete the alias addition.

J1939 Basics

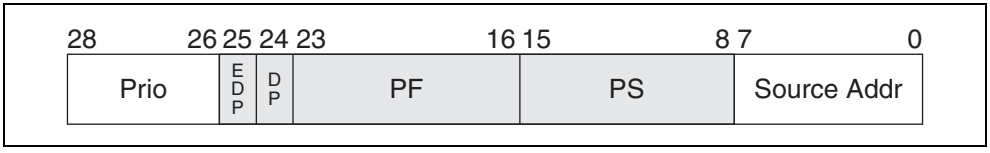
A J1939 network consists of ECUs connected by a CAN bus running at 250 k baud rate. Some newer networks might use a 500 k baud rate. A physical ECU can contain one or more logical ECUs called nodes or Controller Applications. This description refers to it as a node or ECU.

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 transmitting XNET session requires a node 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.

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.

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.

Transmitting Frames

When transmitting frames, the granted address of the node automatically replaces the source address part of the identifier.

Transmitting Frames without Granted Node Address

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. If a transmitting session without an address tries to send a frame without a valid address in the identifier, this returns an error.

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.

NI-XNET API for LabVIEW Reference

This section describes the NI-XNET LabVIEW APIs and properties.

XNET Session Constant

This constant provides the constant form of the XNET Session I/O name. You drag a constant to the block diagram of your VI, then select a session. You can change constants only during configuration, prior to running the VI. For a complete description, refer to *XNET Session I/O Name*.

XNET Create Session.vi

Purpose

Creates an XNET session to read/write data on the network.

Description

The XNET session specifies a relationship between National Instruments interface hardware and frames or signals to access on the external network (cluster). The XNET session also specifies the input/output direction and how data is transferred between your application and the network. For more information about NI-XNET concepts and object classes, refer to *Interfaces*, *Databases*, and *Sessions*.

Use this VI to create a session at run time. Run-time creation is useful when the session configuration must be selected using the front panel. If you prefer to create a session at edit time (static configuration), refer to Appendix E, *LabVIEW Project Provider*.

The instances of this polymorphic VI specify the session mode to create:

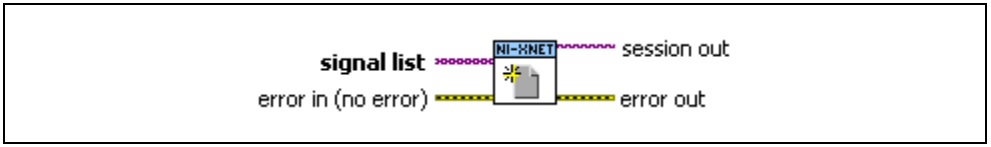
- **XNET Create Session (Signal Input Single-Point).vi**
- **XNET Create Session (Signal Input Waveform).vi**
- **XNET Create Session (Signal Input XY).vi**
- **XNET Create Session (Signal Output Single-Point).vi**
- **XNET Create Session (Signal Output Waveform).vi**
- **XNET Create Session (Signal Output XY).vi**
- **XNET Create Session (Frame Input Stream).vi**
- **XNET Create Session (Frame Input Queued).vi**
- **XNET Create Session (Frame Input Single-Point).vi**
- **XNET Create Session (PDU Input Queued).vi**
- **XNET Create Session (PDU Input Single Point).vi**
- **XNET Create Session (Frame Output Stream).vi**
- **XNET Create Session (Frame Output Queued).vi**
- **XNET Create Session (Frame Output Single-Point).vi**
- **XNET Create Session (PDU Output Queued).vi**
- **XNET Create Session (PDU Output Single-Point).vi**
- **XNET Create Session (Generic).vi**: (This instance is used for advanced applications, when you need to specify the configuration as strings.)
- **XNET Create Session (Conversion).vi**

XNET Create Session (Conversion).vi

Purpose

Creates an XNET session at run time for the Conversion Mode.

Format



Inputs



signal list is the array of XNET signals to convert to or from frames. These signals are specified in your database and describe the values encoded in one or more frames.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the created session.



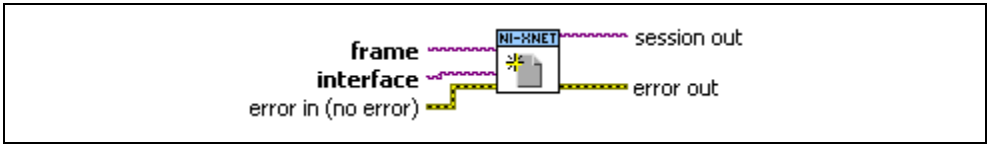
error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (Frame Input Queued).vi

Purpose

Creates an XNET session at run time for the Frame Input Queued Mode.

Format



Inputs

frame is the XNET Frame to read. This mode supports only one frame per session. Your database specifies this frame.

interface is the XNET Interface to use for this session.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the created session.

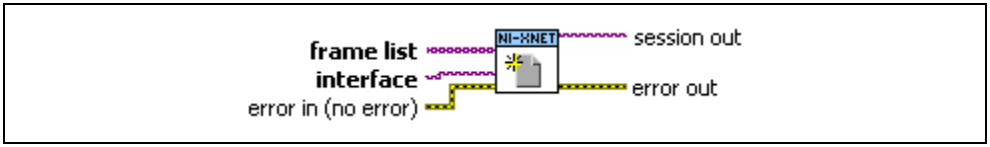
error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (Frame Input Single-Point).vi

Purpose

Creates an XNET session at run time for the Frame Input Single-Point Mode.

Format



Inputs

frame list is the array of XNET Frames to read. Your database specifies these frames.

interface is the XNET Interface to use for this session.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the created session.

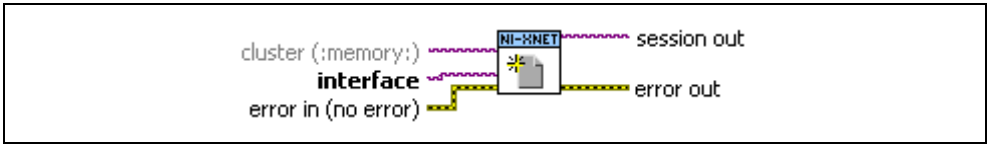
error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (Frame Input Stream).vi

Purpose

Creates an XNET session at run time for the Frame Input Stream Mode.

Format



Inputs

I/O

cluster is the XNET Cluster to use for interface configuration. The default value is *:memory:*, the in-memory database.

There are five options:

- **Empty in-memory database**: **cluster** is unwired, and the in-memory database is empty (**XNET Database Create Object.vi** is not used). This option is supported for CAN only (not FlexRay or LIN). After you create the session, you must set the XNET Session Interface:Baud Rate property using a Session node. You must set the baud rate prior to starting the session.
- **Pre-defined CAN FD in-memory database:** Pass in special in-memory databases :can_fd: and :can_fd_brs:, as the cluster (**XNET Database Create Object.vi** is not used). These databases are similar to the empty in-memory database (:memory:), but configure the cluster in either CAN FD or CAN FD+BRS mode, respectively. After you create the session, you must set the XNET Session Interface:Baud Rate and Interface:CAN:FD Baud Rate properties using a Session node. You must set these baud rates prior to starting the session.
- **Pre-defined SAE J1939 Database:** Pass in the special in-memory database :can_j1939:. This database is similar to the empty in-memory database (:memory:), but configures the cluster in CAN SAE J1939 application protocol mode. After you create the session, you must set the XNET Session Interface:Baud Rate property using a Session node. You must set this baud rate prior to starting the session.
- **Cluster within database file**: **cluster** specifies a cluster within a database file. This is the most common option used with FlexRay. The cluster within the FIBEX database file contains all required properties to configure the interface. For CANdb files, although the file itself

does not specify a CAN baud rate, you provide this when you add an alias to the file within NI-XNET. For LIN, the LDF file format already specifies the baud rate.

- **Nonempty in-memory database**: Call **XNET Database Create Object.vi** to create a cluster within the in-memory database, use the XNET Cluster property node to set properties (such as baud rate), then wire from the Cluster node to this cluster.
- **Subordinate**: Wire in **cluster** of *:subordinate:*. A subordinate session uses the cluster and interface configuration from other sessions. For example, you may have a test application with which the end user specifies the database file, cluster, and signals to read/write. You also have a second application with which you want to log all received frames (input stream), but that application does not specify a database. You run this second application using a subordinate session, meaning it does not configure or start the interface, but depends on the primary test application. For a subordinate session, start and stop of the interface (using **XNET Start.vi**) is ignored. The subordinate session reads frames only when another nonsubordinate session starts the interface.



interface is the XNET Interface to use for this session.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the created session.



error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (PDU Input Queued).vi

Purpose

Creates an XNET session at run time for the Frame Input Queued Mode.

This selection uses a PDU instead of a frame, but otherwise it is the same as **XNET Create Session (Frame Input Queued).vi**. You read PDU data using the **XNET Read.vi** frame selections. The payload in each frame value contains the PDU's data, not the entire frame.

XNET Create Session (PDU Input Single Point).vi

Purpose

Creates an XNET session at run time for the Frame Input Single-Point Mode.

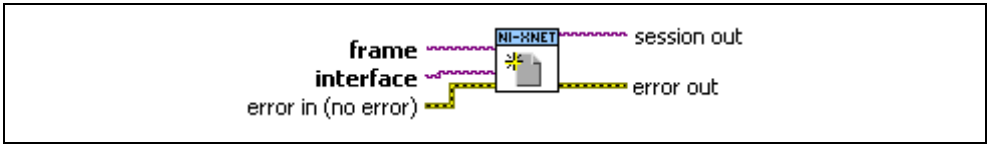
This selection uses one or more PDUs instead of frames, but otherwise it is the same as **XNET Create Session (Frame Input Single-Point).vi**. You read PDU data using the **XNET Read.vi** frame selections. The payload in each frame value contains the PDU's data, not the entire frame.

XNET Create Session (Frame Output Queued).vi

Purpose

Creates an XNET session at run time for the Frame Output Queued Mode.

Format



Inputs

frame is the XNET Frame to write. This mode supports only one frame per session. Your database specifies this frame.

interface is the XNET Interface to use for this session.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the created session.

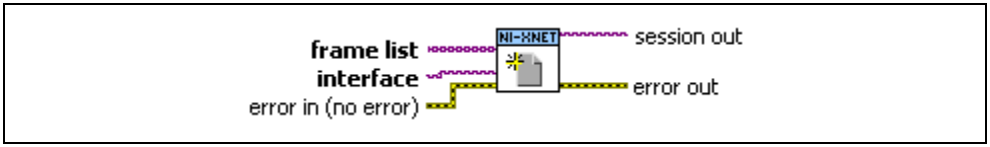
error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (Frame Output Single-Point).vi

Purpose

Creates an XNET session at run time for the Frame Output Single-Point Mode.

Format



Inputs



frame list is the array of XNET Frames to write. Your database specifies these frames.



interface is the XNET Interface to use for this session.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the created session.



error out is the error cluster output (refer to *Error Handling*).

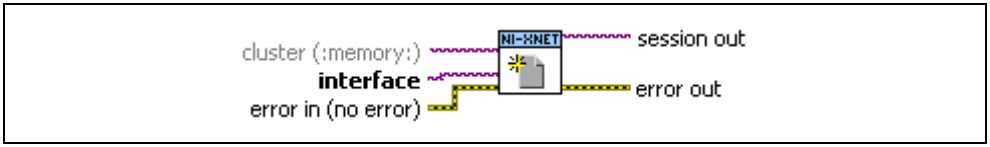
XNET Create Session (Frame Output Stream).vi

Purpose

Creates an XNET session at run time for the Frame Output Stream Mode.

Note This instance is supported for CAN and LIN only (not FlexRay).

Format



Inputs

cluster is the XNET Cluster I/O Name to use for interface configuration. The default value is *:memory:*, the in-memory database.

There are four options:

- **Empty in-memory database**: **cluster** is unwired, and the in-memory database is empty (**XNET Database Create Object.vi** is not used). After you create the session, you must set the XNET Session Interface:Baud Rate property using a Session node. You must set the CAN or LIN baud rate prior to starting the session.
- **Pre-defined CAN FD in-memory database:** Pass in special in-memory databases :can_fd: and :can_fd_brs:, as the cluster (**XNET Database Create Object.vi** is not used). These databases are similar to the empty in-memory database (:memory:), but configure the cluster in either CAN FD or CAN FD+BRS mode, respectively. After you create the session, you must set the XNET Session Interface:Baud Rate and Interface:CAN:FD Baud Rate properties using a Session node. You must set these baud rates prior to starting the session.
- **Pre-defined SAE J1939 Database:** Pass in the special in-memory database :can_j1939:. This database is similar to the empty in-memory database (:memory:), but configures the cluster in CAN SAE J1939 application protocol mode. After you create the session, you must set the XNET Session Interface:Baud Rate property using a Session node. You must set this baud rate prior to starting the session.
- **Cluster within database file**: **cluster** specifies a cluster within a database file. For CANdb files, although the file itself does not specify

a CAN baud rate, you provide this when you add an alias to the file within NI-XNET.

- **Nonempty in-memory databas**e: Call **XNET Database Create Object.vi** to create a cluster within the in-memory database, use the Cluster node to set properties (such as baud rate), then wire from the Cluster node to this cluster.



interface is the XNET Interface to use for this session.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the created session.



error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (PDU Output Queued).vi

Purpose

Creates an XNET session at run time for the Frame Output Queued Mode.

This selection uses a PDU instead of a frame, but otherwise it is the same as **XNET Create Session (Frame Output Queued).vi**. You write PDU data using the **XNET Write.vi** frame selections. The payload in each frame value contains the PDU's data, not the entire frame.

XNET Create Session (PDU Output Single-Point).vi

Purpose

Creates an XNET session at run time for the Frame Output Single-Point Mode.

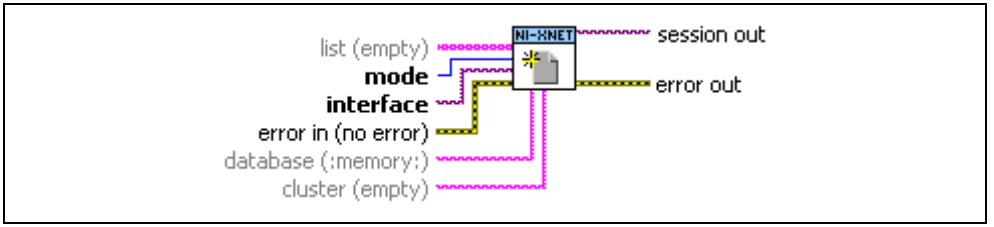
This selection uses a PDU instead of a frame, but otherwise it is the same as **XNET Create Session (Frame Output Single-Point).vi**. You write PDU data using the **XNET Write.vi** frame selections. The payload in each frame value contains the PDU's data, not the entire frame.

XNET Create Session (Generic).vi

Purpose

Creates an XNET session at run time using strings instead of XNET I/O Names. This VI is for advanced applications, when you need to store the configuration as strings (such as within a text file).

Format



Inputs



list provides the list of signals or frames for the session.

The **list** syntax depends on the mode:

Mode	list Syntax
Signal Input Single-Point, Signal Output Single-Point	**list** contains one or more XNET Signal names. If more than one name is provided, a comma must separate each name. Each name must use the <*signal*> or <*frame.signal*> syntax as specified for the I/O name (new line and <*dbSelection*> not included).
Signal Input Waveform, Signal Output Waveform	**list** contains one or more XNET Signal names. If more than one name is provided, a comma must separate each name. Each name must use the <*signal*> or <*frame.signal*> syntax as specified for the I/O name (new line and <*dbSelection*> not included).

Mode	list Syntax
Signal Input XY, Signal Output XY	**list** contains one or more XNET Signal names. If more than one name is provided, a comma must separate each name. Each name must use the <*signal*> or <*frame.signal*> syntax as specified for the I/O name (new line and <*dbSelection*> not included).
Frame Input Stream, Frame Output Stream	**list** is empty (unwired).
Frame Input Queued, Frame Output Queued	**list** contains only one XNET Frame name. Only one name is supported. The frame name must use the <*frame*> syntax as specified for the I/O name (new line and <*dbSelection*> not included).
Frame Input Single-Point, Frame Output Single-Point	**list** contains one or more XNET Frame names. If more than one name is provided, a comma must separate each name. The frame name must use the <*frame*> syntax as specified for the I/O name (new line and <*dbSelection*> not included).



mode is the session mode.



interface is the XNET Interface to use for this session.



database is the XNET Database to use for interface configuration. The database name must use the <*alias*> or <*filepath*> syntax specified for the I/O name. The default value is *:memory:*, the in-memory database.



cluster is the XNET Cluster to use for interface configuration. The cluster name must use the <*cluster*> syntax specified for the I/O name (<*alias*>. prefix not included).



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the created session.



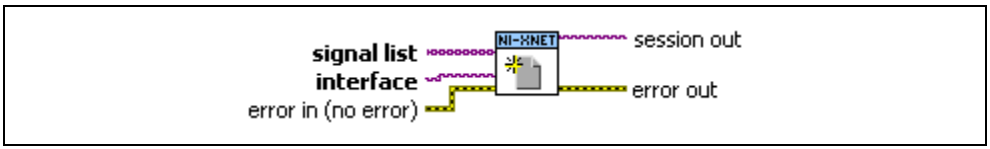
error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (Signal Input Single-Point).vi

Purpose

Creates an XNET session at run time for the Signal Input Single-Point Mode.

Format



Inputs

signal list is the array of XNET Signals to read. These signals are specified in your database and describe the values encoded in one or more frames, or they are trigger signals for frames. For more information about trigger signals, refer to *Signal Input Single-Point Mode*.

interface is the XNET Interface to use for this session.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the created session.

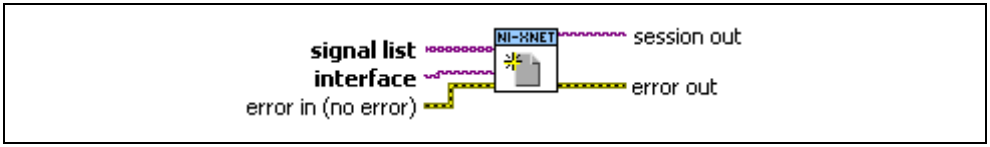
error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (Signal Input Waveform).vi

Purpose

Creates an XNET session at run time for the Signal Input Waveform Mode.

Format



Inputs

signal list is the array of XNET Signals to read. These signals are specified in your database and describe the values encoded in one or more frames.

interface is the XNET Interface to use for this session.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the created session.

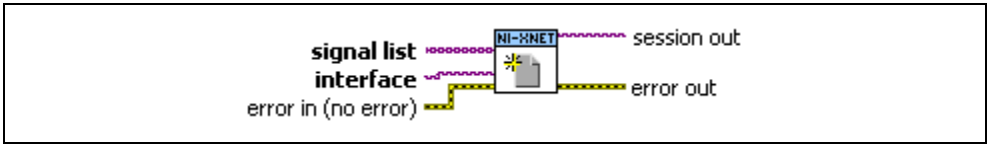
error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (Signal Input XY).vi

Purpose

Creates an XNET session at run time for the Signal Input XY Mode.

Format



Inputs



signal list is the array of XNET Signals to read. These signals are specified in your database and describe the values encoded in one or more frames.



interface is the XNET Interface to use for this session.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the created session.



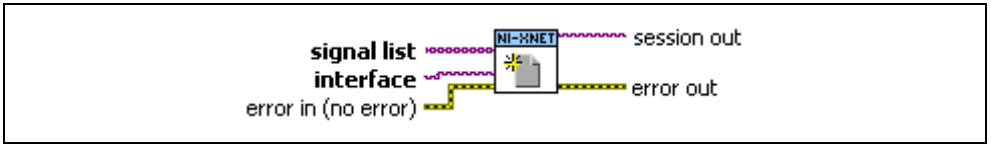
error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (Signal Output Single-Point).vi

Purpose

Creates an XNET session at run time for the Signal Output Single-Point Mode.

Format



Inputs



signal list is the array of XNET Signals to write. These signals are specified in your database and describe the values encoded in one or more frames, or they are trigger signals for frames. For information about trigger signals, refer to *Signal Output Single-Point Mode*.



interface is the XNET Interface to use for this session.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the created session.



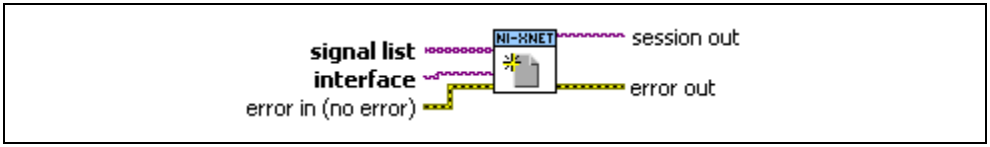
error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (Signal Output Waveform).vi

Purpose

Creates an XNET session at run time for the Signal Output Waveform Mode.

Format



Inputs

signal list is the array of XNET Signals to write. These signals are specified in your database and describe the values encoded in one or more frames.

interface is the XNET Interface to use for this session.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the created session.

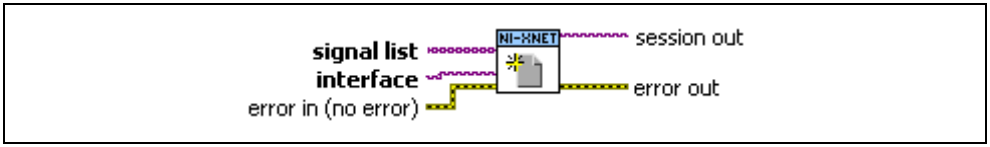
error out is the error cluster output (refer to *Error Handling*).

XNET Create Session (Signal Output XY).vi

Purpose

Creates an XNET session at run time for the Signal Output XY Mode.

Format



Inputs

signal list is the array of XNET Signals to write. These signals are specified in your database and describe the values encoded in one or more frames.

interface is the XNET Interface to use for this session.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the created session.

error out is the error cluster output (refer to *Error Handling*).

XNET Session Property Node

Format



Description

Property node used to read/write properties for an XNET Session I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

Interface Properties

Properties in the Interface category apply to the interface and not the session. If more than one session exists for the interface, changing an interface property affects all the sessions.

CAN Interface Properties

This category includes CAN-specific interface properties.

Interface:CAN:External Transceiver Config

Data Type	Direction	Required?	Default
U32	Write Only	No	0x00000007

Property Class

XNET Session

Short Name

Intf.CAN.ExtTcvrCfg

Description

This property allows you to configure XS series CAN hardware to communicate properly with your external transceiver. The connector on your XS series CAN hardware has five lines for communicating with your transceiver.

Line	Direction	Purpose
Ext_RX	In	Data received from the CAN bus.
Ext_TX	Out	Data to transmit on the CAN bus.
Output0	Out	Generic output used to configure the transceiver mode.
Output1	Out	Generic output used to configure the transceiver mode.
NERR	In	Input to connect to the nERR pin of your transceiver to route status back from the transceiver to the hardware.

The Ext_RX and Ext_TX lines are self explanatory and provide for the transfer of CAN data to and from the transceiver. The remaining three lines are for configuring the transceiver and retrieving status from the transceivers. Not all transceivers use all pins. Typically, a transceiver has one or two lines that can configure the transceiver mode. The NI-XNET driver natively supports five transceiver modes: Normal, Sleep, Single Wire Wakeup, Single Wire High Speed, and Power-On. This property configures how the NI-XNET driver sets the outputs of your external transceiver for each mode.

The configuration is in the form of a u32 written as a bitmask. The u32 bitmask is defined as:

31	30..15	14..12	11..9	8..6	5..3	2..0
nERR Connected	Reserved	PowerOn Configuration	SWHighSpeed Configuration	SWWakeup Configuration	Sleep Configuration	Normal Configuration

Where each configuration is a 3-bit value defined as:

2	1	0
State Supported	Output1 Value	Output0 Value

The Interface:CAN:Transceiver State property changes the transceiver state. Based on the transceiver configuration, if the state is supported, the configuration determines how the two pins are set. If the state is not supported, an error is returned, because you tried to set an invalid configuration. Note that all transceivers must support a Normal state, so the State Supported bit for that configuration is ignored.

Other internal state changes may occur. For example, if you put the transceiver to sleep and a remote wakeup occurs, the transceiver automatically is changed to the normal state. For information about the state machine for the transceiver state, refer to *CAN Transceiver State Machine* in *Additional Topics*.

If nERR Connected is set, the nERR pin into the connector determines a transceiver error. It is active low, meaning a value of 0 on this pin indicates an error. A value of 1 indicates no error. If this line is connected, the NI-XNET driver monitors this line and reports its status via the **Transceiver Error** field of **XNET Read (State CAN Comm).vi**.

Examples

TJA1041 (HS): To connect to the TJA1041 transceiver, connect Output0 to the nSTB pin and Output1 to the EN pin. The TJA1041 does have an nERR pin, so that should be connected to the nERR input. The TJA1041 supports a power-on state, a sleep state, and a normal state. As this is not a single wire transceiver, it does not support any single wire state. For normal operation, the TJA1041 uses a 1 for both nSTB and EN. For sleep, the TJA1041 uses the standby mode, which uses a 0 for both nSTB and EN. For power-on, the TJA1041 uses a 1 for nSTB and a 0 for EN. The final configuration is 0x80005027.

TJA1054 (LS): You can connect and configure the TJA1054 identically to the TJA1041.

AU5790 (SW): To connect to the AU5790 transceiver, connect Output0 to the nSTB pin and Output1 to the EN pin. The AU5790 does not support any transceiver status, so you do not need to connect the nERR pin. The AU5790 supports all states. For normal operation, the AU5790 uses a 1 for both nSTB and EN. For sleep, the AU5790 uses a 0 for both nSTB and EN. For Single Wire Wakeup, the AU5790 requires nSTB to be a 0 and EN to be a 1. For Single Wire High-Speed, the AU5790 requires nSTB to be a 1, and EN to be a 0. For

power-on, the sleep state is used so there is less interference on the bus. The final configuration is 0x00004DA7.

Interface:CAN:FD Baud Rate

Data Type	Direction	Required?	Default
U32	Read/Write	No	0

Property Class

XNET Session

Short Name

Intf.CAN.FdBaudRate

Description

Note You can modify this property only when the interface is stopped.

The Interface:CAN:FD Baud Rate property sets the fast data baud rate for CAN FD + BRS CAN:I/O Mode. The default value for this interface property is the same as the cluster's FD baud rate in the database. Your application can set this interface FD baud rate to override the value in the database.

When the upper nibble (0xF0000000) is clear, this is a numeric baud rate (for example, 500000).

NI-XNET CAN hardware currently accepts the following numeric baud rates: 200000, 250000, 400000, 500000, 800000, 1000000, 1250000, 1600000, 2000000, 2500000, 4000000, 5000000, and 8000000.

Note Not all CAN transceivers are rated to transmit at the requested rate. If you attempt to use a rate that exceeds the transceiver's qualified rate, XNET Start returns a warning. Chapter 3, *NI-XNET Hardware Overview*, describes the CAN transceivers' limitations.

When the upper nibble is set to 0x8 (that is, 0x80000000), the remaining bits provide fields for more custom CAN communication baud rate programming. The fields are shown in the following table:

	31..28	27..26	25..24	23..20	19..16	15...10	9..8	7..0
Normal	b0000	Baud Rate (200 k–8 M)						
Custom	b1000	Res	SJW (0–3)	TSEG2 (0–7)	TSEG1 (1–15)	Res	Tq (25–800)	

- (Re-)Synchronization Jump Width (SJW)
  - Valid programmed values are 0–3.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time Segment 2 (TSEG2) is the time segment after the sample point.
  - Valid programmed values are 0–7.
  - This is the Phase_Seg2(D) from *Bosch's CAN with Flexible Data-Rate* specification, version 1.0.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time Segment 1 (TSEG1) is the time segment before the sample point.
  - Valid programmed values are 1–15.
  - This is the combination of Prop_Seg(D) and Phase_Seg1(D) from Bosch's *CAN with Flexible Data-Rate* specification, version 1.0.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time quantum (Tq) is used to program the baud rate prescaler.
  - Valid programmed values are 25–800, in increments of 25 ns.

Interface:CAN:I/O Mode

Data Type	Direction	Required?	Default
	Read Only	—	Same as XNET Cluster CAN:I/O Mode

Property Class

XNET Session

Short Name

Intf.CAN.IoMode

Description

This property indicates the I/O Mode the interface is using. It is a ring of three values, as described in the following table:

Enumeration	Value	Meaning
CAN	0	This is the default CAN 2.0 A/B standard I/O mode as defined in ISO 11898-1:2003. A fixed baud rate is used for transfer, and the payload length is limited to 8 bytes.
CAN FD	1	This is the CAN FD mode as specified in the *CAN with Flexible Data-Rate* specification, version 1.0. Payload lengths are allowed up to 64 bytes, but they are transmitted at a single fixed baud rate (defined by XNET Cluster Baud Rate or Interface:Baud Rate.)
CAN FD + BRS	2	This is the CAN FD mode as specified in the *CAN with Flexible Data-Rate* specification, version 1.0, with the optional Baud Rate Switching enabled. The same payload lengths as CAN FD mode are allowed; additionally, the data portion of the CAN frame is transferred at a different (higher) baud rate (defined by XNET Cluster CAN:FD Baud Rate or Interface:CAN:FD Baud Rate).

The value is initialized from the database cluster when the session is created and cannot be changed later. However, you can transmit standard CAN frames on a CAN FD network. Refer to the Interface:CAN:Transmit I/O Mode property.

Interface:CAN:Listen Only?

Data Type	Direction	Required?	Default
TF	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.CAN.LstnOnly?

Description

Note You can modify this property only when the interface is stopped.

The Listen Only? property configures whether the CAN interface transmits any information to the CAN bus.

When this property is false, the interface can transmit CAN frames and acknowledge received CAN frames.

When this property is true, the interface can neither transmit CAN frames nor acknowledge a received CAN frame. The true value enables passive monitoring of network traffic, which can be useful for debugging scenarios when you do not want to interfere with a communicating network cluster.

Interface:CAN:Pending Transmit Order

Data Type	Direction	Required?	Default
U32	Read/Write	No	As Submitted

Property Class

XNET Session

Short Name

Intf.CAN.PendTxOrder

Description

Note You can modify this property only when the interface is stopped.

Note Setting this property causes the internal queue to be flushed. If you start a session, queue frames, and then stop the session and change this mode, some frames may be lost. Set this property to the desired value once; do not constantly change modes.

The Pending Transmit Order property configures how the CAN interface manages the internal queue of frames. More than one frame may desire to transmit at the same time. NI-XNET stores the frames in an internal queue and transmits them onto the CAN bus when the bus is idle.

This property modifies how NI-XNET handles this queue of frames. The following table lists the accepted values:

Enumeration	Value
As Submitted	0
By Identifier	1

When you configure this property to be As Submitted, frames are transmitted in the order that they were submitted into the queue. There is no reordering of any frames, and a higher priority frame may be delayed due to the transmission or retransmission of a previously submitted frame. However, this mode has the highest performance.

When you configure this property to be By Identifier, frames with the highest priority identifier (lower CAN ID value) transmit first. The frames are stored in a priority queue sorted by ID. If a frame currently being transmitted requires retransmission (for example, it lost arbitration or failed with a bus error), and a higher priority frame is queued in the meantime,

the lower priority frame is not immediately retried, but the higher priority frame is transmitted instead. In this mode, you can emulate multiple ECUs and still see a behavior similar to a real bus in that the highest priority message is transmitted on the bus. This mode may be slower in performance (possible delays between transmissions as the queue is re-evaluated), and lower priority messages may be delayed indefinitely due to frequent high-priority messages.

Interface:CAN:Single Shot Transmit?

Data Type	Direction	Required?	Default
TF	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.CAN.SingShot?

Description

Note You can modify this property only when the interface is stopped.

Note Setting this property causes the internal queue to be flushed. If you start a session, queue frames, and then stop the session and change this mode, some frames may be lost. Set this property to the desired value once; do not constantly change modes.

The Single Shot Transmit? property configures whether the CAN interface retries failed transmissions.

When this property is false, failed transmissions retry as specified by the CAN protocol (ISO 11898–1, 6.11 *Automatic Retransmission*). If a CAN frame is not transmitted successfully, the interface attempts to retransmit the frame as soon as the bus is idle again. This retransmit process continues until the frame is successfully transmitted.

When this property is true, failed transmissions do not retry. If a CAN frame is not transmitted successfully, no further transmissions are attempted.

Interface:CAN:Termination

Data Type	Direction	Required?	Default
U32	Read/Write	No	Off (0)

Property Class

XNET Session

Short Name

Intf.CAN.Term

Description

Note You can modify this property only when the interface is stopped.

This property does not take effect until the interface is started.

The Termination property configures the onboard termination of the NI-XNET interface CAN connector (port). The enumeration is generic and supports two values: Off and On. However, different CAN hardware has different termination requirements, and the Off and On values have different meanings, as described below.

High-Speed CAN

High-Speed CAN networks are typically terminated on the bus itself instead of within a node. However, NI-XNET allows you to configure termination within the node to simplify testing. If your bus already has the correct amount of termination, leave this property in the default state of Off. However, if you require termination, set this property to On.

Value	Meaning	Description
Off	Disabled	Termination is disabled.
On	Enabled	Termination (120 Ω) is enabled.

Low-Speed/Fault-Tolerant CAN

Every node on a Low-Speed CAN network requires termination for each CAN data line (CAN_H and CAN_L). This configuration allows the Low-Speed/Fault-Tolerant CAN port to provide fault detection and recovery. Refer to *Termination* for more information about low-speed termination. In general, if the existing network has an overall network termination of 125 Ω or less, turn on termination to enable the 4.99 kΩ option. Otherwise, you should select the default 1.11 kΩ option.

Value	Meaning	Description
Off	1.11 kΩ	Termination is set to 1.11 kΩ.
On	4.99 kΩ	Termination is set to 4.99 kΩ.

Single Wire CAN

The ISO standard requires single wire transceivers to have a 9.09 kΩ resistor, and no additional configuration is supported.

Interface:CAN:Transceiver State

Data Type	Direction	Required?	Default
U32	Read/Write	No	Normal (0)

Property Class

XNET Session

Short Name

Intf.CAN.TcvrState

Description

The Transceiver State property configures the CAN transceiver and CAN controller modes. The transceiver state controls whether the transceiver is asleep or communicating, as well as configuring other special modes. The following table lists the accepted values.

Enumeration	Value
Normal	0
Sleep	1
Single Wire Wakeup	2
Single Wire High-Speed	3

Normal

This state sets the transceiver to normal communication mode. If the transceiver is in the Sleep mode, this performs a local wakeup of the transceiver and CAN controller chip.

Sleep

This state sets the transceiver and CAN controller chip to Sleep (or standby) mode. You can set the interface to Sleep mode only while the interface is communicating. If the interface has not been started, setting the transceiver to Sleep mode returns an error.

Before going to sleep, all pending transmissions are transmitted onto the CAN bus. Once all pending frames have been transmitted, the interface and transceiver go into Sleep (or standby) mode. Once the interface enters Sleep mode, further communication is not possible until a wakeup occurs. The transceiver and CAN controller wake from Sleep mode when either a local wakeup or remote wakeup occurs.

A local wakeup occurs when the application sets the transceiver state to either Normal or Single Wire Wakeup.

A remote wakeup occurs when a remote node transmits a CAN frame (referred to as the wakeup frame). The wakeup frame wakes up the NI-XNET interface transceiver and CAN controller chip. The CAN controller chip does not receive or acknowledge the wakeup frame. After detecting the wakeup frame and idle bus, the CAN interface enters Normal mode.

When the local or remote wakeup occurs, frame transmissions resume from the point at which the original Sleep mode was set.

You can use **XNET Read (State CAN Comm).vi** to detect when a wakeup occurs. To suspend the application while waiting for the remote wakeup, use **XNET Wait (CAN Remote Wakeup).vi**.

Single Wire Wakeup

For a remote wakeup to occur for Single Wire transceivers, the node that transmits the wakeup frame first must place the network into the Single Wire Wakeup Transmission mode by asserting a higher voltage.

This state sets a Single Wire transceiver into the Single Wire Wakeup Transmission mode, which forces the Single Wire transceiver to drive a higher voltage level on the network to wake up all sleeping nodes. Other than this higher voltage, this mode is similar to Normal mode. CAN frames can be received and transmitted normally.

If you are not using a Single Wire transceiver, setting this state returns an error. If your current mode is Single Wire High-Speed, setting this mode returns an error because you are not allowed to wake up the bus in high-speed mode.

The application controls the timing of how long the wakeup voltage is driven. The application typically changes to Single Wire Wakeup mode, transmits a single wakeup frame, and then returns to Normal mode.

Single Wire High-Speed

This state sets a Single Wire transceiver into Single Wire High-Speed Communication mode. If you are not using a Single Wire transceiver, setting this state returns an error.

Single Wire High-Speed Communication mode disables the transceiver's internal waveshaping function, allowing the SAE J2411 High Speed baud rate of 83.333 kbytes/s to be used. The disadvantage versus Single Wire Normal Communication mode, which only allows the SAE J2411 baud rate of 33.333 kbytes/s, is degraded EMC performance. Other than the disabled waveshaping, this mode is similar to Normal mode. CAN frames can be received and transmitted normally.

This mode has no relationship to High-Speed transceivers. It is merely a higher speed mode of the Single Wire transceiver, typically used to download data when the onboard network is attached to an offboard tester ECU.

The Single Wire transceiver does not support use of this mode in conjunction with Sleep mode. For example, a remote wakeup cannot transition from sleep to this Single Wire High-Speed mode. Therefore, setting the mode to Sleep from Single Wire High-Speed mode returns an error.

Interface:CAN:Transceiver Type

Data Type	Direction	Required?	Default
	Read/Write	No	High-Speed (0) for High-Speed and XS Hardware; Low-Speed (1) for Low-Speed Hardware

Property Class

XNET Session

Short Name

Intf.CAN.TcvrType

Description



Notes You can modify this property only when the interface is stopped.

For XNET hardware that provides a software-selectable transceiver, the Transceiver Type property allows you to set the transceiver type. Use the XNET Interface CAN.Transceiver Capability property to determine whether your hardware supports a software-selectable transceiver.

You also can use this property to determine the currently configured transceiver type. The following table lists the accepted values:

Enumeration	Value
High-Speed (HS)	0
Low-Speed (LS)	1
Single Wire (SW)	2
External (Ext)	3
Disconnect (Disc)	4

The default value for this property depends on your type of hardware. If you have fixed-personality hardware, the default value is the hardware value. If you have hardware that supports software-selectable transceivers, the default is High-Speed.

This attribute uses the following values:

High-Speed

This configuration enables the High-Speed transceiver. This transceiver supports baud rates of 40 kbaud to 1 Mbaud. When using a High-Speed transceiver, you also can communicate with a CAN FD bus. Refer to Chapter 3, *NI-XNET Hardware Overview*, to determine which CAN FD baud rates are supported.

Low-Speed/Fault-Tolerant

This configuration enables the Low-Speed/Fault-Tolerant transceiver. This transceiver supports baud rates of 40–125 kbaud.

Single Wire

This configuration enables the Single Wire transceiver. This transceiver supports baud rates of 33.333 kbaud and 83.333 kbaud.

External

This configuration allows you to use an external transceiver to connect to your CAN bus. Refer to Interface:CAN:External Transceiver Config for more information.

Disconnect

This configuration allows you to disconnect the CAN controller chip from the connector. You can use this value when you physically change the external transceiver.

Interface:CAN:Transmit I/O Mode

Data Type	Direction	Required?	Default
U32	Read/Write	No	Same as Interface:CAN:I/O Mode

Property Class

XNET Session

Short Name

Intf.CAN.TxIoMode

Description

This property specifies the I/O Mode the interface uses when transmitting a CAN frame. By default, it is the same as the XNET Cluster CAN:I/O Mode property. However, even if the interface is in CAN FD (+ BRS) mode, you can force it to transmit frames in the standard CAN format. For this purpose, set this property to CAN.

Note This property affects only the transmission of frames. Even if you set the transmit I/O mode to CAN, the interface still can receive frames in FD modes (if the XNET Cluster CAN:I/O Mode property is configured in an FD mode).

The Transmit I/O mode may not exceed the mode set by the XNET Cluster CAN:I/O Mode property.

FlexRay Interface Properties

These properties are calculated based on constraints in the *FlexRay Protocol Specification*. To calculate these properties, the constraints use cluster settings and knowledge of the oscillator that the FlexRay interface uses.

At Create Session time, the XNET driver automatically calculates these properties, and they are passed down to the hardware. However, you can use the XNET property node to change these settings.

Note Changing the interface properties can affect the integration and communication of the XNET FlexRay interface with the cluster.

Interface:FlexRay:Accepted Startup Range

Data Type	Direction	Required?	Default
U32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.AccStartRng

Description

Range of measure clock deviation allowed for startup frames during node integration. This property corresponds to the **pdAcceptedStartupRange** node parameter in the *FlexRay Protocol Specification*.

The range for this property is 0–1875 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Allow Halt Due To Clock?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.FlexRay.AlwHltClk?

Description

Controls the FlexRay interface transition to the POC: halt state due to clock synchronization errors. If set to true, the node can transition to the POC: halt state. If set to false, the node does not transition to the POC: halt state and remains in the POC: normal passive state, allowing for self recovery.

This property corresponds to the **pAllowHaltDueToClock** node parameter in the *FlexRay Protocol Specification*.

The property is a Boolean flag.

The default value of this property is false.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Refer to **XNET Read (State FlexRay Comm).vi** for more information about the POC: halt and POC: normal passive states.

Interface:FlexRay:Allow Passive to Active

Data Type	Direction	Required?	Default
	Read/Write	No	0

Property Class

XNET Session

Short Name

Intf.FlexRay.AlwPassAct

Description

Number of consecutive even/odd cycle pairs that must have valid clock correction terms before the FlexRay node can transition from the POC: normal-passive to the POC: normal-active state. If set to zero, the node cannot transition from POC: normal-passive to POC: normal-active.

This property corresponds to the **pAllowPassiveToActive** node parameter in the *FlexRay Protocol Specification*.

The property is expressed as the number of even/odd cycle pairs, with values of 0–31.

The default value of this property is zero.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Refer to **XNET Read (State FlexRay Comm).vi** for more information about the POC: normal-active and POC: normal-passive states.

Interface:FlexRay:Auto Asleep When Stopped

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.AutoAslpStp

Description

This property indicates whether the FlexRay interface (node) automatically places the FlexRay transceiver and controller into sleep when the interface is stopped. The default value of this property is False, and you must handle the wakeup/sleep processing manually using the XNET Session Interface:FlexRay:Sleep property.

When this property is called with the value True while the interface is asleep, the interface is put to sleep immediately. When this property is called with the value False, the interface is set to a local awake state immediately.

If the interface is asleep when **XNET Start.vi** is called, the FlexRay interface waits for a wakeup pattern on the bus before transitioning out of the POC:READY state. To initiate a bus wakeup, you can set the XNET Session Interface:FlexRay:Sleep property with a value of Remote Wake.

After **XNET Stop.vi** is called, if this property is True, the FlexRay interface automatically goes back to sleep to be ready to handle the wakeup on subsequent **XNET Start.vi** calls. When this property is False when **XNET Stop.vi** is called, the FlexRay interface remains in the sleep state it was in prior to the **XNET Stop.vi** call.

You can overwrite the default value by writing this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Cluster Drift Damping

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.ClstDriftDmp

Description

Local cluster drift damping factor used for rate correction.

This property corresponds to the **pAllowPassiveToActive** node parameter in the *FlexRay Protocol Specification*.

The range for the property is 0–20 MT.

The cluster drift damping property should be configured in such a way that the damping values in all nodes within the same cluster have approximately the same duration.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Coldstart?

Data Type	Direction	Required?	Default
	Read	No	False

Property Class

XNET Session

Short Name

Intf.FlexRay.Coldstart?

Description

This property specifies whether the FlexRay interface operates as a coldstart node on the cluster. This property is read only and calculated from the XNET Session Interface:FlexRay:Key Slot Identifier property. If the KeySlot Identifier is 0 (invalid slot identifier), the XNET FlexRay interface does not act as a coldstart node, and this property is false. If the KeySlot Identifier is 1 or more, the XNET FlexRay interface transmits a startup frame from that slot, and the Coldstart? property is true.

This property returns a Boolean flag (true/false).

The default value of this property is false.

Interface:FlexRay:Connected Channels

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.ConnectedChs

Description

This property specifies the channel(s) that the FlexRay interface (node) is physically connected to. The default value of this property is connected to all channels available on the cluster. However, if you are using a node connected to only one channel of a multichannel cluster that uses wakeup, you must set the value properly. If you do not, your node may not wake up, as the wakeup pattern cannot be received on a channel not physically connected.

This property corresponds to the **pChannels** node parameter in the *FlexRay Protocol Specification*.

The values supported for this property (enumeration) are A = 1, B = 2, and A and B = 3.

You can overwrite the default value by writing this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Decoding Correction

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.DecCorr

Description

This property specifies the value that the receiving FlexRay node uses to calculate the difference between the primary time reference point and secondary reference point. The clock synchronization algorithm uses the primary time reference and the sync frame's expected arrival time to calculate and compensate for the node's local clock deviation.

This property corresponds to the **pDecodingCorrection** node parameter in the *FlexRay Protocol Specification*.

The range for the property is 14–143 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Delay Compensation Ch A

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.DelayCompA

Description

This property specifies the value that the XNET FlexRay interface (node) uses to compensate for reception delays on channel A. This takes into account the assumed propagation delay up to the maximum allowed propagation delay (**cPropagationDelayMax**) for microticks in the 0.0125–0.05 range. In practice, you should apply the minimum of the propagation delays of all sync nodes.

This property corresponds to the **pDelayCompensation[A]** node parameter in the *FlexRay Protocol Specification*.

The property range is 0–200 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Delay Compensation Ch B

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.DelayCompB

Description

This property specifies the value that the XNET FlexRay interface (node) uses to compensate for reception delays on channel B. This takes into account the assumed propagation delay up to the maximum allowed propagation delay (**Propagation Delay Max**) for microticks in the 0.0125–0.05 range. In practice, you should apply the minimum of the propagation delays of all sync nodes.

This property corresponds to the **pDelayCompensation[B]** node parameter in the *FlexRay Protocol Specification*.

The property range is 0–200 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Key Slot Identifier

Data Type	Direction	Required?	Default
U32	Read/Write	No	0

Property Class

XNET Session

Short Name

Intf.FlexRay.KeySlotID

Description

This property specifies the FlexRay slot number from which the XNET FlexRay interface transmits a startup frame, during the process of integration with other cluster nodes.

For a network (cluster) of FlexRay nodes to start up for communication, at least two nodes must transmit startup frames. If your application is designed to test only one external ECU, you must configure the XNET FlexRay interface to transmit a startup frame. If the one external ECU does not transmit a startup frame itself, you must use two XNET FlexRay interfaces for the test, each of which must transmit a startup frame.

There are two methods for configuring the XNET FlexRay interface as a coldstart node (transmit startup frame).

Output Session with Startup Frame

Create an output session that contains a startup frame (or one of its signals). The XNET Frame FlexRay:Startup? property is true for a startup frame. If you use this method, this Key Slot Identifier property contains the identifier property of that startup frame. You do not write this property.

Write this Key Slot Identifier Property

This interface uses the identifier (slot) you write to transmit a startup frame using that slot.

Note If you create an output session that contains the startup frame, with the same identifier as that specified in the Key Slot Identifier property, the data you write to the session transmits in the frame. If you do not create an output session that contains the startup frame, the interface transmits a null frame for startup purposes.

If you create an output session that contains a startup frame with an identifier that does not match the Key Slot Identifier property, an error is returned.

The default value of this property is 0 (no startup frame).

You can overwrite the default value by writing an identifier that corresponds to the identifier of a startup frame prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Latest Tx

Data Type	Direction	Required?	Default
U32	Read	No	0

Property Class

XNET Session

Short Name

Intf.FlexRay.LatestTx

Description

This property specifies the number of the last minislot in which a frame transmission can start in the dynamic segment. This is a read-only property, as the FlexRay controller evaluates it based on the configuration of the frames in the dynamic segment.

This property corresponds to the **pLatestTx** node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 0–7981 minislots.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Listen Timeout

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.ListTimo

Description

This property specifies the upper limit for the startup listen timeout and wakeup listen timeout.

Refer to *Summary of the FlexRay Standard* for more information about startup and wakeup procedures within the FlexRay protocol.

This property corresponds to the **pdListenTimeout** node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 1284–1283846 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Macro Initial Offset Ch A

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.MacInitOffA

Description

This property specifies the integer number of macroticks between the static slot boundary and the following macrotick boundary of the secondary time reference point based on the nominal macrotick duration. This property applies only to Channel A.

This property corresponds to the **pMacroInitialOffset[A]** node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 2–72 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Macro Initial Offset Ch B

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.MacInitOffB

Description

This property specifies the integer number of macroticks between the static slot boundary and the following macrotick boundary of the secondary time reference point based on the nominal macrotick duration. This property applies only to Channel B.

This property corresponds to the **pMacroInitialOffset[B]** node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 2–72 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Max Drift

Data Type	Direction	Required?	Default
U32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.MaxDrift

Description

This property specifies the maximum drift offset between two nodes that operate with unsynchronized clocks over one communication cycle.

This property corresponds to the **pdMaxDrift** node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 2–1923 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Micro Initial Offset Ch A

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.MicInitOffA

Description

This property specifies the number of microticks between the closest macrotick boundary described by the Macro Initial Offset Ch A property and the secondary time reference point. This parameter depends on the Delay Compensation property for Channel A, and therefore you must set it independently for each channel.

This property corresponds to the **pMicroInitialOffset[A]** node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 0–240 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Micro Initial Offset Ch B

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.MicInitOffB

Description

This property specifies the number of microticks between the closest macrotick boundary described by the Macro Initial Offset Ch B property and the secondary time reference point. This parameter depends on the Delay Compensation property for Channel B, and therefore you must set it independently for each channel.

This property corresponds to the **pMicroInitialOffset[B]** node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 0–240 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Microtick

Data Type	Direction	Required?	Default
	Read	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.Microtick

Description

This property specifies the duration of a microtick. This property is calculated based on the product of the Samples per Microtick interface property and the BaudRate cluster. This is a read-only property.

This property corresponds to the **pdMicrotick** node parameter in the *FlexRay Protocol Specification*.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Null Frames To Input Stream?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.FlexRay.NullToInStrm?

Description

This property indicates whether the Frame Input Stream Mode session should return FlexRay null frames from **XNET Read.vi**.

When this property uses the default value of false, FlexRay null frames are not returned for a Frame Input Stream Mode session. This behavior is consistent with the other two frame input modes (Frame Input Single-Point Mode and Frame Input Queued Mode), which never return FlexRay null frames from **XNET Read.vi**.

When you set this property to true for a Frame Input Stream Mode session, **XNET Read.vi** returns all FlexRay null frames that are received by the interface. This feature is used to monitor all frames that occur on the network, regardless of whether new payload is available or not. When you use **XNET Read (Frame FlexRay).vi** instance of **XNET Read.vi**, each frame's type field indicates a null frame.

You can overwrite the default value prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Offset Correction

Data Type	Direction	Required?	Default
	Read	No	N/A

Property Class

XNET Session

Short Name

Intf.FlexRay.OffCorr

Description

This property provides the maximum permissible offset correction value, expressed in microticks. The offset correction synchronizes the cycle start time. The value indicates the number of microticks added or subtracted to the offset correction portion of the network idle time, to synchronize the interface to the FlexRay network. The value is returned as a signed 32-bit integer (I32). The offset correction value calculation takes place every cycle, but the correction is applied only at the end of odd cycles. This is a read-only property.

This property can be read anytime prior to closing the FlexRay interface.

Interface:FlexRay:Offset Correction Out

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.OffCorrOut

Description

This property specifies the magnitude of the maximum permissible offset correction value. This node parameter is based on the value of the maximum offset correction for the specific cluster.

This property corresponds to the **pOffsetCorrectionOut** node parameter in the *FlexRay Protocol Specification*.

The value range for this property is 5–15266 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Rate Correction

Data Type	Direction	Required?	Default
	Read	No	N/A

Property Class

XNET Session

Short Name

Intf.FlexRay.RateCorr

Description

Read-only property that provides the rate correction value, expressed in microticks. The rate correction synchronizes frequency. The value indicates the number of microticks added to or subtracted from the configured number of microticks in a cycle, to synchronize the interface to the FlexRay network.

The value is returned as a signed 32-bit integer (I32). The rate correction value calculation takes place in the static segment of an odd cycle, based on values measured in an even-odd double cycle.

This property can be read prior to closing the FlexRay interface.

Interface:FlexRay:Rate Correction Out

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.RateCorrOut

Description

This property specifies the magnitude of the maximum permissible rate correction value. This node parameter is based on the value of the maximum rate correction for the specific cluster. This property corresponds to the **pRateCorrectionOut** node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 2–1923 MT.

This property is calculated from the microticks per cycle and clock accuracy.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Samples Per Microtick

Data Type	Direction	Required?	Default
	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Short Name

Intf.FlexRay.SampPerMicro

Description

This property specifies the number of samples per microtick.

There is a defined relationship between the “ticks” of the microtick timebase and the sample ticks of bit sampling. Specifically, a microtick consists of an integral number of samples.

As a result, there is a fixed phase relationship between the microtick timebase and the sample clock ticks.

This property corresponds to the **pSamplesPerMicrotick** node parameter in the *FlexRay Protocol Specification*.

The supported values for this property are 1, 2, and 4 samples.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Single Slot Enabled?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.FlexRay.SingSlotEn

Description

This property serves as a flag to indicate whether the FlexRay interface (node) should enter single slot mode following startup.

This Boolean property supports a strategy to limit frame transmissions following startup to a single frame (designated by the XNET Session Interface:FlexRay:Key Slot Identifier property). If you leave this property false prior to start (default), all configured output frames transmit. If you set this property to true prior to start, only the key slot transmits. After the interface is communicating (integrated), you can set this property to false at runtime to enable the remaining transmissions (the protocol's ALL_SLOTS command). After the interface is communicating, you cannot set this property from false to true.

This property corresponds to the **pSingleSlotEnabled** node parameter in the *FlexRay Protocol Specification*.

You can overwrite the default value prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Sleep

Data Type	Direction	Required?	Default
	Write Only	No	N/A

Property Class

XNET Session

Short Name

Intf.FlexRay.Sleep

Description

Use the Sleep property to change the NI-XNET FlexRay interface sleep/awake state and optionally to initiate a wakeup on the FlexRay cluster.

The property is a ring (enumerated list) with the following values:

String	Value	Description
Local Sleep	0	Set interface and transceiver(s) to sleep
Local Wake	1	Set interface and transceiver(s) to awake
Remote Wake	2	Set interface and transceivers to awake and attempt to wake up the FlexRay bus by sending the wakeup pattern on the configured wakeup channel

This property is write only. Setting a new value is effectively a request, and the property node returns before the request is complete. To detect the current interface sleep/wake state, use **XNET Read (State FlexRay Comm).vi**.

The FlexRay interface maintains a state machine to determine the action to perform when this property is set (request). The following table specifies the sleep/wake action on the FlexRay interface.

Request	Current Local State	
	Sleep	**Awake**
Local Sleep	No action	Change local state
Local Wake	Attempt to integrate with the bus (move from POC:READY to POC:NORMAL)	No action
Remote Wake	Attempt to wake up the bus followed by an attempt to integrate with the bus (move from POC:READY to POC:NORMAL ACTIVE). If the interface is not yet started, setting Remote Wake schedules a remote wake to be generated once the interface has started.	No action

Interface:FlexRay:Statistics Enabled?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.FlexRay.StatisticsEn?

Description

This XNET Boolean property enables reporting FlexRay error statistics. When this property is false (default), calls to **XNET Read (State FlexRay Statistics).vi** always return zero for each statistic. To enable FlexRay statistics, set this property to true in your application.

You can overwrite the default value prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Symbol Frames To Input Stream?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.FlexRay.SymToInStrm?

Description

This property indicates whether the Frame Input Stream Mode session should return FlexRay symbols from **XNET Read.vi**.

When this property uses the default value of False, FlexRay symbols are not returned for a Frame Input Stream Mode session. This behavior is consistent with the other two frame input modes (Frame Input Single-Point Mode and Frame Input Queued Mode), which never return FlexRay symbols from **XNET Read.vi**.

When you set this property to true for a Frame Input Stream Mode session, **XNET Read.vi** returns all FlexRay symbols the interface receives. This feature detects wakeup symbols and Media Access Test Symbols (MTS). When you use the **XNET Read (Frame FlexRay).vi** instance of **XNET Read.vi**, each frame type field indicates a symbol.

When the frame type is FlexRay Symbol, the first payload byte (offset 0) specifies the type of symbol: 0 for MTS or 1 for wakeup. The frame payload length is 1 or higher, with bytes beyond the first 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).

You can overwrite the default value prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Sync Frames Channel A Even

Data Type	Direction	Required?	Default
	Read	No	N/A

Property Class

XNET Session

Short Name

Intf.FlexRay.SyncChAEven

Description

This property returns an array of sync frames (slot IDs) transmitted or received on channel A during the last even cycle. This read-only property returns an array in which each element holds the slot ID of a sync frame. If the interface is not started, this returns an empty array. If you start the interface, but it fails to communicate (integrate), this property may be helpful in diagnosing the problem.

Refer to Appendix B, *Summary of the FlexRay Standard*, for more information about the FlexRay protocol startup procedure.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Sync Frames Channel A Odd

Data Type	Direction	Required?	Default
	Read	No	N/A

Property Class

XNET Session

Short Name

Intf.FlexRay.SyncChAOdd

Description

This property returns an array of sync frames (slot IDs) transmitted or received on channel A during the last odd cycle. This read-only property returns an array in which each element holds the slot ID of a sync frame. If the interface is not started, this returns an empty array. If you start the interface, but it fails to communicate (integrate), this property may be helpful in diagnosing the problem.

Refer to Appendix B, *Summary of the FlexRay Standard*, for more information about the FlexRay protocol startup procedure.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Sync Frames Channel B Even

Data Type	Direction	Required?	Default
	Read	No	N/A

Property Class

XNET Session

Short Name

Intf.FlexRay.SyncChBEven

Description

This property returns an array of sync frames (slot IDs) transmitted or received on channel B during the last even cycle. This read-only property returns an array in which each element holds the slot ID of a sync frame. If the interface is not started, this returns an empty array. If you start the interface, but it fails to communicate (integrate), this property may be helpful in diagnosing the problem.

Refer to Appendix B, *Summary of the FlexRay Standard*, for more information about the FlexRay protocol startup procedure.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Sync Frames Channel B Odd

Data Type	Direction	Required?	Default
	Read	No	N/A

Property Class

XNET Session

Short Name

Intf.FlexRay.SyncChBOdd

Description

This property returns an array of sync frames (slot IDs) transmitted or received on channel B during the last odd cycle. This read-only property returns an array in which each element holds the slot ID of a sync frame. If the interface is not started, this returns an empty array. If you start the interface, but it fails to communicate (integrate), this property may be helpful in diagnosing the problem.

Refer toAppendix B, *Summary of the FlexRay Standard*, for more information about the FlexRay protocol startup procedure.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Sync Frame Status

Data Type	Direction	Required?	Default
	Read	No	N/A

Property Class

XNET Session

Short Name

Intf.FlexRay.SyncStatus

Description

This property returns the status of sync frames since the interface (enumeration) start. *Within Limits* means the number of sync frames is within the protocol's limits since the interface start. *Below Minimum* means that in at least one cycle, the number of sync frames was below the limit the protocol requires (2 or 3, depending on number of nodes). *Overflow* means that in at least one cycle, the number of sync frames was above the limit set by the XNET Cluster FlexRay:Sync Node Max property. *Both Min and Max* means that both minimum and overflow errors have occurred (this is unlikely).

If the interface is not started, this property returns Within Limits. If you start the interface, but it fails to communicate (integrate), this property may be helpful in diagnosing the problem.

Refer to Appendix B, *Summary of the FlexRay Standard*, for more information about the FlexRay protocol startup and cluster integration procedure.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Termination

Data Type	Direction	Required?	Default
U32	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.FlexRay.Term

Description

This property controls termination at the NI-XNET interface (enumeration) connector (port). This applies to both channels (A and B) on each FlexRay interface. False means the interface is not terminated (default). True means the interface is terminated.

You can overwrite the default value by writing this property prior to starting the FlexRay interface (refer to *Session States* for more information). You can start the FlexRay interface by calling **XNET Start.vi** with **scope** set to either **Normal** or **Interface Only** on the session.

Interface:FlexRay:Wakeup Channel

Data Type	Direction	Required?	Default
U32	Read/Write	No	A

Property Class

XNET Session

Short Name

Intf.FlexRay.WakeupCh

Description

This property specifies the channel the FlexRay interface (node) uses to send a wakeup pattern. This property is used only when the XNET Session Interface:FlexRay:Sleep property is set to Remote Wake.

This property corresponds to the **pWakeupChannel** node parameter in the *FlexRay Protocol Specification*.

The values supported for this property (enumeration) are A = 0 and B = 1.

You can overwrite the default value by writing this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Wakeup Pattern

Data Type	Direction	Required?	Default
	Read/Write	No	2

Property Class

XNET Session

Short Name

Intf.FlexRay.WakeupPtrn

Description

This property specifies the number of repetitions of the wakeup symbol that are combined to form a wakeup pattern when the FlexRay interface (node) enters the POC:wakeup-send state. The POC:wakeup send state is one of the FlexRay controller state transitions during the wakeup process. In this state, the controller sends the wakeup pattern on the specified Wakeup Channel and checks for collisions on the bus.

This property corresponds to the **pWakeupPattern** node parameter in the *FlexRay Protocol Specification*.

The supported values for this property are 2–63.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

LIN Interface Properties

This category includes LIN-specific interface properties.

Properties in the Interface category apply to the interface and not the session. If more than one session exists for the interface, changing an interface property affects all the sessions.

Interface:LIN:Break Length

Data Type	Direction	Required?	Default
	Read/Write	No	13

Property Class

XNET Session

Short Name

Intf.LIN.BreakLen

Description

This property determines the length of the serial break used at the start of a frame header (schedule entry). The value is specified in bit-times.

The valid range is 10–36 (inclusive). The default value is 13, which is the value the LIN standard specifies.

At baud rates below 9600, the upper limit may be lower than 36 to avoid violating hold times for the bus. For example, at 2400 baud, the valid range is 10–14.

This property is applicable only when the interface is the master.

Interface:LIN:DiagP2min

Data Type	Direction	Required?	Default
	Read/Write	No	0.05

Property Class

XNET Session

Short Name

Intf.LIN.DiagP2min

Description

When the interface is the slave, this is the minimum time in seconds between reception of the last frame of the diagnostic request message and transmission of the response for the first frame in the diagnostic response message by the slave.

This property applies only to the interface as slave. An attempt to write the property for interface as master results in error nxErrInvalidPropertyValue being reported.

Interface:LIN:DiagSTmin

Data Type	Direction	Required?	Default
	Read/Write	No	0

Property Class

XNET Session

Short Name

Intf.LIN.DiagSTmin

Description

When the interface is the slave, this property sets the minimum time in seconds it places between the end of transmission of a frame in a diagnostic response message and the start of transmission of the response for the next frame in the diagnostic response message.

When the interface is the master, this property sets the minimum time in seconds it places between the end of transmission of a frame in a diagnostic request message and the start of transmission of the next frame in the diagnostic request message.

Interface:LIN:Master?

Data Type	Direction	Required?	Default
TF	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.LIN.Master?

Description

Note You can set this property only when the interface is stopped.

This Boolean property specifies the NI-XNET LIN interface role on the network: master (true) or slave (false).

In a LIN network (cluster), there always is a single ECU in the system called the master. The master transmits a schedule of frame headers. Each frame header is a remote request for a specific frame ID. For each header, typically a single ECU in the network (slave) responds by transmitting the requested ID payload. The master ECU can respond to a specific header as well, and thus the master can transmit payload data for the slave ECUs to receive. For more information, refer to Appendix C, *Summary of the LIN Standard*.

The default value for this property is false (slave). This means that by default, the interface does not transmit frame headers onto the network. When you use input sessions, you read frames that other ECUs transmit. When you use output sessions, the NI-XNET interface waits for the remote master to send a header for a frame in the output sessions, then the interface responds with data for the requested frame.

If you call **XNET Write (State LIN Schedule Change).vi** to request execution of a schedule, that implicitly sets this property to true (master). You also can set this property to true using a property node, but no schedule is active by default, so you still must call **XNET Write (State LIN Schedule Change).vi** at some point to request a specific schedule.

Regardless of this property's value, you use can input and output sessions. This property specifies which hardware transmits the scheduled frame headers: NI-XNET (true) or a remote master ECU (false).

Interface:LIN:Output Stream Slave Response List By NAD

Data Type	Direction	Required?	Default
	Read/Write	No	Empty Array

Property Class

XNET Session

Short Name

Intf.LIN.OutStrmSlvRspListByNAD

Description

The Output Stream Slave Response List by NAD property provides a list of NADs for use with the replay feature (Interface:Output Stream Timing property set to Replay Exclusive or Replay Inclusive).

For LIN, the array of frames to replay might contain multiple slave response frames, each with the same slave response identifier, but each having been transmitted by a different slave (per the NAD value in the data payload). This means that processing slave response frames for replay requires two levels of filtering. First, you can include or exclude the slave response frame or ID for replay using Interface:Output Stream List or Interface:Output Stream List By ID. If you do not include the slave response frame or ID for replay, no slave responses are transmitted. If you do include the slave response frame or ID for replay, you can use the Output Stream Slave Response List by NAD property to filter which slave responses (per the NAD values in the array) are transmitted. This property is always inclusive, regardless of the replay mode (inclusive or exclusive). If the NAD is in the list and the response frame or ID has been enabled for replay, any slave response for that NAD is transmitted. If the NAD is not in the list, no slave response for that NAD is transmitted. The property's data type is an array of unsigned 32-bit integer (u32). Currently, only byte 0 is required to hold the NAD value. The remaining bits are reserved for future use.

Interface:LIN:Schedules

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Session

Short Name

Intf.LIN.Schedules

Description

This property provides the list of schedules for use when the NI-XNET LIN interface acts as a master (Interface:LIN:Master? is true). When the interface is master, you can wire one of these schedules to **XNET Write (State LIN Schedule Change).vi** to request a schedule change.

When the interface is slave, you cannot control the schedule, and **XNET Write (State LIN Schedule Change).vi** returns an error if it cannot set the interface into master mode (for example, if the interface already is started).

This array of XNET LIN Schedule I/O names is the same list as the XNET Cluster LIN:Schedules property used to configure the session.

Interface:LIN:Sleep

Data Type	Direction	Required?	Default
	Write Only	No	N/A

Property Class

XNET Session

Short Name

Intf.LIN.Sleep

Description

Use the Sleep property to change the NI-XNET LIN interface sleep/awake state and optionally to change remote node (ECU) sleep/awake states.

The property is a ring (enumerated list) with the following values:

String	Value	Description
Remote Sleep	0	Set interface to sleep locally and transmit sleep requests to remote nodes
Remote Wake	1	Set interface to awake locally and transmit wakeup requests to remote nodes
Local Sleep	2	Set interface to sleep locally and not to interact with the network
Local Wake	3	Set interface to awake locally and not to interact with the network

The property is write only. Setting a new value is effectively a request, and the property node returns before the request is complete. To detect the current interface sleep/wake state, use **XNET Read (State LIN Comm).vi**.

The LIN interface maintains a state machine to determine the action to perform when this property is set (request). The following sections specify the action when the interface is master and slave.

Table 4-1. Sleep/Wake Action for Master

Request	Current Local State	
	Sleep	**Awake**
Remote Sleep	No action	Change local state; pause scheduler; transmit go-to-sleep request frame
Remote Wake	Change local state; transmit master wakeup pattern (serial break); resume scheduler	No action
Local Sleep	No action	Change local state
Local Wake	Change local state; resume scheduler	No action

When the master's scheduler pauses, it finishes the pending entry (slot) and saves its current position. When the master's scheduler resumes, it continues with the schedule where it left off (entry after the pause).

The go-to-sleep request is frame ID 63, payload length 8, payload byte 0 has the value 0, and the remaining bytes have the value 0xFF.

If the master is in the Sleep state, and a remote slave (ECU) transmits the slave wakeup pattern, this is equivalent to setting this property to Local Wake. In addition, a pending **XNET Wait (LIN Remote Wakeup).vi** returns. This **XNET Wait** VI does not apply to setting this property, because you know when you set it.

Table 4-2. Sleep/Wake Action for Slave

Request	Current Local State	
	Sleep	**Awake**
Remote Sleep	Error	Error
Remote Wake	Change local state; transmit slave wakeup pattern	No action
Local Sleep	No action	Change local state
Local Wake	Change local state	No action

According to the LIN protocol standard, Remote Sleep is not supported for slave mode, so that request returns an error.

If the slave is in Sleep state, and a remote master (ECU) transmits the master wakeup pattern, this is equivalent to setting this property to Local Wake. In addition, a pending **XNET Wait (LIN Remote Wakeup).vi** returns. This **XNET Wait** VI does not apply to setting this property, because you know when you set it.

Interface:LIN:Start Allowed without Bus Power?

Data Type	Direction	Required?	Default
TF	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.LIN.StrtWoPwr?

Description

Note You can modify this property only when the interface is stopped.

The Start Allowed Without Bus Power? property configures whether the LIN interface does not check for bus power present at interface start, or checks and reports an error if bus power is missing.

When this property is true, the LIN interface does not check for bus power present at start, so no error is reported if the interface is started without bus power.

When this property is false, the LIN interface checks for bus power present at start, and nxErrMissingBusPower is reported if the interface is started without bus power.

Interface:LIN:Termination

Data Type	Direction	Required?	Default
U32	Read/Write	No	Off (0)

Property Class

XNET Session

Short Name

Intf.LIN.Term

Description

Notes You can modify this property only when the interface is stopped.

This property does not take effect until the interface is started.

The Termination property configures the NI-XNET interface LIN connector (port) onboard termination. The enumeration is generic and supports two values: Off (disabled) and On (enabled).

The property is a ring (enumerated list) with the following values:

String	Value
Off	0
On	1

Per the LIN 2.1 standard, the Master ECU has a ~1 kΩ termination resistor between Vbat and Vbus. Therefore, use this property only if you are using your interface as the master and do not already have external termination.

For more information about LIN cabling and termination, refer to *NI-XNET LIN Hardware*.

Source Terminal Interface Properties

This category includes properties to route trigger signals between multiple DAQmx and XNET devices.

Interface:Source Terminal:Start Trigger

Data Type	Direction	Required?	Default
	Read/Write	No	(Disconnected)

Property Class

XNET Session

Short Name

Intf.SrcTerm.StartTrigger

Description

This property specifies the name of the internal terminal to use as the interface Start Trigger. The data type is NI Terminal (DAQmx terminal).

This property is supported for C Series modules in a CompactDAQ chassis. It is not supported for CompactRIO, PXI, or PCI (refer to **XNET Connect Terminals.vi** for those platforms).

The digital trigger signal at this terminal is for the Start Interface transition, to begin communication for all sessions that use the interface. This property routes the start trigger, but not the timebase (used for timestamp of received frames and cyclic transmit of frames). Timebase routing is not required for CompactDAQ, because all modules in the chassis automatically use a shared timebase.

Use this property to connect the interface Start Trigger to triggers in other modules and/or interfaces. When you read this property, you specify the interface Start Trigger as the source of a connection. When you write this property, you specify the interface Start Trigger as the destination of a connection, and the value you write represents the source. For examples that demonstrate use of this property to synchronize NI-XNET and NI-DAQmx hardware, refer to the **Synchronization** category within the NI-XNET examples.

The connection this property creates is disconnected when you clear (close) all sessions that use the interface.

Interface:Baud Rate

Data Type	Direction	Required?	Default
U32	Read/Write	Yes (If Not in Database)	0 (If Not in Database)

Property Class

XNET Session

Short Name

Intf.BaudRate

Description

Note You can modify this property only when the interface is stopped.

The Interface:Baud Rate property sets the CAN, FlexRay, or LIN interface baud rate. The default value for this interface property is the same as the cluster's baud rate in the database. Your application can set this interface baud rate to override the value in the database, or when no database is used.

CAN

When the upper nibble (0xF0000000) is clear, this is a numeric baud rate (for example, 500000).

NI-XNET CAN hardware currently accepts the following numeric baud rates: 33333, 40000, 50000, 62500, 80000, 83333, 100000, 125000, 160000, 200000, 250000, 400000, 500000, 800000, and 1000000.

Note The 33333 baud rate is supported with single-wire transceivers only.

Note Baud rates greater than 125000 are supported with high-speed transceivers only.

When the upper nibble is set to 0x8 (that is, 0x80000000), the remaining bits provide fields for more custom CAN communication baud rate programming. Additionally, if the upper nibble is set to 0xC (that is, 0xC0000000), the remaining bits provide fields for higher-precision custom CAN communication baud rate programming. The higher-precision

bit timings facilitate connectivity to a CAN FD cluster. The baud rate models are shown in the following table:

	31..28	27..26	25..24	23	22..20	19..16	15..14	13..12	11..8	7..4	3..0
Normal	b0000	Baud Rate (33.3 k–1 M)									
Custom	b1000	Res	SJW (0–3)	TSEG2 (0–7)		TSEG1 (1–15)	Res	Tq (125–0x3200)			
High Precision	b1100	SJW (0–15)		TSEG2 (0–15)		TSEG1 (1–63)		Tq (25–0x3200)			

- (Re-)Synchronization Jump Width (SJW)
  - Valid programmed values are 0–3 in normal custom mode and 0–15 in high-precision custom mode.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time Segment 2 (TSEG2), which is the time segment after the sample point
  - Valid programmed values are 0–7 in normal custom mode and 0–15 in high-precision custom mode.
  - This is the Phase_Seg2 time from ISO 11898–1, 12.4.1 *Bit Encoding/Decoding*.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time Segment 1 (TSEG1), which is the time segment before the sample point
  - Valid programmed values are 1–0xF (1–15 decimal) in normal custom mode and 1–0x3F (1–63 decimal) in high-precision custom mode.
  - This is the combination of the Prop_Seg and Phase_Seg1 time from ISO 11898–1, 12.4.1 *Bit Encoding/Decoding*.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time quantum (Tq), which is used to program the baud rate prescaler
  - Valid programmed values are 125–12800, in increments of 0x7D (125 decimal) ns for normal custom mode and 25–12800, in increments of 0x19 (25 decimal) ns for high-precision custom mode.
  - This is the time quantum from ISO 11898–1, 12.4.1 *Bit Encoding/Decoding*.

An advanced baud rate example is 0x8014007D. This example breaks down into the following values:

- SJW = 0x0 (0x01 in hardware, due to the + 1)
- TSEG2 = 0x1 (0x02 in hardware, due to the + 1)

- TSEG 1 = 0x4 (0x05 in hardware, due to the + 1)
- Tq = 0x7D (125 ns in hardware)

Each time quanta is 125 ns. From IS0 11898–1, 12.4.1.2 *Programming of Bit Time*, the nominal time segments length is Sync_Seg(Fixed at 1) + (Prop_Seg + Phase_Seg1)(B) + Phase_Seg2(C) = 1 + 2 + 5 = 8. So, the total time for a bit in this example is 8 * 125 ns = 1000 ns = 1 μs. A 1 μs bit time is equivalent to a 1 MHz baud rate.

LIN

When the upper nibble (0xF0000000) is clear, you can set only baud rates within the LIN-specified range (2400 to 20000) for the interface.

When the upper nibble is set to 0x8 (0x80000000), no check for baud rate within LIN-specified range is performed, and the lowest 16 bits of the value may contain the custom baud rate. Any custom value higher than 65535 is masked to a 16-bit value. As with the noncustom values, the interface internally calculates the appropriate divisor values to program into its UART. Because the interface uses the Atmel ATA6620 LIN transceiver, which is guaranteed to operate within the LIN 2.0 specification limits, there are some special considerations when programming custom baud rates for LIN:

- The ATA6620 transceiver incorporates a TX dominant timeout function to prevent a faulty device that it is built into from holding the LIN dominant indefinitely. If the TX line into the transceiver is held in the dominant state for too long, the transceiver switches its driver to the recessive state. This places a limit on how long the LIN header break field that the interface transmits may be, and thus limits the lowest baud rate you can set. At the point the baud rate or break length is set for the interface, it uses the baud rate bit time and break length settings internally to calculate the resulting break duration and returns an error if that duration is long enough to trigger the TX dominant timeout.
- At the other end of the baud range, the ATA6620 is specified to work up to 20000 baud. While you can use the custom bit to program rates higher than that, the transceiver behavior when operating above that rate is not guaranteed.

Interface:Echo Transmit?

Data Type	Direction	Required?	Default
TF	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.EchoTx?

Description

The Interface:Echo Transmit? property determines whether Frame Input or Signal Input sessions contain frames that the interface transmits.

When this property is true, and a frame transmit is complete for an Output session, the frame is echoed to the Input session. Frame Input sessions can use the Flags field to differentiate frames received from the bus and frames the interface transmits. When using **XNET Read (Frame CAN).vi**, **XNET Read (Frame FlexRay).vi**, or **XNET Read (Frame LIN).vi**, the Flags field is parsed into an **echo?** Boolean in the frame cluster. When using **XNET Read (Frame Raw).vi**, you can parse the Flags manually by reviewing the *Raw Frame Format* section. Signal Input sessions cannot differentiate the origin of the incoming data.

Note Echoed frames are placed into the input sessions only after the frame transmit is complete. If there are bus problems (for example, no listener) such that the frame did not transmit, the frame is not received.

Interface:I/O Name

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

Intf.IOName

Description

The I/O Name property returns a reference to the interface used to create the session.

You can pass this I/O name into an XNET Interface property node to retrieve hardware information for the interface, such as the name and serial number. The I/O Name is the same reference available from the XNET System property node, which is used to read information for all XNET hardware in the system.

You can use this property on the diagram to:

- Display a string that contains the name of the interface as shown in Measurement and Automation Explorer (MAX).
- Provide a refnum you can wire to a property node to read information for the interface.

Interface:Output Stream List

Data Type	Direction	Required?	Default
	Read/Write	No	Empty Array

Property Class

XNET Session

Short Name

Intf.OutStrmList

Description

Note Only CAN and LIN interfaces currently support this property.

The Output Stream List property provides a list of frames for use with the replay feature (Interface:Output Stream Timing property set to Replay Exclusive or Replay Inclusive). In Replay Exclusive mode, the hardware transmits only frames that do not appear in the list. In Replay Inclusive mode, the hardware transmits only frames that appear in the list. For a LIN interface, the header of each frame written to stream output is transmitted, and the Exclusive or Inclusive mode controls the response transmission. Using these modes, you can either emulate an ECU (Replay Inclusive, where the list contains the frames the ECU transmits) or test an ECU (Replay Exclusive, where the list contains the frames the ECU transmits), or some other combination.

This property's data type is an array of XNET Frame from a database. When you are using a database file such as CANdb or FIBEX, each XNET frame uses the string name. If you are not using a database file or prefer to specify the frames using CAN arbitration IDs or LIN unprotected IDs, you can use Interface:Output Stream List By ID instead of this property.

Interface:Output Stream List By ID

Data Type	Direction	Required?	Default
	Read/Write	No	Empty Array

Property Class

XNET Session

Short Name

Intf.OutStrmListById

Description



Note Only CAN and LIN interfaces currently support this property.

The Output Stream List By ID property provides a list of frames for use with the replay feature (Interface:Output Stream Timing property set to Replay Exclusive or Replay Inclusive).

This property serves the same purpose as Interface:Output Stream List, in that it provides a list of frames for replay filtering. This property provides an alternate format for you to specify the frames by their CAN arbitration ID or LIN unprotected ID. The property's data type is an array of unsigned 32-bit integer (u32). Each integer represents a CAN or LIN frame's identifier, using the same encoding as the Raw Frame Format.

Within each CAN frame ID value, 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. LIN frame ID values may be within the range of possible LIN IDs (0–63).

Interface:Output Stream Timing

Data Type	Direction	Required?	Default
U32	Read/Write	No	Immediate

Property Class

XNET Session

Short Name

Intf.OutStrmTimng

Description

Note Only CAN and LIN interfaces currently support this property.

The Output Stream Timing property configures how the hardware transmits frames queued using a Frame Output Stream session. The following table lists the accepted values:

Enumeration	Value
Immediate	0
Replay Exclusive	1
Replay Inclusive	2

When you configure this property to be Immediate, frames are dequeued from the queue and transmitted immediately to the bus. The hardware transmits all frames in the queue as fast as possible.

When you configure this property as Replay Exclusive or Replay Inclusive, the hardware is placed into a Replay mode. In this mode, the hardware evaluates the frame timestamps and attempts to maintain the original transmission times as the timestamp stored in the frame indicates. The actual transmission time is based on the relative time difference between the first dequeued frame and the time contained in the dequeued frame.

When in one of the replay modes, you can use the Interface:Output Stream List property to supply a list. In Replay Exclusive mode, the hardware transmits only frames that do not appear in the list. In Replay Inclusive mode, the hardware transmits only frames that appear in the list. Using these modes, you can either emulate an ECU (Replay Inclusive, where the list contains the frames the ECU transmits) or test an ECU (Replay Exclusive, where the list contains the frames the ECU transmits), or some other combination. You can replay all frames by using Replay Exclusive mode without setting any list.

Runtime Behavior

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 whose timestamp goes backwards relative to the previous timestamp, but still is 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.

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.

When a frame with a Start Trigger frame type is received, the hardware treats this frame as a new first frame and uses the absolute time associated with this frame as the new start time. Subsequent frames are compared to this new start time to determine the transmission time. Using a Start Trigger is especially useful when synchronizing with data acquisition products, so that you can replay the first frame at the correct time relative to the start trigger for accurate synchronized replay.

Special Considerations for LIN

Only LIN interface as Master supports stream output. You do not need to set the interface explicitly to Master if you want to use stream output. Just create a stream output session, and the driver automatically sets the interface to Master at interface start.

You can use immediate mode to transmit a header or full frame. You can transmit only the header for a frame by writing the frame to stream output with the desired ID and an empty

data payload. You can transmit a full frame by writing the frame to stream output with the desired ID and data payload. If you write a full frame for ID *n* to stream output, and you have created a frame output session for frame with ID *n*, the stream output data takes priority (the stream output frame data is transmitted and not the frame output data). If you write a full frame to stream output, but the frame has not been defined in the database, the frame transmits with Enhanced checksum. To control the checksum type transmitted for a frame, you first must create the frame in the database and assign it to an ECU using the LIN specification you desire (the specification number determines the checksum type). You then must create a frame output object to transmit the response for the frame, and use stream output to transmit the header. Similarly, to transmit *n* corrupted checksums for a frame, you first must create a frame object in the database, create a frame output session for it, set the transmit *n* corrupted checksums property, and then use stream output to transmit the header.

Regarding event-triggered frame handling for immediate mode, if the hardware can determine that an ID is for an event-triggered frame, which means an event-triggered frame has been defined for the ID in the database, the frame is processed as if it were in an event-triggered slot in a schedule. If you write a full frame with event-triggered ID, the full frame is transmitted. If there is no collision, the next stream output frame is processed. If there is a collision, the hardware executes the collision-resolving schedule. The hardware retransmits the frame response at the corresponding slot time in the collision resolving schedule. If you write a header frame with an event-triggered ID and there is no collision, the next stream output frame is processed. If there is a collision, the hardware executes the collision-resolving schedule.

You can mix use of the hardware scheduler and stream output immediate mode. Basically, the hardware treats each stream output frame as a separate run-once schedule containing a single slot for the frame. Transmission of a stream output frame may interrupt a run-continuous schedule, but may not interrupt a run-once schedule. Transmission of stream output frames is interleaved with run-continuous schedule slot executions, depending on the application timing of writes to stream output. Stream output is prioritized to the equivalent of the lowest priority level for a run-once schedule. If you write one or more run-once schedules with higher-than-lowest priority and write frames to stream output, all the run-once schedules are executed before stream output transmits anything. If you write one or more run-once schedules with the lowest priority and write frames to stream output, the run-once schedules execute in the order you wrote them, and are interleaved with stream output frames, depending on the application timing of writes to stream output and writes of run-once schedule changes.

In contrast to the immediate mode, neither replay mode allows for the concurrent use of the hardware scheduler, and an error is reported if you attempt to do so. Event-triggered frame handling is different for the replay modes. If the hardware can determine that an ID is for an event-triggered frame, which means an event-triggered frame has been defined for the ID in the database, the frame is transmitted as if it were being transmitted during the collision-resolving schedule for the event triggered frame. The full frame is transmitted with

the Data[0] value (the underlying unconditional frame ID), copied into the header ID. If a frame cannot be found in the database, it is transmitted with Enhanced checksum. Otherwise, it is transmitted with the checksum type defined in the database.

The reply modes provide an easy means to replay headers only, full frames only, or some mix of the two. For either replay mode, the header for each frame is always transmitted and the slot delay is preserved. For replay inclusive, if you want only to replay headers, leave the Interface:Output Stream List property empty. To replay some of the responses, add their frames to Interface:Output Stream List. For frames that are not in Interface:Output Stream List, you are free to create frame output objects for them, for which you can change the checksum type or transmit corrupted checksums.

There is another consideration for the replay of diagnostic slave response frames. Because the master always transmits only the diagnostic slave response header, and a slave transmits the response if its NAD matches the one transmitted in the preceding master request frame, an array of frames for replay might include multiple slave response frames (each having the same slave response header ID) transmitted by different slaves (each having a different NAD value in the data payload). If you are using inclusive mode, you can choose not to replay any slave response frames by not including the slave response frame in Interface:Output Stream List. You can choose to replay some or all of the slave response frames by first including the slave response frame in Interface:Output Stream List, then including the NAD values for the slave responses you want to play back, in Interface:LIN:Output Stream Slave Response List By NAD. In this way, you have complete control over which slave responses are replayed (which diagnostic slaves you emulate). Replay of a diagnostic master request frame is handled like replay of any other frame; the header is always transmitted. Using the inclusive mode as an example, the response may or may not be transmitted depending on whether or not the master request frame is in Interface:Output Stream List.

Restrictions on Other Sessions

When you use Immediate mode, there are no restrictions on frames that you use in other sessions.

When you use Replay Inclusive mode, you can create output sessions that use frames that do not appear in the Interface:Output Stream List property. Attempting to create an output session that uses a frame from the Interface:Output Stream List property results in an error. Input sessions have no restrictions.

When you use Replay Exclusive mode, you cannot create any other output sessions. Attempting to create an output session returns an error. Input sessions have no restrictions.

Interface:Start Trigger Frames to Input Stream?

Data Type	Direction	Required?	Default
TF	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.StartTrigToInStrm?

Description

The Start Trigger Frames to Input Stream? property configures the hardware to place a start trigger frame into the Stream Input queue after it is generated. A Start Trigger frame is generated when the interface is started. The interface start process is described in *Interface Transitions*. For more information about the start trigger frame, refer to *Special Frames*.

The start trigger frame is especially useful if you plan to log and replay CAN data.

Interface:Bus Error Frames to Input Stream?

Data Type	Direction	Required?	Default
TF	Read/Write	No	False

Property Class

XNET Session

Short Name

Intf.BusErrToInStrm?

Description

Note Only CAN and LIN interfaces currently support this property.

The Bus Error Frames to Input Stream? property configures the hardware to place a CAN or LIN bus error frame into the Stream Input queue after it is generated. A bus error frame is generated when the hardware detects a bus error. For more information about the bus error frame, refer to *Special Frames*.

Session:Application Protocol

Data Type	Direction	Required?	Default
	Read Only	N/A	None

Property Class

XNET Session

Short Name

ApplProtocol

Description

This property returns the application protocol that the session uses.

The database used with **XNET Create Session.vi** determines the application protocol.

The values (enumeration) for this property are:

0 None

1 J1939

SAE J1939:ECU

Data Type	Direction	Required?	Default
I/O	Write Only	No	Unassigned

Property Class

XNET Session

Short Name

J1939.ECU

Description

Note This property applies to only the CAN J1939 application protocol.

This property assigns a database ECU to a J1939 session. Setting this property changes the node address and J1939 64-bit ECU name of the session to the values stored in the database ECU object. Changing the node address starts an address claiming procedure, as described in the SAE J1939:Node Address property.

You can assign the same ECU to multiple sessions running on the same CAN interface (for example, CAN1). All sessions with the same assigned ECU represent one J1939 node.

If multiple sessions have been assigned the same ECU, setting the SAE J1939:Node Address property in one session changes the address in all sessions with the same assigned ECU running on the same CAN interface.

For more information, refer to the SAE J1939:Node Address property.

SAE J1939:ECU Busy

Data Type	Direction	Required?	Default
TF	Read/Write	No	False

Property Class

XNET Session

Short Name

J1939.Busy

Description

Note This property applies to only the CAN J1939 application protocol.

Busy is a special ECU state defined in the SAE J1939 standard. A busy ECU receives subsequent RTS messages while handling a previous RTS/CTS communication.

If the ECU cannot respond immediately to an RTS request, the ECU may send CTS Hold messages. In this case, the originator receives information about the busy state and waits until the ECU leaves the busy state. (That is, the ECU no longer sends CTS Hold messages and sends the first CTS message with the requested data.)

Use the ECU Busy property to simulate this ECU behavior. If a busy XNET ECU receives a CTS message, it sends CTS Hold messages instead of CTS data messages immediately. Afterward, if clearing the busy property, the XNET ECU resumes handling the transport protocol starting with CTS data messages, as the originator expects.

SAE J1939:Hold Time Th

Data Type	Direction	Required?	Default
	Read/Write	No	0.5 s

Property Class

XNET Session

Short Name

J1939.HoldTimeTh

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the Hold Time Timeout value at the responder node. The value is the maximum time between a TP.CM_CTS hold message and the next TP.CM_CTS message, in seconds.

This property is related to handling the transport protocol.

SAE J1939:Maximum Repeat CTS

Data Type	Direction	Required?	Default
U32	Read/Write	No	2

Property Class

XNET Session

Short Name

J1939.MAXReptCTS

Description

Note This property applies to only the CAN J1939 application protocol.

This property limits the number of requests for retransmission of data packet(s) using the TP.CM_CTS message.

This property is related to handling the transport protocol.

SAE J1939:Node Address

Data Type	Direction	Required?	Default
U32	Read/Write	No	Null (254)

Property Class

XNET Session

Short Name

J1939.Address

Description

Note This property applies to only the CAN J1939 application protocol.

This property changes the node address of a J1939 session by starting an address claiming procedure. After setting this property to a valid value (≤ 253), reading the property returns the null address (254) until the address is granted. Poll the property and wait until the address gets to a valid value again before starting to write. Refer to the NI-XNET examples that demonstrate this procedure.

The node address value determines the source address in a transmitting session or a destination address in a receiving session. The source address in the extended frame identifier is overwritten with the node address of the session before transmitting.

A session with a null (254) or global address (255) receives all messages sent on the bus, but cannot transmit messages. A session with an assigned address of less than 254 receives only messages sent to this address or global messages, but not messages sent to other nodes. This session also can transmit messages.

In NI-XNET, you can assign the same J1939 node address to multiple sessions running on the same interface (for example, CAN1). Those sessions represent one J1939 node. By assigning different J1939 node addresses to multiple sessions running on the same interface, you also can create multiple nodes on the same interface.

If a J1939 ECU is assigned to multiple sessions, changing the address in one session also changes the address in all other sessions with the same assigned ECU.

For more information, refer to the SAE J1939:ECU property.

SAE J1939:NodeName

Data Type	Direction	Required?	Default
	Read/Write	Yes	0

Property Class

XNET Session

Short Name

J1939.NodeName

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the name value of a J1939 session. The name is an unsigned 64-bit integer value. Beside the SAE J1939:Node Address property, the value is specific to the ECU you want to emulate using the session. That means the session can act as if it were the real-world ECU, using the identical address and name value.

The name value is used within the address claiming procedure. If the ECU (session) wants to claim its address, it sends out an address claiming message. That message contains the ECU address and the name value of the current session's ECU. If there is another ECU within the network with an identical address but lower name value, the current session loses its address. In this case, the session cannot send out further messages, and all addressed messages using the previous address of the current session are addressed to another ECU within the network.

The most significant bit (bit 63) in the Node Name defines the ECU's arbitrary address capability (bit 63 = 1 means it is arbitrary address capable). If the node cannot use the assigned address, it automatically tries to claim another random value between 128 and 247 until it is successful.

If multiple sessions are assigned the same ECU, setting the SAE J1939.NodeName property in one session changes the address in all sessions with the same assigned ECU running on the same CAN interface.

The name value has multiple bit fields, as described in SAE J1939-81 (Network Management). A single 64-bit value represents the name value within XNET.

For more information, refer to the SAE J1939:Node Address property.

SAE J1939:Number of Packets Received

Data Type	Direction	Required?	Default
U32	Read/Write	No	255

Property Class

XNET Session

Short Name

J1939.NumPktsRecv

Description

Note This property applies to only the CAN J1939 application protocol.

This property changes the maximum number of data packet(s) that can be received in one block at the responder node.

This property is related to handling the transport protocol.

SAE J1939:Number of Packets Response

Data Type	Direction	Required?	Default
U32	Read/Write	No	255

Property Class

XNET Session

Short Name

J1939.NumPktsResp

Description

Note This property applies to only the CAN J1939 application protocol.

This property limits the maximum number of packets in a response. This allows the originator node to limit the number of packets in the TP.CM_CTS message. When the responder complies with this limit, it ensures the sender always can retransmit packets that the responder may not have received.

This property is related to handling the transport protocol.

SAE J1939:Response Time Tr_GD

Data Type	Direction	Required?	Default
	Read/Write	No	0.05 s

Property Class

XNET Session

Short Name

J1939.RespTimeTrGD

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the Device Response Time for global destination messages (TP.CM_BAM messages). The value is the maximum delay between sending two TP.CM_BAM messages, in seconds. The recommended range is 0.05–200 s.

This property is related to handling the transport protocol.

SAE J1939:Response Time Tr_SD

Data Type	Direction	Required?	Default
	Read/Write	No	0.05 s

Property Class

XNET Session

Short Name

J1939.RespTimeTrSD

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the Device Response Time value for specific destination messages (TP.CM_RTS/CTS messages). The value is the maximum time between receiving a message and sending the response message, in seconds. The recommended range is 0.05–0.200 s.

This property is related to handling the transport protocol.

SAE J1939:Timeout T1

Data Type	Direction	Required?	Default
DBL	Read/Write	No	0.75 s

Property Class

XNET Session

Short Name

J1939.TimeoutT1

Description

Note This property applies to only the CAN J1939 application protocol.

This property changes the timeout T1 value for the responder node. The value is the maximum gap between two received TP.DT messages in seconds.

This property is related to handling the transport protocol.

SAE J1939:Timeout T2

Data Type	Direction	Required?	Default
	Read/Write	No	1.25 s

Property Class

XNET Session

Short Name

J1939.TimeoutT2

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the timeout T2 value at the responder node. This value is the maximum gap between sending out the TP.CM_CTS message and receiving the next TP.DT message, in seconds.

This property is related to handling the transport protocol.

SAE J1939:Timeout T3

Data Type	Direction	Required?	Default
DBL	Read/Write	No	1.25 s

Property Class

XNET Session

Short Name

J1939.TimeoutT3

Description

Note This property applies to only the CAN J1939 application protocol.

This property changes the timeout T3 value at the originator node. This value is the maximum gap between sending out a TP.CM_RTS message or the last TP.DT message and receiving the TP.CM_CTS response, in seconds.

This property is related to handling the transport protocol.

SAE J1939:Timeout T4

Data Type	Direction	Required?	Default
	Read/Write	No	1.05 s

Property Class

XNET Session

Short Name

J1939.TimeoutT4

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the timeout T4 value at the originator node. This value is the maximum gap between the TP.CM_CTS hold message and the next TP.CM_CTS message, in seconds.

This property is related to handling the transport protocol.

Frame Properties

This section includes the frame-specific properties in the session property node.

CAN Frame Properties

This category includes CAN-specific frame properties.

Frame:CAN:Start Time Offset

Data Type	Direction	Required?	Default
	Write Only	No	–1

Property Class

XNET Session

Short Name

Frm.CAN.StartTimeOff

Description

Use this property to configure the amount of time that must elapse between the session being started and the time that the first frame is transmitted across the bus. This is different than the cyclic rate, which determines the time between subsequent frame transmissions.

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

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

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



Note This property affects the active frame object in the session. Review the Frame:Active property to learn more about setting a property on an active frame.

Frame:CAN:Transmit Time

Data Type	Direction	Required?	Default
	Write Only	No	From Database

Property Class

XNET Session

Short Name

Frm.CAN.TxTime

Description

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

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



Note This property affects the active frame object in the session. Review the Frame:Active property to learn more about setting property on an active frame.



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

Frame:Active

Data Type	Direction	Required?	Default
	Write Only	No	0

Property Class

XNET Session

Short Name

Frm.Active

Description

This property provides access to properties for a specific frame running within the session. Writing this property sets the active frame for subsequent properties in the Frame category.

The string syntax supports the following options:

- **Decimal number**: This is interpreted as the index of the signal or frame in the session's list. If the session is signal I/O, subsequent frame properties change the signal's parent frame.
- **XNET Frame**: If the session is frame I/O, you can wire a frame name from the session's List of Frames property.
- **XNET Signal**: If the session is signal I/O, you can wire a signal name from the session's List of Signals property. Subsequent frame properties change the signal's parent frame.

If the session is Frame Stream Input or Frame Stream Output, this property has no effect, because stream I/O sessions do not use specific frames.

The default value of this property is 0, the first frame or signal in the session's list. If the empty string is wired to this property, this is converted to 0 internally.

Frame:LIN:Transmit N Corrupted Checksums

Data Type	Direction	Required?	Default
U32	Write Only	No	0

Property Class

XNET Session

Short Name

Frm.LIN.TxNCrptChks

Description

When set to a nonzero value, this property causes the next *N* number of checksums to be corrupted. The checksum is corrupted by negating the value calculated per the database; (`EnhancedValue * -1`) or (`ClassicValue * -1`). This property is valid only for output sessions. If the frame is transmitted in an unconditional or sporadic schedule slot, *N* is always decremented for each frame transmission. If the frame is transmitted in an event-triggered slot and a collision occurs, *N* is not decremented. In that case, *N* is decremented only when the collision resolving schedule is executed and the frame is successfully transmitted. If the frame is the only one to transmit in the event-triggered slot (no collision), *N* is decremented at event-triggered slot time.

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

Note This property affects the active frame object in the session. Review the Frame:Active property to learn more about setting a property on an active frame.

Frame:Skip N Cyclic Frames

Data Type	Direction	Required?	Default
U32	Write Only	No	0

Property Class

XNET Session

Short Name

Frm.SkipNCyclic

Description

Note This property is currently supported by CAN interfaces only.

When set to a nonzero value, this property causes the next *N* cyclic frames to be skipped. When the frame's transmission time arrives and the skip count is nonzero, a frame value is dequeued (if this is not a single-point session), and the skip count is decremented, but the frame actually is not transmitted across the bus. When the skip count decrements to zero, subsequent cyclic transmissions resume. This property is valid only for output sessions and frames with cyclic timing (that is, not event-based frames).

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

Note This property affects the active frame object in the session. Review the Frame:Active property to learn more about setting a property on an active frame.

Auto Start?

Data Type	Direction	Required?	Default
	Read/Write	No	True

Property Class

XNET Session

Short Name

AutoStart?

Description

Automatically starts the output session on the first call to **XNET Write.vi**.

For input sessions, start always is performed within the first call to **XNET Read.vi** (if not already started using **XNET Start.vi**). This is done because there is no known use case for reading a stopped input session.

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

When automatic start is performed, it is equivalent to **XNET Start.vi** with **scope** set to Normal. This starts the session itself, and if the interface is not already started, it starts the interface also.

Cluster

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

Cluster

Description

This property returns the cluster (network) used with **XNET Create Session.vi**.

Use this property on the block diagram as follows:

- As a refnum wired to a property node to access information for the cluster and its objects (frames, signals, etc.).
- As a string containing the cluster name. This name typically is the database alias followed by the cluster name.

Database

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

Database

Description

This property returns the database used with **XNET Create Session.vi**.

Use this property on the block diagram as follows:

- As a refnum wired to a property node to access information for the database and its objects (frames, signals, etc.).
- As a string containing the database name. This name is typically a database alias, but it also can be a complete file path.

List of Frames

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

ListFrms

Description

This property returns the list of frames in the session.

This property is valid only for sessions of Frame Input or Frame Output mode. For a Signal Input/Output session, use the List of Signals property.

Use each array element on the block diagram as follows:

- As a refnum wired to a property node to access information for the frame.
- As a string containing the frame name. The name is the one used to create the session.

List of Signals

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

ListSigs

Description

This property returns the list of signals in the session.

This property is valid only for sessions of Signal Input or Signal Output mode. For a Frame Input/Output session, use the List of Frames property.

Use each array element on the block diagram as follows:

- As a refnum wired to a property node to access information for the signal.
- As a string containing the signal name. The name is the one used to create the session.

Mode

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

Mode

Description

This property returns the session mode (ring). You provided this mode when you created the session. For more information, refer to *Session Modes*.

Number in List

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

NumInList

Description

This property returns the number of frames or signals in the session's list. This is a quick way to get the size of the List of Frames or List of Signals property.

Number of Values Pending

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

NumPend

Description

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

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

For output sessions, this is the number of frames/signal values provided to **XNET Write.vi** but not yet transmitted onto the network.

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

The largest possible frames sizes are:

- **CAN FD:** 64 byte payload.
- **FlexRay:** The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay:Payload Length Maximum property provides this value.

The execution time to read this property is sufficient for use in a high-priority loop on LabVIEW Real-Time (RT) (refer to *High Priority Loops* for more information).

Number of Values Unused

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

NumUnused

Description

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

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

For output sessions, this is the number of frame/signal values you can provide to a subsequent call to **XNET Write.vi**. If you call **XNET Write.vi** with this number of values and timeout of 0.0, **XNET Write.vi** should return success.

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

The largest possible frames sizes are:

- **CAN FD:** 64 byte payload.
- **FlexRay:** The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay:Payload Length Maximum property provides this value.

The execution time to read this property is sufficient for use in a high-priority loop on LabVIEW Real-Time (RT) (refer to *High Priority Loops* for more information).

Payload Length Maximum

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

PayldLenMax

Description

This property returns the maximum payload length of all frames in this session, expressed as bytes (0–254).

This property does not apply to Signal sessions (only Frame sessions).

For CAN Stream (Input and Output), this property depends on the XNET Cluster CAN:I/O Mode property. If the I/O mode is CAN, this property is 8 bytes. If the I/O mode is CAN FD or CAN FD + BRS, this property is 64 bytes.

For LIN Stream (Input and Output), this property always is 8 bytes. For FlexRay Stream (Input and Output), this property is the same as the XNET Cluster FlexRay:Payload Length Maximum property value. For Queued and Single-Point (Input and Output), this is the maximum payload of all frames specified in the List of Frames property.

Protocol

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Session

Short Name

Protocol

Description

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

The values (enumeration) for this property are:

0 CAN

1 FlexRay

2 LIN

Queue Size

Data Type	Direction	Required?	Default
	Read/Write	No	Refer to Description

Property Class

XNET Session

Short Name

QueueSize

Description

For output sessions, queues store data passed to **XNET Write.vi** and not yet transmitted onto the network. For input sessions, queues store data received from the network and not yet obtained using **XNET Read.vi**.

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

For signal I/O sessions, this property is the number of signal values stored. This is analogous to the number of values you use with **XNET Read.vi** or **XNET Write.vi**.

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

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

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

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

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not

represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

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

Default Value

You calculate the default queue size based on the following assumptions:

- **Application Time**: The time between calls to **XNET Read.vi**/**XNET Write.vi** in your application.
- **Frame Time**: The time between frames on the network for this session.

The following pseudo code describes the default queue size formula:

```
if (session is Signal I/O Waveform)
    Queue_Size = (Application_Time * Resample_Rate);
else
    Queue_Size = (Application_Time / Frame_Time);
if (Queue_Size < 64)
    Queue_Size = 64;
if (session mode is Frame I/O)
    Queue_Size = Queue_Size * Frame_Size;
```

For Signal I/O Waveform sessions, the initial formula calculates the number of resampled values that occur within the Application Time. This is done by multiplying Application Time by the XNET Session Resample Rate property.

For all other session modes, the initial formula divides Application Time by Frame Time.

The minimum for this formula is 64. This minimum ensures that you can read or write at least 64 elements. If you need to read or write more elements for a slow frame, you can set the Queue Size property to a larger number than the default. If you set a large Queue Size, this may limit the maximum number of frames you can use in all sessions.

For Frame I/O sessions, this formula result is multiplied by each frame value size to obtain a queue size in bytes.

For Signal I/O sessions, this formula result is used directly for the queue size property to provide the number of signal values for **XNET Read.vi** or **XNET Write.vi**. Within the Signal I/O session, the memory allocated for the queue incorporates frame sizes, because the signal values are mapped to/from frame values internally.

Application Time

The LabVIEW target in which your application runs determines the Application Time:

- **Windows**: 400 ms (0.4 s)
- **LabVIEW Real-Time (RT)**: 100 ms (0.1 s)

This works under the assumption that for Windows, more memory is available for input queues, and you have limited control over the application timing. LabVIEW RT targets typically have less available memory, but your application has better control over application timing.

Frame Time

Frame Time is calculated differently for Frame I/O Stream sessions compared to other modes. For Frame I/O Stream, you access all frames in the network (cluster), so the Frame Time is related to the average bus load on your network. For other modes, you access specific frames only, so the Frame Time is obtained from database properties for those frames.

The Frame Time used for the default varies by session mode and protocol, as described below.

CAN, Frame I/O Stream

Frame Time is 100 μs (0.0001 s).

This time assumes a baud rate of 1 Mbps, with frames back to back (100 percent busload).

For CAN sessions created for a standard CAN bus, the Frame Size is 24 bytes. For CAN sessions created for a CAN FD Bus (the cluster I/O mode is CAN FD or CAN FD+BRS), the frame size can vary up to 64 bytes. However, the default queue size is based on the 24-byte frame time. When connecting to a CAN FD bus, you may need to adjust this size as necessary.

When you create an application to stress test NI-XNET performance, it is possible to generate CAN frames faster than 100 μs. For this application, you must set the queue size to larger than the default.

FlexRay, Frame I/O Stream

Frame Time is 20 μs (0.00002 s).

This time assumes a baud rate of 10 Mbps, with a cycle containing static slots only (no minislots or NIT), and frames on channel A only.

Small frames at a fast rate require a larger queue size than large frames at a slow rate. Therefore, this default assumes static slots with 4 bytes, for a Frame Size of 24 bytes.

When you create an application to stress test NI-XNET performance, it is possible to generate FlexRay frames faster than 20 µs. For this application, you must set the queue size to larger than the default.

LIN, Frame I/O Stream

Frame Time is 2 ms (0.002 s).

This time assumes a baud rate of 20 kbps, with 1 byte frames back to back (100 percent busload).

For all LIN sessions, Frame Size is 24 bytes.

CAN, Other Modes

For Frame I/O Queued, Signal I/O XY, and Signal I/O Waveform, the Frame Time is different for each frame in the session (or frame within which signals are contained).

For CAN frames, Frame Time is the frame property CAN:Transmit Time, which specifies the time between successive frames (in floating-point seconds).

If the frame's CAN Transmit Time is 0, this implies the possibility of back-to-back frames on the network. Nevertheless, this back-to-back traffic typically occurs in bursts, and the average rate over a long period of time is relatively slow. To keep the default queue size to a reasonable value, when CAN Transmit Time is 0, the formula uses a Frame Time of 50 ms (0.05 s).

For CAN sessions using a standard CAN cluster, the frame size is 24 bytes. For CAN sessions using a CAN FD cluster, the frame size may differ for each frame in the session. Each frame size is obtained from its XNET Frame Payload Length property in the database.

FlexRay, Other Modes

For Frame I/O Queued, Signal I/O XY, and Signal I/O Waveform, the Frame Time is different for each frame in the session (or frame within which signals are contained).

For FlexRay frames, Frame Time is the time between successive frames (in floating-point seconds), calculated from cluster and frame properties. For example, if a cluster Cycle (cycle duration) is 10000 µs, and the frame Base Cycle is 0 and Cycle Repetition is 1, the frame's Transmit Time is 0.01 (10 ms).

For these session modes, the Frame Size is different for each frame in the session. Each Frame Size is obtained from its XNET Frame Payload Length property in the database.

LIN, Other Modes

For LIN frames, Frame Time is a property of the schedule running in the LIN master node. It is assumed that the Frame Time for a single frame always is larger than 8 ms, so that the default queue size is set to 64 frames throughout.

For all LIN sessions, Frame Size is 24 bytes.

Examples

The following table lists example session configurations and the resulting default queue sizes.

Session Configuration	Default Queue Size	Formula
Frame Input Stream, CAN, Windows	96000	(0.4 / 0.0001) = 4000; 4000 × 24 bytes
Frame Output Stream, CAN, Windows	96000	(0.4 / 0.0001) = 4000; 4000 × 24 bytes; output is always same as input
Frame Input Stream, FlexRay, Windows	480000	(0.4 / 0.00002) = 20000; 20000 × 24 bytes
Frame Input Stream, CAN, LabVIEW RT	24000	(0.1 / 0.0001) = 1000; 1000 × 24 bytes
Frame Input Stream, FlexRay, LabVIEW RT	120000	(0.1 / 0.00002) = 5000; 5000 × 24 bytes
Frame Input Queued, CAN, Transmit Time 0.0, Windows	1536*	(0.4 / 0.05) = 8; Transmit Time 0 uses Frame Time 50 ms; use minimum of 64 frames (64 × 24)
Frame Input Queued, CAN, Transmit Time 0.0005, Windows	19200*	(0.4 / 0.0005) = 800; 800 × 24 bytes
Frame Input Queued, CAN, Transmit Time 1.0 (1 s), Windows	1536*	(0.4 / 1.0) = 0.4; use minimum of 64 frames (64 × 24)
Frame Input Queued, FlexRay, every 2 ms cycle, payload length 4, Windows	4800	(0.4 / 0.002) = 200; 200 × 24 bytes
Frame Input Queued, FlexRay, every 2 ms cycle, payload length 16, LabVIEW RT	2048	(0.1 / 0.002) = 50, use minimum of 64; payload length 16 requires 32 bytes; 64 × 32 bytes

Session Configuration	Default Queue Size	Formula
Signal Input XY, two CAN frames, Transmit Time 0.0 and 0.0005, Windows	64* and 800* (read as 800)	(0.4 / 0.05) = 8, use minimum of 64; (0.4 / 0.0005) = 800; expressed as signal values
Signal Output XY, two CAN frames, Transmit Time 0.0 and 0.0005, Windows	64* and 800* (read as 64)	(0.4 / 0.05) = 8, use minimum of 64; (0.4 / 0.0005) = 800; expressed as signal values
Signal Output Waveform, two CAN frames, 1 ms and 400 ms, resample rate 1000 Hz, Windows	400*	Memory allocation is 400 and 64 frames to provide 0.4 sec of storage, queue size represents number of samples, or (0.4 × 1000.0)
Signal Output Waveform, two CAN frames, 1 ms and 400 ms, resample rate 1000 Hz, Windows	400*	Memory allocation is 400 and 64 frames to provide 0.4 sec of storage, queue size represents number of samples, or (0.4 × 1000.0)

* For a CAN FD cluster, the default queue size is based on the frame's database payload length, which may be larger than 24 bytes (up to 64 bytes).

Resample Rate

Data Type	Direction	Required?	Default
	Read/Write	No	1000.0 (Sample Every Millisecond)

Property Class

XNET Session

Short Name

ResampRate

Description

Rate used to resample frame data to/from signal data in waveforms.

This property applies only when the session mode is Signal Input Waveform or Signal Output Waveform. This property is ignored for all other modes.

The data type is 64-bit floating point (DBL). The units are in Hertz (samples per second).

XNET Read.vi

Purpose

Reads data from the network using an XNET session.

Description

The instances of this polymorphic VI specify the type of data returned.

XNET Read.vi and **XNET Write.vi** are optimized for real-time performance. **XNET Read.vi** executes quickly and avoids access to shared resources that can induce jitter on other VI priorities.

There are three categories of **XNET Read** instance VIs:

- **Signal**: Use when the session mode is Signal Input. The **XNET Read.vi** instance must match the mode exactly (for example, the Signal Waveform instance when mode is Signal Input Waveform).
- **Frame**: Use when the session mode is Frame Input. The **XNET Read.vi** instance specifies the desired data type for frames and is not related to the mode. For an easy-to-use data type, use the CAN, FlexRay, or LIN instance.
- **State**: Use to read state, status, and time information for the session interface. You can use these instances in addition to Signal or Frame instances, and they are not related to the mode. The data these instances return is optimized for performance. Although property nodes may return similar runtime data, those properties are not necessarily optimized for real-time loops.

The **XNET Read** instance VIs are:

- **XNET Read (Frame CAN).vi**: The session uses a CAN interface, and the mode is Frame Input Stream Mode, Frame Input Queued Mode, or Frame Input Single-Point Mode.
- **XNET Read (Frame FlexRay).vi**: The session uses a FlexRay interface, and the mode is Frame Input Stream Mode, Frame Input Queued Mode, Frame Input Single-Point Mode, PDU Input Queued Mode (similar to Frame Input Queued Mode), and PDU Input Single-Point Mode (similar to Frame Input Single-Point Mode).
- **XNET Read (Frame LIN).vi**: The session uses a LIN interface, and the mode is Frame Input Stream Mode, Frame Input Queued Mode, or Frame Input Single-Point Mode
- **XNET Read (Frame Raw).vi**: A data type for frame input that is protocol independent and more efficient than the protocol-specific instances.
- **XNET Read (Signal Single-Point).vi**: The session mode is Signal Input Single-Point.
- **XNET Read (Signal Waveform).vi**: The session mode is Signal Input Waveform.
- **XNET Read (Signal XY).vi**: The session mode is Signal Input XY.

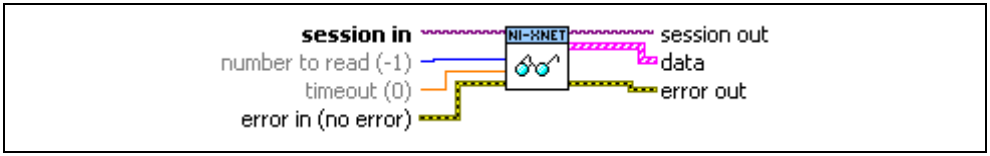
- **XNET Read (State CAN Comm).vi**: Returns the CAN interface's communication state.
- **XNET Read (State FlexRay Comm).vi**: Returns the FlexRay interface's communication state.
- **XNET Read (State LIN Comm).vi**: Returns the LIN interface's communication state.
- **XNET Read (State FlexRay Cycle Macrotick).vi**: Returns the current global time of the session FlexRay interface, represented as cycle and macrotick.
- **XNET Read (State FlexRay Statistics).vi**: Returns the communication statistics for the session FlexRay interface.
- **XNET Read (State Time Comm).vi**: Returns the LabVIEW timestamp at which communication began for the session interface.
- **XNET Read (State Time Current).vi**: Returns the session interface current time as a LabVIEW timestamp.
- **XNET Read (State Time Start).vi**: Returns the LabVIEW timestamp at which communication started for the session interface. This time always precedes the Communication time.
- **XNET Read (State Session Info).vi**: Returns the current state for the session provided.

XNET Read (Frame CAN).vi

Purpose

Reads data from a session as an array of CAN frames. The session must use a CAN interface and Frame Input Stream Mode, Frame Input Queued Mode, or Frame Input Single-Point Mode.

Format



Inputs



session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Frame Input Stream, Frame Input Queued, or Frame Input Single-Point.



number to read is the number of frame values desired.

If **number to read** is positive (or 0), the data array size is no greater than this number.

If **number to read** is negative (typically –1), all available frame values are returned. If **number to read** is negative, you must use **timeout** of 0.

This input is optional. The default value is –1.

If the session mode is Frame Input Single-Point, set **number to read** to either –1 or the number of frames in the sessions list. This ensures that the **XNET Read (Frame CAN).vi** can return the current value of all session frames.



timeout is the time to wait for **number to read** frame values to become available.

The **timeout** is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, the **XNET Read (Frame CAN).vi** waits for **number to read** frame values, then returns that number. If the values do not arrive prior to the **timeout**, an error is returned.

If **timeout** is negative, the **XNET Read (Frame CAN).vi** waits indefinitely for **number to read** frame values.

If **timeout** is zero, the **XNET Read (Frame CAN).vi** does not wait and immediately returns all available frame values up to the limit **number to read** specifies.

This input is optional. The default value is 0.0.

If the session mode is Frame Input Single-Point, you must leave **timeout** unwired (0.0). Because this mode reads the most recent value of each frame, **timeout** does not apply.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



data returns an array of LabVIEW clusters.

Each array element corresponds to a frame the session receives.

For a Frame Input Single-Point session mode, the order of frames in the array corresponds to the order in the session list.

The elements of each cluster are specific to the CAN protocol. For more information, refer to Appendix A, *Summary of the CAN Standard*, or the CAN protocol specification.

The cluster elements are:



identifier is the CAN frame arbitration identifier.

If **extended?** Is false, the identifier uses standard format, so 11 bits of this identifier are valid. If **extended?** Is true, the identifier uses extended format, so 29 bits of this identifier are valid.



extended? is a Boolean value that determines whether the identifier uses extended format (true) or standard format (false).



echo? is a Boolean value that determines whether the frame was an echo of a successful transmit (true), or received from the network (false).

This value is true only when you enable echo of transmitted frames by setting the XNET Session Interface:Echo Transmit? property to True.



type is the frame type (decimal value in parentheses):

CAN Data (0) The CAN data frame contains **payload** data. This is the most commonly used frame type for CAN.

CAN Remote (1) 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**.

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 (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*.



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



payload is the array of **data** bytes for the CAN data frame.

The array size indicates the received frame value payload length. According to the CAN protocol, this payload length range is 0–8. For CAN FD, the range can be 0–8, 12, 16, 20, 24, 32, 48, or 64.

For a received remote frame (**type** of **CAN Remote**), the payload length in the frame value specifies the number of **payload** bytes requested. This payload length is provided to your application by filling **payload** with the requested number of bytes. Your

application can use the **payload** array size, but you must ignore the actual values in the **payload** bytes.

For an example of how this data applies to network traffic, refer to *Frame Input Stream Mode*, *Frame Input Queued Mode*, or *Frame Input Single-Point Mode*.



error out is the error cluster output (refer to *Error Handling*).

Description

The data represents an array of CAN frames. Each CAN frame uses a LabVIEW cluster with CAN-specific elements.

The CAN frames are associated to the session's list of frames as follows:

- Frame Input Stream Mode: Array of all frame values received (list ignored).
- Frame Input Queued Mode: Array of frame values received for the single frame specified in the list.
- Frame Input Single-Point Mode: Array of single frame values, one for each frame specified in the list.

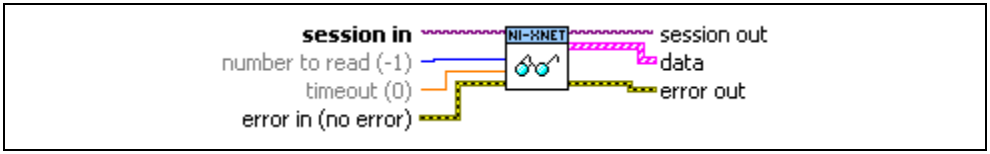
Due to issues with LabVIEW memory allocation for clusters with an array, this **XNET Read.vi** instance can introduce jitter to a high-priority loop on LabVIEW Real-Time (RT) (refer to *High Priority Loops* for more information). The **XNET Read (Frame Raw).vi** instance provides optimal performance for high-priority loops.

XNET Read (Frame FlexRay).vi

Purpose

Reads data from a session as an array of FlexRay frames. The session must use a FlexRay interface and Frame Input Stream Mode, Frame Input Queued Mode, or Frame Input Single-Point Mode.

Format



Inputs

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Frame Input Stream, Frame Input Queued, or Frame Input Single-Point.

number to read is the number of frame values desired.

If **number to read** is positive (or 0), the data array size is no greater than this number.

If **number to read** is negative (typically –1), all available frame values are returned. If **number to read** is negative, you must use a timeout of 0.

This input is optional. The default value is –1.

If the session mode is Frame Input Single-Point, set **number to read** to either –1 or the number of frames in the session list. This ensures that **XNET Read (Frame FlexRay).vi** can return the current value of all session frames.

timeout is the time to wait for **number to read** frame values to become available.

The timeout is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, **XNET Read (Frame FlexRay).vi** waits for **number to read** frame values, then returns that number. If the values do not arrive prior to the timeout, an error is returned.

If **timeout** is negative, **XNET Read (Frame FlexRay).vi** waits indefinitely for **number to read** frame values.

If **timeout** is zero, **XNET Read (Frame FlexRay).vi** does not wait and immediately returns all available frame values up to the limit **number to read** specifies.

This input is optional. The default value is 0.0.

If the session mode is Frame Input Single-Point, you must leave **timeout** unwired (0.0). Because this mode reads the most recent value of each frame, **timeout** does not apply.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



data returns an array of LabVIEW clusters.

Each array element corresponds to a frame the session receives.

For the Frame Input Single-Point and PDU Input Single-Point session modes, the order of frames/payload in the array corresponds to the order in the session list.

The elements of each cluster are specific to the FlexRay protocol. For more information, refer to Appendix B, *Summary of the FlexRay Standard*, or the *FlexRay Protocol Specification*.

The cluster elements are:



slot specifies the slot number within the FlexRay cycle.



cycle count specifies the cycle number.

The FlexRay cycle count increments from 0 to 63, then rolls over back to 0.



startup? is a Boolean value that specifies whether the frame is a startup frame (true) or not (false).



sync? is a Boolean value that specifies whether the frame is a sync frame (true) or not (false).



preamble? is a Boolean value that specifies the value of the payload preamble indicator in the frame header.

If the frame is in the static segment, **preamble?** being true indicates the presence of a network management vector at the beginning of the payload. The XNET Cluster FlexRay:Network Management Vector Length property specifies the number of bytes at the beginning.

If the frame is in the dynamic segment, **preamble?** being true indicates the presence of a message ID at the beginning of the payload. The message ID is always 2 bytes in length.

If **preamble?** is false, the payload does not contain a network management vector or a message ID.



chA is a Boolean value that specifies whether the frame was received on channel A (true) or not (false).



chB is a Boolean value that specifies whether the frame was received on channel B (true) or not (false).



echo? Is a Boolean value that determines whether the frame was an echo of a successful transmit (true) or received from the network (false).

This value is true only when you enable echo of transmitted frames by setting the XNET Session Interface:Echo Transmit? property to true. Frames are echoed only to a session with the Frame Input Stream Mode.



type is the frame type (decimal value in parentheses):

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.

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.

FlexRay Symbol (34) FlexRay symbol frame. The frame contains a symbol received on the FlexRay bus.

For this frame type, the 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 (refer to *Start Interface* for more information). For information about this frame, including the other frame fields, refer to *Special Frames*.



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

While the NI-XNET FlexRay interface is communicating (integrated), this timestamp is normally derived from FlexRay global time, the FlexRay network timebase. Under this configuration, the timestamp does not drift as compared to the FlexRay global time (**XNET Read (State FlexRay Cycle Macrotick).vi**), but it may drift relative to other NI hardware products and the LabVIEW absolute timebase. If you prefer to synchronize this timestamp to other sources, you can use **XNET Connect Terminals.vi** to change the source of the Master Timebase terminal.



payload is the array of data bytes for FlexRay frames of type **FlexRay Data** or **FlexRay Null**.

The array size indicates the received frame value payload length. According to the FlexRay protocol, this length range is 0–254.

For PDU session modes, only the payload for the particular PDU is returned, not the entire frame.

For an example of how this data applies to network traffic, refer to *Frame Input Stream Mode*, *Frame Input Queued Mode*, or *Frame Input Single-Point Mode*.



error out is the error cluster output (refer to *Error Handling*).

Description

The data represents an array of FlexRay frames. Each FlexRay frame uses a LabVIEW cluster with FlexRay-specific elements.

The FlexRay frames are associated to the session list of frames as follows:

- Frame Input Stream Mode: Array of all frame values received (list ignored).
- Frame Input Queued Mode: Array of frame values received for the single frame specified in the list.
- Frame Input Single-Point Mode: Array of single frame values, one for each frame specified in the list.
- PDU Input Queued Mode: Array of frame (PDU payload) values received for the single PDU specified in the list. This mode is similar to Frame Input Queued Mode,
- PDU Input Single-Point Mode: Array of single frame (PDU payload) values, one for each PDU specified in the list. This mode is similar to Frame Input Single-Point Mode.

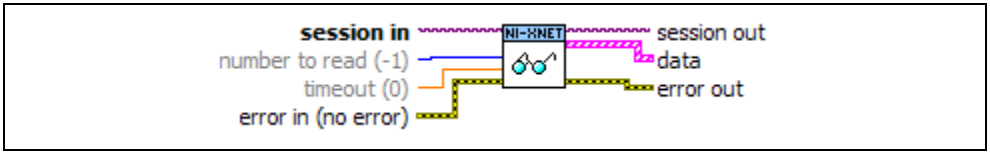
Due to issues with LabVIEW memory allocation for clusters with an array, this **XNET Read.vi** instance can introduce jitter to a high-priority loop on LabVIEW Real-Time (RT) (refer to *High Priority Loops* for more information). The **XNET Read (Frame Raw).vi** instance provides optimal performance for high-priority loops.

XNET Read (Frame LIN).vi

Purpose

Reads data from a session as an array of LIN frames. The session must use a LIN interface and Frame Input Stream Mode, Frame Input Queued Mode, or Frame Input Single-Point Mode.

Format



Inputs

I/O

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Frame Input Stream, Frame Input Queued, or Frame Input Single-Point.

I32

number to read is the number of frame values desired.

If **number to read** is positive (or 0), the **data** array size is no greater than this number.

If **number to read** is negative (typically –1), all available frame values are returned. If **number to read** is negative, you must use a timeout of 0.

This input is optional. The default value is –1.

If the session mode is Frame Input Single-Point, set **number to read** to either –1 or the number of frames in the session list. This ensures that **XNET Read (Frame LIN).vi** can return the current value of all session frames.

DBL

timeout is the time to wait for **number to read** frame values to become available.

The timeout is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, **XNET Read (Frame LIN).vi** waits for **number to read** frame values, then returns that number. If the values do not arrive prior to the timeout, an error is returned.

If **timeout** is negative, **XNET Read (Frame LIN).vi** waits indefinitely for **number to read** frame values.

If **timeout** is zero, **XNET Read (Frame LIN).vi** does not wait and immediately returns all available frame values up to the limit **number to read** specifies.

This input is optional. The default value is 0.0.

If the session mode is Frame Input Single-Point, you must leave **timeout** unwired (0.0). Because this mode reads the most recent value of each frame, **timeout** does not apply.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



data returns an array of LabVIEW clusters.

Each array element corresponds to a frame the session receives.

For a Frame Input Single-Point session mode, the order of frames in the array corresponds to the order in the session list.

The elements of each cluster are specific to the LIN protocol. For more information, refer to Appendix C, *Summary of the LIN Standard*, or the LIN protocol specification.

For the Frame Input Stream session mode, LIN frames are read in their raw form, without interpretation of their elements using the database. For the Frame Input Single-point and Frame Input Queued session modes, information from the database is used to interpret the LIN frames for ease of use.

The following cluster description applies to session modes Frame Input Single-point and Frame Input Queued. For these modes, the cluster elements are:



identifier is the LIN frame identifier.

The identifier is a number from 0 to 63. This number identifies the content of the data contained within **payload**.

The location of this ID within the frame depends on the value of **event slot?**. If **event slot?** is false, this ID is taken from the frame's header. If **event slot?** is true, this ID is taken from the first

payload byte. This ensures that the number identifies the payload, regardless of how it was scheduled.

Regardless of its location, this is the unprotected ID, without parity applied. For more information about LIN ID protection, refer to Appendix C, *Summary of the LIN Standard*.



event slot? is a Boolean value that specifies whether the frame was received within an event-triggered schedule entry (slot). If the value is true, the frame was received within an event-triggered slot. If the value is false, the frame was received within an unconditional or sporadic slot.

When this value is true, **event ID** contains the ID from the frame's header.



event ID is the identifier for an event-triggered slot (**event slot?** true).

When **event slot?** is true, **event ID** is the ID from the frame's header. The **event ID** is a number from 0 to 63. This is the unprotected ID, without parity applied.

When **event slot?** is false, this value does not apply (it is 0).



echo? is a Boolean value that determines whether the frame was an echo of a successful transmit (true), or received from the network (false).

This value is true only when you enable echo of transmitted frames by setting the XNET Session Interface:Echo Transmit? property to True.



type is the frame type (decimal value in parentheses):

LIN Data (64) The LIN data frame contains payload data.

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 (refer to *Start Interface* for more information). For

information about this frame, including the other frame fields, refer to *Special Frames*.

LIN Bus Error (65) 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*.



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



payload is the array of data bytes for the LIN data frame.

The array size indicates the received frame's payload length. According to the LIN protocol, this payload is 0–8 bytes in length.

If the frame payload is used within an event-triggered schedule entry (slot), the first byte of **payload** is the identifier of the frame in its protected form (checksum applied). This is required by the LIN standard even if the frame transmits in an unconditional or sporadic slot. For this type of LIN frame, the actual data (for example, signal values) is limited to 7 bytes.

For example, assume that frame ID 5 is received in an unconditional slot and an event-triggered slot of ID 9. When you receive from the unconditional slot, **identifier** is 5, **event slot?** is false, **event ID** is 0, and the first payload byte contains 5 with checksum applied. When you receive from the event-triggered slot, **identifier** is 5, **event slot?** is true, **event ID** is 9, and the first payload byte contains 5 with checksum applied. Regardless of how the frame is received, you can use the **identifier** to determine the contents of the actual payload data contents in bytes 2–8.

The following cluster description applies to session mode Frame Input Stream. For this mode, the cluster elements are:



identifier is the identifier received within the frame's header.

The identifier is a number from 0 to 63.

If the schedule entry (slot) is unconditional or sporadic, this identifies the payload data (LIN frame). If the schedule entry is event triggered, this identifies the schedule entry itself, and the

protected ID contained in the first payload byte identifies the payload.



event slot? is not used. This element is false.



event ID is not used. This element is 0.



echo? uses the same semantics as the previous description for Frame Input Queued.



type uses the same semantics as the previous description for Frame Input Queued.



timestamp uses the same semantics as the previous description for Frame Input Queued.



payload uses the same semantics as the previous description for Frame Input Queued.

For an example of how this data applies to network traffic, refer to *Frame Input Stream Mode*, *Frame Input Queued Mode*, or *Frame Input Single-Point Mode*.



error out is the error cluster output (refer to *Error Handling*).

Description

The data represents an array of LIN frames. Each LIN frame uses a LabVIEW cluster with LIN-specific elements.

The LIN frames are associated to the session's list of frames as follows:

- Frame Input Stream Mode: Array of all frame values received (list ignored).
- Frame Input Queued Mode: Array of frame values received for the single frame specified in the list.
- Frame Input Single-Point Mode: Array of single frame values, one for each frame specified in the list.

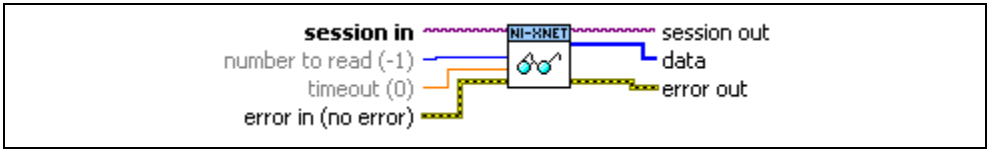
Due to issues with LabVIEW memory allocation for clusters with an array, this **XNET Read.vi** instance can introduce jitter to a high-priority loop on LabVIEW Real-Time (RT) (refer to *High Priority Loops* for more information). The **XNET Read (Frame Raw).vi** instance provides optimal performance for high-priority loops.

XNET Read (Frame Raw).vi

Purpose

Reads data from a session as an array of raw bytes.

Format



Inputs

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Frame Input Stream Mode, Frame Input Queued Mode, or Frame Input Single-Point Mode.

number to read is the number of bytes (U8) desired.

This number does not represent the number of frames to read. As encoded in raw data, each frame can vary in length. Therefore, the number represents the maximum raw bytes to read, not the number of frames.

Standard CAN and LIN frames are always 24 bytes in length. If you want to read a specific number of frames, multiply that number by 24.

CAN FD and FlexRay frames vary in length. For example, if you pass **number to read** of 91, the **data** might return 80 bytes, within which the first 24 bytes encode the first frame, and the next 56 bytes encode the second frame.

If **number to read** is positive (or 0), the data array size is no greater than this number. The minimum size for a single frame is 24 bytes.

If **number to read** is negative (typically –1), all available raw data is returned. If **number to read** is negative, you must use a **timeout** of 0.

This input is optional. The default value is –1.

If the session mode is Frame Input Single-Point, set **number to read** to –1. This ensures that **XNET Read (Frame Raw).vi** can return the current value of all session frames.



timeout is the time to wait for **number to read** frame bytes to become available.

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

The timeout is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, **XNET Read (Frame Raw).vi** waits for **number to read** frame bytes to be received, then returns complete frames up to that number. If the bytes do not arrive prior to the timeout, an error is returned.

If **timeout** is negative, **XNET Read (Frame Raw).vi** waits indefinitely for **number to read** frame bytes.

If **timeout** is zero, **XNET Read (Frame Raw).vi** does not wait and immediately returns all available frame bytes up to the limit **number to read** specifies.

This input is optional. The default value is 0.0.

If the session mode is Frame Input Single-Point, you must leave **timeout** unwired (0.0). Because this mode reads the most recent value of each frame, **timeout** does not apply.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



data returns an array of bytes.

The raw bytes encode one or more frames using the Raw Frame Format. This frame format is the same for read and write of raw data, and it is also used for log file examples.

The data always returns complete frames.

For information about which elements of the raw frame are applicable, refer to the frame read for the protocol in use (**XNET Read (Frame CAN).vi**, **XNET Read (Frame FlexRay).vi**), or **XNET Read (Frame LIN).vi**. For example, when you read FlexRay frames for a Frame Input Queued session, the only frame type is FlexRay Data (other types apply to Frame Input Stream only).

For an example of how this data applies to network traffic, refer to *Frame Input Stream Mode*, *Frame Input Queued Mode*, or *Frame Input Single-Point Mode*.



error out is the error cluster output (refer to *Error Handling*).

Description

The raw bytes encode one or more frames using the Raw Frame Format. The session must use Frame Input Stream Mode, Frame Input Queued Mode, Frame Input Single-Point Mode, PDU Input Queued Mode (similar to Frame Input Queued Mode), or PDU Input Single-Point Mode (similar to Frame Input Single-Point Mode). The raw frame format is protocol independent, so the session can use either a CAN, FlexRay, or LIN interface.

The raw frame format matches the format of data transferred to/from the XNET hardware. Because it is not converted to/from LabVIEW clusters for ease of use, it is more efficient with regard to performance. This **XNET Read.vi** instance typically is used to read raw frame data from the interface and log the data to a file for later analysis. The NI-XNET examples provide code to read the raw frame data from the log file and convert the raw data into protocol-specific LabVIEW clusters.

The raw frames are associated to the session's list of frames as follows:

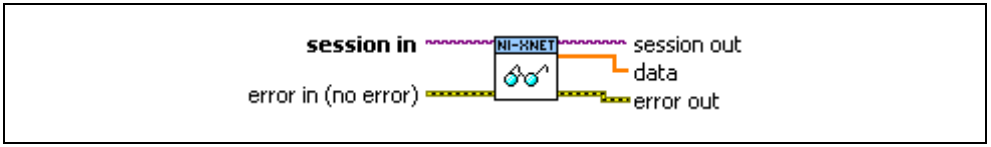
- Frame Input Stream Mode: Array of all frame values received (list ignored).
- Frame Input Queued Mode: Array of frame values received for the single frame specified in the list.
- Frame Input Single-Point Mode: Array of single frame values, one for each frame specified in the list.
- PDU Input Queued Mode: Array of frame (PDU payload) values received for the single PDU specified in the list. This mode is similar to Frame Input Queued Mode.
- PDU Input Single-Point Mode: Array of single frame (PDU payload) values, one for each PDU specified in the list. This mode is similar to Frame Input Single-Point Mode.

XNET Read (Signal Single-Point).vi

Purpose

Reads data from a session of Signal Input Single-Point Mode.

Format



Inputs

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Signal Input Single-Point Mode.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

data returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data returns the most recent value received for each signal. If multiple frames for a signal are received since the previous call to **XNET Read (Signal Single-Point).vi** (or session start), only signal data from the most recent frame is returned.

If no frame is received for the corresponding signals since you started the session, the XNET Signal Default Value is returned.

For an example of how this data applies to network traffic, refer to *Signal Input Single-Point Mode*.

A trigger signal returns a value of 1.0 or 0.0, depending on whether its frame arrived since the last Read (or Start) or not. For more information about trigger signals, refer to *Signal Input Single-Point Mode*.

error out is the error cluster output (refer to *Error Handling*).

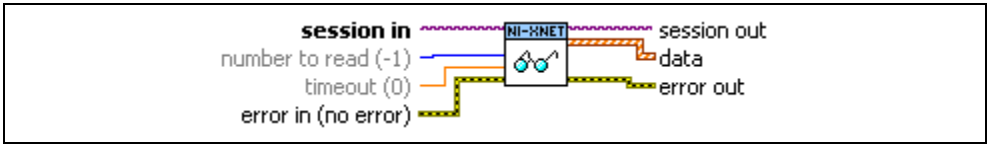
XNET Read (Signal Waveform).vi

Purpose

Reads data from a session of Signal Input Waveform Mode.

The data represents a waveform of resampled values for each signal in the session. You can wire the data directly to a LabVIEW Waveform Graph for display.

Format



Inputs

I/O

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Signal Input Waveform.

I32

number to read is the number of samples desired.

If **number to read** is positive (or 0), the number of samples returned (size of **Y** arrays) is no greater than this number. If **timeout** is nonzero, the number returned is exactly this number on success.

If **number to read** is negative (typically –1), the maximum number of samples is returned. If **number to read** is negative, you must use a **timeout** of zero.

This input is optional. The default value is –1.

DBL

timeout is the time to wait for **number to read** samples to become available.

The timeout is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, **XNET Read (Signal Waveform).vi** waits for **number to read** samples, then returns that number. If the samples do not arrive prior to the timeout, an error is returned.

If **timeout** is negative, **XNET Read (Signal Waveform).vi** waits indefinitely for **number to read** samples.

If **timeout** is zero, **XNET Read (Signal Waveform).vi** does not wait and immediately returns all available samples up to the limit **number to read** specifies.

Because time determines sample availability, typical values for this **timeout** are 0 (return available) or a large positive value such as 100.0 (wait for a specific **number to read**). This input is optional. The default value is 0.0.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



data returns a one-dimensional array of LabVIEW waveforms.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The waveform elements are:



t0 is the waveform start time. This is a LabVIEW absolute timestamp that specifies the time for the first sample in the **Y** array.



dt is the waveform delta time. This is a LabVIEW relative time that specifies the time between each sample in the **Y** array. LabVIEW relative time is represented as 64-bit floating point in units of seconds. The waveform **dt** always is the inverse of the XNET Session Resample Rate property.



Y is the array of resampled signal values. Each signal value is scaled, 64-bit floating point.

The **Y** array size is the same for all waveforms returned, because it is determined based on time, and not the number of frames received.

If no frame is received for the corresponding signals since you started the session, the XNET Signal Default Value is returned.

For an example of how this data applies to network traffic, refer to *Signal Input Waveform Mode*.



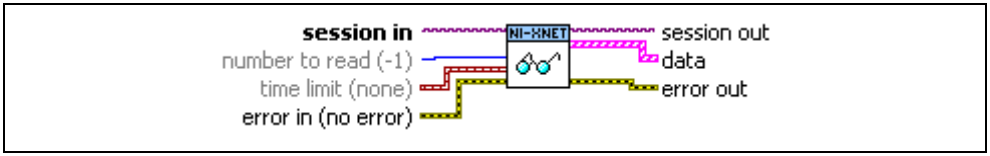
error out is the error cluster output (refer to *Error Handling*).

XNET Read (Signal XY).vi

Purpose

Reads data from a session of Signal Input XY Mode.

Format



Inputs

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Signal Input XY.

number to read is the number of values desired.

If **number to read** is positive (or 0), the size of **value** arrays is no greater than this number.

If **number to read** is negative (typically –1), the maximum number of values is returned.

This input is optional. The default value is –1.

If **number to read** values are received for any signal, **XNET Read (Signal XY).vi** returns those values, even if the **time limit** has not occurred. Therefore, to read values up to the time limit, leave **number to read** unwired (–1).

time limit is the timestamp to wait for before returning signal values.

If **time limit** is valid, **XNET Read (Signal XY).vi** waits for the timestamp to occur, then returns available values (up to **number to read**). If you increment **time limit** by a fixed number of seconds for each call to **XNET Read (Signal XY).vi**, you effectively obtain a moving window of signal values.

If **time limit** is unwired (invalid), **XNET Read (Signal XY).vi** returns immediately all available values up to the current time (up to **number to read**).

This input is optional. The default value is an invalid timestamp.

The **timeout** of other **XNET Read.vi** instances specifies the maximum amount time to wait for a specific **number to read** values. The **time limit** of **XNET Read (Signal XY).vi** does not specify a worst-case timeout value, but rather a specific absolute timestamp to wait for.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



data returns an array of LabVIEW clusters.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

Each cluster contains two arrays, one for **timestamp** and one for **value**. For each signal, the size of the **timestamp** and **value** arrays always is the same, such that it represents a single array of timestamp/value pairs.

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

The cluster elements are:



timestamp is the array of LabVIEW timestamps, one for each frame received that contains the signal.

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



value is the array of signal values, one for each frame received that contains the signal.

Each signal value is scaled, 64-bit floating point.

The **value** array size is the same as the **timestamp** array size.

For an example of how this data applies to network traffic, refer to *Signal Input XY Mode*.

When you use this instance with a session of Signal Input Single-Point Mode, **time limit** and **number to read** are ignored, and the **timestamp** and **value** arrays always contain only one element per signal. This effectively returns a single pair of timestamp and value for every signal.



error out is the error cluster output (refer to *Error Handling*).

Description

You also can use this instance to read data from a session of Signal Input Single-Point Mode, although **XNET Read (Signal Single-Point).vi** is more common for that mode.

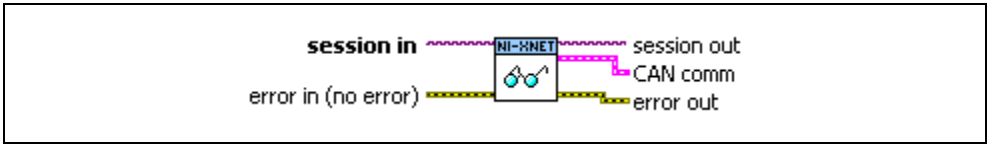
The data represents an XY plot of timestamp/value pairs for each signal in the session. You can wire the data directly to a LabVIEW XY Graph for display.

XNET Read (State CAN Comm).vi

Purpose

Reads the state of CAN communication using an XNET session.

Format



Inputs

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

CAN comm returns a LabVIEW cluster containing the communication elements. The elements are:

communication state specifies the CAN interface state with respect to error confinement (decimal value in parentheses):

Error Active (0) This state reflects normal communication, with few errors detected. The CAN interface remains in this state as long as **receive error counter** and **transmit error counter** are both below 128.

Error Passive (1) If either the **receive error counter** or **transmit error counter** increment above 127, the CAN interface transitions into this state. Although communication proceeds, the CAN device generally is assumed to have problems with receiving frames.

When a CAN interface is in error passive state, acknowledgement errors do not increment the **transmit error counter**.

Therefore, if the CAN interface transmits a frame with no other device (ECU) connected, it eventually enters error passive state due to retransmissions, but does not enter bus off state.

Bus Off (2) If the **transmit error counter** increments above 255, the CAN interface transitions into this state. Communication immediately stops under the assumption that the CAN interface must be isolated from other devices.

When a CAN interface transitions to the bus off state, communication stops for the interface. All NI-XNET sessions for the interface no longer receive or transmit frame values. To restart the CAN interface and all its sessions, call **XNET Start.vi**.

Init (3) This is the CAN interface initial state on power-up. The interface is essentially off, in that it is not attempting to communicate with other nodes (ECUs).

When the start trigger occurs for the CAN interface, it transitions from the Init state to the Error Active state. When the interface stops due to a call to **XNET Stop.vi**, the CAN interface transitions from either Error Active or Error Passive to the Init state. When the interface stops due to the Bus Off state, it remains in that state until you restart.



transceiver error? indicates whether an error condition exists on the physical transceiver. This is typically referred to as the transceiver chip NERR pin. False indicates normal operation (no error), and true indicates an error.



sleep? indicates whether the transceiver and communication controller are in their sleep state. False indicates normal operation (awake), and true indicates sleep.



last error specifies the status of the last attempt to receive or transmit a frame (decimal value in parentheses):

None (0) The last receive or transmit was successful.

Stuff (1) More than 5 equal bits have occurred in sequence, which the CAN specification does not allow.

Form (2) A fixed format part of the received frame used the wrong format.

Ack (3) Another node (ECU) did not acknowledge the frame transmit.

If you call **XNET Write.vi** and do not have a cable connected, or the cable is connected to a node that is not communicating, you see this error repeatedly. The CAN **communication state** eventually transitions to Error Passive, and the frame transmit retries indefinitely.

Bit 1 (4) During a frame transmit (with the exception of the arbitration ID field), the interface wanted to send a recessive bit (logical 1), but the monitored bus value was dominant (logical 0).

Bit 0 (5) During a frame transmit (with the exception of the arbitration ID field), the interface wanted to send a dominant bit (logical 0), but the monitored bus value was recessive (logical 1).

CRC (6) The CRC contained within a received frame does not match the CRC calculated for the incoming bits.



The **receive error counter** begins at 0 when communication starts on the CAN interface. The counter increments when an error is detected for a received frame and decrements when a frame is received successfully. The counter increases more for an error than it is decreased for success. This ensures that the counter generally increases when a certain ratio of frames (roughly 1/8) encounter errors.



The **transmit error counter** begins at 0 when communication starts on the CAN interface. The counter increments when an error is detected for a transmitted frame and decrements when a frame transmits successfully. The counter increases more for an error than it is decreased for success. This ensures that the counter generally increases when a certain ratio of frames (roughly 1/8) encounter errors.

When **communication state** transitions to Bus Off, the **transmit error counter** no longer is valid.



fault? indicates that a fault occurred, and its code is available as **fault code**.



fault code returns a numeric code you can use to obtain a description of the fault. If **fault?** is false, the fault code is 0.

A fault is an error that occurs asynchronously to the NI-XNET VIs your application calls. The fault cause may be related to CAN communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, **XNET Read (State CAN Comm).vi** provides a detection method distinct from the error out of NI-XNET VIs, yet easy to use alongside the common practice of checking the communication state.

To obtain a fault description, wire the **fault code** into the LabVIEW **Simple Error Handler.vi error code** input and view the resulting **message**. You also can bundle the **fault code** into a LabVIEW error cluster as the **code** element and use front panel features to view the error description.



error out is the error cluster output (refer to *Error Handling*).

Description

You can use **XNET Read (State CAN Comm).vi** with any XNET session mode, as long as the session interface is CAN. Because the state reflects the CAN interface, it can apply to multiple sessions.

Your application can use **XNET Read (State CAN Comm).vi** to check for problems on the CAN network independently from other aspects of your application. For example, you intentionally may introduce noise into the CAN cables to test how your ECU behaves under these conditions. When you do this, you do not want the **error out** of NI-XNET VIs to return errors, because this may cause your application to stop. Your application can use **XNET Read (State CAN Comm).vi** to read the CAN network state quickly as data, so that it does not introduce errors into the flow of your LabVIEW VIs.

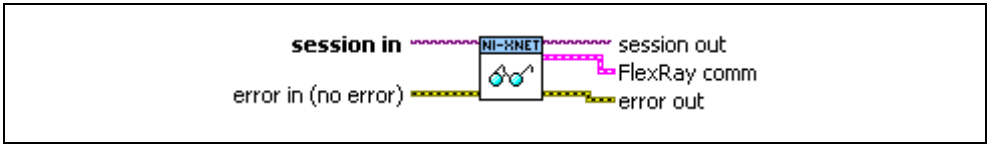
Alternately, to log bus errors, you can set the Interface:Bus Error Frames to Input Stream? property to cause CAN bus errors to be logged as a special frame (refer to *Special Frames* for more information) into a Frame Stream Input queue.

XNET Read (State FlexRay Comm).vi

Purpose

Reads the state of FlexRay communication using an XNET session.

Format



Inputs

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

FlexRay comm returns a LabVIEW cluster containing the communication elements. The elements are:

POC state specifies the FlexRay interface state (decimal value in parentheses):

Default Config (0) This is the FlexRay interface initial state on power-up. The interface is essentially off, in that it is not configured and is not attempting to communicate with other nodes (ECUs).

Ready (1) When the interface starts, it first enters Config state to validate the FlexRay cluster and interface properties. Assuming the properties are valid, the interface transitions to this Ready state.

In the Ready state, the FlexRay interface attempts to integrate (synchronize) with other nodes in the network cluster. This integration process can take several

	FlexRay cycles, up to 200 ms. If the integration succeeds, the interface transitions to Normal Active. You can use **XNET Read (State Time Start).vi** to read the time when the FlexRay interface entered Ready. If integration succeeds, you can use **XNET Read (State Time Comm).vi** to read the time when the FlexRay entered Normal Active.
Normal Active (2)	This is the normal operation state. The NI-XNET interface is adequately synchronized to the cluster to allow continued frame transmission without disrupting the transmissions of other nodes (ECUs). If synchronization problems occur, the interface can transition from this state to Normal Passive.
Normal Passive (3)	Frame reception is allowed, but frame transmission is disabled due to degraded synchronization with the cluster remainder. If synchronization improves, the interface can transition to Normal Active. If synchronization continues to degrade, the interface transitions to Halt.
Halt (4)	Communication halted due to synchronization problems. When the FlexRay interface is in Halt state, all NI-XNET sessions for the interface stop, and no frame values are received or transmitted. To restart the FlexRay interface, you must restart the NI-XNET sessions. If you clear (close) all NI-XNET sessions for the interface, it transitions from Halt to Default Config state.
Config (15)	This state is transitional when configuration is valid. If you detect this state after starting the interface, it typically indicates a problem with the configuration.

Check the **fault?** output for a fault. If no fault is returned, check your FlexRay cluster and interface properties. You can check the validity of these properties using the NI-XNET Database Editor, which displays invalid configuration properties.

In the FlexRay specification, this value is referred to as the Protocol Operation Control (POC) state. For more information about the FlexRay POC state, refer to Appendix B, *Summary of the FlexRay Standard*.



clock correction failed returns the number of consecutive even/odd cycle pairs that have occurred without successful clock synchronization.

If this count reaches the value in the XNET Cluster FlexRay:Max Without Clock Correction Passive property, the FlexRay interface POC state transitions from Normal Active to Normal Passive state. If this count reaches the value in the XNET cluster FlexRay:Max Without Clock Correction Fatal property, the FlexRay interface POC state transitions from Normal Passive to Halt state.

In the FlexRay specification, this value is referred to as *vClockCorrectionFailed*.



passive to active count returns the number of consecutive even/odd cycle pairs that have occurred with successful clock synchronization.

This count increments while the FlexRay interface is in POC state Error Passive. If the count reaches the value in the XNET Session Interface:FlexRay:Allow Passive to Active property, the interface POC state transitions to Normal Active.

In the FlexRay specification, this value is referred to as *vAllowPassiveToActive*.



fault? indicates that a fault occurred, and its code is available is fault code.



fault code returns a numeric code you can use to obtain a fault description. If **fault?** is false, the fault code is 0.

A fault is an error that occurs asynchronously to the NI-XNET VIs your application calls. The fault cause may be related to FlexRay communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, **XNET Read (State FlexRay Comm).vi** provides a detection method distinct from the **error out** of NI-XNET VIs, yet easy to use alongside the common practice of checking the communication state.

To obtain a fault description fault, wire the **fault code** into the LabVIEW **Simple Error Handler.vi error code** input and view the resulting **message**. You also can bundle the **fault code** into a LabVIEW error cluster as the **code** element and use front panel features to view the error description.



channel A sleep? indicates whether channel A currently is asleep.



channel B sleep? indicates whether channel B currently is asleep.



error out is the error cluster output (refer to *Error Handling*).

Description

You can use **XNET Read (State FlexRay Comm).vi** with any XNET session mode, as long as the session interface is FlexRay. Because the state reflects the FlexRay interface, it can apply to multiple sessions.

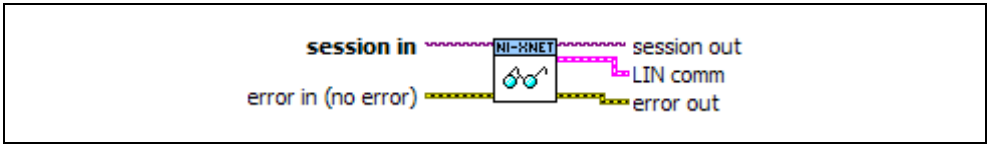
Your application can use **XNET Read (State FlexRay Comm).vi** to check for problems on the FlexRay network independently from the other aspects of your application. For example, you intentionally may introduce noise into the FlexRay cables to test how your ECU behaves under these conditions. When you do this, you do not want the **error out** of NI-XNET VIs to return errors, because this may cause your application to stop. Your application can use **XNET Read (State FlexRay Comm).vi** to read the FlexRay network state quickly as data, so that it does not introduce errors into the flow of your LabVIEW VIs.

XNET Read (State LIN Comm).vi

Purpose

Reads the state of LIN communication using an XNET session.

Format



Inputs

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session must use a LIN interface.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

LIN comm returns a LabVIEW cluster containing the communication elements. The elements are:

communication state specifies the LIN interface state (decimal value in parentheses):

Idle (0): This is the LIN interface initial state on power-up. The interface is essentially off, in that it is not attempting to communicate with other nodes (ECUs).

When the start trigger occurs for the LIN interface, it transitions from the Idle state to the Active state. When the interface stops due to a call to XNET Stop, the LIN interface transitions from either Active or Inactive to the Idle state.

Active (1): This state reflects normal communication. The LIN interface remains in this state as long as bus activity is detected (frame headers received or transmitted).

Inactive (2): This state indicates that no bus activity has been detected in the past four seconds.

Regardless of whether the interface acts as a master or slave, it transitions to this state after four seconds of bus inactivity. As soon as bus activity is detected (break or frame header), the interface transitions to the Active state.

The LIN interface does not go to sleep automatically when it transitions to Inactive. To place the interface into sleep mode, set the XNET Session Interface:LIN:Sleep property when you detect the Inactive state.



sleep? indicates whether the transceiver and communication controller are in their sleep state. False indicates normal operation (awake), and true indicates sleep.

This Boolean value changes from false to true only when you set the XNET Session Interface:LIN:Sleep property to **Remote Sleep** or **Local Sleep**.

This Boolean value changes from true to false when one of the following occurs:

- You set the XNET Session Interface:LIN:Sleep property to **Remote Wake** or **Local Wake**.
- The interface receives a remote wakeup pattern (break). In addition to this **XNET Read** VI, you can wait for a remote wakeup event using **XNET Wait (LIN Remote Wakeup).vi**.



transceiver ready? indicates whether the LIN transceiver is powered from the bus.

True indicates the bus power exists, so it is safe to start communication on the LIN interface.

If this value is false, you cannot start communication successfully. Wire power to the LIN transceiver and run your application again.



last error specifies the status of the last attempt to receive or transmit a frame. It is an enumeration (ring data type). For a table of all values for **last error**, refer to the *Description* section.



last received returns the value received from the network when **last error** occurred.



last expected returns the value that the LIN interface expected to see (instead of **last received**).



last identifier returns the frame identifier in which the last error occurred.



fault? indicates that a fault occurred, and its code is available as **fault code**.



fault code returns a numeric code you can use to obtain a description of the fault. If **fault?** is false, the fault code is 0.

A fault is an error that occurs asynchronously to the NI-XNET VIs your application calls. The fault cause may be related to LIN communication, but it also can be related to XNET hardware, such as a fault in the onboard processor. Although faults are extremely rare, the **XNET Read (State LIN Comm).vi** provides a detection method distinct from the error out of NI-XNET VIs, yet easy to use alongside the common practice of checking the communication state.

To obtain a fault description, wire the **fault code** into the LabVIEW **Simple Error Handler.vi error code** input and view the resulting **message**. You also can bundle the **fault code** into a LabVIEW error cluster as the **code** element and use front panel features to view the error description.

For more information, refer to *Fault Handling*.



schedule index indicates the LIN schedule that the interface is currently running.

This index refers to a LIN schedule that you requested using **XNET Write (State LIN Schedule Change).vi**. It indexes the array of schedules that are represented in the XNET Session Interface:LIN:Schedules property.

This index applies only when the LIN interface is running as a master. If the LIN interface is running as a slave only, this element should be ignored.



error out is the error cluster output (refer to *Error Handling*).

Description

You can use **XNET Read (State LIN Comm).vi** with any XNET session mode, as long as the session interface is LIN. Because the state reflects the LIN interface, it can apply to multiple sessions.

Your application can use **XNET Read (State LIN Comm).vi** to check for problems on the LIN network independently from other aspects of your application. For example, you intentionally may introduce noise into the LIN cables to test how your ECU behaves under these conditions. When you do this, you do not want the **error out** of NI-XNET VIs to return errors, because this may cause your application to stop. Your application can use **XNET Read (State LIN Comm).vi** to read the LIN network state quickly as data, so that it does not introduce errors into the flow of your LabVIEW VIs.

The following table lists each value for **last error**, along with a description, and applicable use of **last received**, **last expected**, and **last identifier**. In the **last error** column, the decimal value is shown in parentheses after the string name.

Last Error	Description	Last Received	Last Expected	Last Identifier
None (0)	No bus error has occurred since the previous communication state read.	0 (N/A)	0 (N/A)	0 (N/A)
Unknown ID (1)	Received a frame identifier that is not valid (0–63).	0 (N/A)	0 (N/A)	0 (N/A)
Form (2)	The form of a received frame is incorrect. For example, the database specifies 8 bytes of payload, but you receive only 4 bytes.	0 (N/A)	0 (N/A)	Received frame ID
Framing (3)	The byte framing is incorrect (for example, a missing stop bit).	0 (N/A)	0 (N/A)	Received frame ID
Readback (4)	The interface transmitted a byte, but the value read back from the transceiver was different. This often is caused by a cabling problem, such as noise.	Value read back	Value transmitted	Received frame ID

Last Error	Description	Last Received	Last Expected	Last Identifier
Timeout (5)	Receiving the frame took longer than the LIN-specified timeout.	0 (N/A)	0 (N/A)	Received frame ID
Checksum (6)	The received checksum was different than the expected checksum.	Received checksum	Calculated checksum	Received frame ID

If the bus error is detected at time when no frame ID is received (such as wakeup), **last identifier** uses the special value 64.

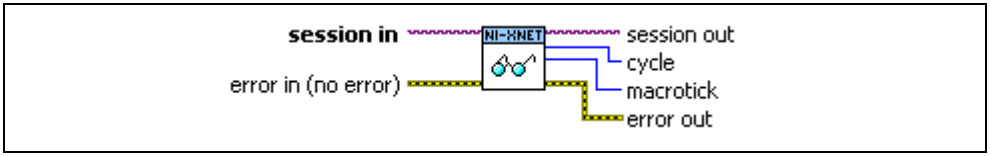
Alternately, to log bus errors, you can set the Interface:Bus Error Frames to Input Stream? property to cause LIN bus errors to be logged as a special frame (refer to *Special Frames* for more information) into a Frame Stream Input queue.

XNET Read (State FlexRay Cycle Macrotick).vi

Purpose

Reads the current FlexRay global time using an XNET session.

Format



Inputs

session in is the session to read. This session is selected from a LabVIEW project or returned from **XNET Create Session.vi**.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

cycle returns the current FlexRay cycle counter. The cycle counter range is 0–63. In the FlexRay specification, the current cycle counter is referred to as *vCycleCounter*.

The XNET Cluster FlexRay:Cycle property returns the cycle length in microseconds.

macrotick returns the current FlexRay macrotick. In the FlexRay specification, the current macrotick is referred to as *vMacrotick*.

The XNET Cluster FlexRay:Macro Per Cycle property returns the number of macroticks in the cycle. The current macrotick returned from this **XNET Read.vi** instance ranges from 0 to (FlexRay:Macro Per Cycle – 1).

The XNET Cluster FlexRay:Macrotick property returns the macrotick length in floating-point seconds.

error out is the error cluster output (refer to *Error Handling*).

Description

Global time represents the timebase that all ECUs on the FlexRay network cluster share. You use sync frames to synchronize the global time. The global time components are the current cycle counter and macrotick within the cycle. For more information about global time, refer to Appendix B, *Summary of the FlexRay Standard*.

You can use this **XNET Read.vi** instance with any XNET session mode, as long as the session interface is FlexRay. Because the state reflects the FlexRay interface, it can apply to multiple sessions.

For this VI to operate properly, you must connect FlexRay global time as the FlexRay interface timebase source. To do this, you must call **XNET Connect Terminals.vi** with a source of FlexRay Macrotick and destination of Master Timebase. If the terminals are not connected in this manner, this **XNET Read.vi** instance returns an error.

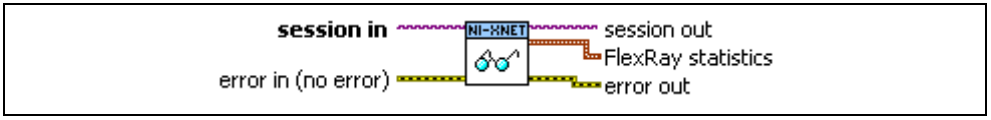
When using LabVIEW Real-Time, this VI often is useful in conjunction with **XNET Create Timing Source (FlexRay Cycle).vi**. The FlexRay Cycle timing source enables a LabVIEW timed loop to execute at a specific macrotick within the cycle. Only one FlexRay Cycle timing source is allowed within the cycle. Within the timed loop, you can read the current FlexRay global time to measure performance or synchronize LabVIEW code to additional macroticks in the cycle.

XNET Read (State FlexRay Statistics).vi

Purpose

Reads statistics for FlexRay communication using an XNET session.

Format



Inputs

session in is the session to read. This session is selected from a LabVIEW project or returned from **XNET Create Session.vi**.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

FlexRay statistics returns a LabVIEW cluster that contains the statistical elements. The elements are:

num syntax error ch A is the number of syntax errors that have occurred on channel A since communication started.

A syntax error occurs if:

- A node starts transmitting while the channel is not in the idle state.
- There is a decoding error.
- A frame is decoded in the symbol window or in the network idle time.
- A symbol is decoded in the static segment, dynamic segment, or network idle time.
- A frame is received within the slot after reception of a semantically correct frame (two frames in one slot).
- Two or more symbols are received within the symbol window.

num syntax error ch B is the number of syntax errors that have occurred on channel B since communication started.



num content error ch A is the number of content errors that have occurred on channel A since communication started.

A content error occurs if:

- In a static segment, the payload length of a frame does not match the global cluster property.
- In a static segment, the Startup indicator (bit) is 1 while the Sync indicator is 0.
- A frame ID encoded in the frame header does not match the current slot.
- A cycle count encoded in the frame's header does not match the current cycle count.
- In a dynamic segment, the Sync indicator is 1.
- In a dynamic segment, the Startup indicator is 1.
- In a dynamic segment, the Null indicator is 0.



num content error ch B is the number of content errors that have occurred on channel B since communication started.



num slot boundary violation ch A is the number of slot boundary violations that have occurred on channel A since communication started.

A slot boundary violation error occurs if the interface does not consider the channel to be idle at the boundary of a slot (either beginning or end).



num slot boundary violation ch B is the number of slot boundary violations that have occurred on channel B since communication started.

For more information about these statistics, refer to Appendix B, *Summary of the FlexRay Standard*.



error out is the error cluster output (refer to *Error Handling*).

Description

You can use this **XNET Read.vi** instance with any XNET session mode, as long as the session's interface is FlexRay. Because the state reflects the FlexRay interface, it can apply to multiple sessions.

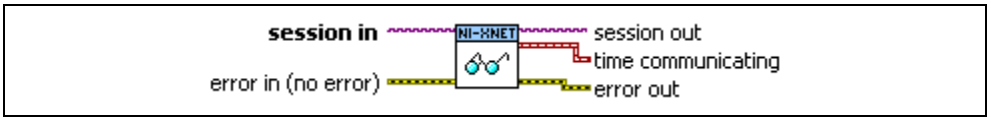
Like other **XNET Read.vi** instances, this VI executes quickly, so it is appropriate for real-time loops. The statistical information is updated during the Network Idle Time (NIT) of each FlexRay cycle.

XNET Read (State Time Comm).vi

Purpose

Reads the time at which the session's interface completed its integration with the network cluster. This represents the time at which communication began.

Format



Inputs



session in is the session to read. This session is selected from a LabVIEW project or returned from **XNET Create Session.vi**.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



time communicating returns the communication time of the interface as a LabVIEW absolute timestamp.

If the interface is not communicating when this read is called, **time communicating** returns an invalid time (0).



error out is the error cluster output (refer to *Error Handling*).

Description

You can use this **XNET Read.vi** instance with any XNET session mode. Because the time is associated with the interface, it can apply to multiple sessions.

This **XNET Read.vi** instance returns time as a LabVIEW absolute timestamp data type.

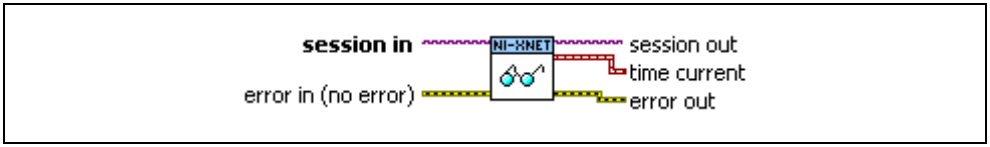
After your application starts the XNET interface hardware, the communication controller begins to integrate with ECUs in the network. The timestamp at which this integration starts is available using **XNET Read (State Time Start).vi**. Once the XNET interface is fully integrated and communicating on the network (transmitting and receiving frames), this VI captures and returns the time. For the CAN protocol, the time difference between Start and Communicating is very small. For the FlexRay protocol, the time difference can be many milliseconds due to factors such as clock synchronization and cycle length.

XNET Read (State Time Current).vi

Purpose

Reads the current interface timestamp using an XNET session.

Format



Inputs

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

time current returns the current interface timestamp as a LabVIEW absolute time. If the interface is not started when **XNET Read (State Time Current).vi** is called, **time current returns** an invalid time (0).

error out is the error cluster output (refer to *Error Handling*).

Description

You can use **XNET Read (State Time Current).vi** with any XNET session mode. Because the time is associated with the interface, it can apply to multiple sessions.

XNET Read (State Time Current).vi returns time as a LabVIEW absolute timestamp data type. The timestamp represents absolute time that the interface timebase source drives. You use the timebase source to timestamp frames the interface receives. For a CAN interface, you use the timebase source to schedule cyclic frame transmit.

The interface timebase source is not necessarily connected to the LabVIEW CPU clock, so this timestamp can drift relative to the LabVIEW time used for internally sourced timed loops and **Get Date/Time in Seconds.vi**.

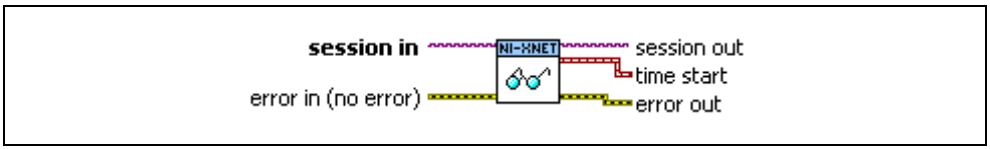
For more information about the interface timebase source, refer to **XNET Connect Terminals.vi**.

XNET Read (State Time Start).vi

Purpose

Reads the time when the session interface started its integration.

Format



Inputs



session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



time start returns the interface start time as a LabVIEW absolute timestamp.

If the interface is not started when **XNET Read (State Time Start).vi** is called, **time start** returns an invalid time (0).



error out is the error cluster output (refer to *Error Handling*).

Description

You can use **XNET Read (State Time Start).vi** with any XNET session mode. Because the time is associated with the interface, it can apply to multiple sessions.

XNET Read (State Time Start).vi returns time as a LabVIEW absolute timestamp data type.

Your application typically starts the interface simply by calling **XNET Read.vi** or **XNET Write.vi**, because the XNET Session Auto Start? property is true by default. If you set Auto Start? to false, you start the interface using **XNET Start.vi**. If you use **XNET Connect Terminals.vi** to import a start trigger for the interface, all sessions for that interface wait for the trigger to occur before starting the interface.

Once the interface starts, this VI captures and returns the time. Unless you connect a start trigger, this time generally is known, so this VI may not be useful.

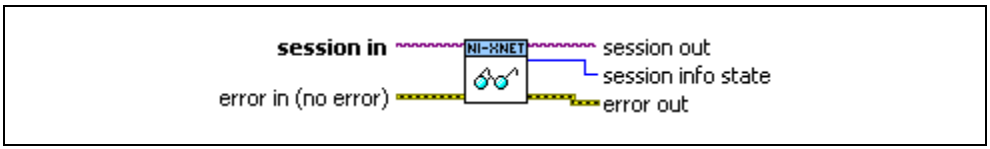
After the XNET interface starts, the communication controller begins to integrate with ECUs in the network. After this integration is complete, the time is captured and available using **XNET Read (State Time Comm).vi**. That time often is useful for FlexRay, because it indicates the time when true communication began.

XNET Read (State Session Info).vi

Purpose

Returns the current state for the session provided.

Format



Inputs

session in is the session to read. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

session info state returns the state of the provided session.

Stopped (0) All frames in the session are stopped.

Started (1) All frames in the session are started.

Mix (2) Some frames in the session are started while other frames are stopped.

error out is the error cluster output (refer to *Error Handling*).

Description

You can use **XNET Read (State Session Info).vi** with any XNET session mode.

XNET Read (State Session Info).vi returns the state of the session's objects. A mixed state may occur when using **XNET Start.vi** or **XNET Stop.vi** with the Session Only option. By reading this state, your application can ensure that all frames in the session have started or stopped.

If the session is started with any option other than Session Only, the state is known, so this VI may not be useful.

XNET Write.vi

Purpose

Writes data to the network using an XNET session.

Description

The instances of this polymorphic VI specify the type of data provided.

XNET Read.vi and **XNET Write.vi** are optimized for real-time performance. **XNET Write.vi** executes quickly and avoids access to shared resources that can induce jitter on other VI priorities.

The **XNET Write.vi** instances are asynchronous, in that data is written to a hardware buffer, but the **XNET Write.vi** returns before the corresponding frames are transmitted onto the network. If you need to wait for the data provided to **XNET Write.vi** to transmit onto the network, use **XNET Wait (Transmit Complete).vi**.

There are two categories of **XNET Write** instance VIs:

- **Signal**: Use when the session mode is Signal Output. The **XNET Write.vi** instance must match the mode exactly (for example, the instance is Signal Waveform when the mode is Signal Output Waveform).
- **Frame**: Use when the session mode is Frame Output. The **XNET Write.vi** instance specifies the desired data type for frames and is not related to the mode. For an easy-to-use data type, use the CAN, FlexRay, or LIN instance.
- **State**: Use to change the session's interface state. You can use these instances in addition to Signal or Frame instances, and they are not related to the mode. These instances are optimized for performance. Although property nodes may provide write access to similar runtime data, those properties are not necessarily optimized for real-time loops.

The **XNET Write** instance VIs are:

- **XNET Write (Signal Single-Point).vi**: The session mode is Signal Output Single-Point.
- **XNET Write (Signal Waveform).vi**: The session mode is Signal Output Waveform.
- **XNET Write (Signal XY).vi**: The session mode is Signal Output XY.
- **XNET Write (Frame CAN).vi**: The session uses a CAN interface, and the mode is Frame Output Stream Mode, Frame Output Single-Point Mode, or Frame Output Queued Mode. Additionally, **XNET Write (Frame CAN).vi** can be called on any signal or frame input session if it contains one or more Event Remote frames (refer to CAN:Timing Type). In this case, it signals an event to transmit those remote frames.
- **XNET Write (Frame FlexRay).vi**: The session uses a FlexRay interface, and the mode is Frame Output Single-Point Mode, Frame Output Queued Mode, PDU Output

Single-Point Mode (similar to Frame Output Single-Point Mode), or PDU Output Queued Mode (similar to Frame Output Queued Mode).

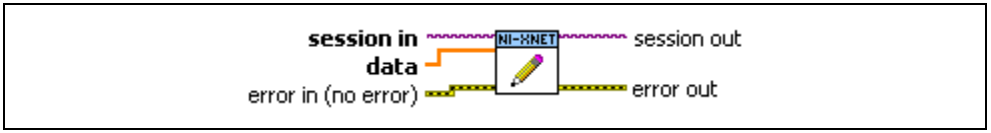
- **XNET Write (Frame LIN).vi**: The session uses a LIN interface, and the mode is Frame Output Stream Mode, Frame Output Single-Point Mode, or Frame Output Queued Mode.
- **XNET Write (Frame Raw).vi**: A data type for frame output that is protocol independent and more efficient than the CAN, FlexRay, and LIN instances.
- **XNET Write (State FlexRay Symbol).vi**: Writes a request for the FlexRay interface to transmit a symbol on the FlexRay bus.
- **XNET Write (State LIN Schedule Change).vi**: Submits a request for the LIN interface to change the running schedule.

XNET Write (Signal Single-Point).vi

Purpose

Writes data to a session of Signal Output Single-Point Mode.

Format



Inputs

session in is the session to write. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Signal Output Single-Point.

data provides a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data provides the value for the next transmit of each signal. If **XNET Write (Signal Single-Point).vi** is called twice before the next transmit, the transmitted frame uses signal values from the second call to **XNET Write (Signal Single-Point).vi**.

For an example of how this data applies to network traffic, refer to *Signal Output Single-Point Mode*.

A trigger signal written a value of 0.0 suppresses writing of its frame's data; writing a value not equal to 0.0 enables it. For more information about trigger signals, refer to *Signal Output Single-Point Mode*.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

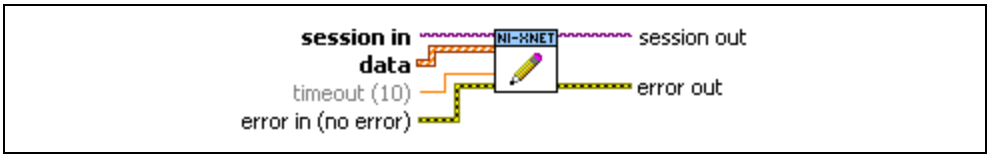
error out is the error cluster output (refer to *Error Handling*).

XNET Write (Signal Waveform).vi

Purpose

Writes data to a session of Signal Output Waveform Mode. The data represents a waveform of resampled values for each signal in the session.

Format



Inputs

session in is the session to write. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Signal Output Waveform.

data provides a one-dimensional array of LabVIEW waveforms.

The data you write is queued up for transmit on the network. Using the default queue configuration for this mode, and assuming a 1000 Hz resample rate, you can safely write 64 frames if you have a sufficiently long timeout. To write more data, refer to the XNET Session Number of Values Unused property to determine the actual amount of queue space available for writing.

For an example of how this data applies to network traffic, refer to *Signal Output Waveform Mode*.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The waveform elements are:

t0 is the waveform start time. This is a LabVIEW absolute timestamp.

This start time is unused (reserved) for Signal Output Waveform mode. If you change it from its default value of 0 (invalid), **XNET Write (Signal Waveform).vi** returns an error.

dt is the waveform delta time. This is a LabVIEW relative time that specifies the time between each sample in the **Y** array.

LabVIEW relative time is represented as 64-bit floating point in units of seconds.

This delta time is unused (reserved) for Signal Output Waveform mode. If you change it from its default value of 0, **XNET Write (Signal Waveform).vi** returns an error.



Y is the array of resampled signal values. Each signal value is scaled, 64-bit floating point.

The **Y** array size must be the same for all waveforms, because the size determines the total timeline for **XNET Write (Signal Waveform).vi**. If the **Y** array sizes are not the same, **XNET Write (Signal Waveform).vi** returns an error.



timeout is the time to wait for the data to be queued for transmit. The timeout does not wait for frames to be transmitted on the network (refer to **XNET Wait (Transmit Complete).vi**).

The timeout is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, **XNET Write (Signal Waveform).vi** waits up to that timeout for space to become available in queues. If the space is not available prior to the timeout, a timeout error is returned.

If **timeout** is negative, **XNET Write (Signal Waveform).vi** waits indefinitely for space to become available in queues.

If **timeout** is 0, **XNET Write (Signal Waveform).vi** does not wait and immediately returns an error if all data cannot be queued. Regardless of the timeout used, if a timeout error occurs, none of the data is queued, so you can attempt to call **XNET Write (Signal Waveform).vi** again at a later time with the same data.

This input is optional. The default value is 10.0 (10 seconds).



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



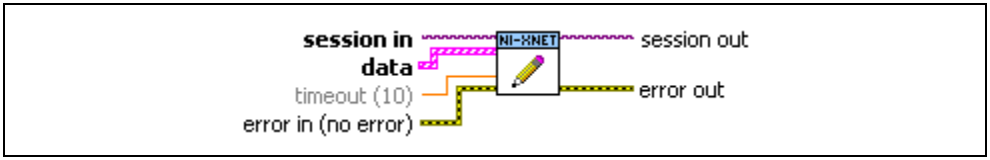
error out is the error cluster output (refer to *Error Handling*).

XNET Write (Signal XY).vi

Purpose

Writes data to a session of Signal Output XY Mode. The data represents a sequence of signal values for transmit using each frame's timing as the database specifies.

Format



Inputs

session in is the session to write. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Signal Output XY.

data provides an array of LabVIEW clusters.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data you write is queued up for transmit on the network. Using the default queue configuration for this mode, you can safely write 64 elements if you have a sufficiently long timeout. To write more data, refer to the XNET Session Number of Values Unused property to determine the actual amount of queue space available for writing.

For an example of how this data applies to network traffic, refer to *Signal Output XY Mode*.

Each cluster contains two arrays, one for value, and one for timestamp. Each value is mapped to a frame for transmit. When signals exist in different frames, the array sizes may be different from one cluster (signal) to another.

The cluster elements are:

timestamp is the array of LabVIEW timestamps.

The **timestamp** array is unused (reserved) for Signal Output XY. If you change it from its default value of empty, **XNET Write (Signal XY).vi** returns an error.



value is the array of signal values, one for each frame that contains the signal. Frame transmission is timed according to the frame properties in the database.

Each signal value is scaled, 64-bit floating point.



timeout is the time to wait for the data to be queued for transmit. The timeout does not wait for frames to be transmitted on the network (refer to **XNET Wait (Transmit Complete).vi**).

The timeout is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, **XNET Write (Signal XY).vi** waits up to that timeout for space to become available in queues. If the space is not available prior to the timeout, a timeout error is returned.

If timeout is negative, **XNET Write (Signal XY).vi** waits indefinitely for space to become available in queues.

If **timeout** is 0, **XNET Write (Signal XY).vi** does not wait and immediately returns with a timeout error if all data cannot be queued. Regardless of the timeout used, if a timeout error occurs, none of the data is queued, so you can attempt to call **XNET Write (Signal XY).vi** again at a later time with the same data.

This input is optional. The default value is 10.0 (10 seconds).



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



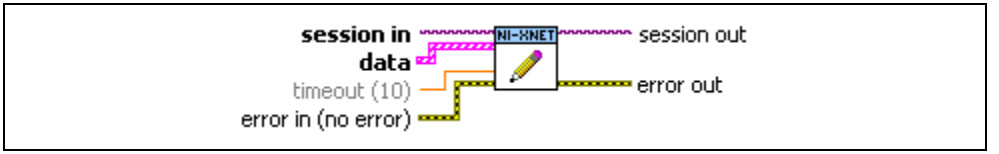
error out is the error cluster output (refer to *Error Handling*).

XNET Write (Frame CAN).vi

Purpose

Writes data to a session as an array of CAN frames. The session must use a CAN interface and Frame Output Stream Mode, Frame Output Queued Mode, or Frame Output Single-Point Mode.

Format



Inputs



session in is the session to write. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Frame Output Stream, Frame Output Queued, or Frame Output Single-Point.



data provides the array of LabVIEW clusters.

Each array element corresponds to a frame value to transmit.

For a Frame Output Single-Point session mode, the order of frames in the array corresponds to the order in the session list.

The data you write is queued up for transmit on the network. Using the default queue configuration for this mode, you can safely write 64 frames if you have a sufficiently long timeout. To write more data, refer to the XNET Session Number of Values Unused property to determine the actual amount of queue space available for write.

For an example of how this data applies to network traffic, refer to *Frame Output Stream Mode*, *Frame Output Queued Mode*, or *Frame Output Single-Point Mode*.

Additionally, **XNET Write (Frame CAN).vi** can be called on any signal or frame input session if it contains one or more Event Remote frames (refer to CAN:Timing Type). In this case, it signals an event to transmit those remote frames. The **data** parameter is ignored in this case, and you can set it to an empty array.

The elements of each cluster are specific to the CAN protocol. For more information, refer to Appendix A, *Summary of the CAN Standard*, or the CAN protocol specification.

The cluster elements are:



identifier is the CAN frame arbitration identifier.

If **extended?** is false, the identifier uses standard format, so 11 bits of this identifier are valid.

If **extended?** is true, the identifier uses extended format, so 29 bits of this identifier are valid.



extended? is a Boolean value that determines whether the identifier uses extended format (true) or standard format (false).



echo? is not used for transmit. You must set this element to false.



type is the frame type (decimal value in parentheses):

CAN Data (0) The CAN data frame contains **payload** data. This is the most commonly used frame type for CAN.

CAN Remote (1) CAN remote frame. Your application transmits a CAN remote frame to request data for the corresponding **identifier**. A remote ECU should respond with a CAN data frame for the **identifier**, which you can obtain using **XNET Read.vi**.



timestamp represents absolute time using the LabVIEW absolute timestamp type. **timestamp** is not used for transmit. You must set this element to the default value, invalid (0).



payload is the array of data bytes for a CAN data frame.

The array size indicates the payload length of the frame value to transmit. According to the CAN protocol, the payload length range is 0–8. For CAN FD, the range can be 0–8, 12, 16, 20, 24, 32, 48, or 64.

When the session mode is Frame Output Single-Point or Frame Output Queued, the number of bytes in the **payload** array must be less than or equal to the Payload Length property of the corresponding frame. You can leave all other CAN frame cluster elements uninitialized. For more information, refer to the section for each mode.

For a transmitted remote frame (**CAN Remote type**), the payload length in the frame value specifies the number of payload bytes requested. Your application provides this payload length by filling **payload** with the requested number of bytes. This enables your application to specify the frame payload length, but the actual values in the **payload** bytes are ignored (not contained in the transmitted frame).



timeout is the time to wait for the CAN frame data to be queued up for transmit.

The timeout is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, **XNET Write (Frame CAN).vi** waits up to that timeout for space to become available in queues. If the space is not available prior to the timeout, a timeout error is returned.

If **timeout** is negative, **XNET Write (Frame CAN).vi** waits indefinitely for space to become available in queues.

If **timeout** is 0, **XNET Write (Frame CAN).vi** does not wait and immediately returns with a timeout error if all data cannot be queued. Regardless of the timeout used, if a timeout error occurs, none of the data is queued, so you can attempt to call **XNET Write (Frame CAN).vi** again at a later time with the same data.

This input is optional. The default value is 10.0 (10 seconds).

If the session mode is Frame Output Single-Point, you must set **timeout** to 0.0. Because this mode writes the most recent value of each frame, **timeout** does not apply.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

The data represents an array of CAN frames. Each CAN frame uses a LabVIEW cluster with CAN-specific elements.

The CAN frames are associated to the session's list of frames as follows:

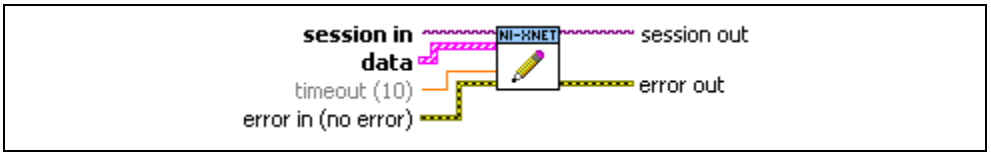
- Frame Output Stream Mode: Array of all frame values for transmit (list ignored).
- Frame Output Queued Mode: Array of frame values to transmit for the single frame specified in the list.
- Frame Output Single-Point Mode: Array of single frame values, one for each frame specified in the list.
- Any signal or frame input mode: The **data** parameter is ignored, and you can set it to an empty array. The VI transmits an event remote frame.

XNET Write (Frame FlexRay).vi

Purpose

Writes data to a session as an array of FlexRay frames. The session must use a FlexRay interface and Frame Output Queued Mode or Frame Output Single-Point Mode.

Format



Inputs



session in is the session to write. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Frame Output Queued or Frame Output Single-Point.

Frame Output Stream mode is not supported for FlexRay.



data provides the array of LabVIEW clusters.

Each array element corresponds to a frame value to transmit.

For a Frame Input Single-Point session mode, the order of frames in the array corresponds to the order in the session list.

The data you write is queued up for transmit on the network. Using the default queue configuration for this mode, and assuming frames with 8 bytes of payload, you can safely write 64 frames if you have a sufficiently long timeout. To write more data, refer to the XNET Session Number of Values Unused property to determine the actual amount of queue space available for write.

For an example of how this data applies to network traffic, refer to *Frame Output Queued Mode* or *Frame Output Single-Point Mode*.

The elements of each cluster are specific to the FlexRay protocol. For more information, refer to Appendix B, *Summary of the FlexRay Standard*, or the *FlexRay Protocol Specification*.

The cluster elements are:



slot specifies the slot number within the FlexRay cycle.



cycle count specifies the cycle number.

The FlexRay cycle count increments from 0 to 63, then rolls over back to 0.



startup? is a Boolean value that specifies whether the frame is a startup frame (true) or not (false).



sync? is a Boolean value that specifies whether the frame is a sync frame (true) or not (false).



preamble? is a Boolean value that specifies the value of the payload preamble indicator in the frame header.

If the frame is in the static segment, **preamble?** being true indicates the presence of a network management vector at the beginning of the payload. The XNET Cluster FlexRay:Network Management Vector Length property specifies the number of bytes at the beginning.

If the frame is in the dynamic segment, **preamble?** being true indicates the presence of a message ID at the beginning of the payload. The message ID is always 2 bytes in length.

If **preamble?** is false, the payload does not contain a network management vector or a message ID.



chA is a Boolean value that specifies whether to transmit the frame on channel A (true) or not (false).



chB is a Boolean value that specifies whether to transmit the frame on channel B (true) or not (false).



echo? is not used for transmit. You must set this element to false.



type is the frame type. **type** is not used for transmit, so you must leave this element uninitialized. All frame values are assumed to be the **FlexRay Data** type. Frames of **FlexRay Data** type contain payload data.

The **FlexRay Null** type is not transmitted based on this type. As specified in the XNET Frame FlexRay:Timing Type property, the FlexRay null frame is transmitted when a cyclically timed frame does not have new data.



timestamp represents absolute time using the LabVIEW absolute timestamp type. **timestamp** is not used for transmit. You must set this element to the default value, invalid (0).

The slot and cycle count specify when the frame transmits in FlexRay global time.



payload is the array of data bytes for FlexRay frames of type **FlexRay Data**.

The array size indicates the payload length of the frame value to transmit. According to the FlexRay protocol, the length range is 0–254.

For PDU output session mode, the payload is the array of data bytes for the specific PDU, not the entire frame.

When the session mode is Frame Output Single-Point, Frame Output Queued, PDU Output Single-Point, or PDU Output Queued, the number of bytes in the payload array must match the Payload Length property of the corresponding frame. You can leave all other FlexRay frame cluster elements uninitialized. For more information, refer to the section for each mode.



timeout is the time to wait for the FlexRay frame data to be queued up for transmit.

The timeout is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, **XNET Write (Frame FlexRay).vi** waits up to that timeout for space to become available in queues. If the space is not available prior to the timeout, a timeout error is returned.

If **timeout** is negative, **XNET Write (Frame FlexRay).vi** waits indefinitely for space to become available in queues.

If **timeout** is 0, **XNET Write (Frame FlexRay).vi** does not wait and immediately returns with a timeout error if all data cannot be queued. Regardless of the timeout used, if a timeout error occurs, none of the data is queued, so you can attempt to call **XNET Write (Frame FlexRay).vi** again at a later time with the same data.

This input is optional. The default value is 10.0 (10 seconds).

If the session mode is Frame Output Single-Point, you must set **timeout** to 0.0. Because this mode writes the most recent value of each frame, **timeout** does not apply.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

The data represents an array of FlexRay frames. Each FlexRay frame uses a LabVIEW cluster with FlexRay-specific elements.

The FlexRay frames are associated to the session's list of frames as follows:

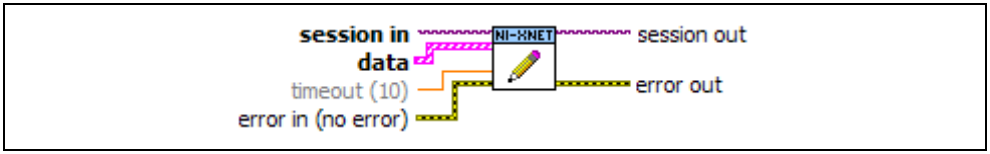
- Frame Output Queued Mode: Array of frame values to transmit for the single frame specified in the list.
- Frame Output Single-Point Mode: Array of single frame values, one for each frame specified in the list.
- PDU Output Queued Mode: Array of frame (PDU payload) values to transmit for the single PDU specified in the list. This mode is similar to Frame Output Queued Mode.
- PDU Output Single-Point Mode: Array of single frame (PDU payload) values, one for each PDU specified in the list. This mode is similar to Frame Output Single-Point Mode.

XNET Write (Frame LIN).vi

Purpose

Writes data to a session as an array of LIN frames. The session must use a LIN interface and Frame Output Stream Mode, Frame Output Queued Mode, or Frame Output Single-Point Mode.

Format



Inputs



session in is the session to write. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Frame Output Stream, Frame Output Queued, or Frame Output Single-Point.



data provides the array of LabVIEW clusters.

Each array element corresponds to a frame value to transmit.

For a Frame Input Single-Point session mode, the order of frames in the array corresponds to the order in the session list.

For Frame Output Queued session mode, the data you write is queued up for transmit on the network. Using the default queue configuration for this mode, you can safely write 64 frames if you have a sufficiently long timeout. To write more data, refer to the XNET Session Number of Values Unused property to determine the actual amount of queue space available for write.

For an example of how this data applies to network traffic, refer to *Frame Output Stream Mode*, *Frame Output Queued Mode*, or *Frame Output Single-Point Mode*.

The elements of each cluster are specific to the LIN protocol. For more information, refer to Appendix C, *Summary of the LIN Standard*, or the LIN protocol specification.

The cluster elements are:



identifier is not used for transmit. You must set this element to 0.

Each frame is identified based on the list of frames or signals provided for the session. The actual identifier to transmit is taken from the database (frame and schedule properties). Therefore, this **identifier** in the frame value is ignored.



event slot? is not used for transmit. You must set this element to false.

The currently running schedule is used to map the specific frame to a corresponding schedule entry (slot). The schedule entry itself determines whether the slot is unconditional, sporadic, or event triggered.



event ID is not used for transmit. You must set this element to 0.



echo? is not used for transmit. You must set this element to false.



type is the frame type (decimal value in parentheses):

LIN Data (64) The LIN data frame contains **payload** data. This is currently the only frame type for LIN.



timestamp represents absolute time using the LabVIEW absolute timestamp type. **timestamp** is not used for transmit. You must set this element to the default value, invalid (0).



payload is the array of data bytes for a LIN data frame.

The array size indicates the payload length of the frame value to transmit. According to the LIN protocol, the payload length range is 0–8.

The number of bytes in the **payload** array must match the Payload Length property of the corresponding frame. You can leave all other LIN frame cluster elements uninitialized. For more information, refer to the topic for each mode.

If you use the frame payload within an event-triggered schedule entry (slot), the first byte of data on the network is the frame's payload identifier. The LIN standard requires this even if the frame transmits in an unconditional or sporadic slot. For this type of LIN frame, the actual data (for example, signal values) is limited to 7 bytes. For this type of frame, you must write the first byte (**payload** of 8 bytes even if only the last 7 are used), but

NI-XNET ignores the value and fills in the first byte for you, using the known frame ID from the session's configuration.



timeout is the time to wait for the LIN frame data to be queued up for transmit.

The timeout is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, **XNET Write (Frame LIN).vi** waits up to that timeout for space to become available in queues. If the space is not available prior to the timeout, a timeout error is returned.

If **timeout** is negative, **XNET Write (Frame LIN).vi** waits indefinitely for space to become available in queues.

If **timeout** is 0, **XNET Write (Frame LIN).vi** does not wait and immediately returns with a timeout error if all data cannot be queued. Regardless of the timeout used, if a timeout error occurs, none of the data is queued, so you can attempt to call **XNET Write (Frame LIN).vi** again at a later time with the same data.

This input is optional. The default value is 10.0 (10 seconds).

If the session mode is Frame Output Single-Point, you must set **timeout** to 0.0. Because this mode writes the most recent value of each frame, **timeout** does not apply.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

The data represents an array of LIN frames. Each LIN frame uses a LabVIEW cluster with LIN-specific elements.

The LIN frames are associated to the session's list of frames as follows:

- Frame Output Stream Mode: Array of all frame values for transmit (list ignored). If the payload is an empty array, only the header part of the LIN frame is transmitted. If the payload is not an empty array, the header and response parts of the LIN frame are transmitted.

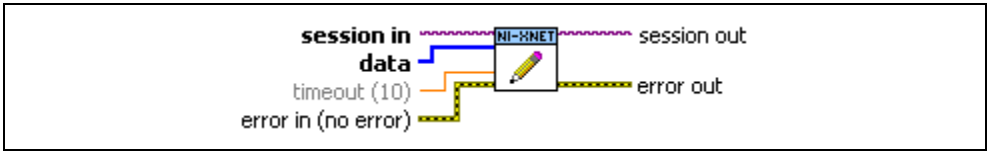
- Frame Output Queued Mode: Array of frame values to transmit for the single frame specified in the list.
- Frame Output Single-Point Mode: Array of single frame values, one for each frame specified in the list.

XNET Write (Frame Raw).vi

Purpose

Writes data to a session as an array of raw bytes.

Format



Inputs



session in is the session to write. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session mode must be Frame Output Stream Mode, Frame Output Queued Mode, or Frame Output Single-Point Mode.



data provides the array of bytes, representing frames to transmit.

The raw bytes encode one or more frames using the Raw Frame Format. This frame format is the same for read and write of raw data and also is used for log file examples.

If needed, you can write data for a partial frame. For example, if a complete raw frame is 24 bytes, you can write 12 bytes, then write the next 12 bytes. You typically do this when you are reading raw frame data from a logfile and want to avoid iterating through the data to detect the start and end of each frame.

For information about which elements of the raw frame are applicable, refer to the **XNET Write.vi** instance for the protocol in use (**XNET Write (Frame CAN).vi**, **XNET Write (Frame FlexRay).vi**, or **XNET Write (Frame LIN).vi**).

The data you write is queued up for transmit on the network. Using the default queue configuration for this mode, you can safely write 1536 frames if you have a sufficiently long timeout. To write more data, refer to the XNET Session Number of Values Unused property to determine the actual amount of queue space available for writing.

For an example of how this data applies to network traffic, refer to *Frame Output Stream Mode*, *Frame Output Queued Mode*, or *Frame Output Single-Point Mode*.



timeout is the time to wait for the raw data to be queued up for transmit.

The timeout is a LabVIEW relative time, represented as 64-bit floating-point in units of seconds.

If **timeout** is positive, **XNET Write (Frame Raw).vi** waits up to that timeout for space to become available in queues. If the space is not available prior to the timeout, a timeout error is returned.

If **timeout** is negative, **XNET Write (Frame Raw).vi** waits indefinitely for space to become available in queues.

If **timeout** is 0, **XNET Write (Frame Raw).vi** does not wait and immediately returns with a timeout error if all data cannot be queued. Regardless of the timeout used, if a timeout error occurs, none of the data is queued, so you can attempt to call **XNET Write (Frame Raw).vi** again at a later time with the same data.

This input is optional. The default value is 10.0 (10 seconds).

If the session mode is Frame Output Single-Point, you must set **timeout** to 0.0. Because this mode writes the most recent value of each frame, **timeout** does not apply.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

The raw bytes encode one or more frames using the Raw Frame Format. The session must use a mode of Frame Output Stream, Frame Output Queued, or Frame Output Single-Point. The raw frame format is protocol independent, so the session can use either a CAN, FlexRay, or LIN interface.

The raw frame format matches the format of data transferred to/from the XNET hardware. Because it is not converted to/from LabVIEW clusters for ease of use, it is more efficient with regard to performance. This instance typically is used to read raw frame data from a log file and write the data to the interface for transmit (replay).

The raw frames are associated to the session's list of frames as follows:

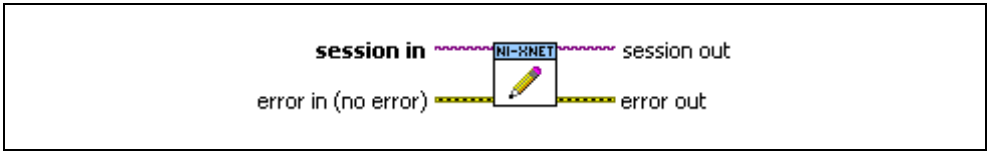
- Frame Output Stream Mode: Array of all frame values for transmit (list ignored). For LIN, if the payload element is an empty array, only the header part of the LIN frame is transmitted. If the payload element is not an empty array, the header and response parts of the LIN frame are transmitted.
- Frame Output Queued Mode: Array of frame values to transmit for the single frame specified in the list.
- Frame Output Single-Point Mode: Array of single frame values, one for each frame specified in the list.
- PDU Output Queued Mode: Array of frame (PDU payload) values to transmit for the single PDU specified in the list. This mode is similar to Frame Output Queued Mode.
- PDU Output Single-Point Mode: Array of single frame (PDU payload) values, one for each PDU specified in the list. This mode is similar to Frame Output Single-Point Mode.

XNET Write (State FlexRay Symbol).vi

Purpose

Writes a request for the FlexRay interface to transmit a symbol on the FlexRay bus. You can use this **XNET Write** VI with any input or output session for FlexRay.

Format



Inputs



session in is the session to use for the symbol write. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session must use a FlexRay interface.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

You can use **XNET Write (State FlexRay Symbol).vi** with any XNET session mode, as long as the session interface is FlexRay. Because the symbol write applies to the FlexRay interface, it can apply to multiple sessions.

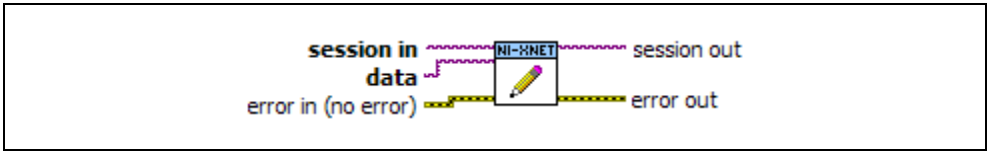
After calling **XNET Write (State FlexRay Symbol).vi**, the XNET interface transmits the symbol during the symbol window of the FlexRay cycle following the currently executing cycle. If you call **XNET Write (State FlexRay Symbol).vi** multiple times, only the most recent symbol is transmitted.

XNET Write (State LIN Schedule Change).vi

Purpose

Write a request for the LIN interface to change the running schedule. You can use this **XNET Write** VI with any input or output session for LIN.

Format



Inputs

session in is the session to use for the schedule change. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session must use a LIN interface.

data is the XNET LIN schedule. Although the data type for this input is the XNET LIN Schedule I/O Name, you also can wire a string.

The **data** input supports the following options:

- **XNET LIN Schedule I/O Name**: You can use the complete I/O name. This provides features such as the ability to choose from LIN schedules in a selected database.
- **String with XNET LIN short name**: If you prefer to use the XNET LIN Schedule Name (Short) property, you can wire in the property as a string.
- **String with decimal number**: This is interpreted as an index into the XNET Cluster LIN:Schedules property used for this session. If you are editing your database file to add/remove LIN schedules, this index may change, in which case the name is the recommended option.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

error out is the error cluster output (refer to *Error Handling*).

Description

You can use **XNET Write (State LIN Schedule Change).vi** with any XNET session mode, as long as the session interface is LIN. Because the schedule change applies to the LIN interface, it can apply to multiple sessions.

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

The **XNET Write (State LIN Schedule Change).vi** behavior depends on the Run Mode property of the XNET LIN schedule that you wire in as data:

- **Continuous**: This mode changes the single run-continuous schedule for the interface. The single run-continuous schedule executes all its entries (slots) repetitively, starting over with the first entry after running the last entry.

  The run-continuous schedule is handled as if it is lowest priority. If you write a run-once schedule in the middle of a run-continuous execution, the run-continuous schedule is interrupted after the current slot finishes. The scheduler switches to the run-once schedule, and when all run-once schedules are done, the scheduler returns to the slot in the run-continuous schedule where it left off. For example, if run-continuous schedule A has 4 slots, and it is executing slot 2 when a run-once schedule B is written, slot 2 of A finishes, then all slots of schedule B run, then the scheduler returns to slot 3 of schedule A.

  Only one run-continuous schedule exists at a time. If you change from one run-continuous schedule to another in the middle of a run, the current schedule completes all of its slots, then the scheduler changes to the new run-continuous schedule.

- **Once**: This mode writes a request for a run-once schedule. Multiple run-once schedules can be pending for execution. Each run-once schedule executes all its entries (slots), and then it is considered done.

  Each run-once schedule has a priority from 1 to 254. Lower values correspond to higher priority (1 is highest). The LIN interface's scheduler maintains a priority queue of run-once schedule requests. This means the highest-priority run-once schedule executes first, followed by the next run-once in priority, and when no run-once schedules are pending, the interface returns to the run-continuous schedule.

  A run-once schedule cannot interrupt another run-once schedule. For example, if run-once schedule X has 3 slots and is executing slot 0 when a run-once schedule Y with higher priority is written, slots 0, 1, and 2 of X finish, then all slots of schedule Y run.

- **Null**: This mode stops scheduler execution after the current slot is finished. The queue of run-once schedules is flushed (all elements discarded).

  The null schedule is considered the highest priority schedule. It overrides the single run-continuous schedule, thus acting as the default scheduling behavior. For example, if you write a null schedule, then write a run-once schedule, the run-once schedule executes all its slots, then communication stops (returns to null schedule).

  XNET Write (State LIN Schedule Change).vi does not wait for the requested schedule to finish execution prior to return. The VI does not wait for the schedule to begin execution, because in the case of run-once schedules, that may take a long time (depending on priority). Because this VI simply writes a schedule request and returns, it is safe to use within a high-priority loop in LabVIEW Real-Time.

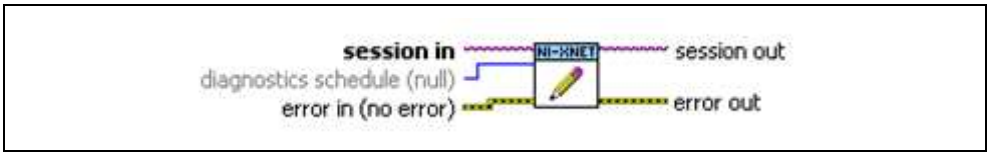
Node configuration is handled using **XNET Write (State LIN Schedule Change).vi** instead of **XNET Write (State LIN Diagnostic Schedule Change).vi**. Wire the node configuration schedule defined for the LIN cluster into **XNET Write (State LIN Schedule Change).vi** so that it is the first schedule executed for the LIN, with a run mode of once. The data for each node configuration service request entry in the node configuration schedule is automatically transmitted by the master. After the node configuration schedule has completed, use **XNET Write (State LIN Diagnostic Schedule Change).vi** to write master request messages and query for slave response messages, or **XNET Write (State LIN Schedule Change).vi** to run normal schedules.

XNET Write (State LIN Diagnostic Schedule Change).vi

Purpose

Write a request for the LIN interface to change the diagnostic schedule. You can use this **XNET Write** VI with any input or output session for LIN.

Format



Inputs



session in is the session to use for the diagnostic schedule change. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**. The session must use a LIN interface.



diagnostic schedule is a ring (enumerated list) with the following values:

String	Value
Null	0
Master Request	1
Slave Response	2

This specifies which diagnostic schedule the master executes:

- **Null:** The master does not execute any diagnostic schedule. No master request or slave response headers are transmitted on the LIN.
- **Master Request:** The master executes a diagnostic master request schedule (transmits a master request header onto the LIN) if it can. First, a master request schedule must be defined for the LIN cluster in the imported or in-memory database. Otherwise, error nxErrDiagnosticScheduleNotDefined is returned when attempting to set this value. Second, the master must have a frame output queued session created for the master request frame, and there must be one or more new master request frames pending in the queue. If no new frames are pending in the output queue, no master request header is transmitted. This allows the timing of master request header

transmission to be controlled by the timing of master request frame writes to the output queue.

If there are no normal schedules pending, the master is effectively in diagnostics-only mode, and master request headers are transmitted at a rate determined by the slot delay defined for the master request frame slot in the master request schedule or the nxPropSession_IntfLINDiagSTmin time, whichever is greater, and the state of the master request frame output queue as described above.

If there are normal schedules pending, the master is effectively in diagnostics-interleaved mode, and a master request header transmission is inserted between each complete execution of a run-once or run-continuous schedule. This happens as long as the nxPropSession_IntfLINDiagSTmin time has been met, and there are one or more new master request frames pending in the master request frame output queue.

- **Slave Response:** The master executes a diagnostic slave response schedule (transmits a slave response header onto the LIN) if it can. A slave response schedule must be defined for the LIN cluster in the imported or in-memory database. Otherwise, error nxErrDiagnosticScheduleNotDefined is returned when attempting to set this value.

  If there are no normal schedules pending, the master is effectively in diagnostics-only mode, and slave response headers are transmitted at the rate of the slot delay defined for the slave response frame slot in the slave response schedule. The addressed slave may or may not respond to each header, depending on its specified P2min and STmin timings.

  If there are normal schedules pending, the master is effectively in diagnostics-interleaved mode, and a slave response header transmission is inserted between each complete execution of a run-once or run-continuous schedule. Here again, the addressed slave may or may not respond to each header, depending on its specified P2min and STmin timings.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

You can use **XNET Write (State LIN Diagnostic Schedule Change).vi** with any XNET session mode, as long as the session interface is LIN. Because the schedule change applies to the LIN interface, it can apply to multiple sessions.

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

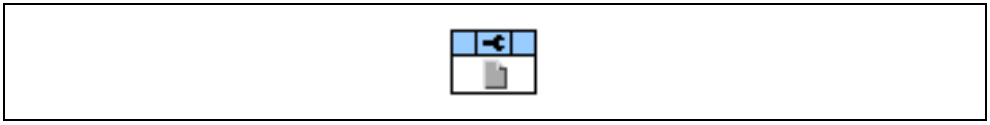
Use **XNET Write (State LIN Diagnostic Schedule Change).vi** to transmit master request messages and query for slave response messages after node configuration has been performed. Node configuration should be handled using **XNET Write (State LIN Schedule Change).vi**. Wire the node configuration schedule defined for the LIN cluster into that VI so that it is the first schedule executed for the LIN. Refer to the description for **XNET Write (State LIN Schedule Change).vi** for more information about using it to perform node configuration.

Database Subpalette

This subpalette includes functions for accessing databases that specify the embedded network configuration, including frame and signal data that is transferred. You can use these functions to retrieve information from database files, create new database objects in LabVIEW, and edit and save new database files.

XNET Database Property Node

Format



Description

Property node used to read/write properties for an XNET Database I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

Clusters

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Database

Short Name

Clsts

Description

Returns an array of I/O names of XNET Clusters in this database.

A cluster is assigned to a database when the cluster object is created. You cannot change this assignment afterwards.

You can use an array element to read or write the cluster properties (for example, cluster protocol or cluster frames). Refer to *XNET Cluster I/O Name* for information about using XNET I/O names.

FIBEX files can contain any number of clusters, and each cluster uses a unique name.

For CANdb (`.dbc`), LDF (`.ldf`), or NI-CAN (`.ncd`) files, the file contains only one cluster, and no cluster name is stored in the file. For these database formats, NI-XNET uses the name *Cluster* for the single cluster.

ShowInvalidFromOpen?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Database

Short Name

ShowInvalid?

Description

Shows frames and signals that are invalid at database open time.

After opening a database, this property always is set to false, meaning that invalid clusters, frames, and signals are not returned in properties that return XNET I/O Names for the database (for example, XNET Cluster Frames and XNET Frame Signals). Invalid clusters, frames, and signals are incorrectly defined and therefore cannot be used in the bus communication. The false setting is recommended when you use the database to create XNET sessions.

In case the database was opened to correct invalid configuration (for example, in a database editor), you must set the property to true prior to reading properties that return XNET I/O Names for the database (for example, XNET Cluster Frames and XNET Frame Signals).

For invalid objects, the XNET Cluster Configuration Status, XNET Frame Configuration Status, and XNET Signal Configuration Status properties return an error code that explains the problem. For valid objects, Configuration Status returns success (no error).

Clusters, frames, and signals that became invalid after the database is opened are still returned from the XNET Database Clusters, XNET Cluster Frames, and XNET Frame Signals properties, even if ShowInvalidFromOpen? is false and Configuration Status returns an error code. For example, if you open the frame with valid properties, then you set the Start Bit beyond the payload length, the Configuration Status returns an error, but the frame is returned from XNET Cluster Frames.

XNET Database Constant

This constant provides the constant form of the XNET Database I/O name. You drag a constant to the block diagram of your VI, then select a database. You can change constants only during configuration, prior to running the VI. For a complete description, refer to *XNET Database I/O Name*.

XNET Cluster Property Node

Format



Description

Property node used to read/write properties for an XNET Cluster I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

FlexRay Properties

This section includes the XNET Cluster FlexRay properties.

FlexRay:Action Point Offset

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.ActPtOff

Description

This property specifies the number of macroticks (MT) that the action point is offset from the beginning of a static slot or symbol window.

This property corresponds to the global cluster parameter **gdActionPointOffset** in the *FlexRay Protocol Specification*.

The action point is that point within a given slot where the actual transmission of a frame starts. This is slightly later than the start of the slot, to allow for a clock drift between the network nodes.

The range for this property is 1–63 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:CAS Rx Low Max

Data Type	Direction	Required?	Default
U32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.CASRxLMax

Description

This property specifies the upper limit of the collision avoidance symbol (CAS) acceptance window. The CAS symbol is transmitted by the FlexRay interface (node) during the symbol window within the communication cycle. A receiving FlexRay interface considers the CAS to be valid if the pattern's low level is within 29 gdBit (**cdCASRxLowMin**) and CAS Rx Low Max.

This property corresponds to the global cluster parameter **gdCASRxLowMax** in the *FlexRay Protocol Specification*.

The values for this property are in the range 67–99 gdBit.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Channels

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.Channels

Description

This property specifies the FlexRay channels used in the cluster. Frames defined in this cluster are expected to use the channels this property specifies. Refer to the XNET Frame FlexRay:Channel Assignment property.

This property corresponds to the global cluster parameter **gChannels** in the *FlexRay Protocol Specification*.

A FlexRay cluster supports two independent network wires (channels A and B). You can choose to use both or only one in your cluster.

The values (enumeration) for this property are:

1 Channel A only

2 Channel B only

3 Channels A and B

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Cluster Drift Damping

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.ClstDriftDmp

Description

This property specifies the cluster drift damping factor, based on the longest microtick used in the cluster. Use this global FlexRay parameter to compute the local cluster drift damping factor for each cluster node. You can access the local cluster drift for the XNET FlexRay interface from the XNET Session Interface:FlexRay:Cluster Drift Damping property.

This property corresponds to the global cluster parameter **gdClusterDriftDamping** in the *FlexRay Protocol Specification*.

The values for this property are in the range 0–5 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Cold Start Attempts

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.ColdStAts

Description

This property specifies the maximum number of times a node in this cluster can start the cluster by initiating schedule synchronization. This global cluster parameter is applicable to all cluster notes that can perform a coldstart (send startup frames).

This property corresponds to the global cluster parameter **gColdStartAttempts** in the *FlexRay Protocol Specification*.

The values for this property are in the range 2–31.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Cycle

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.Cycle

Description

This property specifies the duration of one FlexRay communication cycle, expressed in microseconds.

This property corresponds to the global cluster parameter **gdCycle** in the *FlexRay Protocol Specification*.

All frame transmissions complete within a cycle. After this time, the frame transmissions restart with the first frame in the next cycle. The communication cycle counts increment from 0–63, after which the cycle count resets back to 0.

The range for this property is 10–16000 µs.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Dynamic Segment Start

Data Type	Direction	Required?	Default
	Read Only	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Short Name

FlexRay.DynSegStart

Description

This property specifies the start of the dynamic segment, expressed as the number of macroticks (MT) from the start of the cycle.

The range for this property is 8–15998 MT.

This property is calculated from other cluster properties. It is based on the total static segment size. It is set to 0 if the FlexRay:Number of Minislots property is 0 (no dynamic segment exists).

FlexRay:Dynamic Slot Idle Phase

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.DynSlotIdlPh

Description

This property specifies the dynamic slot idle phase duration.

This property corresponds to the global cluster parameter **gdDynamicSlotIdlePhase** in the *FlexRay Protocol Specification*.

The values for this property are in the range 0–2 minislots.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Latest Guaranteed Dynamic Slot

Data Type	Direction	Required?	Default
	Read Only	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Short Name

FlexRay.LatestGuarDyn

Description

This property specifies the highest slot ID in the dynamic segment that still can transmit a full-length (for example, Payload Length Dynamic Maximum) frame, provided all previous slots in the dynamic segment have transmitted full-length frames also.

A larger slot ID cannot be guaranteed to transmit a full-length frame in each cycle (although a frame might go out depending on the dynamic segment load).

The range for this property is 2–2047 slots.

This read-only property is calculated from other cluster properties. If the Number of Minislots is zero, no dynamic slots exist, and this property returns 0. Otherwise, the Number of Minislots is used along with Payload Length Dynamic Maximum to determine the latest dynamic slot guaranteed to transmit in the next cycle. In other words, when all preceding dynamic slots transmit with Payload Length Dynamic Maximum, this dynamic slot also can transmit with Payload Length Dynamic Maximum, and its frame ends prior to the end of the dynamic segment.

FlexRay:Latest Usable Dynamic Slot

Data Type	Direction	Required?	Default
	Read	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Short Name

FlexRay.LatestUsableDyn

Description

This property specifies the highest slot ID in the dynamic segment that can still transmit a full-length (that is, Payload Length Dynamic Maximum) frame, provided no other frames have been sent in the dynamic segment.

A larger slot ID cannot transmit a full-length frame (but could probably still transmit a shorter frame).

The range for this property is 2–2047.

This read-only property is calculated from other cluster properties. If the Number of Minislots is zero, no dynamic slots exist, and this property returns 0. Otherwise, Number of Minislots is used along with Payload Length Dynamic Maximum to determine the latest dynamic slot that can be used when all preceding dynamic slots are empty (zero payload length). In other words, this property is calculated under the assumption that all other dynamic slots use only one minislot, and this dynamic slot uses the number of minislots required to deliver the maximum payload. The frame for this dynamic slot must end prior to the end of the dynamic segment. Any frame transmitted in a preceding dynamic slot is likely to preclude this slot's frame.

FlexRay:Listen Noise

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.LisNoise

Description

This property specifies the upper limit for the startup and wakeup listen timeout in the presence of noise. It is used as a multiplier for the Interface:FlexRay:Listen Timeout property.

This property corresponds to the global cluster parameter **gListenNoise** in the *FlexRay Protocol Specification*.

The values for this property are in the range 2–16.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Macro Per Cycle

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.MacroPerCycle

Description

This property specifies the number of macroticks in a communication cycle. For example, if the FlexRay cycle has a duration of 5 ms (5000 µs), and the duration of a macrotick is 1 µs, the XNET Cluster FlexRay:Macro Per Cycle property is 5000.

This property corresponds to the global cluster parameter **gMacroPerCycle** in the *FlexRay Protocol Specification*.

The macrotick (MT) is the basic timing unit in the FlexRay cluster. Nearly all timing dependent properties are expressed in terms of macroticks.

The range for this property is 10–16000 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Macrotick

Data Type	Direction	Required?	Default
	Read	N/A	Calculated from Other Cluster Parameters

Property Class

XNET Cluster

Short Name

FlexRay.Macrotick

Description

This property specifies the duration of the clusterwide nominal macrotick, expressed in microseconds.

This property corresponds to the global cluster parameter **gdMacrotick** in the *FlexRay Protocol Specification*.

The macrotick (MT) is the basic timing unit in the FlexRay cluster. Nearly all timing-dependent properties are expressed in terms of macroticks.

The range for this property is 1–6 µs.

This property is calculated from the FlexRay:Cycle and FlexRay:Macro Per Cycle properties and rounded to the nearest permitted value.

FlexRay:Max Without Clock Correction Fatal

Data Type	Direction	Required?	Default
U32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.MaxWoClkCorFat

Description

This property defines the number of consecutive even/odd cycle pairs with missing clock correction terms that cause the controller to transition from the Protocol Operation Control status of Normal Active or Normal Passive to the Halt state. Use this global parameter as a threshold for testing the clock correction failure counter.

This property corresponds to the global cluster parameter **gMaxWithoutClockCorrectionFatal** in the *FlexRay Protocol Specification*.

The values for this property are in the range 1–15 even/odd cycle pairs.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Max Without Clock Correction Passive

Data Type	Direction	Required?	Default
U32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.MaxWoClkCorPas

Description

This property defines the number of consecutive even/odd cycle pairs with missing clock correction terms that cause the controller to transition from the Protocol Operation Control status of Normal Active to Normal Passive. Use this global parameter as a threshold for testing the clock correction failure counter.

Note This property, Max Without Clock Correction Passive, <= Max Without Clock Correction Fatal <= 15.

This property corresponds to the global cluster parameter **gMaxWithoutClockCorrectionPassive** in the *FlexRay Protocol Specification*.

The values for this property are in the range 1–15 even/odd cycle pairs.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Minislot Action Point Offset

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

MinislotActPt

Description

This property specifies the number of macroticks (MT) the minislot action point is offset from the beginning of a minislot.

This property corresponds to the global cluster parameter **gdMinislotActionPointOffset** in the *FlexRay Protocol Specification*.

The action point is that point within a given slot where the actual transmission of a frame starts. This is slightly later than the start of the slot to allow for a clock drift between the network nodes.

The range for this property is 1–31 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Minislot

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.Minislot

Description

This property specifies the duration of a minislot, expressed in macroticks (MT).

This property corresponds to the global cluster parameter **gdMinislot** in the *FlexRay Protocol Specification*.

In the dynamic segment of the FlexRay cycle, frames can have variable payload length.

Minislots are the dynamic segment time increments. In a minislot, a dynamic frame can start transmission, but it usually spans several minislots. If no frame transmits, the slot counter (slot ID) is incremented to allow for the next frame.

The total dynamic segment length is determined by multiplying this property by the Number Of Minislots property. The total dynamic segment length must be shorter than the Macro Per Cycle property minus the total static segment length.

The range for this property is 2–63 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Network Management Vector Length

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.NMVecLen

Description

This property specifies the length of the Network Management vector (NMVector) in a cluster.

Only frames transmitted in the static segment of the communication cycle use the NMVector. The NMVector length specifies the number of bytes in the payload segment of the FlexRay frame transmitted in the status segment that can be used as the NMVector.

This property corresponds to the global cluster parameter **gNetworkManagementVectorLength** in the *FlexRay Protocol Specification*.

The range for this property is 0–12 bytes.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:NIT Start

Data Type	Direction	Required?	Default
	Read	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Short Name

FlexRay.NITStart

Description

This property specifies the start of the Network Idle Time (NIT), expressed as the number of macroticks (MT) from the start of the cycle.

The NIT is a period at the end of a FlexRay communication cycle where no frames are transmitted. The network nodes use it to re-sync their clocks to the common network time.

The range for this property is 8–15998 MT.

This property is calculated from other cluster properties. It is the total size of the static and dynamic segments plus the symbol window length, which is optional in a FlexRay communication cycle.

FlexRay:NIT

Data Type	Direction	Required?	Default
U32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.NIT

Description

This property is the Network Idle Time (NIT) duration, expressed in macroticks (MT).

This property corresponds to the global cluster parameter **gdNIT** in the *FlexRay Protocol Specification*.

The NIT is a period at the end of a FlexRay communication cycle where no frames are transmitted. The network nodes use it to re-sync their clocks to the common network time.

Configure the NIT to be the Macro Per Cycle property minus the total static and dynamic segment lengths minus the optional symbol window duration.

The range for this property is 2–805 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Number of Minislots

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.NumMinislt

Description

This property specifies the number of minislots in the dynamic segment.

This property corresponds to the global cluster parameter **gNumberOfMinislots** in the *FlexRay Protocol Specification*.

In the FlexRay cycle dynamic segment, frames can have variable payload lengths.

Minislots are the dynamic segment time increments. In a minislot, a dynamic frame can start transmission, but it usually spans several minislots. If no frame transmits, the slot counter (slot ID) is incremented to allow for the next frame.

The total dynamic segment length is determined by multiplying this property by the Minislot property. The total dynamic segment length must be shorter than the Macro Per Cycle property minus the total static segment length.

The range for this property is 0–7986.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Number of Static Slots

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.NumStatSlt

Description

This property specifies the number of static slots in the static segment.

This property corresponds to the global cluster parameter **gNumberOfStaticSlots** in the *FlexRay Protocol Specification*.

Each static slot is used to transmit one (static) frame on the bus.

The total static segment length is determined by multiplying this property by the Static Slot property. The total static segment length must be shorter than the Macro Per Cycle property.

The range for this property is 2–1023.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Offset Correction Start

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.OffCorSt

Description

This property specifies the start of the offset correction phase within the Network Idle Time (NIT), expressed as the number of macroticks (MT) from the start of the cycle.

This property corresponds to the global cluster parameter **gOffsetCorrectionStart** in the *FlexRay Protocol Specification*.

The NIT is a period at the end of a FlexRay communication cycle where no frames are transmitted. The network nodes use it to re-sync their clocks to the common network time.

The Offset Correction Start is usually configured to be NITStart + 1, but can deviate from that value.

The range for this property is 9–15999 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Payload Length Dynamic Maximum

Data Type	Direction	Required?	Default
U32	Read/Write	N/A	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.PayldLenDynMax

Description

This property specifies the maximum of the payload lengths of all dynamic frames.

In the FlexRay cycle dynamic segment, frames can have variable payload length.

The range for this property is 0–254 bytes (even numbers only).

The value returned for this property is the maximum of the payload lengths of all frames defined for the dynamic segment in the database.

Use this property to calculate the Latest Usable Dynamic Slot and Latest Guaranteed Dynamic Slot properties.

You may temporarily set this to a larger value (if it is not yet the maximum), and then this value is returned for this property. But this setting is lost once the database is closed, and after a reopen, the maximum of the frames is returned again. The changed value is returned from the FlexRay:Payload Length Dynamic Maximum property until the database is closed.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Payload Length Maximum

Data Type	Direction	Required?	Default
	Read	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Short Name

FlexRay.PayldLenMax

Description

This property returns the payload length of any frame (static or dynamic) in this cluster with the longest payload. The payload specifies that the frame transfers the data.

The range for this property is 0–254 bytes (even numbers only).

FlexRay:Payload Length Static

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.PayldLenSt

Description

This property specifies the payload length of a static frame. All static frames in a cluster have the same payload length.

This property corresponds to the global cluster parameter **gPayloadLengthStatic** in the *FlexRay Protocol Specification*.

The range for this property is 0–254 bytes (even numbers only).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Static Slot

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.StatSlot

Description

This property specifies the duration of a slot in the static segment in macroticks (MT).

This property corresponds to the global cluster parameter **gdStaticSlot** in the *FlexRay Protocol Specification*.

Each static slot is used to transmit one (static) frame on the bus.

The static slot duration takes into account the Payload Length Static and Action Point Offset properties, as well as maximum propagation delay.

In the FlexRay cycle static segment, all frames must have the same payload length; therefore, the duration of a static frame is the same.

The total static segment length is determined by multiplying this property by the Number Of Static Slots property. The total static segment length must be shorter than the Macro Per Cycle property.

The range for this property is 4–661 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Symbol Window Start

Data Type	Direction	Required?	Default
	Read	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Short Name

FlexRay.SymWinStart

Description

This property specifies the macrotick offset at which the symbol window begins from the start of the cycle. During the symbol window, a channel sends a single Media Test Access Symbol (MTS).

The range for this property is 8–15998 MT.

This property is calculated from other cluster properties. It is based on the total static and dynamic segment size. It is set to zero if the Symbol Window property is 0 (no symbol window exists).

FlexRay:Symbol Window

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.SymWin

Description

This property specifies the symbol window duration, expressed in macroticks (MT).

This property corresponds to the global cluster parameter **gdSymbolWindow** in the *FlexRay Protocol Specification*.

The symbol window is a slot after the static and dynamic segment, and is used to transmit Collision Avoidance symbols (CAS) and/or Media Access Test symbol (MTS). The symbol window is optional for a given cluster (the Symbol Window property can be zero). A symbol transmission starts at the action point offset within the symbol window.

The range for this property is 0–142 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Sync Node Max

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.SyncNodeMax

Description

This property specifies the maximum number of nodes that may send frames with the sync frame indicator bit set to one.

This property corresponds to the global cluster parameter **gSyncNodeMax** in the *FlexRay Protocol Specification*.

Sync frames define the zero points for the clock drift measurement. Startup frames are special sync frames transmitted first after a network startup. There must be at least two startup nodes in a network.

The range for this property is 2–15.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:TSS Transmitter

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.TSSTx

Description

This property specifies the number of bits in the Transmission Start Sequence (TSS). A frame transmission may be truncated at the beginning. The amount of truncation depends on the nodes involved and the channel topology layout. For example, the purpose of the TSS is to "open the gates" of an active star (that is, to cause the star to properly set up input and output connections). During this setup, an active star truncates a number of bits at the beginning of a communication element. The TSS prevents the frame or symbol content from being truncated.You must set this property to be greater than the expected worst case truncation of a frame.

This property corresponds to the global cluster parameter **gdTSSTransmitter** in the *FlexRay Protocol Specification*.

The range for this property is 3–15 bit.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Use Wakeup

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Cluster

Short Name

FlexRay.UseWakeup?

Description

This property indicates whether the FlexRay cluster supports wakeup. This value is set to True if the WAKE-UP tree is present in the FIBEX file. This value is set to False if the WAKE-UP tree is not present in the FIBEX file.

When this property is True, the FlexRay cluster uses wakeup functionality; otherwise, the FlexRay cluster does not use wakeup functionality.

When creating a new database, the default value of this property is False. However, if you set any wakeup parameter (for example, FlexRay:Wakeup Symbol Rx Low), this property is set to True automatically, and the WAKE-UP tree is saved in the FIBEX file when saved.

FlexRay:Wakeup Symbol Rx Idle

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.WakeSymRxIdl

Description

This property specifies the number of bits the node uses to test the idle portion duration of a received wakeup symbol. Collisions, clock differences, and other effects can deform the transmitted wakeup pattern.

This property corresponds to the global cluster parameter **gdWakeupSymbolRxIdle** in the *FlexRay Protocol Specification*.

The range for this property is 14–59 gdBit (bit duration).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Wakeup Symbol Rx Low

Data Type	Direction	Required?	Default
U32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.WakeSymRxLow

Description

This property specifies the number of bits the node uses to test the low portion duration of a received wakeup symbol. This lower limit of zero bits must be received for the receiver to detect the low portion. Active starts, clock differences, and other effects can deform the transmitted wakeup pattern.

This property corresponds to the global cluster parameter **gdWakeupSymbolRxLow** in the *FlexRay Protocol Specification*.

The range for this property is 10–55 gdBit (bit duration).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Wakeup Symbol Rx Window

Data Type	Direction	Required?	Default
U32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.WakeSymRxWin

Description

This property specifies the size of the window used to detect wakeups. Detection of a wakeup requires a low and idle period from one WUS (wakeup symbol) and a low period from another WUS, to be detected entirely within a window of this size. Clock differences and other effects can deform the transmitted wakeup pattern.

This property corresponds to the global cluster parameter **gdWakeupSymbolRxWindow** in the *FlexRay Protocol Specification*.

The range for this property is 76–301 gdBit (bit duration).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Wakeup Symbol Tx Idle

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.WakeSymTxIdl

Description

This property specifies the number of bits the node uses to transmit the wakeup symbol idle portion.

This property corresponds to the global cluster parameter **gdWakeupSymbolTxIdle** in the *FlexRay Protocol Specification*.

The range for this property is 45–180 gdBit (bit duration).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Wakeup Symbol Tx Low

Data Type	Direction	Required?	Default
	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Short Name

FlexRay.WakeSymTxLow

Description

This property specifies the number of bits the node uses to transmit the wakeup symbol low phase.

This property corresponds to the global cluster parameter **gdWakeupSymbolTxLow** in the *FlexRay Protocol Specification*.

The range for this property is 15–60 gdBit (bit duration).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Application Protocol

Data Type	Direction	Required?	Default
	Read/Write	No	Read from Database

Property Class

XNET Cluster

Short Name

ApplProtocol

Description

This property specifies the application protocol. It is a ring of two values:

Enumeration	Value	Meaning
None	0	The default application protocol.
J1939	1	Indicates J1939 clusters. The value enables the following features: • Sending/receiving long frames as the SAE J1939 specification specifies, using the J1939 transport protocol. • Using a special notation for J1939 identifiers. • Using J1939 address claiming.

Baud Rate

Data Type	Direction	Required?	Default
U32	Read/Write	No	0

Property Class

XNET Cluster

Short Name

BaudRate

Description

The Baud Rate property sets the baud rate all cluster nodes use. This baud rate represents the rate from the database, so it is read-only from the session. Use a session interface property (for example, Interface:Baud Rate) to override the database baud rate with an application-specific baud rate.

CAN

For CAN, this rate can be 33333, 40000, 50000, 62500, 80000, 83333, 100000, 125000, 160000, 200000, 250000, 400000, 500000, 800000, or 1000000. Some transceivers may only support a subset of these values.

If you need values other than these, use the custom settings as described in the Interface:Baud Rate property.

FlexRay

For FlexRay, this rate can be 2500000, 5000000, or 10000000.

LIN

For LIN, this rate can be 2400–20000 inclusive.

If you need values other than these, use the custom settings as described in the Interface:Baud Rate property.

CAN:FD Baud Rate

Data Type	Direction	Required?	Default
	Read/Write	No	0

Property Class

XNET Cluster

Short Name

CAN.FdBaudRate

Description

The FD Baud Rate property sets the fast data baud rate for the CAN FD + BRS CAN:I/O Mode property. This property represents the database fast data baud rate for the CAN FD + BRS I/O Mode. Refer to the CAN:I/O Mode property for a description of this mode. Use a session interface property (for example, Interface:CAN:FD Baud Rate) to override the database fast baud rate with an application-specific fast baud rate.

NI-XNET CAN hardware currently accepts the following numeric baud rates: 200000, 250000, 400000, 500000, 800000, 1000000, 1250000, 1600000, 2000000, 2500000, 4000000, 5000000, and 8000000. Some transceivers may support only a subset of these values.

If you need values other than these, use the custom settings as described in the Interface:CAN:FD Baud Rate property.

CAN:I/O Mode

Data Type	Direction	Required?	Default
	Read/Write	No	Read from Database

Property Class

XNET Cluster

Short Name

CAN.IoMode

Description

This property specifies the CAN I/O Mode of the cluster. It is a ring of three values:

Enumeration	Value	Meaning
CAN	0	This is the default CAN 2.0 A/B standard I/O mode as defined in ISO 11898-1:2003. A fixed baud rate is used for transfer, and the payload length is limited to 8 bytes.
CAN FD	1	This is the CAN FD mode as specified in the *CAN with Flexible Data-Rate* specification, version 1.0. Payload lengths up to 64 are allowed, but they are transmitted at a single fixed baud rate (defined by the XNET Cluster Baud Rate or XNET Session Interface:Baud Rate properties).
CAN FD + BRS	2	This is the CAN FD as specified in the *CAN with Flexible Data-Rate* specification, version 1.0, with the optional Baud Rate Switching enabled. The same payload lengths as CAN FD mode are allowed; additionally, the data portion of the CAN frame is transferred at a different (higher) baud rate (defined by the CAN:FD Baud Rate or XNET Session Interface:CAN:FD Baud Rate properties).

Comment

Data Type	Direction	Required?	Default
	Read/Write	No	Empty String

Property Class

XNET Cluster

Short Name

Comment

Description

A comment describing the cluster object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Cluster

Short Name

ConfigStatus

Description

The cluster object configuration status.

Configuration Status returns an NI-XNET error code. You can pass the value to the error code input of **Simple Error Handler.vi** to convert it to a text description (on message output) of the configuration problem.

By default, incorrectly configured clusters in the database are not returned from the XNET Database Clusters property because they cannot be used in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When the configuration status of a cluster becomes invalid after the database has been opened, the cluster still is returned from the XNET Database Clusters property even if ShowInvalidFromOpen? is false.

Database

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Cluster

Short Name

Database

Description

I/O name of the cluster parent database.

The parent database is defined when the cluster object is created. You cannot change it afterwards.

ECUs

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Cluster

Short Name

ECUs

Description

ECUs in this cluster.

Returns an array of I/O names of all ECUs defined in this cluster. An ECU is assigned to a cluster when the ECU object is created. You cannot change this assignment afterwards.

To add an ECU to a cluster, use **XNET Database Create (ECU).vi**. To remove an ECU from the cluster, use **XNET Database Delete (ECU).vi**.

Frames

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Cluster

Short Name

Frms

Description

Frames in this cluster.

Returns an array of I/O names of all frames defined in this cluster. A frame is assigned to a cluster when the frame object is created. You cannot change this assignment afterwards.

To add a frame to a cluster, use **XNET Database Create (Frame).vi**. To remove a frame from a cluster, use **XNET Database Delete (Frame).vi**.

LIN:Schedules

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Cluster

Short Name

LIN.Schedules

Description

Array of LIN schedules defined in this cluster. A LIN schedule is assigned to a cluster when the LIN schedule object is created. You cannot change this assignment afterwards. The schedules in this array are sorted alphabetically by schedule name.

While the XNET interface is running, you can use **XNET Write (State LIN Schedule Change).vi** to change the running schedule. No schedule runs by default, so you must write a schedule request at least once in your application.

For **XNET Write (State LIN Schedule Change).vi**, if you use an index to specify the schedule, that index is the position in this array (starting at 0).

LIN:Tick

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET Cluster

Short Name

LIN.Tick

Description

Relative time between LIN ticks (f64, relative time in seconds). The XNET LIN Schedule Entry Delay property must be a multiple of this tick.

This tick is referred to as the “timebase” in the LIN specification.

The XNET ECU LIN:Master? property defines the LIN:Tick property in this cluster. You cannot use the LIN:Tick property when there is no LIN:Master? property defined in this cluster.

Name (Short)

Data Type	Direction	Required?	Default
	Read/Write	Yes	Defined in Create Object

Property Class

XNET Cluster

Short Name

NameShort

Description

String identifying the cluster object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

If you use a FIBEX file, the short name comes from the file. If you use a CANdb (`.dbc`), LDF (`.ldf`), or NI-CAN (`.ncd`) file, no cluster name is stored in the file, so NI-XNET uses the name *Cluster*. If you create the cluster yourself, it comes from **Name** input of **XNET Database Create (Cluster).vi**.

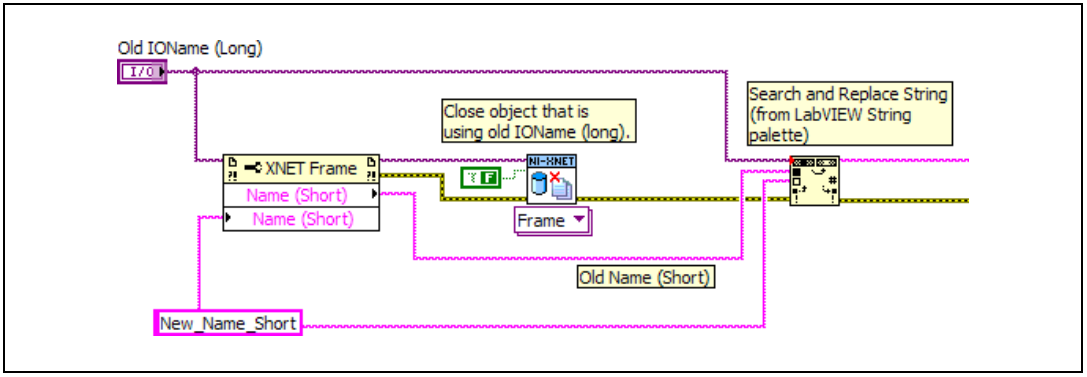
A cluster name must be unique for all clusters in a database.

This short name does not include qualifiers to ensure that it is unique, such as the database name. It is for display purposes. The fully qualified name is available by using the XNET Cluster I/O name as a string.

You can write this property to change the cluster's short name. When you do this, then use the original XNET Cluster that contains the old name, errors can result because the old name cannot be found. Follow these steps to avoid this problem:

1. Get the old Name (Short) property using the property node.
2. Set the new Name (Short) property for the object.
3. Close the object using **XNET Database Close.vi**. Wire the **close all?** input as false to close the renamed object only.
4. Wire the XNET Cluster as the input string to **Search and Replace String Function.vi** with the old **Name** as the search string and the new **Name** as the replacement string. This replaces the short name in the XNET Cluster, while retaining the other text that ensures a unique name.

The following diagram demonstrates steps 1 through 4 for an XNET Frame I/O name:



PDUs

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Cluster

Short Name

PDUs

Description

PDUs in this cluster.

Returns an array of I/O names of all PDUs defined in this cluster. A PDU is assigned to a cluster when the PDU object is created. You cannot change this assignment afterwards.

To add a PDU to a cluster, use **XNET Database Create (PDU).vi**. To remove a PDU from a cluster, use the **XNET Database Delete (PDU).vi**.

PDUs Required?

Data Type	Direction	Required?	Default
TF	Read Only	N/A	N/A

Property Class

XNET Cluster

Short Name

PDUsReqd?

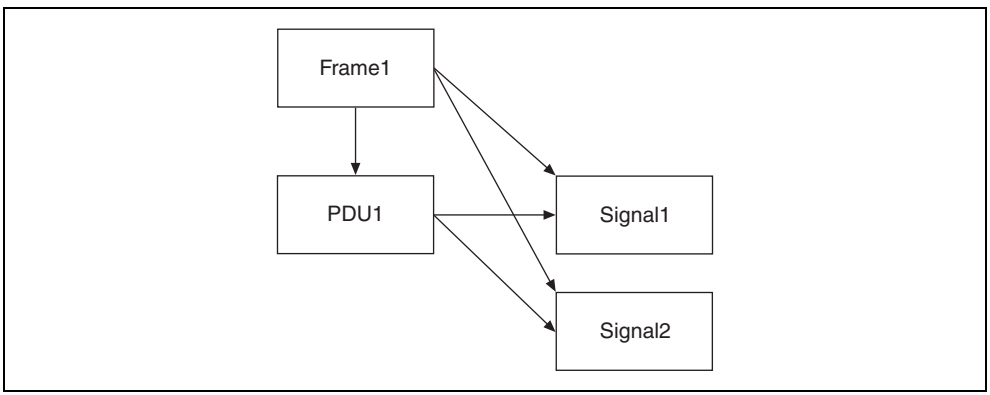
Description

Determines whether using PDUs in the database API is required for this cluster.

If this property returns false, it is safe to use signals as child objects of a frame without PDUs. This behavior is compatible with NI-XNET 1.1 or earlier. Clusters from `.dbc`, `.ncd`, or FIBEX 2 files always return false for this property, so using PDUs from those files is not required.

If this property returns true, the cluster contains PDU configuration, which requires reading the PDUs as frame child objects and then signals as PDU child objects, as shown in the following figure.

Internally, the database always uses PDUs, but shows the same signal objects also as children of a frame.



The following conditions must be fulfilled for all frames in the cluster to return false from the PDUs Required? property:

- Only one PDU is mapped to the frame.
- This PDU is not mapped to other frames.
- The PDU Start Bit in the frame is 0.
- The PDU Update Bit is not used.

If the conditions are not fulfilled for a given frame, signals from the frame are still returned, but reading the property returns a warning.

The NI-XNET session supports frames requiring PDUs only for FlexRay. For frames requiring PDUs on a CAN or LIN cluster, the XNET Frame Configuration Status property and **XNET Create Session.vi** return an error.

Protocol

Data Type	Direction	Required?	Default
	Read/Write	No	CAN

Property Class

XNET Cluster

Short Name

Protocol

Description

Determines the cluster protocol.

The values (enumeration) for this property are:

0 CAN

1 FlexRay

2 LIN

Signals

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Cluster

Short Name

Sigs

Description

This property returns an array of I/O names of all XNET Signals defined in this cluster.

A signal is assigned to a cluster when the signal object is created. You cannot change this assignment afterwards.

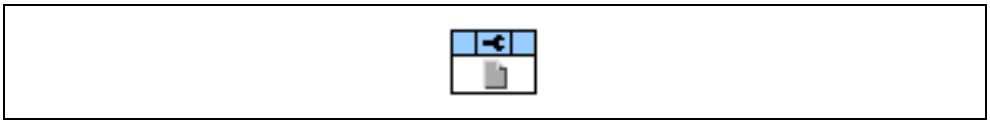
To add a signal to a cluster, use **XNET Database Create (Signal).vi**. To remove a signal from a cluster use **XNET Database Delete (Signal).vi**.

XNET Cluster Constant

This constant provides the constant form of the XNET Cluster I/O name. You drag a constant to the block diagram of your VI, then select a cluster. You can change constants only during configuration, prior to running the VI. For a complete description, refer to *XNET Cluster I/O Name*.

XNET ECU Property Node

Format



Description

Property node used to read/write properties for an XNET ECU I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

Cluster

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET ECU

Short Name

Cluster

Description

I/O name of the parent cluster to which the ECU is connected.

The parent cluster is determined when the ECU object is created. You cannot change it afterwards.

FlexRay:Coldstart?

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET ECU

Short Name

FlexRay.Coldstart?

Description

Indicates that the ECU is sending a startup frame.

This property is valid only for ECUs connected to a FlexRay bus. It returns true when one of the frames this ECU transmits (refer to the XNET ECU Frames Transmitted property) has the XNET Frame FlexRay:Startup? property set to true. You can determine the frame transmitting the startup using the XNET ECU FlexRay:Startup Frame property. An ECU can send only one startup frame on the FlexRay bus.

FlexRay:Connected Channels

Data Type	Direction	Required?	Default
U32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET ECU

Short Name

FlexRay.ConnectedChs

Description

This property specifies the channel(s) that the FlexRay ECU (node) is physically connected to. The default value of this property is connected to all channels available on the cluster.

This property corresponds to the **pChannels** node parameter in the *FlexRay Protocol Specification*.

The values supported for this property (enumeration) are A = 1, B = 2, and A and B = 3.

FlexRay:Startup Frame

Data Type	Direction	Required?	Default
I/O	Read Only	N/A	N/A

Property Class

XNET ECU

Short Name

FlexRay.StartupFrm

Description

Returns the I/O name of the startup frame the ECU sends.

This property is valid only for ECUs connected to a FlexRay bus. If the ECU transmits a frame (refer to the XNET ECU Frames Transmitted property) with the XNET Frame FlexRay:Startup? property set to true, this property returns this frame. Otherwise, it is empty.

FlexRay:Wakeup Channels

Data Type	Direction	Required?	Default
U32	Read/Write	No	None

Property Class

XNET ECU

Short Name

FlexRay.WakeupChs

Description

This property specifies the channel(s) on which the FlexRay ECU (node) is allowed to generate the wake-up pattern. The default value of this property is not to be a wakeup node.

When importing from a FIBEX file, this parameter corresponds to a WAKE-UP-CHANNEL being set to True for each connected channel.

The values supported for this property (enumeration) are A = 1, B = 2, A and B = 3, and None = 4.

FlexRay:Wakeup Pattern

Data Type	Direction	Required?	Default
U32	Read/Write	No	2

Property Class

XNET ECU

Short Name

FlexRay.WakeupPtrn

Description

This property specifies the number of repetitions of the wakeup symbol that are combined to form a wakeup pattern when the FlexRay ECU (node) enters the POC:WAKEUP_SEND state. The POC:WAKEUP_SEND state is one of the FlexRay controller state transitions during the wakeup process. In this state, the controller sends the wakeup pattern on the specified Wakeup Channel and checks for collisions on the bus.

This property is relevant only when FlexRay:Wakeup Channels is set to a value other than None and FlexRay:Use Wakeup is True.

This property corresponds to the **pWakeupPattern** node parameter in the *FlexRay Protocol Specification*.

The supported values for this property are 2–63.

Comment

Data Type	Direction	Required?	Default
	Read/Write	No	Empty String

Property Class

XNET ECU

Short Name

Comment

Description

Comment describing the ECU object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET ECU

Short Name

ConfigStatus

Description

The ECU object configuration status.

Configuration Status returns an NI-XNET error code. You can pass the value to **Simple Error Handler.vi** error code input to convert the value to a text description (on message output) of the configuration problem.

By default, incorrectly configured ECUs in the database are not returned from the XNET Cluster ECUs property because they cannot be used in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When the configuration status of an ECU became invalid after the database is opened, the ECU still is returned from the XNET Cluster ECUs property even if ShowInvalidFromOpen? is false.

Frames Received

Data Type	Direction	Required?	Default
	Read/Write	No	Empty Array

Property Class

XNET ECU

Short Name

FrmsRx

Description

Returns an array of I/O names of frames the ECU receives.

This property defines all frames the ECU receives. All frames an ECU receives in a given cluster must be defined in the same cluster.

Frames Transmitted

Data Type	Direction	Required?	Default
	Read/Write	No	Empty Array

Property Class

XNET ECU

Short Name

FrmsTx

Description

Returns an array of I/O names of frames the ECU transmits.

This property defines all frames the ECU transmits. All frames an ECU transmits in a given cluster must be defined in the same cluster.

LIN:Master?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET ECU

Short Name

LIN.Master?

Description

Determines whether the ECU is a LIN master (true) or slave (false).

LIN:Protocol Version

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET ECU

Short Name

LIN.ProtclVer

Description

The LIN standard version this ECU uses.

This property is a ring (enumerated list) with the following values:

String	Value
1.2	2
1.3	3
2.0	4
2.1	5

LIN:Initial NAD

Data Type	Direction	Required?	Default
U32	Read Only	N/A	N/A

Property Class

XNET ECU

Short Name

InitialNAD

Description

Initial NAD of a LIN slave node. NAD is the address of a slave node and is used in diagnostic services. Initial NAD is replaced by configured NAD with node configuration services.

Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

LIN:Configured NAD

Data Type	Direction	Required?	Default
U32	Read Only	N/A	N/A

Property Class

XNET ECU

Short Name

ConfigNAD

Description

Configured NAD of a LIN slave node. NAD is the address of a slave node and is used in diagnostic services. Initial NAD is replaced by configured NAD with node configuration services.

Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

LIN:Supplier ID

Data Type	Direction	Required?	Default
U32	Read Only	N/A	N/A

Property Class

XNET ECU

Short Name

SupplierID

Description

Supplier ID is a 16-bit value identifying the supplier of the LIN node (ECU).

Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

LIN:Function ID

Data Type	Direction	Required?	Default
U32	Read Only	N/A	N/A

Property Class

XNET ECU

Short Name

FunctionID

Description

Function ID is a 16-bit value identifying the function of the LIN node (ECU).

Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

LIN:P2min

Data Type	Direction	Required?	Default
DBL	Read Only	N/A	N/A

Property Class

XNET ECU

Short Name

P2min

Description

The minimum time in seconds between reception of the last frame of the diagnostic request and the response sent by the node.

Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

LIN:STmin

Data Type	Direction	Required?	Default
DBL	Read Only	N/A	N/A

Property Class

XNET ECU

Short Name

STmin

Description

The minimum time in seconds the node requires to prepare for the next frame of the diagnostic service.

Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

Name (Short)

Data Type	Direction	Required?	Default
	Read/Write	Yes	Defined in Create Object

Property Class

XNET ECU

Short Name

NameShort

Description

String identifying the ECU object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

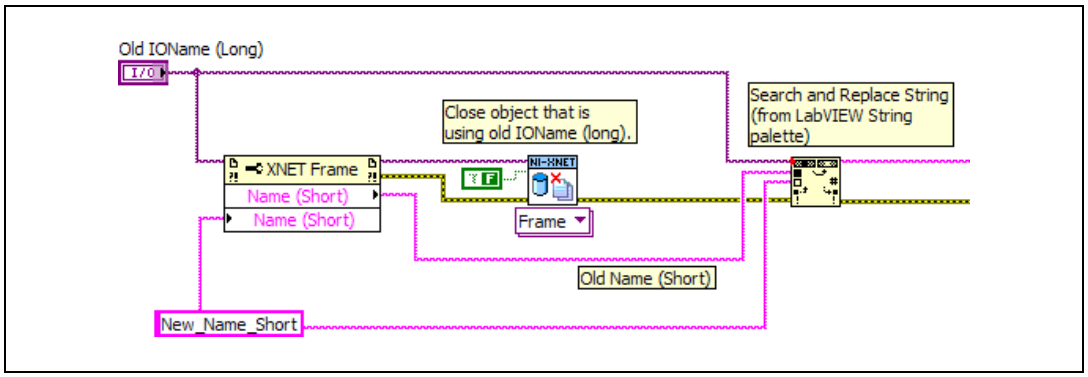
An ECU name must be unique for all ECUs in a cluster.

This short name does not include qualifiers to ensure that it is unique, such as the database and cluster name. It is for display purposes. The fully qualified name is available by using the XNET ECU I/O name as a string.

You can write this property to change the ECU's short name. When you do this and then use the original XNET ECU that contains the old name, errors can result because the old name cannot be found. Follow these steps to avoid this problem:

1. Get the old Name (Short) property using the property node.
2. Set the new Name (Short) property for the object.
3. Close the object using **XNET Database Close.vi**. Wire the **close all?** input as false to close the renamed object only.
4. Wire the XNET ECU as the input string to **Search and Replace String Function.vi** with the old **Name** as the search string and the new **Name** as the replacement string. This replaces the short name in the XNET ECU, while retaining the other text that ensures a unique name.

The following diagram demonstrates steps 1 through 4 for an XNET Frame I/O name:



XNET ECU Constant

This constant provides the constant form of the XNET ECU I/O name. You drag a constant to the block diagram of your VI, then select an ECU. You can change constants only during configuration, prior to running the VI. For a complete description, refer to *XNET ECU I/O Name*.

XNET Frame Property Node

Format



Description

Property node used to read/write properties for an XNET Frame I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

CAN:Extended Identifier?

Data Type	Direction	Required?	Default
TF	Read/Write	No	False

Property Class

XNET Frame

Short Name

CAN.ExtID?

Description

This property determines whether the XNET Frame Identifier property in a CAN cluster represents a standard 11-bit (false) or extended 29-bit (true) arbitration ID.

CAN:Timing Type

Data Type	Direction	Required?	Default
	Read/Write	No	Event Data (If Not in Database)

Property Class

XNET Frame

Short Name

CAN.TimingType

Description

Specifies the CAN frame timing.

Because this property specifies the behavior of the frame's transfer within the embedded system (for example, a vehicle), it describes the transfer between ECUs in the network. In the following description, *transmitting ECU* refers to the ECU that transmits the CAN data frame (and possibly receives the associated CAN remote frame). *Receiving ECU* refers to an ECU that receives the CAN data frame (and possibly transmits the associated CAN remote frame).

When you use the frame within an NI-XNET session, an output session acts as the transmitting ECU, and an input session acts as a receiving ECU. For a description of how these CAN timing types apply to the NI-XNET session mode, refer to *CAN Timing Type and Session Mode*.

The CAN timing types (decimal value in parentheses) are:

Cyclic Data (0) The transmitting ECU transmits the CAN data frame in a cyclic (periodic) manner. The XNET Frame CAN:Transmit Time property defines the time between cycles. The transmitting ECU ignores CAN remote frames received for this frame.

Event Data (1) The transmitting ECU transmits the CAN data frame in an event-driven manner. The XNET Frame CAN:Transmit Time property defines the minimum interval. For NI-XNET, the event occurs when you call **XNET Write.vi**. The transmitting ECU ignores CAN remote frames received for this frame.

Cyclic Remote (2) The receiving ECU transmits the CAN remote frame in a cyclic (periodic) manner. The XNET Frame CAN:Transmit Time property defines the time between cycles. The transmitting ECU responds to each CAN remote frame by transmitting the associated CAN data frame.

Event Remote (3) The receiving ECU transmits the CAN remote frame in an event-driven manner. The XNET Frame CAN:Transmit Time property defines the minimum interval. For NI-XNET, the event occurs when you call **XNET Write.vi**. The transmitting ECU responds to each CAN remote frame by transmitting the associated CAN data frame.

Cyclic/Event (4) This timing type is a combination of the cyclic and event timing. The frame is transmitted when you call **XNET Write.vi**, but also periodically sending the last recent values written. The XNET Frame CAN:Transmit Time property defines the cycle period. There is no minimum interval time defined in this mode, so be careful not to write too frequently to avoid creating a high busload.

If you are using a FIBEX database, this property is a required part of the XML schema for a frame, so the default (initial) value is obtained from the file.

If you are using a CANdb (`.dbc`) database, this property is an optional attribute in the file. If NI-XNET finds an attribute named GenMsgSendType, that attribute is the default value of this property. If the GenMsgSendType attribute begins with cyclic, this property's default value is Cyclic Data; otherwise, it is Event Data. If the CANdb file does not use the GenMsgSendType attribute, this property uses a default value of Event Data, which you can change in your application.

If you are using an `.ncd` database or an in-memory database (XNET Create Frame), this property uses a default value of Event Data. Within your application, change this property to the desired timing type.

CAN:Transmit Time

Data Type	Direction	Required?	Default
	Read/Write	No	0.1 (If Not in Database)

Property Class

XNET Frame

Short Name

CAN.TxTime

Description

Specifies the time between consecutive frames from the transmitting ECU.

The data type is 64-bit floating point (DBL). The units are in seconds.

Although the fractional part of the DBL data type can provide resolution of picoseconds, the NI-XNET CAN transmit supports an accuracy of 500 µs. Therefore, when used within an NI-XNET output session, this property is rounded to the nearest 500 µs increment (0.0005).

For a CAN:Timing Type of Cyclic Data or Cyclic Remote, this property specifies the time between consecutive data/remote frames. A time of 0.0 is invalid.

For a CAN:Timing Type of Event Data or Event Remote, this property specifies the minimum time between consecutive data/remote frames when the event occurs quickly. This is also known as the debounce time or minimum interval. The time is measured from the end of previous frame (acknowledgment) to the start of the next frame. A time of 0.0 specifies no minimum (back to back frames allowed).

If you are using a FIBEX database, this property is a required part of the XML schema for a frame, so the default (initial) value is obtained from the file.

If you are using a CANdb (`.dbc`) database, this property is an optional attribute in the file. If NI-XNET finds an attribute named GenMsgCycleTime, that attribute is interpreted as a number of milliseconds and used as the default value of this property. If the CANdb file does not use the GenMsgCycleTime attribute, this property uses a default value of 0.1 (100 ms), which you can change in your application.

If you are using a `.ncd` database or an in-memory database (XNET Create Frame), this property uses a default value of 0.1 (100 ms). Within your application, change this property to the desired time.

Application Protocol

Data Type	Direction	Required?	Default
	Read/Write	No	Read from Database

Property Class

XNET Frame

Short Name

ApplProtocol

Description

This property specifies the frame's application protocol. It is a ring of two values:

Enumeration	Value	Meaning
None	0	The default application protocol.
J1939	1	Indicates J1939 frames. The value enables the following features: • Sending/receiving long frames as the SAE J1939 specification specifies, using the J1939 transport protocol. • Using a special notation for J1939 identifiers.

Cluster

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Frame

Short Name

Cluster

Description

This property returns the I/O name of the parent cluster in which the frame has been created. You cannot change the parent cluster after the frame object has been created.

Comment

Data Type	Direction	Required?	Default
	Read/Write	No	Empty String

Property Class

XNET Frame

Short Name

Comment

Description

Comment describing the frame object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Frame

Short Name

ConfigStatus

Description

The frame object configuration status.

Configuration Status returns an NI-XNET error code. You can pass the value to **Simple Error Handler.vi** error code input to convert the value to a text description (on message output) of the configuration problem.

By default, incorrectly configured frames in the database are not returned from the XNET Cluster Frames property because they cannot be used in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When a frame configuration status became invalid after the database is opened, the frame still is returned from the XNET Cluster Frames property even if ShowInvalidFromOpen? is false.

Examples of invalid frame configuration:

- A required property of the frame or an object contained in this frame has not been defined. For example, Frame Payload Length.
- The number of bytes specified for this frame is incorrect. CAN frames must use 0 to 8 bytes. FlexRay frames must use 0 to 254 bytes (even numbers only).
- The CAN arbitration ID is invalid. The standard ID is greater than 0x7FF (11 bits) or the extended ID is greater than 0x1FFFFFFF (29 bits).
- The FlexRay frame is specified to use channels not defined in the cluster. For example, the XNET Cluster FlexRay:Channels property is set to Channel A only, but the XNET Frame FlexRay:Channel Assignment property is set to Channel A and B.
- The XNET Frame FlexRay:Channel Assignment property in this dynamic FlexRay frame is set to Channel A and B, but dynamic frames can be sent on only one channel (A or B).

Default Payload

Data Type	Direction	Required?	Default
	Read/Write	No	Array of All 0

Property Class

XNET Frame

Short Name

DefaultPayload

Description

The frame default payload, specified as an array of bytes (U8).

The number of bytes in the array must match the XNET Frame Payload Length property.

This property's initial value is an array of all 0. For the database formats NI-XNET supports, this property is not provided in the database file.

When you use this frame within an NI-XNET session, this property's use varies depending on the session mode. The following sections describe this property's behavior for each session mode.

Frame Output Single-Point and Frame Output Queued Modes

Use this property when a frame transmits prior to a call to **XNET Write.vi**. This can occur when you set the XNET Session Auto Start? property to false and call **XNET Start.vi** prior to **XNET Write.vi**. When Auto Start? is true (default), the first call to **XNET Write.vi** also starts frame transmit, so this property is not used.

The following frame configurations potentially can transmit prior to a call to **XNET Write.vi**:

- CAN:Timing Type of Cyclic Data
- CAN:Timing Type of Cyclic Remote (for example, a remote frame received prior to a call to **XNET Write.vi**)
- CAN:Timing Type of Event Remote (for example, a remote frame received prior to a call to **XNET Write.vi**)
- FlexRay:Timing Type of Cyclic
- LIN frame in a schedule entry of Type unconditional

The following frame configurations cannot transmit prior to a call to **XNET Write.vi**, so this property is not used:

- CAN:Timing Type of Event Data
- FlexRay:Timing Type of Event
- LIN frame in a schedule entry of Type sporadic or event triggered

Frame Output Stream Mode

This property is not used. Transmit is limited to frames provided to **XNET Write.vi**.

Signal Output Single-Point, Signal Output Waveform, and Signal Output XY Modes

Use this property when a frame transmits prior to a call to **XNET Write.vi**. Refer to *Frame Output Single-Point and Frame Output Queued Modes* for a list of applicable frame configurations.

This property is used as the initial payload, then each XNET Signal Default Value is mapped into that payload, and the result is used for the frame transmit.

Frame Input Stream and Frame Input Queued Modes

This property is not used. These modes do not return data prior to receiving frames.

Frame Input Single-Point Mode

This property is used for frames **XNET Read.vi** returns prior to receiving the first frame.

Signal Input Single-Point, Signal Input Waveform, and Signal Input XY Modes

This property is not used. Each XNET Signal Default Value is used when **XNET Read.vi** is called prior to receiving the first frame.

FlexRay:Base Cycle

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET Frame

Short Name

FlexRay.BaseCycle

Description

The first communication cycle in which a frame is sent.

In FlexRay, a communication cycle contains a number of slots in which a frame can be sent. Every node on the bus provides a 6-bit cycle counter that counts the cycles from 0 to 63 and then restarts at 0. The cycle number is common for all nodes on the bus.

NI-XNET has two mechanisms for changing the frame sending frequency:

- If the frame should be sent faster than the cycle period, use In-Cycle Repetition (refer to the XNET Frame FlexRay:In Cycle Repetitions:Identifiers property).
- If the frame should be sent slower than the cycle period, use this property and the XNET Frame FlexRay:Cycle Repetition property.

The second method is called cycle multiplexing. It allows sending multiple frames in the same slot, but on different cycle counters.

If a frame should be sent in every cycle, set this property to 0 and the XNET Frame FlexRay:Cycle Repetition property to 1. For cycle multiplexing, set the XNET Frame FlexRay:Cycle Repetition property to 2, 4, 8, 16, 32, or 64.

Example:

- FrameA and FrameB are both sent in slot 12.
- **FrameA**: The XNET Frame FlexRay:Base Cycle property is 0 and XNET Frame FlexRay:Cycle Repetition property is 2. This frame is sent when the cycle counter has the value 0, 2, 4, 6, ...
- **FrameB**: The XNET Frame FlexRay:Base Cycle property is 1 and XNET Frame FlexRay:Cycle Repetition property is 2. This frame is sent when the cycle counter has the value 1, 3, 5, 7, ...

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Channel Assignment

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET Frame

Short Name

FlexRay.ChAssign

Description

This property determines on which FlexRay channels the frame must be transmitted. A frame can be transmitted only on existing FlexRay channels, configured in the XNET Cluster FlexRay:Channels property.

Frames in the dynamic FlexRay segment cannot be sent on both channels; they must use either channel A or B. Frames in the dynamic segment use slot IDs greater than **the number of static slots cluster** parameter.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Cycle Repetition

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET Frame

Short Name

FlexRay.CycleRep

Description

The number of cycles after which a frame is sent again.

In FlexRay, a communication cycle contains a number of slots in which a frame can be sent. Every node on the bus provides a 6-bit cycle counter that counts the cycles from 0 to 63 and then restarts at 0. The cycle number is common for all nodes on the bus.

NI-XNET has two mechanisms for changing the frame sending frequency:

- If the frame should be sent faster than the cycle period, use In-Cycle Repetition (refer to the XNET Frame FlexRay:In Cycle Repetitions:Identifiers property).
- If the frame should be sent slower than the cycle period, use the XNET Frame FlexRay:Base Cycle property and this property.

The second method is called cycle multiplexing. It allows sending multiple frames in the same slot, but on different cycle counters.

If a frame should be sent in every cycle, set the XNET Frame FlexRay:Base Cycle property to 0 and this property to 1. For cycle multiplexing, set this property to 2, 4, 8, 16, 32, or 64.

Examples:

- FrameA and FrameB are both sent in slot 12.
- FrameA: The XNET Frame FlexRay:Base Cycle property is set to 0 and XNET Frame FlexRay:Cycle Repetition property is set to 2. This frame is sent when the cycle counter has the value 0, 2, 4, 6, ...
- FrameB: The XNET Frame FlexRay:Base Cycle property is set to 1 and XNET Frame FlexRay:Cycle Repetition property is set to 2. This frame is sent when the cycle counter has the value 1, 3, 5, 7, ...

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Payload Preamble?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Frame

Short Name

FlexRay.Preamble?

Description

This property determines whether payload preamble is used in a FlexRay frame:

- For frames in the static segment, it indicates that the network management vector is transmitted at the beginning of the payload.
- For frames in the dynamic segment, it indicates that the message ID is transmitted at the beginning of the payload.

FlexRay:Startup?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Frame

Short Name

FlexRay.Startup?

Description

This property determines whether the frame is a FlexRay startup frame. FlexRay startup frames always are FlexRay sync frames also.

- When this property is set to true, the XNET Frame FlexRay:Sync? property automatically is set to true.
- When this property is set to false, the XNET Frame FlexRay:Sync? property is not changed.
- When the XNET Frame FlexRay:Sync? property is set to false, this property automatically is set to false.
- When the XNET Frame FlexRay:Sync? property is set to true, this property is not changed.

An ECU can send only one startup frame. The startup frame, if an ECU transmits it, is returned from the XNET ECU FlexRay:Startup Frame property.

FlexRay:Sync?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Frame

Short Name

FlexRay.Sync?

Description

This property determines whether the frame is a FlexRay sync frame. FlexRay startup frames always are FlexRay sync frames also:

- When this property is set to false, the XNET Frame FlexRay:Startup? property is automatically set to false.
- When this property is set to true, the XNET Frame FlexRay:Startup? property is not changed.
- When the XNET Frame FlexRay:Startup? property is set to true, this property is set to true.
- When the XNET Frame FlexRay:Startup? property is set to false, this property is not changed.

An ECU can send only one sync frame.

FlexRay:Timing Type

Data Type	Direction	Required?	Default
	Read/Write	No	Cyclic in Static Segment, Event in Dynamic Segment

Property Class

XNET Frame

Short Name

FlexRay.TimingType

Description

Specifies the FlexRay frame timing (decimal value in parentheses):

Cyclic (0) Payload data transmits on every occurrence of the frame's slot.

Event (1) Payload data transmits in an event-driven manner. Within the ECU that transmits the frame, the event typically is associated with the availability of new data.

This property's behavior depends on the FlexRay segment where the frame is located: static or dynamic. If the frame's Identifier (slot) is less than or equal to the cluster's Number Of Static Slots, the frame is static.

Static

Cyclic means no null frame is transmitted. If new data is not provided for the cycle, the previous payload data transmits again.

Event means a null frame is transmitted when no event is pending for the cycle.

This property's default value for the static segment is Cyclic.

Dynamic

Cyclic means the frame transmits in its minislot on every cycle.

Event means the frame transmits in the minislot when the event is pending for the cycle.

This property's default value for the dynamic segment is Event.

For a description of how these FlexRay timing types apply to the NI-XNET session mode, refer to *FlexRay Timing Type and Session Mode*.

FlexRay:In Cycle Repetitions:Channel Assignments

Data Type	Direction	Required?	Default
	Read/Write	No	Empty Array

Property Class

XNET Frame

Short Name

FlexRay.InCycRep.ChAssigns

Description

FlexRay channels for in-cycle frame repetition.

A FlexRay frame can be sent multiple times per cycle. The XNET Frame FlexRay:Channel Assignment property defines the first channel assignment in the cycle. This property defines subsequent channel assignments. The XNET Frame FlexRay:In Cycle Repetitions:Identifiers property defines the corresponding slot IDs. Both properties are arrays of maximum three values, determining the slot ID and channel assignments for the frame. Values at the same array position are corresponding; therefore, both arrays must have the same size.

You must set the FlexRay:Channel Assignment property before setting this property. FlexRay:Channel Assignment is a required property that is undefined when a new frame is created. When FlexRay:Channel Assignment is undefined, setting FlexRay:In Cycle Repetitions:Channel Assignments returns an error. For convenience, you can set both properties in one XNET Frame property node, setting the FlexRay:Channel Assignment first (the properties in a property node are set starting from top position to bottom).

FlexRay:In Cycle Repetitions:Enabled?

Data Type	Direction	Required?	Default
	Read Only	No	False

Property Class

XNET Frame

Short Name

FlexRay.InCycRep.Enabled?

Description

FlexRay in-cycle frame repetition is enabled.

A FlexRay frame can be sent multiple times per cycle. The XNET Frame Identifier property defines the first slot ID in the cycle. The XNET Frame FlexRay:In Cycle Repetitions:Identifiers property can define the subsequent slot IDs, and the XNET Frame FlexRay:In Cycle Repetitions:Channel Assignments property defines the corresponding FlexRay channels. Both properties are arrays of maximum three values determining the slot ID and FlexRay channels for the frame. Values at the same array position are corresponding; therefore, both arrays must have the same size.

This property returns true when at least one in-cycle repetition has been defined, which means that both the FlexRay:In Cycle Repetitions:Identifiers and FlexRay:In Cycle Repetitions:Channel Assignments arrays are not empty.

This property returns false when at least one of the previously mentioned arrays is empty. In this case, in-cycle-repetition is not used.

FlexRay:In Cycle Repetitions:Identifiers

Data Type	Direction	Required?	Default
	Read/Write	No	Empty Array

Property Class

XNET Frame

Short Name

FlexRay.InCycRep.IDs

Description

FlexRay in-cycle repetition slot IDs.

A FlexRay frame can be sent multiple times per cycle. The XNET Frame Identifier property defines the first slot ID in the cycle. The FlexRay:In Cycle Repetitions:Identifiers property defines subsequent slot IDs. The XNET Frame FlexRay:In Cycle Repetitions:Channel Assignments property defines the corresponding FlexRay channel assignments. Both properties are arrays of maximum three values, determining the subsequent slot IDs and channel assignments for the frame. Values at the same array position are corresponding; therefore, both arrays must have the same size.

You must set the XNET Frame Identifier property before setting the FlexRay:In Cycle Repetitions:Identifiers property. Identifier is a required property that is undefined when a new frame is created. When Identifier is undefined, setting in-cycle repetition slot IDs returns an error. For your convenience, you can set both properties in one XNET Frame property node, setting the Identifier first (the properties in a property node are set starting from top position to bottom).

Identifier

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET Frame

Short Name

ID

Description

Determines the frame identifier.

This property is required. If the property does not contain a valid value, and you create an XNET Session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information on using database files and in-memory databases, refer to *Databases*.

CAN

For CAN frames, this is the Arbitration ID.

When the XNET Frame CAN:Extended Identifier? property is set to false, this is the standard CAN identifier with a size of 11 bits, which results in allowed range of 0–2047. However, the CAN standard disallows identifiers in which the first 7 bits are all recessive, so the working range of identifiers is 0–2031.

When the XNET Frame CAN:Extended Identifier? property is set to true, this is the extended CAN identifier with a size of 29 bits, which results in allowed range of 0–536870911.

FlexRay

For FlexRay frames, this is the Slot ID in which the frame is sent. The valid value range for a FlexRay Slot ID is 1–2047.

You also can send a FlexRay frame in multiple slots per cycle. You can define subsequent slot IDs for the frame in the XNET Frame FlexRay:In Cycle Repetitions:Identifiers property. Use this concept to increase a frame's sending frequency. To decrease a frame's sending frequency and share the same slot for different frames depending on the cycle counter, refer to the XNET Frame FlexRay:Base Cycle and FlexRay:Cycle Repetition properties.

The slot ID determines whether a FlexRay frame is sent in a static or dynamic segment. If the slot ID is less than or equal to the XNET Cluster FlexRay:Number of Static Slots property, the frame is sent in the communication cycle static segment; otherwise, it is sent in the dynamic segment.

If the frame identifier is not in the allowed range, this is reported as an error in the XNET Frame Configuration Status property.

LIN

For LIN frames, this is the frame's ID (unprotected). The valid range for a LIN frame ID is 0–63 (inclusive).

LIN:Checksum

Data Type	Direction	Required?	Default
U32	Read Only	N/A	Enhanced

Property Class

XNET Frame

Short Name

LIN.Checksum

Description

Determines whether the LIN frame transmitted checksum is classic or enhanced. The enhanced checksum considers the protected identifier when it is generated.

This property is a ring (enumerated list) with the following values:

String	Value
Classic	0
Enhanced	1

The checksum is determined from the LIN version of ECUs transmitting and receiving the frame. The lower version of both ECUs is significant. If the LIN version of both ECUs is 2.0 or higher, the checksum type is enhanced; otherwise, the checksum type is classic.

Diagnostic frames (with decimal identifier 60 or 61) always use classic checksum, even on LIN 2.*x*.

Mux:Data Multiplexer Signal

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Frame

Short Name

DataMuxSig

Description

Data multiplexer signal in the frame.

This property returns an I/O name of the data multiplexer signal. If the data multiplexer is not defined in the frame, the I/O control is empty. Use the XNET Frame Mux:Is Data Multiplexed? property to determine whether the frame contains a multiplexer signal.

You can create a data multiplexer signal by creating a signal and then setting the XNET Signal Mux:Data Multiplexer? property to true.

A frame can contain only one data multiplexer signal.

Mux:Is Data Multiplexed?

Data Type	Direction	Required?	Default
	Read Only	No	False

Property Class

XNET Frame

Short Name

Mux.IsMuxed?

Description

Frame is data multiplexed.

This property returns true if the frame contains a multiplexer signal. Frames containing a multiplexer contain subframes that allow using bits of the frame payload for different information (signals) depending on the multiplexer value.

Mux:Static Signals

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Frame

Short Name

Mux.StatSigs

Description

Static signals in the frame.

Returns an array of I/O names of signals in the frame that do not depend on the multiplexer value. Static signals are contained in every frame transmitted, as opposed to dynamic signals, which are transmitted depending on the multiplexer value.

You can create static signals by specifying the frame as the parent object. You can create dynamic signals by specifying a subframe as the parent.

If the frame is not multiplexed, this property returns the same array as the XNET Frame Signals property.

Mux:Subframes

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Frame

Short Name

Mux.Subframes

Description

Returns an array of I/O names of subframes in the frame. A subframe defines a group of signals transmitted using the same multiplexer value. Only one subframe at a time is transmitted in the frame.

A subframe is defined by creating a subframe object as a child of a frame.

Name (Short)

Data Type	Direction	Required?	Default
	Read/Write	Yes	Defined in Create Object

Property Class

XNET Frame

Short Name

NameShort

Description

String identifying a frame object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

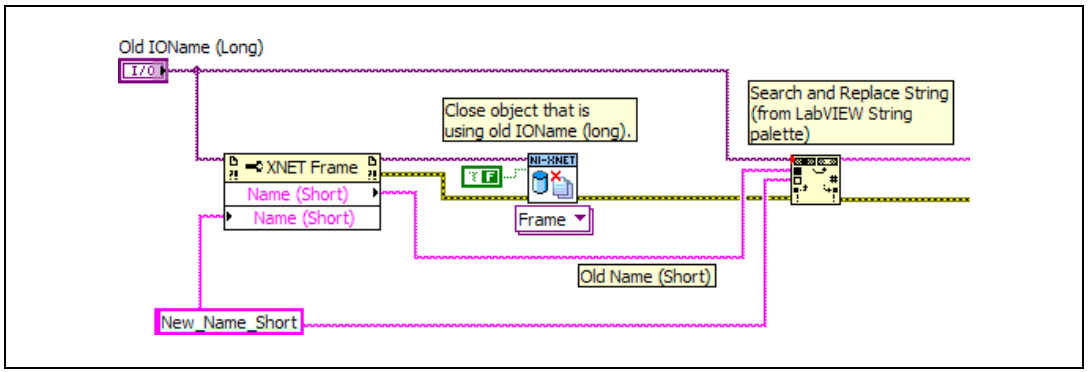
A frame name must be unique for all frames in a cluster.

This short name does not include qualifiers to ensure that it is unique, such as the database and cluster name. It is for display purposes. The fully qualified name is available by using the XNET Frame I/O name as a string.

You can write this property to change the frame's short name. When you do this and then use the original XNET Frame that contains the old name, errors can result because the old name cannot be found. Follow these steps to avoid this problem:

1. Get the old Name (Short) property using the property node.
2. Set the new Name (Short) property for the object.
3. Close the object using **XNET Database Close.vi**. Wire the **close all?** input as false to close the renamed object only.
4. Wire the XNET Frame as the input string to **Search and Replace String Function.vi** with the old **Name** as the search string and the new **Name** as the replacement string. This replaces the short name in the XNET Frame, while retaining the other text that ensures a unique name.

The following diagram demonstrates steps 1 through 4 for an XNET Frame I/O name:



Payload Length

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET Frame

Short Name

PayldLen

Description

Number of bytes of data in the payload.

For CAN or LIN, this is 0–8.

For FlexRay, this is 0–254. As encoded on the FlexRay bus, all frames use an even payload (16-bit words), and the payload of all static slots must be the same. Nevertheless, this property specifies the number of payload bytes used within the frame, so its value can be odd. For example, if a FlexRay cluster uses static slots of 18 bytes, it is valid for this property to be 15, which specifies that the last 3 bytes are unused.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

PDU_Mapping

Data Type	Direction	Required?	Default
	Read/Write	No	Empty Array

Property Class

XNET Frame

Short Name

PDU_Mapping

Description

This property maps existing PDUs to a frame. A mapped PDU is transmitted inside the frame payload when the frame is transmitted. You can map one or more PDUs to a frame and one PDU to multiple frames.

One PDU_Mapping cluster (a LabVIEW cluster, as opposed to a database cluster object) from the array assigns one PDU to the frame. The cluster contains the following elements:

- **PDU**: A string using the PDU I/O name syntax. If you wire an I/O name input to a string output, LabVIEW converts the I/O name to a string.
- **Start Bit**: Defines the start bit of the PDU inside the frame.
- **Update Bit**: Defines the update bit for the PDU inside the frame. If the update bit is not used, set the value to –1. (Refer to *Update Bit* for more information.)

Databases imported from FIBEX prior to version 3.0 from DBC, NCD, or LDF files have a strong one-to-one relationship between frames and PDUs. Every frame has exactly one PDU mapped, and every PDU is mapped to exactly one frame.

To unmap PDUs from a frame, set this property to an empty array. A frame without mapped PDUs contains no signals.

NI-XNET supports advanced PDU configuration (multiple PDUs in one frame or one PDU used in multiple frames) only for FlexRay. Refer to the XNET Cluster PDUs Required? property.

For CAN and LIN, NI-XNET supports only a one-to-one relationship between frames and PDUs. For those interfaces, advanced PDU configuration returns an error from the XNET Frame Configuration Status property and **XNET Create Session.vi**. If you do not use advanced PDU configuration, you can avoid using PDUs in the database API and create signals and subframes directly on a frame.

Signals

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Frame

Short Name

Sigs

Description

I/O names of all signals in the frame.

This property returns an array referencing all signals in the frame, including static and dynamic signals and the multiplexer signal.

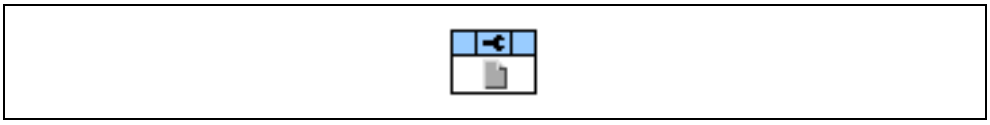
This property is read only. You can add signals to a frame using **XNET Database Create Object.vi** and remove them using **XNET Database Delete Object.vi**.

XNET Frame Constant

This constant provides the constant form of the XNET Frame I/O name. You drag a constant to the block diagram of your VI, then select a frame. You can change constants only during configuration, prior to running the VI. For a complete description, refer to *XNET Frame I/O Name*.

XNET PDU Property Node

Format



Description

Property node used to read/write properties for an XNET PDU I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

Cluster

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET PDU

Short Name

Cluster

Description

This property returns the I/O name to the parent cluster in which the PDU has been created. You cannot change the parent cluster after creating the PDU object.

Comment

Data Type	Direction	Required?	Default
	Read/Write	No	Empty String

Property Class

XNET PDU

Short Name

Comment

Description

Comment describing the PDU object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET PDU

Short Name

ConfigStatus

Description

The PDU object's configuration status.

Configuration Status returns an NI-XNET error code. The value can be passed to the **Simple Error Handler.vi** error code input to convert it to a text description (on message output) of the configuration problem.

By default, incorrectly configured PDUs in the database are not returned from the XNET Cluster PDUs property because they cannot be used in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When a PDU's configuration status became invalid after the database has been opened, the PDU still is returned from the Cluster PDUs property even if ShowInvalidFromOpen? is false.

Examples of invalid PDU configuration:

- You have not defined a required property of the PDU (for example, PDU Payload Length).
- The number of bytes specified for this PDU is incorrect. CAN PDUs must use 0 to 8 bytes. FlexRay PDUs must use 0 to 254 bytes (PDUs payload must fit into a frame).

Frames

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET PDU

Short Name

Frms

Description

I/O names of all frames to which the PDU is mapped. A PDU is transmitted within the frames to which it is mapped.

To map a PDU to a frame, use the XNET Frame PDU_Mapping property. You can map one PDU to multiple frames.

Mux:Data Multiplexer Signal

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET PDU

Short Name

DataMuxSig

Description

Data multiplexer signal in the PDU.

This property returns the data multiplexer signal I/O name. If the data multiplexer is not defined in the PDU, the I/O control is empty. Use the XNET PDU Mux:Is Data Multiplexed? property to determine whether the PDU contains a multiplexer signal.

You can create a data multiplexer signal by creating a signal and then setting the XNET Signal Mux:Data Multiplexer? property to true.

A PDU can contain only one data multiplexer signal.

Mux:Is Data Multiplexed?

Data Type	Direction	Required?	Default
	Read Only	No	False

Property Class

XNET PDU

Short Name

Mux.IsMuxed?

Description

PDU is data multiplexed.

This property returns true if the PDU contains a multiplexer signal. PDUs containing a multiplexer contain subframes that allow using bits of the payload for different information (signals), depending on the multiplexer value.

Mux:Static Signals

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET PDU

Short Name

Mux.StatSigs

Description

Static signals in the PDU.

Returns an array of I/O names of signals in the PDU that do not depend on the multiplexer value. Static signals are contained in every PDU transmitted, as opposed to dynamic signals, which are transmitted depending on the multiplexer value.

You can create static signals by specifying the PDU as the parent object. You can create dynamic signals by specifying a subframe as the parent.

If the PDU is not multiplexed, this property returns the same array as the XNET PDU Signals property.

Mux:Subframes

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET PDU

Short Name

Mux.Subframes

Description

Returns an array of I/O names of subframes in the PDU. A subframe defines a group of signals transmitted using the same multiplexer value. Only one subframe is transmitted in the PDU at a time.

You can define a subframe by creating a subframe object as a child of a PDU.

Name (Short)

Data Type	Direction	Required?	Default
	Read/Write	Yes	Defined in Create Object

Property Class

XNET PDU

Short Name

NameShort

Description

String identifying a PDU object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A PDU name must be unique for all PDUs in a cluster.

You can write this property to change the PDU's short name. When you do this and then use the original XNET PDU that contains the old name, errors can result because the old name cannot be found. Follow these steps to avoid this problem:

1. Get the old Name (Short) property using the property node.
2. Set the new Name (Short) property for the object.
3. Wire the XNET PDU as the input string to **Search and Replace String Function.vi** with the old **Name** as the search string and the new **Name** as the replace string. This replaces the short name in the XNET PDU, while retaining the other text that ensures a unique name.
4. Wire the result from **Search and Replace String Function.vi** to XNET String to IO Name.vi. This casts the string back to a valid XNET PDU.

Payload Length

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET PDU

Short Name

PayldLen

Description

Determines the size of the PDU data in bytes.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this PDU, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Signals

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET PDU

Short Name

Sigs

Description

I/O names of all signals in the PDU.

This property returns an array referencing to all signals in the PDU, including static and dynamic signals and the multiplexer signal.

This property is read only. You can add signals to a PDU using **XNET Database Create Object.vi** and remove them using **XNET Database Delete Object.vi**.

XNET PDU Constant

This constant provides the constant form of the XNET PDU I/O name. You drag a constant to the block diagram of your VI, then select a PDU. You can change constants only during configuration, prior to running the VI. For a complete description, refer to *XNET PDU I/O Name*.

XNET Subframe Property Node

Format



Description

Property node used to read/write properties for an XNET Subframe I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

Dynamic Signals

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Subframe

Short Name

DynSig

Description

Dynamic signals in the subframe.

This property returns an array of I/O names of dynamic signals in the subframe. Those signals are transmitted when the multiplexer signal in the frame has the multiplexer value defined in the subframe.

Dynamic signals are created with **XNET Database Create Object.vi** by specifying a subframe as the parent.

Frame

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Subframe

Short Name

Frame

Description

Returns the I/O name of the parent frame. The parent frame is defined when the subframe is created, and you cannot change it afterwards.

Multiplexer Value

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET Subframe

Short Name

MuxValue

Description

Multiplexer value for this subframe.

This property specifies the multiplexer signal value used when the dynamic signals in this subframe are transmitted in the frame. Only one subframe is transmitted at a time in the frame.

There is also a multiplexer value for a signal object as a read-only property. It reflects the value set on the parent subframe object.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this subframe, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Name (Short)

Data Type	Direction	Required?	Default
	Read/Write	Yes	Defined in Create Object

Property Class

XNET Subframe

Short Name

NameShort

Description

String identifying a subframe object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

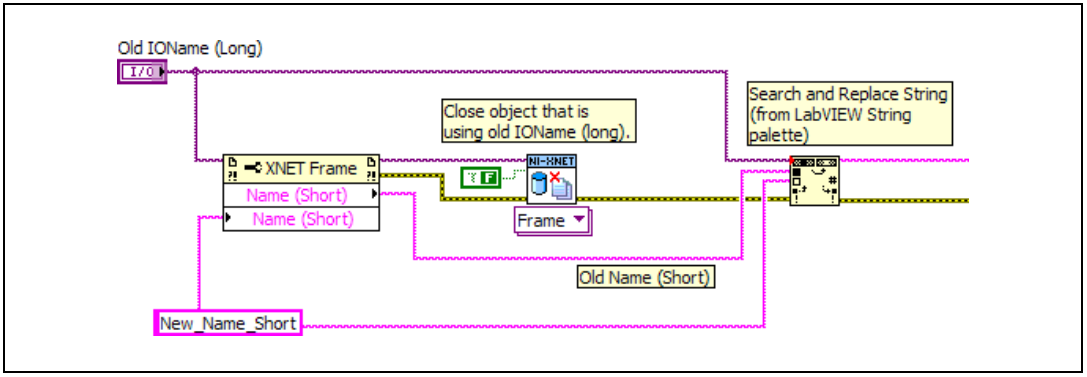
A subframe name must be unique for all subframes in a frame.

This short name does not include qualifiers to ensure that it is unique, such as the database, cluster, and frame name. It is for display purposes. The fully qualified name is available by using the XNET Subframe I/O name as a string.

You can write this property to change the subframe's short name. When you do this and then use the original XNET Subframe that contains the old name, errors can result because the old name cannot be found. Follow these steps to avoid this problem:

1. Get the old Name (Short) property using the property node.
2. Set the new Name (Short) property for the object.
3. Close the object using **XNET Database Close.vi**. Wire the **close all?** input as false to close the renamed object only.
4. Wire the XNET Subframe as the input string to **Search and Replace String Function.vi** with the old **Name** as the search string and the new **Name** as the replacement string. This replaces the short name in the XNET Subframe, while retaining the other text that ensures a unique name.

The following diagram demonstrates steps 1 through 4 for an XNET Frame I/O name:



PDU

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Subrame

Short Name

PDU

Description

I/O name of the subframe's parent PDU.

This property returns the I/O name of the subframe's parent PDU. The parent PDU is defined when the subframe object is created. You cannot change it afterwards.

XNET Signal Property Node

Format



Description

Property node used to read/write properties for an XNET Signal I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

Byte Order

Data Type	Direction	Required?	Default
U32	Read/Write	Yes	N/A

Property Class

XNET Signal

Short Name

ByteOrdr

Description

Signal byte order in the frame payload.

This property defines how signal bytes are ordered in the frame payload when the frame is loaded in memory.

- **Little Endian**: Higher significant signal bits are placed on higher byte addresses. In NI-CAN, this was called Intel Byte Order.

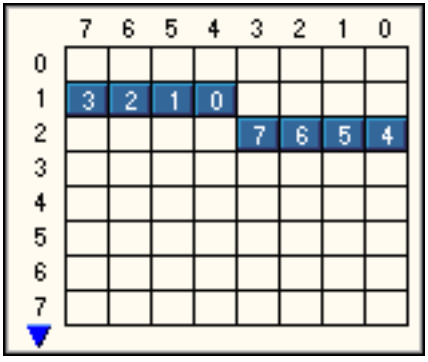


Figure 4-8. Little Endian Signal with Start Bit 12

- **Big Endian**: Higher significant signal bits are placed on lower byte addresses. In NI-CAN, this was called Motorola Byte Order.

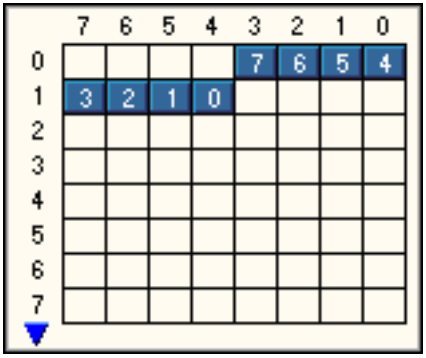


Figure 4-9. Big Endian Signal with Start Bit 12

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Comment

Data Type	Direction	Required?	Default
	Read/Write	No	Empty String

Property Class

XNET Signal

Short Name

Comment

Description

Comment describing the signal object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Signal

Short Name

ConfigStatus

Description

The signal object configuration status.

Configuration Status returns an NI-XNET error code. You can pass the value to **Simple Error Handler.vi** error code input to convert the value to a text description (on message output) of the configuration problem.

By default, incorrectly configured signals in the database are not returned from the XNET Frame Signals property because they cannot be used in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When a signal configuration status becomes invalid after the database is opened, the signal still is returned from the XNET Frame Signals property even if the XNET Database ShowInvalidFromOpen? property is false.

Examples of invalid signal configuration:

- The signal is specified using bits outside the frame payload.
- The signal overlaps another signal in the frame. For example, two multiplexed signals with the same multiplexer value are using the same bit in the frame payload.
- The signal with integer data type (signed or unsigned) is specified with more than 52 bits. This is not allowed due to internal limitation of the double data type that NI-XNET uses for signal values.
- The frame containing the signal is invalid (for example, a CAN frame is defined with more than 8 payload bytes).

Data Type

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET Signal

Short Name

DataType

Description

The signal data type.

This property determines how the bits of a signal in a frame must be interpreted to build a value.

- **Signed**: Signed integer with positive and negative values.
- **Unsigned**: Unsigned integer with no negative values.
- **IEEE Float**: Float value with 7 or 15 significant decimal digits (32 bit or 64 bit).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Default Value

Data Type	Direction	Required?	Default
	Read/Write	No	0.0 (If Not in Database)

Property Class

XNET Signal

Short Name

Default

Description

The signal default value, specified as scaled floating-point units.

The data type is 64-bit floating point (DBL).

The initial value of this property comes from the database. If the database does not provide a value, this property uses a default value of 0.0.

For all three signal output sessions, this property is used when a frame transmits prior to a call to **XNET Write.vi**. The XNET Frame Default Payload property is used as the initial payload, then the default value of each signal is mapped into that payload using this property, and the result is used for the frame transmit.

For all three signal input sessions, this property is returned for each signal when **XNET Read.vi** is called prior to receiving the first frame.

For more information about when this property is used, refer to the discussion of Read/Write for each session mode.

Mux:Dynamic?

Data Type	Direction	Required?	Default
	Read Only	No	False

Property Class

XNET Signal

Short Name

Mux.Dynamic?

Description

Use this property to determine if a signal is static or dynamic. Dynamic signals are transmitted in the frame when the multiplexer signal in the frame has a given value specified in the subframe. Use the Multiplexer Value property to determine with which multiplexer value the dynamic signal is transmitted.

This property is read only. To create a dynamic signal, create the signal object as a child of a subframe instead of a frame. The dynamic signal cannot be changed to a static signal afterwards.

In NI-CAN, dynamic signals were called mode-dependent signals.

Frame

Data Type	Direction	Required?	Default
	Read Only	N/A	Parent Frame

Property Class

XNET Signal

Short Name

Frame

Description

I/O name of the signal's parent frame.

This property returns the I/O name of the signal's parent frame. The parent frame is defined when the signal object is created. You cannot change it afterwards.

Maximum Value

Data Type	Direction	Required?	Default
	Read/Write	No	1000.0

Property Class

XNET Signal

Short Name

Max

Description

The scaled signal value maximum.

XNET Read.vi and **XNET Write.vi** do not limit the signal value to a maximum value. Use this database property to set the maximum value.

In LabVIEW, you can use this property to set the limits of front panel controls and indicators.

Minimum Value

Data Type	Direction	Required?	Default
	Read/Write	No	0.0

Property Class

XNET Signal

Short Name

Min

Description

The scaled signal value minimum.

XNET Read.vi and **XNET Write.vi** do not limit the signal value to a minimum value. Use this database property to set the minimum value.

In LabVIEW, you can use this property to set the limits of front panel controls and indicators.

Mux:Multiplexer Value

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Signal

Short Name

Mux.MuxValue

Description

The multiplexer value applies to dynamic signals only (the XNET Signal Mux:Dynamic? property returns true). This property defines which multiplexer value is transmitted in the multiplexer signal when this dynamic signal is transmitted in the frame.

The multiplexer value is determined in the subframe. All dynamic signals that are children of the same subframe object use the same multiplexer value.

Dynamic signals with the same multiplexer value may not overlap each other, the multiplexer signal, or static signals.

Mux:Data Multiplexer?

Data Type	Direction	Required?	Default
	Read/Write	No	False

Property Class

XNET Signal

Short Name

Mux.Muxer?

Description

This property defines the signal that is a multiplexer signal. A frame containing a multiplexer value is called a multiplexed frame.

A multiplexer defines an area within the frame to contain different information (dynamic signals) depending on the multiplexer signal value. Dynamic signals with a different multiplexer value (defined in a different subframe) can share bits in the frame payload. The multiplexer signal value determines which dynamic signals are transmitted in the given frame.

To define dynamic signals in the frame transmitted with a given multiplexer value, you first must create a subframe in this frame and set the multiplexer value in the subframe. Then you must create dynamic signals using **XNET Database Create (Dynamic Signal).vi** to create child signals of this subframe.

Multiplexer signals may not overlap other static or dynamic signals in the frame.

Dynamic signals may overlap other dynamic signals when they have a different multiplexer value.

A frame may contain only one multiplexer signal.

The multiplexer signal is not scaled. Scaling factor and offset do not apply.

In NI-CAN, the multiplexer signal was called mode channel.

Name (Short)

Data Type	Direction	Required?	Default
	Read/Write	Yes	Defined in Create Object

Property Class

XNET Signal

Short Name

NameShort

Description

String identifying a signal object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

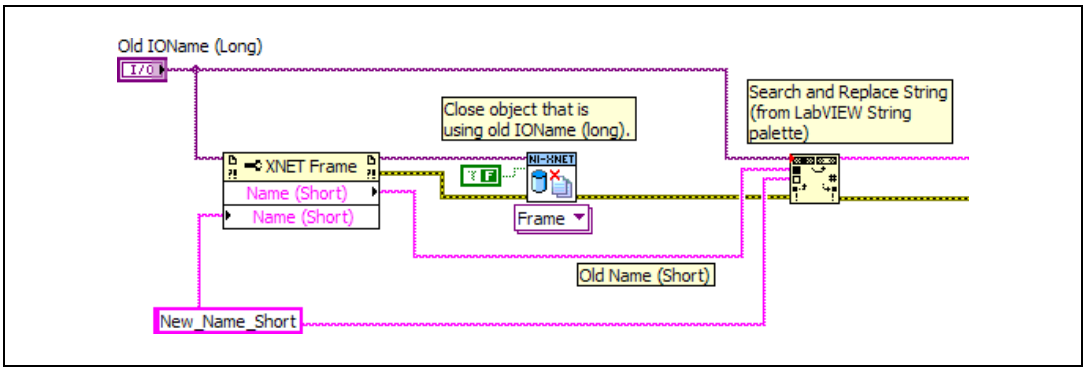
A signal name must be unique for all signals in a frame.

This short name does not include qualifiers to ensure that it is unique, such as the database, cluster, and frame name. It is for display purposes. The fully qualified name is available by using the XNET Signal I/O name as a string.

You can write this property to change the signal's short name. When you do this and then use the original XNET Signal that contains the old name, errors can result because the old name cannot be found. Follow these steps to avoid this problem:

1. Get the old Name (Short) property using the property node.
2. Set the new Name (Short) property for the object.
3. Close the object using **XNET Database Close.vi**. Wire the **close all?** input as false to close the renamed object only.
4. Wire the XNET Signal as the input string to **Search and Replace String Function.vi** with the old **Name** as the search string and the new **Name** as the replacement string. This replaces the short name in the XNET Signal, while retaining the other text that ensures a unique name.

The following diagram demonstrates steps 1 through 4 for an XNET Frame I/O name:



Number of Bits

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET Signal

Short Name

NumBits

Description

The number of bits the signal uses in the frame payload.

IEEE Float numbers are limited to 32 bit or 64 bit.

Integer (signed and unsigned) numbers are limited to 1–52 bits. NI-XNET converts all integers to doubles (64-bit IEEE Float). Integer numbers with more than 52 bits (the size of the mantissa in a 64-bit IEEE Float) cannot be converted exactly to double, and vice versa; therefore, NI-XNET does not support this.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

PDU

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET Signal

Short Name

PDU

Description

I/O name of the signal's parent PDU.

This property returns the I/O name of the signal's parent PDU. The parent PDU is defined when the signal object is created. You cannot change it afterwards.

Scaling Factor

Data Type	Direction	Required?	Default
	Read/Write	No	1.0

Property Class

XNET Signal

Short Name

ScaleFac

Description

Factor *a* for linear scaling *ax*+*b*.

Linear scaling is applied to all signals with the IEEE Float data type, unsigned and signed. For identical scaling 1.0*x*+0.0, NI-XNET optimized scaling routines do not perform the multiplication and addition.

Scaling Offset

Data Type	Direction	Required?	Default
	Read/Write	No	0.0

Property Class

XNET Signal

Short Name

ScaleOff

Description

Offset *b* for linear scaling *ax*+*b*.

Linear scaling is applied to all signals with the IEEE Float data type, unsigned and signed. For identical scaling 1.0*x*+0.0, NI-XNET optimized scaling routines do not perform the multiplication and addition.

Start Bit

Data Type	Direction	Required?	Default
U32	Read/Write	Yes	N/A

Property Class

XNET Signal

Short Name

StartBit

Description

The least significant signal bit position in the frame payload.

This property determines the signal starting point in the frame. For the integer data type (signed and unsigned), it means the binary signal representation least significant bit position. For IEEE Float signals, it means the mantissa least significant bit.

The NI-XNET Database Editor shows a graphical overview of the frame. It enumerates the frame bytes on the left and the byte bits on top. The bit number in the frame is calculated as byte number × 8 + bit number. The maximum bit number in a CAN or LIN frame is 63 (7 × 8 + 7); the maximum bit number in a FlexRay frame is 2031 (253 × 8 + 7).

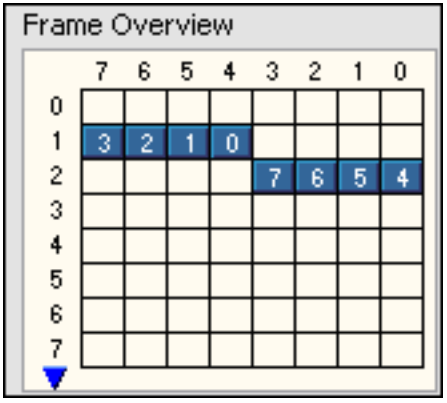


Figure 4-10. Frame Overview in the NI-XNET Database Editor with a Signal Starting in Bit 12

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Mux:Subframe

Data Type	Direction	Required?	Default
	Read Only	N/A	Parent Subframe

Property Class

XNET Signal

Short Name

Mux.Subfrm

Description

I/O name of the subframe parent.

This property is valid only for dynamic signals that have a subframe parent. For static signals or the multiplexer signal, this I/O name is empty.

Unit

Data Type	Direction	Required?	Default
	Read/Write	No	Empty String

Property Class

XNET Signal

Short Name

Unit

Description

This property describes the signal value unit. NI-XNET does not use the unit internally for calculations. You can use the string to display the signal value along with the unit on the front panel.

XNET Signal Constant

This constant provides the constant form of the XNET Signal I/O name. You drag a constant to the block diagram of your VI, then select a signal. You can change constants only during configuration, prior to running the VI. For a complete description, refer to *XNET Signal I/O Name*.

XNET Database Open.vi

Purpose

Opens an object from a database file.

Description

This VI is not required for LabVIEW 2009 or newer. It is provided only for backward compatibility of VIs written in LabVIEW versions prior to 2009. Newer versions of LabVIEW can detect the I/O name's first use as a refnum and open it automatically.

In addition to opening the refnum automatically, LabVIEW also closes it automatically.

XNET Database Close.vi

Purpose

Closes an object from a database, or all database objects.

Description

The instances of this polymorphic VI specify which objects to close:

- **XNET Database Close (Cluster).vi**
- **XNET Database Close (Database).vi**
- **XNET Database Close (ECU).vi**
- **XNET Database Close (Frame).vi**
- **XNET Database Close (PDU).vi**
- **XNET Database Close (Signal).vi**
- **XNET Database Close (Subframe).vi**
- **XNET Database Close (LIN Schedule).vi**
- **XNET Database Close (LIN Schedule Entry).vi**

XNET Database Close (Cluster).vi

Purpose

Closes a cluster from a database, or all database objects.

Format



Inputs

cluster in is the cluster to close.

close all? indicates that all open database objects will be closed. This is the default.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI closes a cluster object from a database (or all database objects). It is an instance of the **XNET Database Close** poly VI.

To simplify the task of closing all database objects you opened, use the **close all?** parameter set to true (default); otherwise, only the single database object wired in is closed.

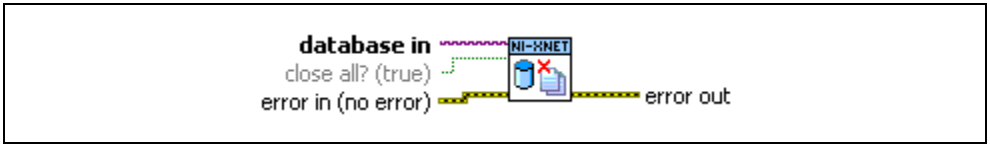
Database objects are closed automatically when the top-level VI terminates, so using this VI is optional. However, you may want to close database objects to free their memory prior to starting a session. You can use this VI to do this.

XNET Database Close (Database).vi

Purpose

Closes an XNET database, or all database objects.

Format



Inputs



database in is the database to close.



close all? indicates that all open database objects will be closed. This is the default.



error in is the error cluster input (refer to *Error Handling*).

Outputs



error out is the error cluster output (refer to *Error Handling*).

Description

This VI closes an XNET database (or all database objects). It is an instance of the **XNET Database Close** poly VI.

To simplify the task of closing all database objects you opened, you can use the **close all?** parameter set to true (default); otherwise, only the single database object wired in is closed.



Note Even if the database has been closed (using **close all?** set to false), all database objects retrieved from this database must be closed separately.

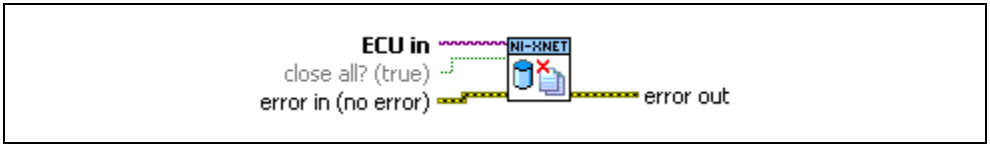
Database objects are closed automatically when the top-level VI terminates, so using this VI is optional. However, you may want to close database objects to free their memory prior to starting a session. You can use this VI to do this.

XNET Database Close (ECU).vi

Purpose

Closes an ECU from a database, or all database objects.

Format



Inputs

ECU in is the ECU to close.

close all? indicates that all open database objects will be closed. This is the default.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI closes an ECU object from a database (or all database objects). It is an instance of the **XNET Database Close** poly VI.

To simplify the task of closing all database objects you opened, you can use the **close all?** parameter set to true (default); otherwise, only the single database object wired in is closed.

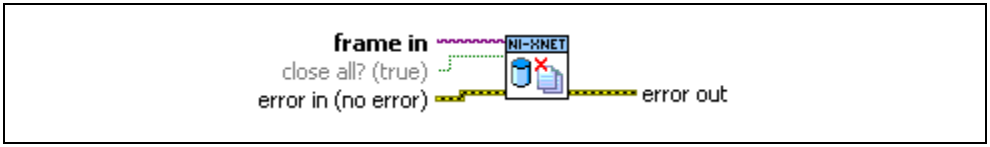
Database objects are closed automatically when the top-level VI terminates, so using this VI is optional. However, you may want to close database objects to free their memory prior to starting a session. You can use this VI to do this.

XNET Database Close (Frame).vi

Purpose

Closes a frame from a database, or all database objects.

Format



Inputs



frame in is the frame to close.



close all? indicates that all open database objects will be closed. This is the default.



error in is the error cluster input (refer to *Error Handling*).

Outputs



error out is the error cluster output (refer to *Error Handling*).

Description

This VI closes a frame object from a database (or all database objects). It is an instance of the **XNET Database Close** poly VI.

To simplify the task of closing all database objects you opened, you can use the **close all?** parameter set to true (default); otherwise, only the single database object wired in is closed.

Database objects are closed automatically when the top-level VI terminates, so using this VI is optional. However, you may want to close database objects to free their memory prior to starting a session. You can use this VI to do this.

XNET Database Close (PDU).vi

Purpose

Closes a PDU from a database, or all database objects.

Format



Inputs

PDU in is the PDU to close.

close all? indicates that all open database objects will be closed. This is the default.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI closes a PDU object from a database (or all database objects). It is an instance of the **XNET Database Close** poly VI.

To simplify the task of closing all database objects you opened, you can use the **close all?** parameter set to true (default); otherwise, only the single database object wired in is closed.

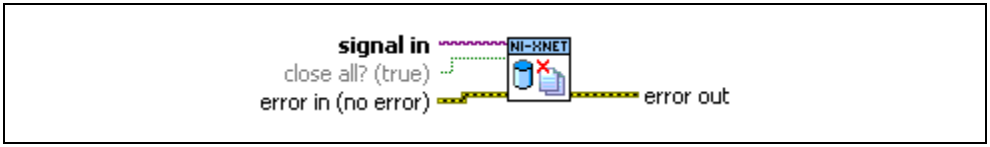
Database objects are closed automatically when the top-level VI terminates, so using this VI is optional. However, you may want to close database objects to free their memory prior to starting a session. You can use this VI to do this.

XNET Database Close (Signal).vi

Purpose

Closes a signal from a database, or all database objects.

Format



Inputs

signal in is the signal to close.

close all? indicates that all open database objects will be closed. This is the default.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI closes a signal object from a database (or all database objects). It is an instance of the **XNET Database Close** poly VI.

To simplify the task of closing all database objects you opened, you can use the **close all?** parameter set to true (default); otherwise, only the single database object wired in is closed.

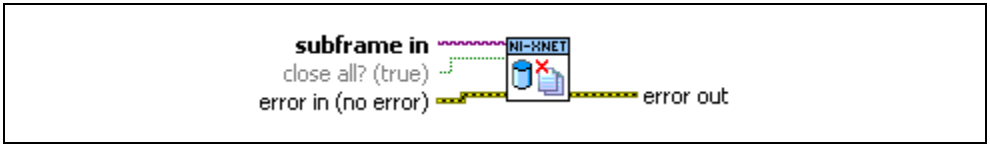
Database objects are closed automatically when the top-level VI terminates, so using this VI is optional. However, you may want to close database objects to free their memory prior to starting a session. You can use this VI to do this.

XNET Database Close (Subframe).vi

Purpose

Closes a subframe from a database, or all database objects.

Format



Inputs

subframe in is the subframe to close.

close all? indicates that all open database objects will be closed. This is the default.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI closes a subframe object from a database (or all database objects). It is an instance of the **XNET Database Close** poly VI.

To simplify the task of closing all database objects you opened, you can use the **close all?** parameter set to true (default); otherwise, only the single database object wired in is closed.

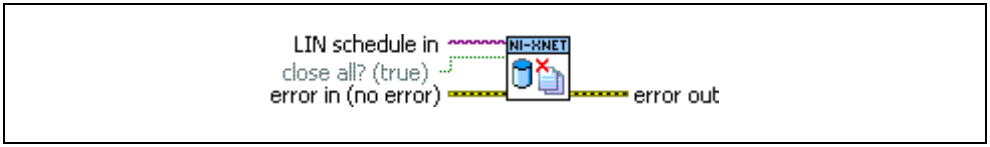
Database objects are closed automatically when the top-level VI terminates, so using this VI is optional. However, you may want to close database objects to free their memory prior to starting a session. You can use this VI to do this.

XNET Database Close (LIN Schedule).vi

Purpose

Closes a LIN schedule object from a database, or all database objects.

Format



Inputs



LIN schedule in is the schedule to close.



close all? indicates that all open database objects will be closed. This is the default.



error in is the error cluster input (refer to *Error Handling*).

Outputs



error out is the error cluster output (refer to *Error Handling*).

Description

This VI closes a LIN schedule object from a database (or all database objects). It is an instance of the **XNET Database Close** poly VI.

To simplify the task of closing all database objects you opened, you can use the **close all?** parameter set to true (default); otherwise, only the single database object wired in is closed.

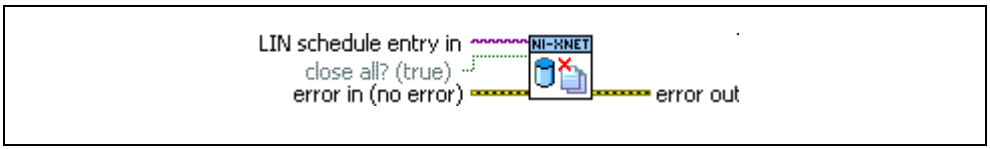
Database objects are closed automatically when the top-level VI terminates, so using this VI is optional. However, you may want to close database objects to free their memory prior to starting a session. You can use this VI to do this.

XNET Database Close (LIN Schedule Entry).vi

Purpose

Closes a LIN schedule entry from a database, or all database objects.

Format



Inputs



LIN schedule entry in is the schedule entry to close.



close all? indicates that all open database objects will be closed. This is the default.



error in is the error cluster input (refer to *Error Handling*).

Outputs



error out is the error cluster output (refer to *Error Handling*).

Description

This VI closes a LIN schedule entry object from a database (or all database objects). It is an instance of the **XNET Database Close** poly VI.

To simplify the task of closing all database objects you opened, you can use the **close all?** parameter set to true (default); otherwise, only the single database object wired in is closed.

Database objects are closed automatically when the top-level VI terminates, so using this VI is optional. However, you may want to close database objects to free their memory prior to starting a session. You can use this VI to do this.

XNET Database Create Object.vi

Purpose

Creates a new database object.

Description

The instances of this polymorphic VI specify which database objects to create:

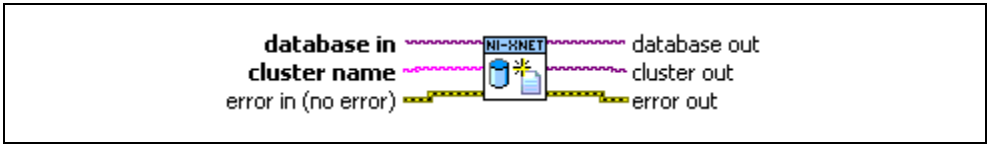
- **XNET Database Create (Cluster).vi**
- **XNET Database Create (Dynamic Signal).vi**
- **XNET Database Create (ECU).vi**
- **XNET Database Create (Frame).vi**
- **XNET Database Create (PDU).vi**
- **XNET Database Create (Signal).vi**
- **XNET Database Create (Subframe).vi**
- **XNET Database Create (LIN Schedule).vi**
- **XNET Database Create (LIN Schedule Entry).vi**

XNET Database Create (Cluster).vi

Purpose

Creates a new XNET cluster.

Format



Inputs

database in is the parent database object. **database in** can be an existing file. You can create a new database in memory by specifying *:memory:* for **database in** and create an entire hierarchy of objects in memory, without using a file on the disk.

cluster name is the name of the cluster to create. The name must be unique for all clusters in a database. Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The name is limited to 128 characters.

error in is the error cluster input (refer to *Error Handling*).

Outputs

database out is a copy of the **database in** parameter. You can use this output to wire the VI to subsequent VIs.

cluster out is I/O name of the newly created cluster object.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI creates an XNET cluster object. It is an instance of **XNET Database Create Object.vi**.

The **cluster name** input becomes the Name (Short) property of the created object. This is distinct from the string contained within **cluster out**, which uses the syntax described in *XNET Cluster I/O Name*.

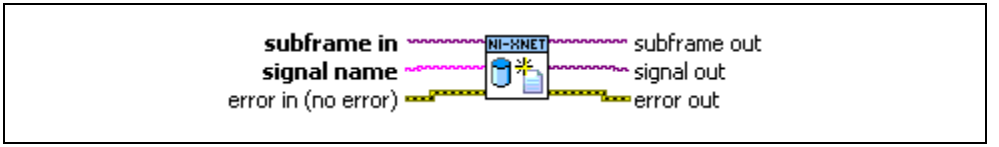
The cluster object is created and remains in memory until the database is closed. This VI does not change the open database file on disk. To save the newly created object to the file, use **XNET Database Save.vi**.

XNET Database Create (Dynamic Signal).vi

Purpose

Creates a new XNET dynamic signal.

Format



Inputs

subframe in is the subframe parent object.

signal name is the name of the signal to create. The name must be unique for all signals in a frame in which the subframe parent was defined, including the static signals and the multiplexer signal. Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The name is limited to 128 characters.

error in is the error cluster input (refer to *Error Handling*).

Outputs

subframe out is a copy of the subframe in parameter. You can use this parameter to wire the VI to subsequent VIs.

signal out is the I/O name of the newly created signal object.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI creates an XNET dynamic signal object. It is an instance of **XNET Database Create Object.vi**.

The **signal name** input becomes the Name (Short) property of the created object. This is distinct from the string contained within **signal out**, which uses the syntax described in *XNET Signal I/O Name*.

The signal object is created and remains in memory until the database is closed. This VI does not change the open database file on disk. To save the newly created object to the file, use **XNET Database Save.vi**.

Dynamic Signal is transmitted in the frame when the multiplexer signal contains the multiplexer value defined in the subframe.

In NI-CAN, dynamic signals were called mode-dependent channels.

XNET Database Create (ECU).vi

Purpose

Creates a new XNET ECU.

Format



Inputs

cluster in is the cluster parent object.

ECU name is the name of the ECU to create. The name must be unique for all ECUs in a cluster. Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The name is limited to 128 characters.

error in is the error cluster input (refer to *Error Handling*).

Outputs

cluster out is a copy of the **cluster in** parameter. You can use this output to wire the VI to subsequent VIs.

ECU out is the I/O name of the newly created ECU object.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI creates an XNET ECU object. It is an instance of **XNET Database Create Object.vi**.

The **ECU name** input becomes the Name (Short) property of the created object. This is distinct from the string contained within **ECU out**, which uses the syntax described in *XNET ECU I/O Name*.

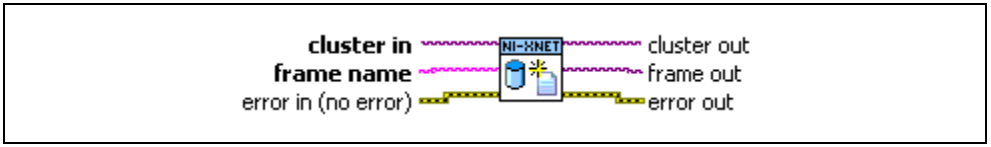
The ECU object is created and remains in memory until the database is closed. This VI does not change the open database file on disk. To save the newly created object to the file, use **XNET Database Save.vi**.

XNET Database Create (Frame).vi

Purpose

Creates a new XNET frame.

Format



Inputs

cluster in is the cluster parent object.

frame name is the name of the frame to create. The name must be unique for all frames in a cluster. Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The name is limited to 128 characters.

error in is the error cluster input (refer to *Error Handling*).

Outputs

cluster out is a copy of the **cluster in** parameter. You can use this output to wire the VI to subsequent VIs.

frame out is the I/O name of the newly created frame object.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI creates an XNET frame object. It is an instance of **XNET Database Create Object.vi**.

The **frame name** input becomes the Name (Short) property of the created object. This is distinct from the string contained within **frame out**, which uses the syntax described in *XNET Frame I/O Name*.

The frame object is created and remains in memory until the database is closed. This VI does not change the open database file on disk. To save the newly created object to the file, use **XNET Database Save.vi**.

XNET Database Create (PDU).vi

Purpose

Creates a new XNET PDU.

Format



Inputs

cluster in is the cluster parent object.

PDU name is the name of the PDU to create. The name must be unique for all PDUs in a cluster. Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The name is limited to 128 characters.

error in is the error cluster input (refer to *Error Handling*).

Outputs

cluster out is a copy of the **cluster in** parameter. You can use this output to wire the VI to subsequent VIs.

PDU out is the reference to the newly created PDU object.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI creates an XNET PDU object. It is an instance of **XNET Database Create Object.vi**.

The **PDU name** input becomes the Name (Short) property of the created object. This is distinct from the string contained within **PDU out**, which uses the syntax described in *XNET PDU I/O Name*.

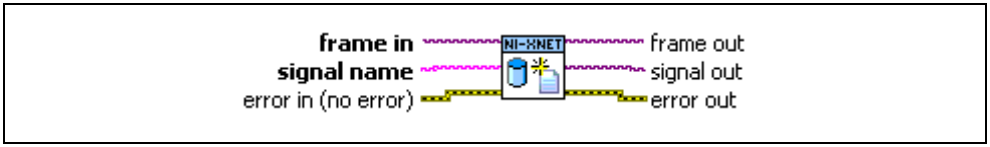
The PDU object is created and remains in memory until the database is closed. This VI does not change the open database file on disk. To save the new created object to the file, use **XNET Database Save.vi**.

XNET Database Create (Signal).vi

Purpose

Creates a new XNET signal.

Format



Inputs

frame in is the frame parent object.

signal name is the name of the signal to create. Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The name is limited to 128 characters.

error in is the error cluster input (refer to *Error Handling*).

Outputs

frame out is a copy of the **frame in** parameter. You can use this parameter to wire the VI to subsequent VIs.

signal out is the I/O name of the newly created signal object.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI creates an XNET signal object. It is an instance of**XNET Database Create Object.vi**.

The **signal name** input becomes the Name (Short) property of the created object. This is distinct from the string contained within **signal out**, which uses the syntax described in *XNET Session I/O Name*.

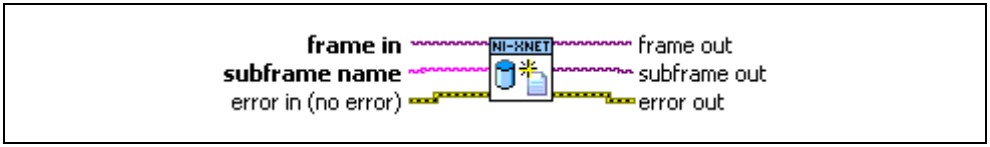
The signal object is created and remains in memory until the database is closed. This VI does not change the open database file on disk. To save the newly created object to the file, use **XNET Database Save.vi**.

XNET Database Create (Subframe).vi

Purpose

Creates a new XNET subframe.

Format



Inputs

frame in is the frame parent object.

subframe name is the name of the subframe to create. The name must be unique for all subframes in a frame. Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The name is limited to 128 characters.

error in is the error cluster input (refer to *Error Handling*).

Outputs

frame out is a copy of the frame in parameter. You can use this parameter to wire the VI to subsequent VIs.

subframe out is the I/O name of the newly created subframe object.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI creates an XNET subframe object. It is an instance of **XNET Database Create Object.vi**.

The **subframe name** input becomes the Name (Short) property of the created object.

The subframe object is created and remains in memory until the database is closed. This VI does not change the open database file on disk. To save the newly created object to the file, use **XNET Database Save.vi**.

A subframe defines the multiplexer value for all dynamic signals in this subframe. Dynamic signals within a subframe inherit the multiplexer value from the subframe parent.

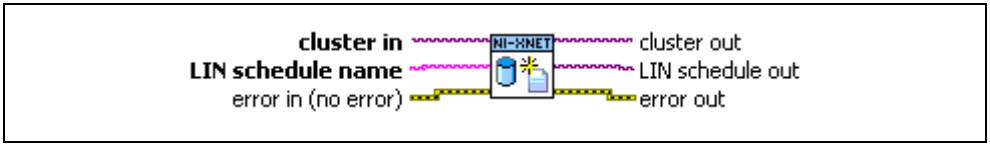
In NI-CAN, a subframe was called a mode.

XNET Database Create (LIN Schedule).vi

Purpose

Creates a new XNET LIN schedule.

Format



Inputs

cluster in is the cluster parent object.

LIN schedule name is the name of the schedule to create. The name must be unique for all schedules in a cluster. Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The name is limited to 128 characters.

error in is the error cluster input (refer to *Error Handling*).

Outputs

cluster out is a copy of the **cluster in** parameter. You can use this output to wire the VI to subsequent VIs.

LIN schedule out is the I/O name of the newly created LIN schedule object.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI creates an XNET LIN schedule object. It is an instance of **XNET Database Create Object.vi**.

The **LIN schedule name** input becomes the Name (Short) property of the created object. This is distinct from the string contained within **LIN schedule out**, which uses the syntax described in XNET LIN Schedule I/O Name.

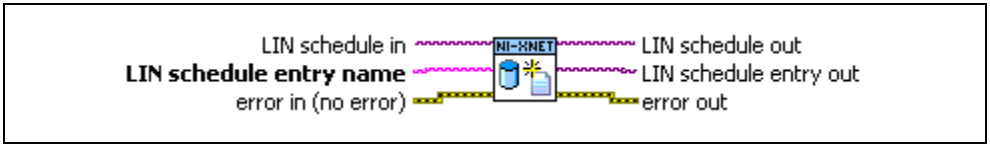
The schedule object is created and remains in memory until the database is closed. This VI does not change the open database file on disk. To save the newly created object to the file, use **XNET Database Save.vi**.

XNET Database Create (LIN Schedule Entry).vi

Purpose

Creates a new XNET LIN schedule entry object.

Format



Inputs

LIN schedule in is the schedule parent object.

LIN schedule entry name is the name of the schedule entry to create. The name must be unique for all entries in a schedule. Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The name is limited to 128 characters.

error in is the error cluster input (refer to *Error Handling*).

Outputs

LIN schedule out is a copy of the **LIN schedule in** parameter. You can use this parameter to wire the VI to subsequent VIs.

LIN schedule entry out is the I/O name of the newly created LIN schedule entry object.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI creates an XNET schedule entry object. It is an instance of **XNET Database Create Object.vi**.

Schedule entries is an ordered array in a schedule. The schedule is being processed in the order of this array. A newly created entry always is added to the last position of the array.

The **LIN schedule entry name** input becomes the Name (Short) property of the created object. This is distinct from the string contained in **LIN schedule entry out**, which uses the syntax described in XNET LIN Schedule Entry I/O Name.

The schedule object is created and remains in memory until the database is closed. This VI does not change the open database file on disk. To save the newly created object to the file, use **XNET Database Save.vi**.

XNET Database Delete Object.vi

Purpose

Deletes a database object.

Description

The instances of this polymorphic VI specify which database objects to delete:

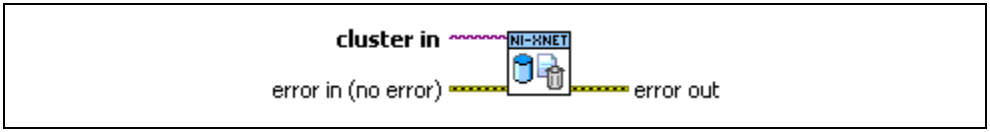
- **XNET Database Delete (Cluster).vi**
- **XNET Database Delete (ECU).vi**
- **XNET Database Delete (Frame).vi**
- **XNET Database Delete (PDU).vi**
- **XNET Database Delete (Signal).vi**
- **XNET Database Delete (Subframe).vi**
- **XNET Database Delete (LIN Schedule).vi**
- **XNET Database Delete (LIN Schedule Entry).vi**

XNET Database Delete (Cluster).vi

Purpose

Deletes an XNET cluster and all child objects in this cluster.

Format



Inputs



cluster in is the I/O name of the cluster to delete.



error in is the error cluster input (refer to *Error Handling*).

Outputs



error out is the error cluster output (refer to *Error Handling*).

Description

This VI deletes an XNET cluster object with all frames, PDUs, signals, subframes, and ECUs in this cluster. It is an instance of **XNET Database Delete Object.vi**.

Upon deletion, the I/O names of all deleted objects are closed and no longer can be used.

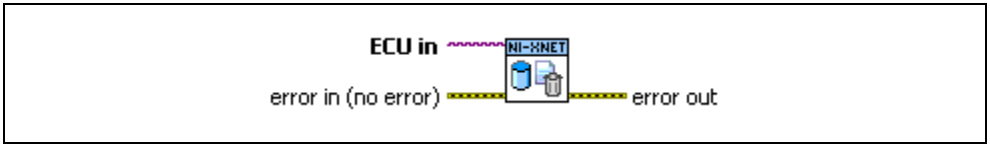
The objects are deleted from a database in memory. The change is in force until the database is closed. This VI does not change the open database file on disk. To save the changed database to the file, use **XNET Database Save.vi**.

XNET Database Delete (ECU).vi

Purpose

Deletes an XNET ECU.

Format



Inputs

ECU in is the I/O name of the ECU to delete.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI deletes an XNET ECU object. It is an instance of **XNET Database Delete Object.vi**.

Upon deletion, the I/O name of the ECU is closed and no longer can be used.

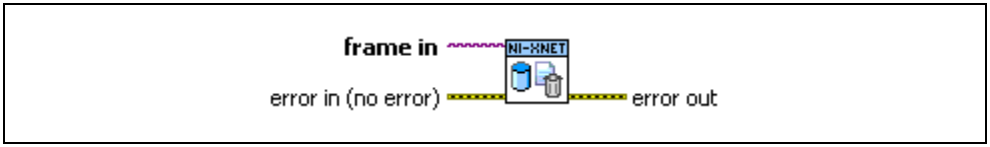
The ECU object is deleted from a database in memory and is in force until the database is closed. This VI does not change the open database file on disk. To save the changed database to the file, use **XNET Database Save.vi**.

XNET Database Delete (Frame).vi

Purpose

Deletes an XNET frame and all child objects in the frame.

Format



Inputs

frame in is the I/O name of the frame to delete.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI deletes an XNET frame object with all mapped PDUs, including signals and subframes in those PDUs. It is an instance of **XNET Database Delete Object.vi**. To avoid deleting PDUs with the frame, unmap the PDUs from the frame before deleting the frame (set the XNET Frame PDU_Mapping property to an empty array).

Upon deletion, the I/O names of all deleted objects are closed and no longer can be used.

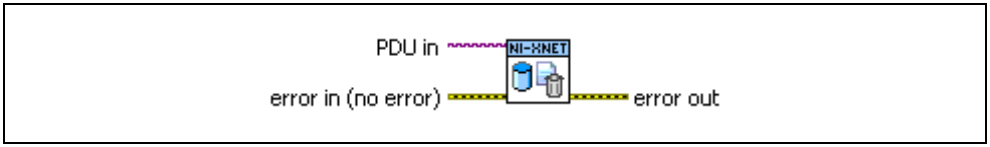
The objects are deleted from a database in memory. The change is in force until the database is closed. This VI does not change the open database file on disk. To save the changed database to the file, use **XNET Database Save.vi**.

XNET Database Delete (PDU).vi

Purpose

Delete an XNET PDU and all child objects in this PDU.

Format



Inputs



PDU in references the PDU to delete.



error in is the error cluster input (refer to *Error Handling*).

Outputs



error out is the error cluster output (refer to *Error Handling*).

Description

This VI deletes an XNET PDU object with all signals and subframes in this PDU. It is an instance of **XNET Database Delete Object.vi**.

Upon deletion, the I/O names to all deleted objects are closed and no longer can be used.

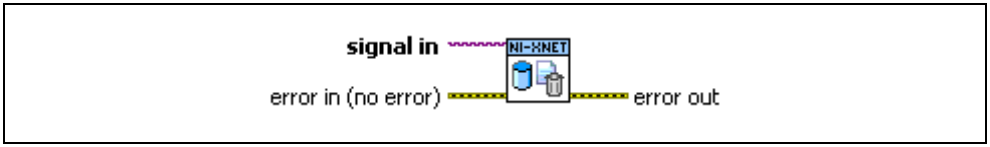
The objects are deleted from a database in memory. The change is in force until the database is closed. This VI does not change the open database file on disk. To save the changed database to the file, use **XNET Database Save.vi**.

XNET Database Delete (Signal).vi

Purpose

Deletes an XNET signal.

Format



Inputs

signal in is the I/O name of the signal to delete.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI deletes an XNET signal object. It is an instance of **XNET Database Delete Object.vi**.

Upon deletion, the I/O name of the signal is closed and no longer can be used.

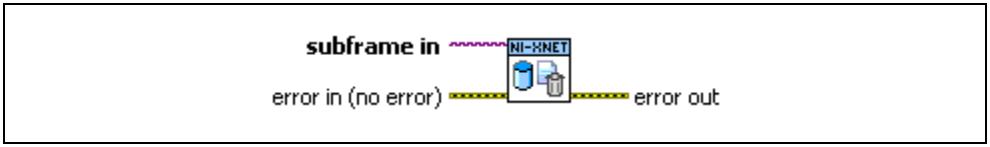
The signal object is deleted from a database in memory and is in force until the database is closed. This VI does not change the open database file on disk. To save the changed database to the file, use **XNET Database Save.vi**.

XNET Database Delete (Subframe).vi

Purpose

Deletes an XNET subframe and all dynamic signals in the subframe.

Format



Inputs



subframe in is the I/O name of the subframe to delete.



error in is the error cluster input (refer to *Error Handling*).

Outputs



error out is the error cluster output (refer to *Error Handling*).

Description

This VI deletes an XNET subframe object and all dynamic signals in this subframe. It is an instance of **XNET Database Delete Object.vi**.

Upon deletion, the I/O names of the subframe and related dynamic signals are closed and no longer can be used.

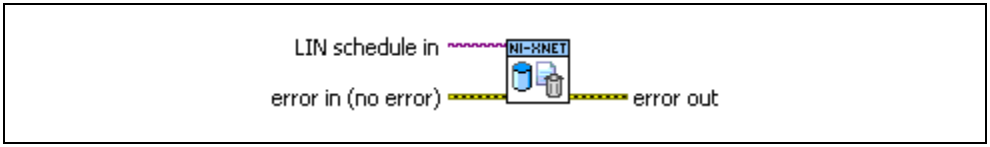
The objects are deleted from a database in memory. The change is in force until the database is closed. This VI does not change the open database file on disk. To save the changed database to the file, use **XNET Database Save.vi**.

XNET Database Delete (LIN Schedule).vi

Purpose

Deletes an XNET LIN schedule and all LIN schedule entry objects in this schedule.

Format



Inputs

LIN schedule in is the I/O name of the LIN schedule to delete.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI deletes an XNET LIN schedule object and the entries it contains. It is an instance of **XNET Database Delete Object.vi**.

Upon deletion, the I/O names of all deleted objects are closed, and you no longer can use them.

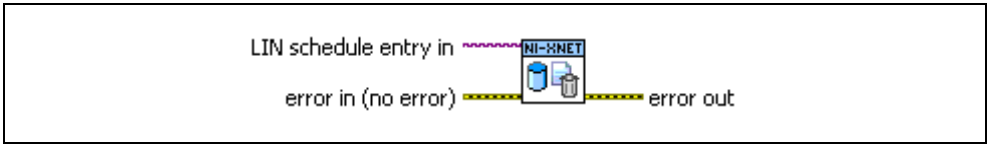
The LIN schedule object is deleted from a database in memory and is in force until the database is closed. This VI does not change the open database file on disk. To save the changed database to the file, use **XNET Database Save.vi**.

XNET Database Delete (LIN Schedule Entry).vi

Purpose

Deletes an XNET schedule entry object.

Format



Inputs



LIN schedule entry in is the I/O name of the LIN schedule entry to delete.



error in is the error cluster input (refer to *Error Handling*).

Outputs



error out is the error cluster output (refer to *Error Handling*).

Description

This VI deletes an XNET LIN schedule entry object. It is an instance of **XNET Database Delete Object.vi**.

Upon deletion, the I/O name of the deleted object is closed, and you no longer can use it.

The objects are deleted from a database in memory. The change is in force until the database is closed. This VI does not change the open database file on disk. To save the changed database to the file, use **XNET Database Save.vi**.

XNET Database Merge.vi

Purpose

Merges database objects and related child objects from the source to the destination cluster.

Description

The instances of this polymorphic VI specify which database objects to merge:

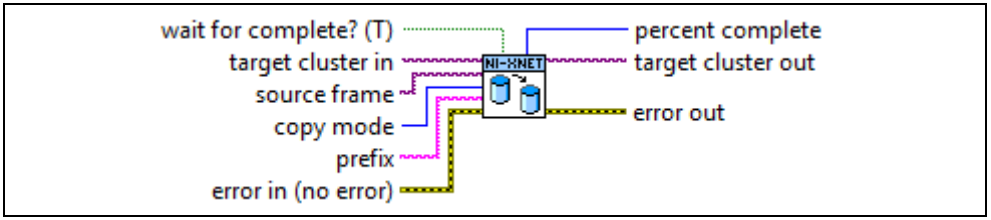
- **XNET Database Merge (Frame).vi**
- **XNET Database Merge (PDU).vi**
- **XNET Database Merge (ECU).vi**
- **XNET Database Merge (LIN Schedule).vi**
- **XNET Database Merge (Cluster).vi**

XNET Database Merge (Frame).vi

Purpose

Merges a frame object with all child objects into the destination cluster.

Format



Inputs



wait for complete? Use this input only if the source object is a cluster (refer to **XNET Database Merge (Cluster).vi**).



target cluster in is the I/O name of the cluster where the source frame is merged.



source frame is the I/O name of the frame to be merged into the target cluster.



copy mode defines the merging behavior if the target cluster already contains a frame with the same name.



prefix is added to the source frame name if a frame with the same name exists in the target cluster.



error in is the error cluster input (refer to *Error Handling*).

Outputs



percent complete is used when **wait for complete?** is false. (This output does not apply to the frame instance.)



target cluster out is a copy of **target cluster in**. You can use this output to wire the VI to subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

This VI merges a frame with all dependent child objects (PDUs, subframes, and signals) to the target cluster.

If the source frame name was not used in the target cluster, this VI copies the source frame with the child objects to the target. If a frame with the same name exists in the target cluster, you can avoid name collisions by specifying the prefix to be added to the name.

If a frame with the same name exists in the target cluster, the merge behavior depends on the **copy mode** input:

- **Copy using source**: The target frame with all dependent child objects is removed from the target cluster and replaced by the source objects.
- **Copy using destination**: The source frame is ignored (the target cluster frame with child objects remains unchanged).
- **Merge using source**: This adds child objects from the source frame to child objects from the destination frame. If the target frame contains a child object with the same name, it is replaced by the child object from the source frame. The source frame properties (for example, payload length) replace the target frame properties.
- **Merge using destination**: This adds child objects from the source frame to child objects from the destination frame. If the target frame contains a child object with the same name, it remains unchanged. The target frame properties remain unchanged (for example, payload length).

Example

Target frame F1(v1) has signals S1 and S2(v1). Source frame F1(v2) has signals S2(v2) and S3.

(v1) and (v2) are two versions of one object with same name, but with different properties.

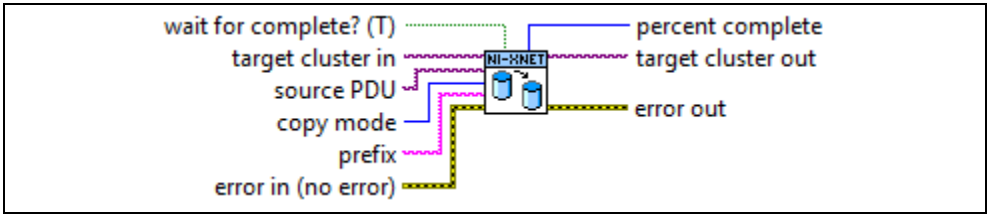
- Result of **Copy using source**: F1(v2), S2(v2), S3.
- Result of **Copy using destination**: F1(v1), S1, S2(v1).
- Result of **Merge using source**: F1(v2), S1, S2(v2), S3.
- Result of **Merge using destination**: F1(v1), S1, S2(v1), S3.

XNET Database Merge (PDU).vi

Purpose

Merges a PDU object with all child objects into the destination cluster.

Format



Inputs

wait for complete? Use this input only if the source object is a cluster (refer to **XNET Database Merge (Cluster).vi**).

target cluster in is the I/O name of the cluster where the source PDU is merged.

source PDU is the I/O name of the PDU to be merged into the target cluster.

copy mode defines the merging behavior if the target cluster already contains a PDU with the same name.

prefix is added to the source PDU name if a PDU with the same name exists in the target cluster.

error in is the error cluster input (refer to *Error Handling*).

Outputs

percent complete is used when **wait for complete?** is false. (This output does not apply to the PDU instance.)

target cluster out is a copy of **target cluster in**. You can use this output to wire the VI to subsequent VIs.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI merges a PDU with all dependent child objects (subframes and signals) to the target cluster.

If the source PDU name was not used in the target cluster, this VI copies the source PDU with the child objects to the target. If a PDU with the same name exists in the target cluster, you can avoid name collisions by specifying the prefix to be added to the name.

If a PDU with the same name exists in the target cluster, the merge behavior depends on the **copy mode** input:

- **Copy using source**: The target PDU with all dependent child objects is removed from the target cluster and replaced by the source objects.
- **Copy using destination**: The source PDU is ignored (the target cluster PDU with child objects remains unchanged).
- **Merge using source**: This adds child objects from the source PDU to child objects from the destination PDU. If the target PDU contains a child object with the same name, it is replaced by the child object from the source PDU. The source PDU properties (for example, payload length) replace the target PDU properties.
- **Merge using destination**: This adds child objects from the source PDU to child objects from the destination PDU. If the target PDU contains a child object with the same name, it remains unchanged. The target PDU properties remain unchanged (for example, payload length).

Example

Target PDU Pdu1(v1) has signals S1 and S2(v1). Source PDU Pdu1(v2) has signals S2(v2) and S3.

(v1) and (v2) are two versions of one object with same name but with different properties.

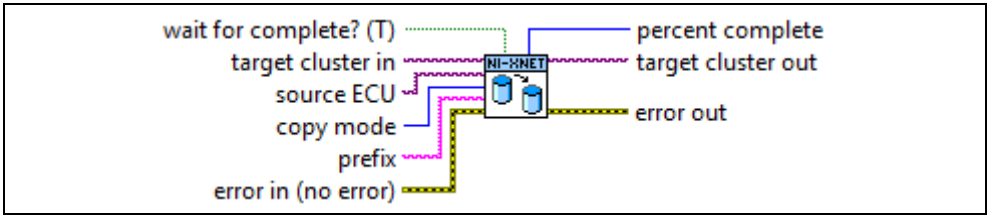
- Result of **Copy using source**: Pdu1(v2), S2(v2), S3.
- Result of **Copy using destination**: Pdu1(v1), S1, S2(v1).
- Result of **Merge using source**: Pdu1(v2), S1, S2(v2), S3.
- Result of **Merge using destination**: Pdu1(v1), S1, S2(v1), S3.

XNET Database Merge (ECU).vi

Purpose

Merges an ECU object with Tx/Rx frames into the destination cluster.

Format



Inputs

wait for complete? Use this input only if the source object is a cluster (refer to **XNET Database Merge (Cluster).vi**).

target cluster in is the I/O name of the cluster where the source ECU is merged.

source ECU is the I/O name of the ECU to be merged into the target cluster.

copy mode defines the merging behavior if the target cluster already contains an ECU with the same name.

prefix is added to the source ECU name if an ECU with the same name exists in the target cluster.

error in is the error cluster input (refer to *Error Handling*).

Outputs

percent complete is used when **wait for complete?** is false. (This output does not apply to the ECU instance.)

target cluster out is a copy of **target cluster in**. You can use this output to wire the VI to subsequent VIs.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI merges an ECU with all Tx/Rx frames to the target cluster. It does not merge the frames itself, but only the transmitting or receiving information. This happens based on frame names. If the source cluster defines new frames not contained in the destination cluster, they should be merged before merging the ECU; otherwise, the Tx/Rx information is removed.

If the source ECU name was not used in the target cluster, this VI copies the source ECU to the target. If an ECU with the same name exists in the target cluster, you can avoid name collisions by specifying the prefix to be added to the name.

If an ECU with the same name exists in the target cluster, the merge behavior depends on the **copy mode** input:

- **Copy using source**: The target ECU with all Tx/Rx information is removed from the target cluster and replaced by the source objects.
- **Copy using destination**: The source ECU is ignored (the target cluster ECU with child objects remains unchanged).
- **Merge using source**: This adds Tx/Rx frames from the source ECU to Tx/Rx from the destination ECU. The source ECU properties (for example, comment) replace the target ECU properties.
- **Merge using destination**: This adds Tx/Rx frames from the source ECU to Tx/Rx from the destination ECU. The target ECU properties remain unchanged (for example, comment).

Example

Target ECU Ecu1(v1) has Tx frames F1 and F2. Source ECU Ecu1(v2) has Tx frames F2 and F3.

(v1) and (v2) are two versions of one object with same name but with different properties.

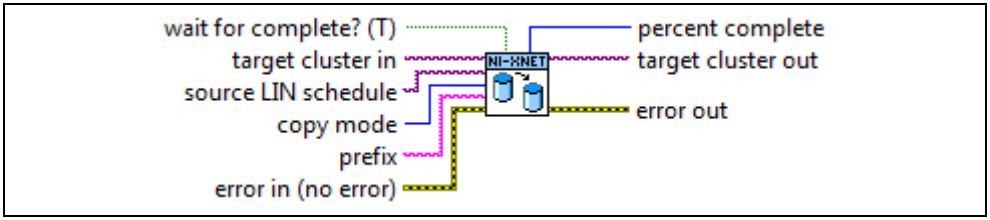
- Result of **Copy using source**: Ecu1(v2), F2, F3.
- Result of **Copy using destination**: Ecu1(v1), F1, F2.
- Result of **Merge using source**: Ecu1(v2), F1, F2, F3.
- Result of **Merge using destination**: Ecu1(v1), F1, F2, F3.

XNET Database Merge (LIN Schedule).vi

Purpose

Merges a LIN schedule object with all child objects into the destination cluster.

Format



Inputs

wait for complete? Use this input only if the source object is a cluster (refer to **XNET Database Merge (Cluster).vi**).

target cluster in is the I/O name of the cluster where the source LIN schedule is merged.

source LIN schedule is the I/O name of the LIN schedule to be merged into the target cluster.

copy mode defines the merging behavior if the target cluster already contains a LIN schedule with the same name.

prefix is added to the source LIN schedule name if a LIN schedule with the same name exists in the target cluster.

error in is the error cluster input (refer to *Error Handling*).

Outputs

percent complete is used when **wait for complete?** is false. (This output does not apply to the LIN schedule instance.)

target cluster out is a copy of **target cluster in**. You can use this output to wire the VI to subsequent VIs.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI merges a LIN schedule with all schedule entries to the target cluster. Frames referenced in the schedule entries should be merged before merging the LIN schedule; otherwise, the reference get lost.

If the source LIN schedule name was not used in the target cluster, this VI copies the source LIN schedule with the entries to the target. If a LIN schedule with the same name exists in the target cluster, you can avoid name collisions by specifying the prefix to be added to the name.

If a LIN schedule with the same name exists in the target cluster, the merge behavior depends on the **copy mode** input:

- **Copy using source**: The target LIN schedule with entries is removed from the target cluster and replaced by the source objects.
- **Copy using destination**: The source LIN schedule is ignored (the target cluster schedule with entries remains unchanged).
- **Merge using source**: This adds schedule entries from the source schedule at the end of the destination schedule table. The copied entries become new names, so all entry names in the schedule are unique. The source schedule properties replace the target schedule properties (comment, priority, run mode).
- **Merge using destination**: This adds schedule entries from the source schedule at the end of the destination schedule table. The copied entries become new names, so all entry names in the schedule are unique. The target schedule properties (comment, priority, run mode) remain unchanged.

Example

Target LIN schedule LS1(v1) has entries e1, e2. Source LIN schedule LS1(v2) has entries e3, e4.

(v1) and (v2) are two versions of one object with same name but with different properties.

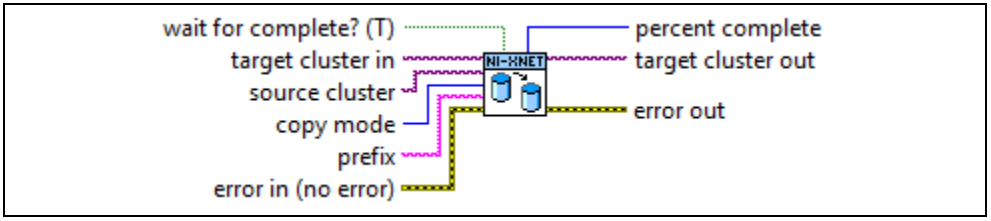
- Result of **Copy using source**: LS1(v1), e1, e2.
- Result of **Copy using destination**: LS1(v2), e3, e4.
- Result of **Merge using source**: LS1(v2),e1, e2, e3, e4.
- Result of **Merge using destination**: LS1(v1), e1, e2, e3, e4.

XNET Database Merge (Cluster).vi

Purpose

Merges a source cluster with all child objects into the destination cluster.

Format



Inputs



wait for complete? Use this input to split the merging process into parts (for example, to display a progress bar).



target cluster in is the I/O name of the cluster where the source cluster is merged.



source cluster is the I/O name of the cluster to be merged into the target cluster.



copy mode defines the merging behavior if the target cluster already contains elements with the same name.



prefix is added to the source cluster name if an element with the same name exists in the target cluster.



error in is the error cluster input (refer to *Error Handling*).

Outputs



percent complete is used when **wait for complete?** is false.



target cluster out is a copy of **target cluster in**. You can use this output to wire the VI to subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

This VI merges all objects contained in the source cluster into the target cluster.

The following VIs merge the objects with dependent-child objects:

- **XNET Database Merge (Frame).vi**
- **XNET Database Merge (PDU).vi**
- **XNET Database Merge (ECU).vi**
- **XNET Database Merge (LIN Schedule).vi**

Copy mode and **prefix** are passed to the appropriate VI for the merging process.

If the copy mode is set to **Copy using source** or **Merge using source**, all cluster properties including the name are copied from the source to the target cluster.

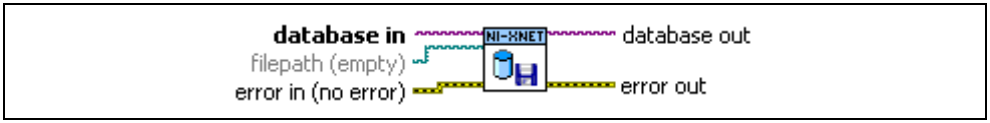
Depending on the number of contained objects in the source and destination clusters, the execution can take longer. If **wait for complete?** is true, this VI waits until the merging process gets completed. If the execution completes without errors, **percent complete** returns 100. If **wait for complete?** is false, the function returns quickly and **percent complete** returns values less than 100. You must call **XNET Database Merge.vi** repeatedly until **percent complete** returns 100. You can use the time between calls to update a progress bar.

XNET Database Save.vi

Purpose

Saves the open database to a FIBEX 3.1.1 file.

Format



Inputs

database in is the I/O name of the database.

filepath contains the pathname to the FIBEX file or is empty (saves to the original filepath).

error in is the error cluster input (refer to *Error Handling*).

Outputs

database out is a copy of the **database in** parameter. You can use this parameter to wire the VI to subsequent VIs.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI saves the XNET database current state to a FIBEX 3.1.1 file. The file extension must be `.xml`. If the target file exists, it is overwritten.

XNET saves to the FIBEX file only features that XNET sessions use to communicate on the network. If the original file was created using non-XNET software, the target file may be missing details from the original file. For example, NI-XNET supports only linear scaling. If the original FIBEX file used a rational equation that cannot be expressed as a linear scaling, XNET converts this to a linear scaling with factor 1.0 and offset 0.0.

If **filepath** is empty, the file is saved to the same FIBEX file specified when opened. If opened as a file path, it uses that file path. If opened as an alias, it uses the file path registered for that alias.

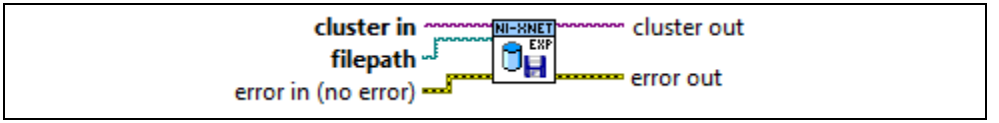
Saving a database is not supported in LabVIEW Real-Time, but you can deploy and use a database saved on Windows in LabVIEW Real-Time (refer to **XNET Database Deploy.vi**).

XNET Database Export.vi

Purpose

Exports a cluster from the open database to a file in a specific format.

Format



Inputs



cluster in is the I/O name of the cluster.



filepath contains the pathname to the file to be created.



error in is the error cluster input (refer to *Error Handling*).

Outputs



cluster out is a copy of the **cluster in** parameter. You can use this parameter to wire the VI to subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

This VI exports a cluster from an XNET database to a specific file format. A CAN cluster is exported as CANdb++ database (`.dbc`). A LIN cluster is exported as a LIN database file (`.ldf`). A FlexRay cluster cannot be exported and returns an error. If the target file exists, it is overwritten. The filepath parameter is required; you cannot accidentally overwrite the original file by specifying an empty filepath.

XNET saves to the file only features that XNET sessions use to communicate on the network. If the original file was created using non-XNET software, the target file may be missing details from the original file. For example, NI-XNET supports only linear scaling. If the original FIBEX file used a rational equation that cannot be expressed as a linear scaling, XNET converts this to a linear scaling with factor 1.0 and offset 0.0.

Exporting a database is not supported in LabVIEW Real-Time, but you can deploy and use a database saved on Windows in LabVIEW Real-Time (refer to **XNET Database Deploy.vi** for more information).

File Management Subpalette

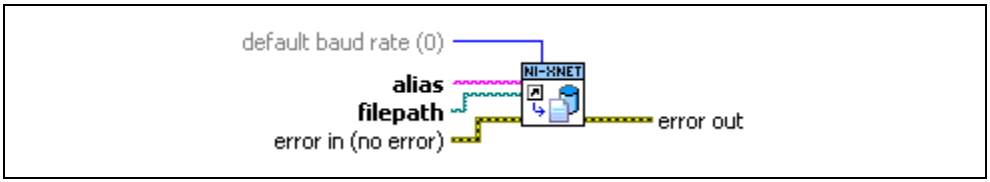
This subpalette includes VIs to manage database aliases and deploy or undeploy a database file to LabVIEW Real-Time (RT).

XNET Database Add Alias.vi

Purpose

Adds a new alias to a database file.

Format



Inputs



default baud rate provides the default baud rate, used when **filepath** refers to a CANdb database (`.dbc`) or an NI-CAN database (`.ncd`). These database formats are specific to CAN and do not specify a cluster baud rate. Use this default baud rate to specify a default CAN baud rate to use with this alias. If **filepath** refers to a FIBEX database (`.xml`) or LIN LDF file, the **default baud rate** parameter is ignored. The FIBEX and LDF database formats require a valid baud rate for every cluster, and NI-XNET uses that baud rate as the default.



alias provides the desired alias name. Unlike the name of other XNET database objects, the alias name can use special characters such as space and dash. If the alias name already exists, this VI changes the previous filepath to the specified filepath.



filepath provides the path to the CANdb, FIBEX, or LDF file.



error in is the error cluster input (refer to *Error Handling*).

Outputs



error out is the error cluster output (refer to *Error Handling*).

Description

NI-XNET uses alias names for database files. The alias names provide a shorter name for display, allow for changes to the file system without changing the application, and enable efficient deployment to LabVIEW Real-Time (RT) targets.

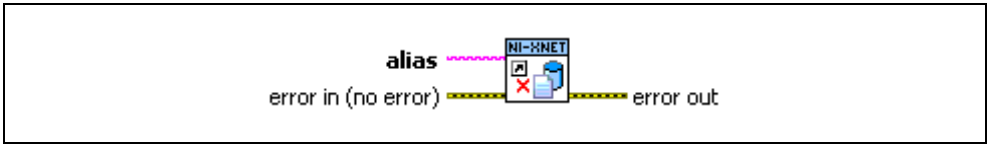
This VI is supported on Windows only. For LabVIEW RT, you can pass the new alias to **XNET Database Deploy.vi** to transfer an optimized binary image of the database to the LabVIEW RT target. After deploying the database, you can use the alias name in any VI for the Windows host and the LabVIEW RT target.

XNET Database Remove Alias.vi

Purpose

Removes a database alias from the system.

Format



Inputs

alias is the name of the alias to delete.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI removes the alias from NI-XNET, but does not affect the database text file. It just removes the alias association to the database filepath.

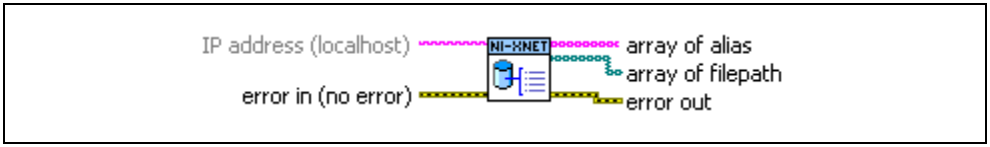
This VI is supported on Windows only, and the alias is removed from Windows only (not LabVIEW RT targets). Use **XNET Database Undeploy.vi** to remove an alias from a LabVIEW Real-Time (RT) target.

XNET Database Get List.vi

Purpose

Gets the current list of databases on a system.

Format



Inputs

IP address is the target IP address.

If **IP address** is unwired (empty), this VI retrieves aliases and file paths for the local Windows system.

If **IP address** is a valid IP address, this VI retrieves aliases and file paths for the remote LabVIEW RT target. You can find this IP address using MAX or VIs in the LabVIEW Real-Time palettes.

error in is the error cluster input (refer to *Error Handling*).

Outputs

array of alias returns an array of strings, one for every alias registered in the system. If no aliases are registered, the array is empty.

array of filepath returns an array of strings that contain the file paths and filenames of the databases assigned to the aliases, one for every alias registered in the system.

If no aliases are registered, the array is empty. This parameter applies to Windows targets only; on RT targets, this array always is empty.

error out is the error cluster output (refer to *Error Handling*).

Description

For a local Windows call (IP address empty), **array of filepath** returns an array of file paths. The size of this array is the same as **array of alias**. It provides the Windows file path for each corresponding alias.

For a remote call to LabVIEW RT, **array of filepath** is empty. NI-XNET handles the file system on the LabVIEW RT target automatically, so that only the alias is needed.

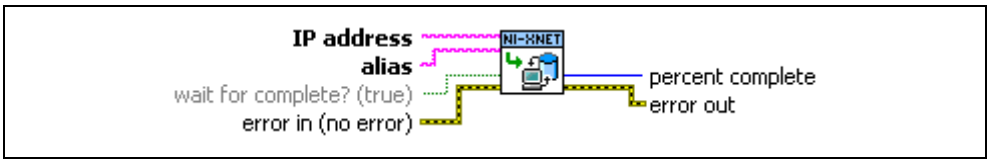
This VI is supported on Windows only. LabVIEW RT database deployments are managed remotely from Windows.

XNET Database Deploy.vi

Purpose

Deploys a database to a remote LabVIEW Real-Time (RT) target.

Format



Inputs

IP address is the target IP address.

alias provides the database alias name. To deploy a database text file, first add an alias using **XNET Database Add Alias.vi**.

wait for complete? determines whether the VI returns directly or waits until the entire transmission is completed.

error in is the error cluster input (refer to *Error Handling*).

Outputs

percent complete indicates the deployment progress.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI transfers an optimized binary image of the database to the LabVIEW RT target. After deploying the database, you can use the alias name in any VI for the Windows host and the LabVIEW RT target.

This VI is supported on Windows only. LabVIEW RT database deployments are managed remotely from Windows.

This VI must access the remote LabVIEW RT target from Windows, so **IP address** must specify a valid IP address for the LabVIEW RT target. You can find this IP address using MAX or VIs in the LabVIEW Real-Time palettes.

If the LabVIEW RT target access is password protected, use the following syntax for the IP address to deploy an alias: *[user:password@]IPaddress*.

Remote file transfer can take a few seconds, especially when the RT target is far away.

If **wait for complete?** is true, this VI waits for the entire transfer to complete, then returns. **error out** reflects the deployment status, and **percent complete** is 100.

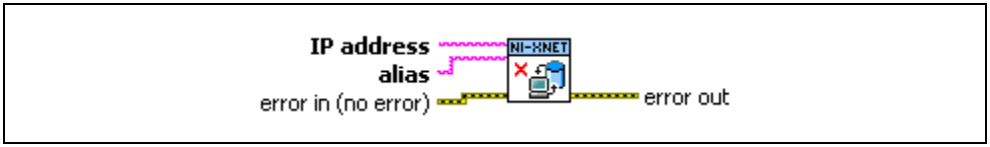
If **wait for complete?** is false, this VI transfers a portion of the database and returns before it is complete. For an incomplete transfer, **error out** returns success, and **percent complete** is less than 100. You can use **percent complete** to display transfer progress on your front panel. You must call **XNET Database Deploy.vi** in a loop until **percent complete** is returned as 100, at which time **error out** reflects the entire deployment status.

XNET Database Undeploy.vi

Purpose

Undeploys a database from a remote LabVIEW Real-Time (RT) target.

Format



Inputs

IP address is the target IP address.

alias provides the database alias name.

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

This VI completely deletes the database file and its alias from the LabVIEW RT target.

This VI is supported on Windows only. LabVIEW RT database deployments are managed remotely from Windows.

This VI must access the remote LabVIEW RT target from Windows, so **IP address** must specify a valid IP address for the LabVIEW RT target. You can find this IP address using MAX or VIs in the LabVIEW Real-Time palettes.

If the LabVIEW RT target access is password protected, you can use the following syntax for the IP address to deploy an alias: *[user:password@]IPaddress*.

XNET LIN Schedule Property Node

Format



Description

Property node used to read/write properties for an XNET LIN Schedule I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

Cluster

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET LIN Schedule

Short Name

Cluster

Description

This property returns the I/O name to the parent cluster in which the schedule has been created. You cannot change the parent cluster after creating the schedule object.

Comment

Data Type	Direction	Required?	Default
	Read/Write	No	Empty String

Property Class

XNET LIN Schedule

Short Name

Comment

Description

Comment describing the schedule object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET LIN Schedule

Short Name

ConfigStatus

Description

The LIN schedule object configuration status.

Configuration Status returns an NI-XNET error code. The value can be passed to the error code input of **Simple Error Handler.vi** to convert it to a text description (on message output) of the configuration problem.

By default, the XNET Cluster LIN:Schedules property does not return incorrect configured schedules in the database because you cannot use them in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When a schedule's configuration status becomes invalid after the database is opened, the XNET Cluster LIN:Schedules property still returns the schedule even if ShowInvalidFromOpen? is false.

An example of an invalid schedule configuration is when a required schedule property is not defined (for example, a schedule entry within this schedule has an undefined delay time).

Entries

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET LIN Schedule

Short Name

Entries

Description

Array of entries for this LIN schedule.

Each entry's position in this array specifies the position in the schedule. The database file and/or the order that you create entries at runtime determine the position.

Name (Short)

Data Type	Direction	Required?	Default
	Read/Write	Yes	Defined in Create Object

Property Class

XNET LIN Schedule

Short Name

NameShort

Description

String identifying the XNET LIN schedule object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A schedule name must be unique for all schedules in a cluster.

You can write this property to change the schedules's short name. When you do this and then use the original XNET LIN schedule that contains the old name, errors can result because the old name cannot be found. Follow these steps to avoid this problem:

1. Get the old Name (Short) property using the property node.
2. Set the new Name (Short) property for the object.
3. Wire the XNET LIN schedule as the input string to **Search and Replace String Function.vi** with the old **Name** as the search string and the new **Name** as the replace string. This replaces the short name in the XNET LIN schedule, while retaining the other text that ensures a unique name.
4. Wire the result from **Search and Replace String Function.vi** to the **XNET String to IO Name.vi**. This casts the string back to a valid XNET LIN schedule.

Priority

Data Type	Direction	Required?	Default
U32	Read/Write	No	42

Property Class

XNET LIN Schedule

Short Name

Priority

Description

Priority of this run-once LIN schedule when multiple run-once schedules are pending for execution.

The valid range for this property is 1–254. Lower values correspond to higher priority.

This property applies only when the Run Mode property is Once. Run-once schedule requests are queued for execution based on this property. When all run-once schedules have completed, the master returns to the previously running continuous schedule (or null).

Run-continuous schedule requests are not queued. Only the most recent run-continuous schedule is used, and it executes only if no run-once schedule is pending. Therefore, a run-continuous schedule has an effective priority of 255, but this property is not used.

Null schedule requests take effect immediately and supercede any running run-once or run-continuous schedule. The queue of pending run-once schedule requests is flushed (emptied without running them). Therefore, a null schedule has an effective priority of 0, but this property is not used.

This property is not read from the database, but is handled like a database property. After opening the database, the default value is returned, and you can change the property. But similar to database properties, you cannot change it after a session is created.

Run Mode

Data Type	Direction	Required?	Default
	Read/Write	No	See Description

Property Class

XNET LIN Schedule

Short Name

RunMode

Description

This property is a ring (enumerated list) with the following values:

String	Value
Continuous	0
Once	1
Null	2

This property specifies how the master runs this schedule:

- **Continuous**: The master runs the schedule continuously. When the last entry executes, the schedule starts again with the first entry.
- **Once**: The master runs the schedule once (all entries), then returns to the previously running continuous schedule (or null). If requests are submitted for multiple run-once schedules, each run-once executes in succession based on its Priority, then the master returns to the continuous schedule (or null).
- **Null**: All communication stops immediately. A schedule with this run mode is called a null schedule.

This property is not read from the database, but is handled like a database property. After opening the database, the default value is returned, and you can change the property. But similar to database properties, you cannot change it after a session is created.

Usually, the default value for the run mode is Continuous. If the schedule is configured to be a collision resolving table for an event-triggered entry, the default is Once.

XNET LIN Schedule Entry Property Node

Format



Description

Property node used to read/write properties for an XNET LIN Schedule Entry I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

Collision Resolving Schedule

Data Type	Direction	Required?	Default
	Read/Write	No	Empty I/O Name

Property Class

XNET LIN Schedule Entry

Short Name

CollResSched

Description

LIN schedule that resolves a collision for this event-triggered entry.

This property applies only when the entry Type is event triggered. When a collision occurs for the event-triggered entry in this schedule, the master must switch to the collision resolving schedule to transfer the unconditional frames successfully. If the XNET interface is acting as the master on the LIN cluster, NI-XNET automatically writes a schedule request for this collision resolving schedule.

The collision resolving schedule Run Mode must be Once.

When the entry type is any value other than event triggered, this property returns an empty entry (invalid).

Delay

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET LIN Schedule Entry

Short Name

Delay

Description

Time from the start of this entry (slot) to the start of the next entry.

The property uses a double value in seconds, with the fractional part used for milliseconds or microseconds.

Event Identifier

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET LIN Schedule Entry

Short Name

EventID

Description

The event-triggered entry identifier. This identifier is unprotected (NI-XNET handles the protection).

This property applies only when the entry type is event triggered. This identifier is for the event-triggered entry itself, and the first payload byte is for the protected identifier of the contained unconditional frame.

Frames

Data Type	Direction	Required?	Default
	Read/Write	No	Empty Array

Property Class

XNET LIN Schedule Entry

Short Name

Frames

Description

Array of frames for this LIN schedule entry.

If the entry Type is unconditional, this array contains one element, which is the single unconditional frame for this entry.

If the entry Type is sporadic, this array contains one or more frames for this entry. When multiple frames are pending for this entry, the order in the array determines the priority to transmit.

If the entry Type is event triggered, this array contains one or more frames for this entry. When multiple frames are pending for this entry, a collision typically occurs on the bus. When the XNET interface is acting as master, and a collision occurs, the master automatically writes a schedule request for the Collision Resolving Schedule. This resolves the collision automatically so that your application can proceed.

Name (Short)

Data Type	Direction	Required?	Default
	Read/Write	Yes	Defined in Create Object

Property Class

XNET LIN Schedule Entry

Short Name

NameShort

Description

String identifying the LIN schedule entry object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A schedule entry name must be unique for all entries in the same schedule.

You can write this property to change the schedule entry's short name. When you do this and then use the original XNET LIN schedule entry that contains the old name, errors can result because the old name cannot be found. Follow these steps to avoid this problem:

1. Get the old Name (Short) property using the property node.
2. Set the new Name (Short) property for the object.
3. Wire the XNET LIN schedule entry as the input string to **Search and Replace String Function.vi** with the old **Name** as the search string and the new **Name** as the replace string. This replaces the short name in the XNET LIN schedule entry, while retaining the other text that ensures a unique name.
4. Wire the result from **Search and Replace String Function.vi** to **XNET String to IO Name.vi**. This casts the string back to a valid XNET LIN schedule entry.

Node Configuration:Free Format:Data Bytes

Data Type	Direction	Required?	Default
	Read/Write	Yes	N/A

Property Class

XNET LIN Schedule Entry

Short Name

NodeConfFFDataBytes

Description

An array of 8 bytes containing raw data for LIN node configuration.

Node configuration defines a set of services used to configure slave nodes in the cluster. Every service has a specific set of parameters coded in this byte array. In the LDF file, those parameters are stored, for example, in the node (ECU) or the frame object. NI-XNET LDF reader composes those parameters to the byte values like they are sent on the bus. The LIN specification document describes the node configuration services and the mapping of the parameters to the raw format bytes.

The node configuration service is executed only if the Schedule Entry Type is set to Node Configuration.



Caution This property is not saved to the FIBEX file. If you write this property, save the database, and reopen it, the node configuration services are not contained in the database. Writing this property is useful only in the NI-XNET session immediately following.

Schedule

Data Type	Direction	Required?	Default
	Read Only	N/A	N/A

Property Class

XNET LIN Schedule Entry

Short Name

Schedule

Description

LIN schedule that uses this entry.

This LIN schedule is considered this entry's parent. You define the parent schedule when creating the entry object. You cannot change it afterwards.

Type

Data Type	Direction	Required?	Default
	Read/Write	No	Unconditional

Property Class

XNET LIN Schedule Entry

Short Name

Type

Description

The LIN schedule entry type determines the mechanism used to transfer frames in this entry (slot). The values (enumeration) for this property are:

0 **Unconditional**: A single frame transfers in this entry (slot).

1 **Sporadic**: The master transmits in this entry (slot). The master selects among multiple frames to transmit. Only updated frames are transmitted. When more than one frame has been updated, the master decides by priority which frame to transmit. The other updated frames remain pending and can be sent when this schedule entry executes again. The order of frames in the LIN Schedule Entry Frames property (the first frame has the highest priority) determines the frame priority.

2 **Event triggered**: Multiple slaves can transmit a frame in this entry (slot). Each slave transmits when the frame's data has been updated. When a collision occurs (multiple slaves try to transmit in the same slot), this is detected and resolved using a different schedule specified in the LIN Schedule Entry Collision Resolving Schedule property. The resolving schedule runs once, starting in the subsequent slot after the collision, and automatically turns back to the previous schedule at the position where the collision occurred.

3 **Node configuration**: The schedule entry contains a node configuration service. The node configuration service is defined as raw data bytes in the XNET LIN Schedule Entry Node Configuration:Free Format:Data Bytes property.

A LIN frame can exist in multiple schedules and multiple schedule entries. For example, if a frame exists in an event-triggered entry in schedule A, it also exists in an unconditional entry of a different schedule B, so that event-triggered collisions in schedule A can be resolved by switching to schedule B.

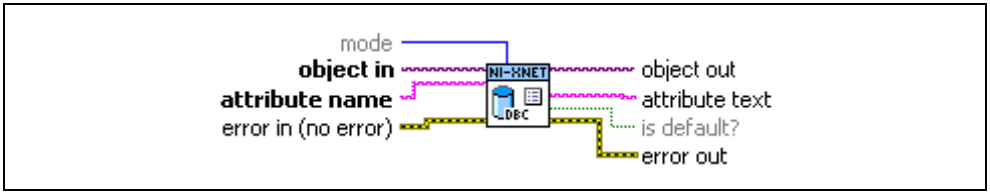
For information about how LIN frame timing compares to the Timing Type property of CAN and FlexRay frames, refer to *Cyclic and Event Timing*.

XNET Database Get DBC Attribute.vi

Purpose

Reads the attribute value, attribute enumeration, defined attributes, or signal value table from a DBC file.

Format



Inputs

U32

mode is the mode specification of this VI. Depending on this value, the VI returns the following data:

- **Mode 0: Get Attribute Value**: For a given object (for example, a signal), the VI returns the attribute value assigned to the object. The attribute values always are returned as text in **attribute text**. The DBC specification also allows defining other data types, such as integer or float. If necessary, you can convert the data to a number by using, for example, the **Scan From String** VI in the **String** palette. If the attribute is defined as an enumeration of text strings, the attribute value returned here is the index to the enumeration list, which you can retrieve using Mode 1 of the VI.
- **Mode 1: Get Enumeration**: For a given attribute name, the VI returns the enumeration text table as a comma-separated string in **attribute text**. Because for a given attribute name, the enumeration is the same for all objects of the same type, **object in** can point to any object with the given class (**object in** specifies the class). If no enumeration is defined for an attribute, the VI returns an empty string.
- **Mode 2: Get Attribute Name List**: Returns all attribute names defined for the given object type as a comma-separated string. **object in** can point to any object in the database of the given class (**object in** specifies the object class). **attribute name** is ignored (it should be set to empty string).

- **Mode 3: Get Signal Value Table**: This is valid only when **object in** points to a signal. **attribute name** is ignored (it should be set to empty string). If the given signal contains a value table, the function returns a comma-separated list in the form *[value,string]{,<value>,<string>}*. The list contains any number of corresponding *value,string* pairs. If no value table is defined for the signal, the result is an empty string.



object in is the database object (cluster, frame, signal, or ECU).



attribute name is the attribute name.



error in is the error cluster input (refer to *Error Handling*).

Outputs



object out is a copy of the **object in** parameter. You can use this output to wire the VI to subsequent VIs.



attribute text is the attribute value.



is default? indicates that a default value is used instead of a specific value for this object. DBC files define a default value for an attribute with the given name, and then specific values for particular objects. If the specific value for an object is not defined, the default value is returned. **is default?** has no meaning if the **mode** parameter is not 0 (refer to the **mode** description above).



error out is the error cluster output (refer to *Error Handling*).

Description

Depending on the **mode** parameter, this VI reads an attribute value, attribute enumeration, list of existing attributes, or value table of a signal from a DBC file. Refer to the **mode** input description above for details.

Attributes are supported for the following object types:

- Cluster (DBC file: network attribute)
- Frame (DBC file: message attribute)
- Signal (DBC file: signal attribute)
- ECU (DBC file: node attribute)

Databases other than DBC do not support attributes. Attributes are not saved to a FIBEX file when you open and save a DBC file.

Notify Subpalette

This subpalette includes functions for waiting on events from XNET hardware, including creation of a LabVIEW timing source.

XNET Wait.vi

Purpose

Waits for an event to occur.

Description

The instances of this polymorphic VI specify the event to wait for:

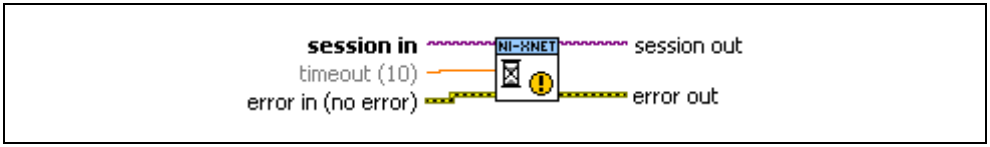
- **XNET Wait (Transmit Complete).vi**
- **XNET Wait (Interface Communicating).vi**
- **XNET Wait (CAN Remote Wakeup).vi**
- **XNET Wait (LIN Remote Wakeup).vi**

XNET Wait (Transmit Complete).vi

Purpose

Waits for previously written data to be transmitted on the cluster.

Format



Inputs



session in is the session to apply the wait.



timeout specifies the maximum amount of time in seconds to wait.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

Waits for all data provided to **XNET Write.vi** before this **XNET Wait.vi** call to be transmitted on the CAN, FlexRay, or LIN network. Depending on the bus or configuration properties such as Interface:CAN:Single Shot Transmit?, the data may or may not have been successfully transmitted; however, if this wait returns successfully, it indicates that the session is making no more attempts to transmit the data. This wait applies to only the current XNET session, and not other sessions used for the same interface.

After using **XNET Write.vi** to provide data for this session, you can use this VI to wait for that data to transmit to remote ECUs. You can use this VI to guarantee that all frames have been transmitted before stopping this session.

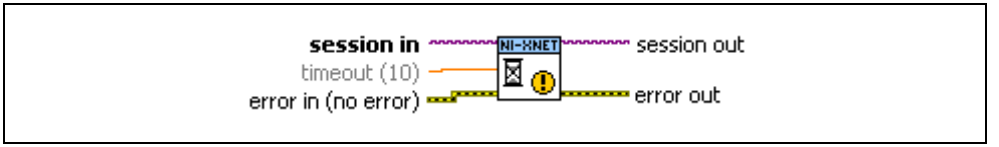
The **timeout** parameter provides the maximum number of seconds to wait. The default value is 10 (10 seconds).

XNET Wait (Interface Communicating).vi

Purpose

Waits for the interface to begin communication on the cluster.

Format



Inputs

session in is the session to apply the wait.

timeout specifies the maximum amount of time in seconds to wait.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

error out is the error cluster output (refer to *Error Handling*).

Description

Waits for the interface to begin communication on the cluster. After the interface is started, the controller connects to the cluster and starts communication. This wait returns after communication with the cluster has been established.

Note For some buses (for example, CAN), the communication may occur within a few microseconds of starting the interface. For other buses, this could be delayed. An example of a bus where the communication time is delayed from the start time is FlexRay, where the interface must perform a startup routine that may take several cycles to complete. A FlexRay interface attempts integration with the remaining nodes in the cluster when it is started. If the FlexRay interface can coldstart, it sends out startup frames when started and synchronizes its clock with other startup nodes in the cluster. Once the FlexRay interface has successfully integrated, the interface is ready to start transmitting and receiving frames. Reading the XNET FlexRay interface Protocol Operation Control (POC) state (refer to the **XNET Read (State FlexRay Comm).vi** description), once the interface has successfully integrated, returns Normal-Active.

Note If a start trigger is configured for the interface, the interface start occurs after the start trigger is received.

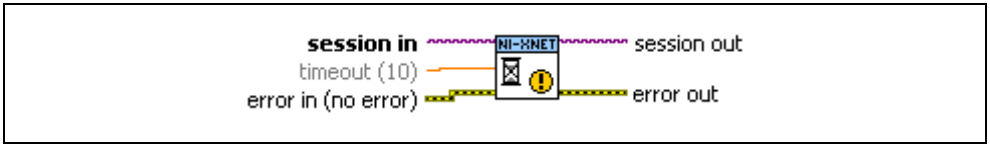
The timeout parameter provides the maximum number of seconds to wait. The default value is 10 (10 seconds).

XNET Wait (CAN Remote Wakeup).vi

Purpose

Waits for the CAN interface to wake up due to activity by a remote ECU on the network.

Format



Inputs



session in is the session to apply the wait.



timeout specifies the maximum amount of time in seconds to wait.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

This wait is used when you set the XNET Session Interface:CAN:Transceiver State property to Sleep. When asleep, the interface and transceiver go into a low-powered mode. If a remote CAN ECU transmits a frame, the transceiver detects this transmission, and both the controller and transceiver wake up. This wait detects that remote wakeup.

Note The interface neither receives nor acknowledges the transmission that caused the wakeup. However, after the interface wakes up, the transceiver automatically is placed into normal mode, and communication is restored.

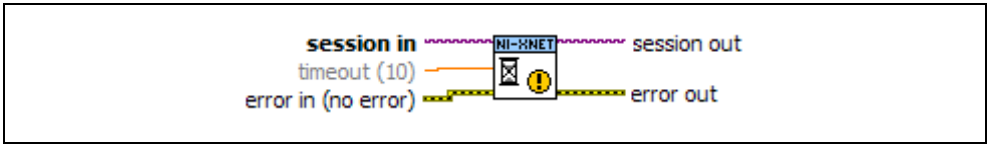
The **timeout** parameter provides the maximum number of seconds to wait. This value must be 1.0 (one second) or greater. The default value is 10 (10 seconds).

XNET Wait (LIN Remote Wakeup).vi

Purpose

Waits for the LIN interface to wake up due to activity by a remote ECU on the network.

Format



Inputs



session in is the session to apply the wait. The wait applies to the LIN interface, so you can use any session.



timeout specifies the maximum amount of time in seconds to wait.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

This wait is used when you set the XNET Session Interface:LIN:Sleep property to **Remote Sleep** or **Local Sleep**. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

The **timeout** parameter provides the maximum number of seconds to wait. This value must be 1.0 (one second) or greater. The default value is 10 (10 seconds).

XNET Create Timing Source.vi

Purpose

Creates a timing source for a LabVIEW Timed Loop.

Description

The instances of this polymorphic VI specify the timing source to create:

- **XNET Create Timing Source (FlexRay Cycle).vi**

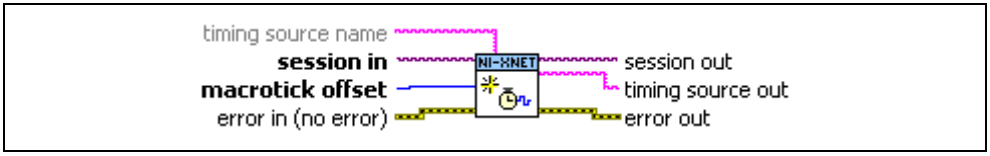
XNET Create Timing Source (FlexRay Cycle).vi

Purpose

Creates a timing source for a LabVIEW Timed Loop.

The timing source is based on the FlexRay communication cycle. The timing source sends a tick to the Timed Loop at a specific offset in time within the FlexRay cycle. The offset within the cycle is specified in FlexRay macroticks.

Format



Inputs

timing source name is the timing source name, returned as timing source out if this VI succeeds.

This input is optional. If you leave **timing source name** unwired (empty), **timing source out** uses the session name (session in).

session in is the session to use for creating the timing source.

You must configure the session to use a FlexRay interface, because the timing source is based on that interface's communication cycle. You can create only one FlexRay cycle timing source for each interface.

This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.



macrotick offset is the offset within each FlexRay cycle that you want the timing source to tick.

The minimum value is zero (0), which specifies a tick at the start of every FlexRay cycle. The value cannot be equal to or greater than the number of macroticks in the cycle, which you can read from the XNET Cluster the session uses, from the FlexRay:Macro Per Cycle property.

For further recommendations about selecting a value, refer to *Macrotick Offset*.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



timing source out is the timing source name. You wire this name to the **Source Name** of the input node outside the Timed Loop.

For more information about the Timed Loop nodes, refer to *Using the Timed Loop*.

If this VI returns an error (**status** true in **error out**), **timing source out** is empty, which indicates to the Timed Loop that no valid timing source exists.



error out is the error cluster output (refer to *Error Handling*).

Description

Use this VI to synchronize your LabVIEW Real-Time application to the deterministic FlexRay cycle. Because the FlexRay cycle repeats every few milliseconds, real-time execution is required, and therefore this VI is not supported on Windows.

You can create only one FlexRay Cycle timing source for each FlexRay interface. You can wire a single timing source to multiple Timed Loops.

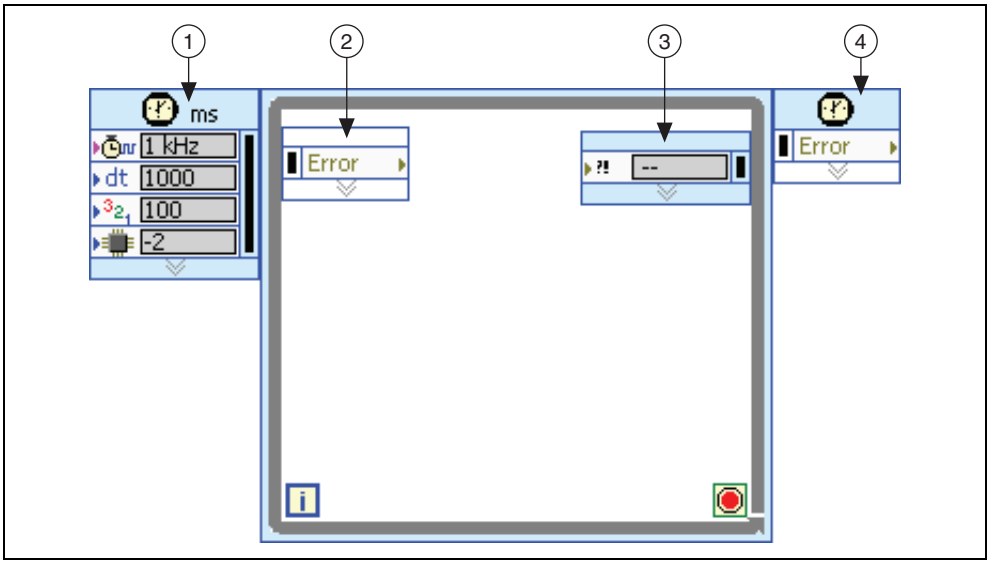
The following sections include more detailed information about using this VI:

- *Using the Timed Loop*
- *Session Start and Stop*
- *Macrotick Offset*

Using the Timed Loop

This section includes guidelines for using the LabVIEW Timed Loop with the NI-XNET FlexRay Cycle timing source. For complete information, refer to the LabVIEW help topics for the Timed Loop.

The Timed Loop contains the nodes described below.



1	Input Node	3	Right Data Node
2	Left Data Node	4	Output Node

Figure 4-11. Timed Loop Nodes

Input Node

Source Name: Wire the **timing source name** output of this VI to this terminal on the Timed Loop input node. This specifies the XNET timing source and overrides the default built-in timing source (1 kHz).

Period: For most applications, you wire the constant 1 to this terminal, which overrides the default of 1000. The **Period** specifies the number of timing source ticks that must occur for the loop to iterate. A value of 1 iterates the Timed Loop on every FlexRay cycle. Higher values skip FlexRay cycles (for example, 2 iterates the loop every other FlexRay cycle).

Timeout: For most applications, you wire the constant 300 to this terminal, which overrides the default of –1. The **Timeout** specifies the maximum number of milliseconds to wait for a tick. For this FlexRay cycle timing source, this timeout primarily applies to the first loop iteration. According to the FlexRay specification, the process of fully synchronizing the

distributed network clocks can take as long as 200 ms. This network clock synchronization is required for the NI-XNET interface to detect the first FlexRay cycle and send a tick to the Timed Loop. If network communication problems occur (for example, noise on the cable), the first tick does not occur. Using a value of 300 for this terminal ensures that if problems occur on the FlexRay network, the Timed Loop can recover (refer to *Wake-Up Reason* in *Left Data Node*).

Error: Use this terminal to propagate errors through the Timed Loop. The Timed Loop does not execute if this terminal receives an error condition. You typically wire the **error out** from this **XNET Create Timing Source (FlexRay Cycle).vi** to this terminal. This avoids the need for alternate error propagation techniques, such as a shift register.

Left Data Node

Error: Propagates errors through the structure. Wire this terminal to error in of the first VI within the subdiagram.

Wake-Up Reason: If the first Timed Loop iteration encounters a **Timeout** due to problems on the FlexRay network, this terminal returns a value of 5 (**Timeout**). When the timeout occurs, the Timed Loop does not return an error condition from **Error**. The timeout causes the iteration to execute untimed, then try again on the next iteration. If the FlexRay tick occurs as expected, Wake-Up Reason returns a value of 0 (Normal).

Right Data Node

Error: Propagates errors from the subdiagram out of the Timed Loop. If **Error** receives an error condition, the Timed Loop finishes executing the current iteration untimed, exits the loop, and returns the error condition on the Output Node. If you want the Timed Loop to exit on error, wire **error out** from the last VI in the subdiagram to this terminal.

Output Node

Error: Propagates errors the Timed Loop receives and returns errors from the subdiagram.

Session Start and Stop

When the Timed Loop input node executes, the XNET session for the timing source is started automatically. This auto-start is equivalent to calling **XNET Start.vi** (Normal). This auto-start is performed even if the session's Auto Start? property is false. Because the Timed Loop uses an execution priority that typically is higher than the VIs that precede it, starting FlexRay communication within the Timed Loop ensures that you do not miss the first FlexRay cycle. Due to these factors, do not call **XNET Start.vi** prior to the Timed Loop (use the Timed Loop auto-start instead).

After the initial session and interface auto-start, the Timed Loop **Timeout** is used to wait for communication to begin.

When the Timed Loop exits to its output node, the XNET session remains in its current state. The Timed Loop does not stop or clear the session, so you can continue to use the session in VIs that follow.

Macrotick Offset

To set the **macrotick offset**, it helps to understand some NI-XNET implementation aspects. When the FlexRay Communication Controller (CC) receives a frame, the NI-XNET hardware immediately transfers that frame to LabVIEW Real-Time (RT). This transfer is performed using DMA (Direct Memory Access) on the PXI backplane, so that it occurs quickly and with negligible jitter to your LabVIEW RT execution.

Figure 4-12 shows the effects of this implementation. In this example, the **macrotick offset** is set to occur at the end of slot 1. The subdiagram in the Timed Loop calls **XNET Read.vi** to read the value received from slot 1.

For better visibility in Figures 4-12, 4-13, and 4-14, the NI-XNET blocks (Read/Write, DMA I/O, an dCC I/O) are longer than actual performance. When using a PXI controller for LabVIEW Real-Time, your results typically will be faster. This is especially true if your application does not transfer data on the PXI backplane continuously (for example, streaming analog, vision, or TCP/IP data), as this sort of transfer can adversely impact the NI-XNET DMA latencies.

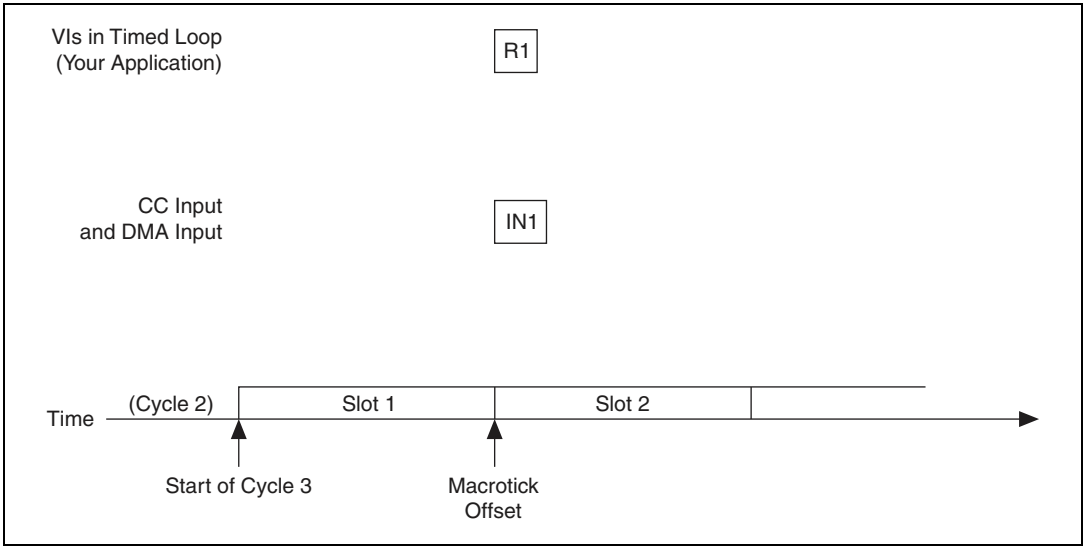


Figure 4-12. FlexRay Frame Timed Read

Figure 4-12 shows that the DMA input transfer for slot 1 (IN1) occurs at the same time as **XNET Read.vi** for slot 1 (R1). Depending on which one completes first, **XNET Read.vi** may return a value from the current cycle (3) or the previous cycle (2).

To prevent this uncertainty, **macrotick offset** must be large enough to ensure that the frame DMA input is complete. Relative to Figure 4-12, setting **macrotick offset** to the end of slot 2 would suffice.

When your LabVIEW RT application calls **XNET Write.vi**, the frame values are transferred immediately using DMA. The frame values are transferred to the NI-XNET hardware onboard processor memory. For efficiency reasons, this onboard processor waits until the FlexRay cycle Network Idle Time (NIT) to transfer the frame values from its memory to the FlexRay Communication Controller (CC). The FlexRay Communication Controller transmits each frame value according to its slot configuration in the cycle.

Figure 4-13 shows the effects of this implementation. This example expands on Figure 4-12 by calling **XNET Write.vi** with a value for slot 8. **XNET Write.vi** (W8) is called well in advance of slot 8 in the cycle. The DMA output transfer for the value of slot 8 (D8) occurs immediately after **XNET Write.vi**. Nevertheless, the value for slot 8 is not placed into the FlexRay Communication Controller until the NIT time, shown as C8. This means that although **XNET Write.vi** was called before slot 8's occurrence in the current cycle 3, that value does not transmit until the subsequent cycle 4.

This implementation for output means that it is not necessarily urgent to call **XNET Write.vi** before the relevant slot. You merely need to provide time for **XNET Write.vi** and the related DMA output to complete prior to the NIT.

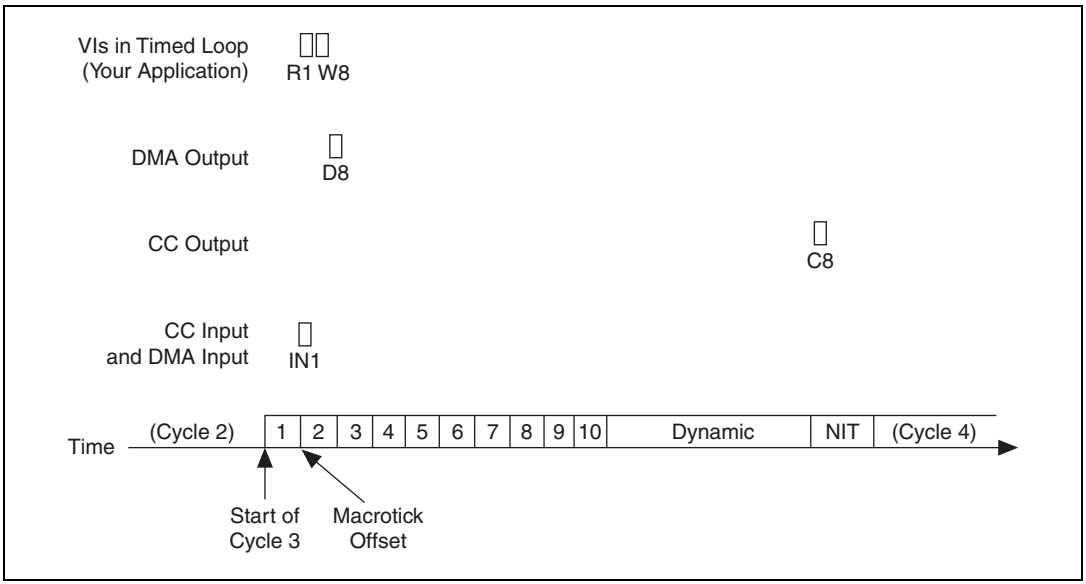


Figure 4-13. FlexRay Frame Timed Write

Taking these implementation considerations into account, the typical **macrotick offset** goal is a value that executes the Timed Loop after the last cycle input DMA and prior to the NIT. Ideally, the **macrotick offset** provides sufficient time for input DMA, **XNET Read.vi**, LabVIEW code within the Timed Loop (for example, a simulation model), **XNET Write.vi**, and DMA output.

To find a value for **macrotick offset**, you can use the XNET Cluster property node. The FlexRay:NIT Start property provides the **macrotick offset** for the start of NIT, which is your upper limit. To determine the lower limit, the FlexRay:Static Slot property provides the number of macroticks for each static slot. Static slot numbers begin at 1. Assuming static slot *X* is the last slot that you read, the lower limit for **macrotick offset** is (*X* × FlexRay:Static Slot).

The following example demonstrates a technique for calculating **macrotick offset**. The example uses a simple FlexRay cluster configured as follows:

- **Baud Rate**—5000000 bps (5 Mbps)
- **Macrotick**—1 (1 µs duration)
- **Macro Per Cycle**—1000 (1 ms)
- **Number of Static Slots**—10
- **Number of Minislots**—80
- **Static Slot**—58 MT (16 byte payload)

- **NIT Start**—900 MT offset
- **NIT**—100 MT (duration)

Within the Timed Loop, the example does the following:

- Reads a Signal Input Single-Point session for frames in static slots 2, 3, and 4.
- Executes a simulation model (passes in inputs and obtains outputs).
- Writes a Signal Output Single-Point session for frames in static slots 8, 9, and 10.

Assume that you test the simulation model performance and determine that it takes 100 µs (including jitter). Using the cluster configuration and the time required for the simulation model, select a macrotick offset that locates the simulation model at the midpoint between the end of slot 4 (the last input frame) and the start of NIT. This provides the maximum time possible for **XNET Read.vi**/**XNET Write.vi**, DMA input/output, and CC input/output.

EndOfSlot4	= (4 × Static_Slot)
	= (4 × 58)
	= 232
Midpoint	= EndOfSlot4 + ((NIT_Start – EndOfSlot4) / 2)
	= 232 + ((900 – 232) / 2)
	= 232 + 334
	= 566
macrotick offset	= (Midpoint – (SimModelTime / 2))
	= (566 – (100 / 2))
	= 516

Figure 4-14 shows the Timed Loop timing diagram. Notice that the simulation model is synchronized deterministically with the FlexRay cycle. The Timed Loop code reads inputs from the current cycle, calculates outputs, and then writes the output for the next cycle.

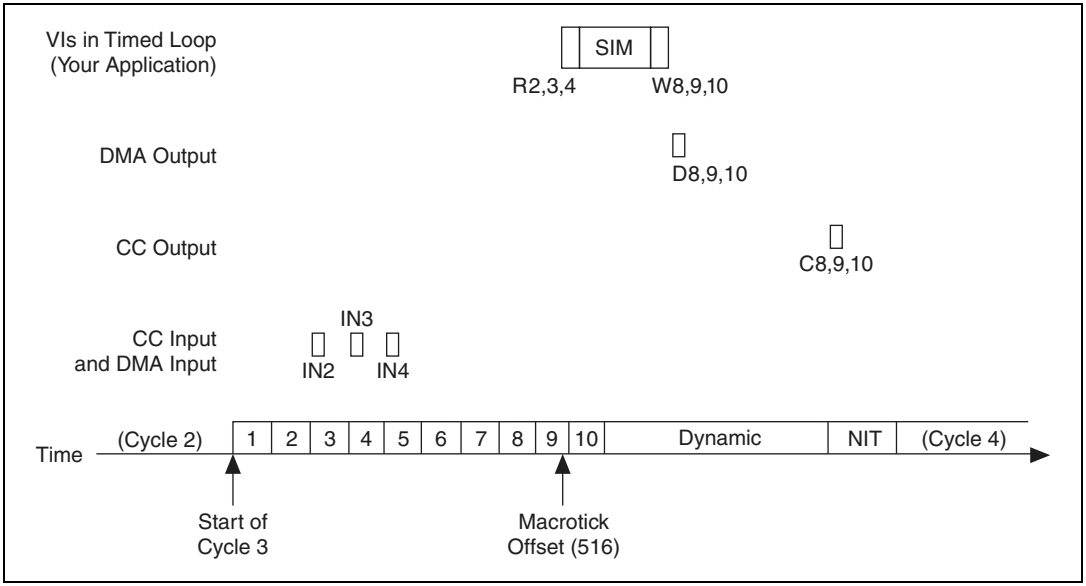


Figure 4-14. Timing Source Example

Reading from the FlexRay Communication Controller (and performing the corresponding DMA) for frames 2, 3, and 4 is shown as blocks IN2, IN3, and IN4. **XNET Read.vi** for frames 2, 3, and 4 is shown as block R2,3,4. The simulation model execution is shown as block SIM. The start of SIM is halfway between the end of slot 4 and the start of NIT. **XNET Write.vi** for frames 8, 9, and 10 is shown as block W8,9,10. The corresponding DMA output for these frames is shown as block D8,9,10. The FlexRay Communication Controller update during the NIT is shown as block C8,9,10.

As with any performance-sensitive configuration, you should measure using your own hardware and application to calculate the best **macrotick offset** value. To determine the current cycle and macrotick within the Timed Loop for measurement purposes, use **XNET Read (State FlexRay Cycle Macrotick).vi**.

Advanced Subpalette

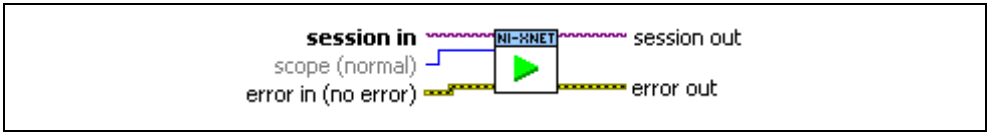
This subpalette includes advanced functions for controlling the state of NI-XNET sessions, connecting hardware terminals, and retrieving information about the XNET hardware in your system.

XNET Start.vi

Purpose

Starts communication for the specified XNET session.

Format



Inputs



session in is the session to start. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.



scope describes the impact of this operation on the underlying state models for the session and its interface.

Normal (0) The session is started followed by starting the interface. This is equivalent to calling **XNET Start.vi** with the **Session Only** scope followed by calling **XNET Start.vi** with the **Interface Only** scope.

This is the default value for **scope** if it is unwired.

Session Only (1) The session is placed into the Started state (refer to *State Models*). If the interface is in the Stopped state before this VI runs, the interface remains in the Stopped state, and no communication occurs with the bus. To have multiple sessions start at exactly the same time, start each session with the **Session Only** scope. When you are ready for all sessions to start communicating on the associated interface, call **XNET Start.vi** with the **Interface Only** scope. Starting a previously started session is considered a no-op. This operation sends the command to start the

session, but does not wait for the session to be started. It is ideal for a real-time application where performance is critical.

Interface Only (2) If the underlying interface is not previously started, the interface is placed into the Started state (refer to *State Models*). After the interface starts communicating, all previously started sessions can transfer data to and from the bus. Starting a previously started interface is considered a no-op.

Session Only Blocking (3) The session is placed into the Started state (refer to *State Models*). If the interface is in the Stopped state before this VI runs, the interface remains in the Stopped state, and no communication occurs with the bus. To have multiple sessions start at exactly the same time, start each session with the Session Only scope. When you are ready for all sessions to start communicating on the associated interface, call **XNET Start.vi** with the Interface Only scope. Starting a previously started session is considered a no-op. This operation waits for the session to start before completing.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

Because the session is started automatically by default, this VI is optional. This VI is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the Auto Start? property.

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

- **Sessions**: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.
- **Interface**: The interface physically connects to the bus and transmits (or receives) data for the sessions.

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

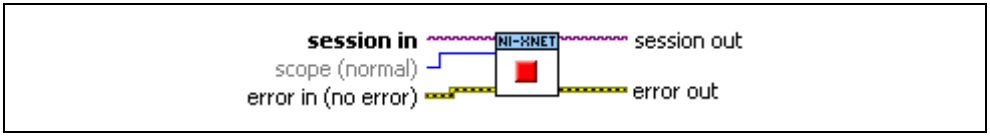
If an output session starts before you write data, or you read an input session before it receives a frame, default data is used. For more information, refer to the XNET Frame Default Payload and XNET Signal Default Value properties.

XNET Stop.vi

Purpose

Stops communication for the specified XNET session.

Format



Inputs

I/O

session in is the session to stop. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.

U16

scope describes the impact of this operation on the underlying state models for the session and its interface.

Normal (0) The session is stopped. If this is the last session stopped on the interface, the interface is also stopped. If any other sessions are running on the interface, this call is treated just like the **Session Only** scope, to avoid disruption of communication on the other sessions.

This is the default value for **scope** if it is unwired.

Session Only (1) The session is placed in the Stopped state (refer to *State Models*). If the interface was in the Started or Running state before this VI is called, the interface remains in that state and communication continues, but data from this session does not transfer. This scope generally is not necessary, as the **Normal** scope only stops the interface if there are no other running sessions. This operation sends the command to stop the session, but does not wait for the session to be stopped. It is ideal for a real-time application where performance is critical.

Interface Only (2) The underlying interface is placed in the Stopped state (refer to *State Models*). This prevents all communication on the bus, for all sessions. This allows you to modify certain properties that require the interface to be stopped (for example, CAN baud rate).

All sessions remain in the Started state. To have multiple sessions stop at exactly the same time, first stop the interface with the **Interface Only** scope and then stop each session with either the Normal or **Session Only** scope.

Session Only Blocking (3) The session is placed in the Stopped state (refer to *State Models*). If the interface was in the Started or Running state before this VI is called, the interface remains in that state and communication continues, but data from this session does not transfer. This scope generally is not necessary, as the Normal scope stops the interface only if there are no other running sessions. This operation waits for the session to stop before completing.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, for use with subsequent VIs.



error out is the error cluster output (refer to *Error Handling*).

Description

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

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

- **Sessions**: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.
- **Interface**: The interface physically connects to the bus and transmits (or receives) data for the sessions.

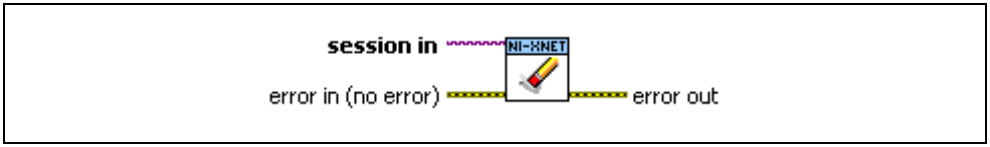
You can stop each logical unit separately. When a session is stopped, all contained frames or signals are placed in a state where they are no longer ready to communicate. When the interface is stopped, it no longer takes data from sessions to communicate with other nodes on the bus. For a specification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

XNET Clear.vi

Purpose

Clears (closes) the XNET session.

Format



Inputs



session in is the session to clear. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.



error in is the error cluster input (refer to *Error Handling*).

Outputs



error out is the error cluster output (refer to *Error Handling*).

Description

This VI stops communication for the session and releases all resources the session uses. **XNET Clear.vi** internally calls **XNET Stop.vi** with normal scope, so if this is the last session using the interface, communication stops.

When your application is finished (the top-level VI is idle), LabVIEW automatically clears all XNET sessions within that VI and its subVIs. Therefore, **XNET Clear.vi** is rarely needed in your application.

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

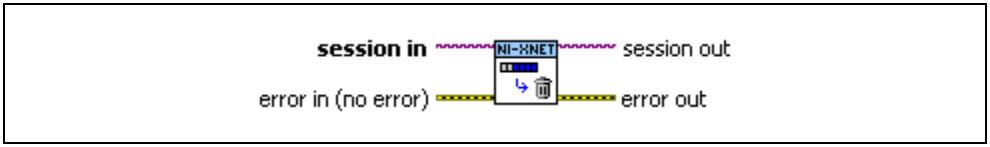
This VI disconnects terminals that you connected using **XNET Connect Terminals.vi**.

XNET Flush.vi

Purpose

Flushes (empties) all XNET session queues.

Format



Inputs

session in is the session to flush. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

error out is the error cluster output (refer to *Error Handling*).

Description

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

XNET Start.vi and **XNET Stop.vi** have no effect on these queues. Use **XNET Flush.vi** to discard all values in the session's queues.

For example, if you call **XNET Write.vi** to write three frames, then immediately call **XNET Stop.vi**, then call **XNET Start.vi** a few seconds later, the three frames transmit. If you call **XNET Flush.vi** between **XNET Stop.vi** and **XNET Start.vi**, no frames transmit.

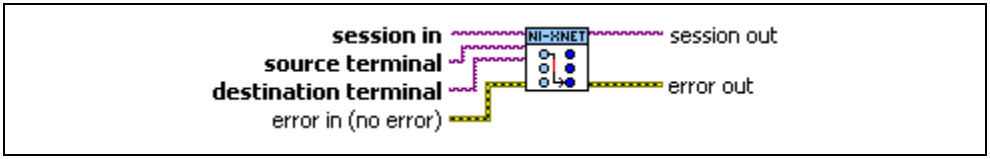
As another example, if you receive three frames, then call **XNET Stop.vi**, the three frames remains in the queue. If you call **XNET Start.vi** a few seconds later, then call **XNET Read.vi**, you obtain the three frames received earlier, potentially followed by other frames received after calling **XNET Start.vi**. If you call **XNET Flush.vi** between **XNET Stop.vi** and **XNET Start.vi**, **XNET Read.vi** returns only frames received after the calling **XNET Start.vi**.

XNET Connect Terminals.vi

Purpose

Connects terminals on the XNET interface.

Format



Inputs

session in is the session to use for the connection. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.

source terminal is the connection source.

destination terminal is the connection destination.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is a duplicate of the session in, provided for simpler wiring.

error out is the error cluster output (refer to *Error Handling*).

Description

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

The terminal inputs use the XNET Terminal I/O name, so you can select from possible values using the drop-down list. Typically, one of the pair is an internal and the other an external.

Valid Combinations of Source/Destination

The following table lists all valid combinations of **source terminal** and **destination terminal**.

Source	Destination				
	PXI_Trig*x*	**FrontPanel0 FrontPanel1**	**Start Trigger**	**Master Timebase**	**Log Trigger**
PXI_Trig*x*	X	X	✓	✓	✓
FrontPanel0 FrontPanel1	X	X	✓	✓	✓
PXI_Star[1]	X	X	✓	X	X
PXI_Clk10[1]	X	X	X	✓	X
StartTrigger	✓	✓	X	X	X
CommTrigger	✓	✓	X	X	X
FlexRayStartCycle[2]	✓	✓	X	X	X
FlexRayMacrotick[2]	✓	✓	X	✓	X
1MHzTimebase	✓	✓	X	X	X
10MHzTimebase	✓	X	X	X	X

[1] Valid only on PXI hardware.

[2] Valid only on FlexRay hardware.

Source Terminals

The following table describes the valid source terminals.

Source Terminal	Description
PXI_Trig*x*	Selects a general-purpose trigger line as the connection source (input), where x is a number from 0 to 7. For PCI cards, these are the RTSI lines. For PXI cards, these are the PXI Trigger lines. For C Series modules in a CompactDAQ chassis, all modules in the chassis automatically share a common timebase. For information about routing the StartTrigger for CompactDAQ, refer to the XNET Session Interface:Source Terminal:Start Trigger property.
FrontPanel0 FrontPanel1	Selects a general-purpose Front Panel Trigger line as the connection source (input).
PXI_Star	Selects the PXI star trigger signal. Within a PXI chassis, some PXI products can source star trigger from Slot 2 to all higher-numbered slots. PXI_Star enables the PXI XNET hardware to receive the star trigger when it is in Slot 3 or higher. **Note**: You cannot use this terminal with a PCI device.
PXI_Clk10	Selects the PXI 10 MHz backplane clock. The only valid **destination terminal** for this source is MasterTimebase. This routes the 10 MHz PXI backplane clock for use as the XNET card timebase. When you use PXI_Clk10 as the XNET card timebase, you also must use PXI_Clk10 as the timebase for other PXI cards to perform synchronized input/output. **Note**: You cannot use this terminal with a PCI device.
StartTrigger	Selects the start trigger, which is the event set when the when the Start Interface transition occurs. The start trigger is the same for all sessions using a given interface. You can route the start trigger of this XNET card to the start trigger of other XNET or DAQ cards to ensure that sampling begins at the same time on both cards. For example, you can synchronize two XNET cards by routing StartTrigger as the **source terminal** on one XNET card and then routing StartTrigger as the **destination terminal** on the other XNET card, with both cards using the same RTSI line for the connections.

Source Terminal	Description
CommTrigger	Selects the communicating trigger, which is the event set when the Comm State Communicating transition occurs. The communicating trigger is the same for all sessions using a given interface. You can route the communicating trigger of this XNET card to the start trigger of other XNET or DAQ cards to ensure that sampling begins at the same time on both cards. The communicating trigger is similar to a start trigger, but is used if your clock source is the FlexRayMacrotick, which is not available until the interface is properly integrated into the bus. Because you cannot generate a start trigger to another interface until the synchronization clock is also available, you can use this trigger to allow for the clock under this special circumstance.
FlexRayStartCycle	Selects the FlexRay Start of Cycles as an advanced trigger source. This generates a repeating pulse that external hardware can use to synchronize a measurement or other action with each FlexRay cycle. **Note**: You can use this terminal only with a FlexRay device.
FlexRayMacrotick	Selects the FlexRay Macrotick as a timing source. The FlexRay Macrotick is the basic unit of time in a FlexRay network. You can use this **source terminal** to synchronize other measurements to the actual time on the FlexRay bus. In this scenario, you would configure the FlexRayMacrotick as the **source terminal** and route it to a RTSI or front panel terminal. After the interface is communicating to the FlexRay network, the Macrotick signal becomes available. You also can connect the FlexRayMacrotick to the MasterTimebase. This configures the counter that timestamps received frames to run synchronized to FlexRay time, and also allows you to read the FlexRay cycle macrotick to do additional synchronization with the FlexRay bus in your application. **Note**: You can use this terminal only with a FlexRay device.

Source Terminal	Description
1MHzTimebase	Selects the XNET card's local 1 MHz oscillator. The only valid **destination terminals** for this source are PXI Trigger0–PXI Trigger7. This **source terminal** routes the XNET card local 1 MHz clock so that other NI cards can use it as a timebase. For example, you can synchronize two XNET cards by connecting 1MHzTimebase to PXI_Trig*x* on one XNET card and then connecting PXI_Trig*x* to MasterTimebase on the other XNET card.
10MHzTimebase	Selects the XNET card's local 10 MHz oscillator. This routes the XNET card local 10 MHz clock for use as a timebase by other NI cards. For example, you can synchronize two XNET cards by connecting 10MHzTimebase to PXI_Trig*x* on one XNET card and then connecting PXI_Trig*x* to MasterTimebase on the other XNET card.

Destination Terminals

The following table describes the valid **destination terminals**.

Destination Terminal	Description
PXI_Trig*x*	Selects a general-purpose trigger line as the connection destination (output), where x is a number from 0 to 7. For PCI cards, these are the RTSI lines. For PXI cards, these are the PXI Trigger lines. For C Series modules in a CompactDAQ chassis, all modules in the chassis automatically share a common timebase. For information about routing the StartTrigger for CompactDAQ, refer to the XNET Session Interface:Source Terminal:Start Trigger property.
FrontPanel0 FrontPanel1	Selects a general-purpose Front Panel Trigger line as the connection destination (output).

Destination Terminal	Description
StartTrigger	Selects the start trigger, which is the event that allows the interface to begin communication. The start trigger occurs on the first **source terminal** low-to-high transition. The start trigger is the same for all sessions using a given interface. This causes the Start Interface transition to occur. You can route the start trigger of another XNET or DAQ card to ensure that sampling begins at the same time on both cards. For example, you can synchronize with an M-Series DAQ MIO card by routing the AI start trigger of the MIO card to a RTSI line and then routing the same RTSI line with StartTrigger as the **destination terminal** on the XNET card. The default (disconnected) state of this destination means the start trigger occurs when XNET Start.vi is invoked with the scope set to either Normal or Interface Only. Alternately, if Auto Start? is enabled, reading or writing to a session may start the interface.
MasterTimebase	MasterTimebase instructs the XNET card to use the connection **source terminal** as the master timebase. The XNET card uses this master timebase for input sampling (including timestamps of received messages) as well as periodic output sampling. Your XNET hardware supports incoming frequencies of 1 MHz, 10 MHz, and 20 MHz, and automatically detects the frequency without any additional configuration. For example, you can synchronize a CAN and DAQ M Series MIO card by connecting the 10 MHz oscillator (board clock) of the DAQ card to a PXI_Trig line, and then connecting the same PXI_Trig line as the **source terminal**. For PXI form factor hardware, you also can use PXI_Clk10 as the **source terminal**. This receives the PXI 10 MHz backplane clock for use as the master timebase. MasterTimebase applies separately to each port of a multiport XNET card, meaning you could run each port off of a separate incoming (or onboard) timebase signal. If you are using a PCI board, the default connection to the Master Timebase is the local oscillator. If you are using a PXI board, the default connection to the MasterTimebase is the PXI_Clk10 signal, if it is available. Some chassis allow PXI_Clk10 to be turned off. In this case, the hardware automatically uses the local oscillator as the default MasterTimebase.

Destination Terminal	Description
Log Trigger	The Log Trigger terminal generates a frame when it detects a rising edge. When connected, this frame is transferred into the Frame Stream Input session's queue if the session is started. For information about this frame, including the frame payload interpretation, refer to *Special Frames*.

XNET Disconnect Terminals.vi

Purpose

Disconnects terminals on the XNET interface.

Format



Inputs

session in is the session to use for the connection. This session is selected from the LabVIEW project or returned from **XNET Create Session.vi**.

source terminal is the connection source.

destination terminal is the connection destination.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is a duplicate of the session in, provided for simpler wiring.

error out is the error cluster output (refer to *Error Handling*).

Description

This VI disconnects a specific pair of source/destination terminals previously connected with **XNET Connect Terminals.vi**.

When the final session for a given interface is cleared (either by the VI going idle or by explicit calls to **XNET Clear.vi**), NI-XNET automatically disconnects all terminal connections for that interface. Therefore, **XNET Disconnect Terminals.vi** is not required for most applications.

This VI typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using **XNET Stop.vi** with the Interface Only scope. Then you can call **XNET Disconnect Terminals.vi** and **XNET Connect Terminals.vi** to adjust terminal connections. Finally, you can call **XNET Start.vi** with the Interface Only scope to restart the interface.

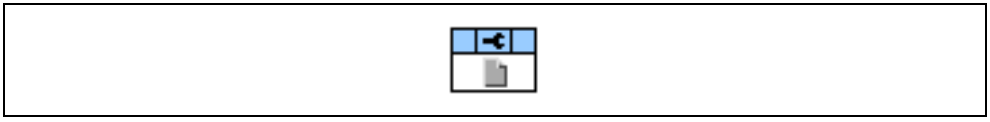
You can disconnect only a terminal that has been previously connected. Attempting to disconnect a nonconnected terminal results in an error.

XNET Terminal Constant

This constant provides the constant form of the XNET Terminal I/O name. You drag a constant to the block diagram of your VI, then select a terminal. You can change constants only during configuration, prior to running the VI. For a complete description, refer to *XNET Terminal I/O Name*.

XNET System Property Node

Format



Description

The XNET System property node provides information about all NI-XNET hardware in your system, including all devices and interfaces.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

Devices

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET System

Short Name

Devices

Description

Returns an array of physical XNET devices in the system. Each physical XNET board is a hardware product such as a PCI/PXI board.

The system refers to the execution target of this property node. If this property is run on an RT target, it reports the RT system hardware.

You can wire the XNET Device I/O name to the XNET Device property node to access properties of the device.

Interfaces (FlexRay)

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET System

Short Name

IntfFlexRay

Description

Returns an array of all available interfaces on the system that support the FlexRay protocol.

The system refers to the execution target of this property node. If this property node executes on an RT target, it reports interfaces physically on the RT target.

Interfaces (All)

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET System

Short Name

IntfAll

Description

Returns an array of all available interfaces on the system.

The system refers to the execution target of this property node. If this property node executes on an RT target, it reports interfaces physically on the RT target.

Interfaces (CAN)

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET System

Short Name

IntfCAN

Description

Returns an array of all available interfaces on the system that support the CAN Protocol.

The system refers to the execution target of this property node. If this property node executes on an RT target, it reports interfaces physically on the RT target.

Interfaces (LIN)

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET System

Short Name

IntfLIN

Description

Returns an array of all available interfaces on the system that support the LIN Protocol.

The system refers to the execution target of this property node. If this property node executes on an RT target, it reports interfaces physically on the RT target.

Version:Build

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET System

Short Name

Ver.Build

Description

Returns the driver version [Build] as a u32.

Remarks

The driver version is specified in the following format:
[Major].[Minor].[Update][Phase][Build].

For example, 1.2.3f4 returns:

- [Major] = 1
- [Minor] = 2
- [Update] = 3
- [Phase] = Final/Release
- [Build] = 4

A larger version number implies a newer XNET driver version.

Use this property for:

- Determining driver functionality or release date.
- Determining upgrade availability.

Version:Major

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET System

Short Name

Ver.Major

Description

Returns the driver version [Major] as a u32.

Remarks

The driver version is specified in the following format:
[Major].[Minor].[Update][Phase][Build].

For example, 1.2.3f4 returns:

- [Major] = 1
- [Minor] = 2
- [Update] = 3
- [Phase] = Final/Release
- [Build] = 4

A larger version number implies a newer XNET driver version.

Use this property for:

- Determining driver functionality or release date.
- Determining upgrade availability.

Version:Minor

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET System

Short Name

Ver.Minor

Description

Returns the driver version [Minor] as a u32.

Remarks

The driver version is specified in the following format:
[Major].[Minor].[Update][Phase][Build].

For example, 1.2.3f4 returns:

- [Major] = 1
- [Minor] = 2
- [Update] = 3
- [Phase] = Final/Release
- [Build] = 4

A larger version number implies a newer XNET driver version.

Use this property for:

- Determining driver functionality or release date.
- Determining upgrade availability.

Version:Phase

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET System

Short Name

Ver.Phase

Description

Returns the driver version [Phase] as an enumeration.

Enumeration	Value
Development	0
Alpha	1
Beta	2
Release	3



Note The driver's official version always has a phase of Release.

Remarks

The driver version is specified in the following format:
[Major].[Minor].[Update][Phase][Build].

For example, 1.2.3f4 returns:

- [Major] = 1
- [Minor] = 2
- [Update] = 3
- [Phase] = Final/Release
- [Build] = 4

A larger version number implies a newer XNET driver version.

Use this property for:

- Determining driver functionality or release date.
- Determining upgrade availability.

Version:Update

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET System

Short Name

Ver.Update

Description

Returns the driver version [Update] as a u32.

Remarks

The driver version is specified in the following format:
[Major].[Minor].[Update][Phase][Build].

For example, 1.2.3f4 returns:

- [Major] = 1
- [Minor] = 2
- [Update] = 3
- [Phase] = Final/Release
- [Build] = 4

A larger version number implies a newer XNET driver version.

Use this property for:

- Determining driver functionality or release date.
- Determining upgrade availability.

XNET Device Property Node

Format



Description

Property node used to read/write properties for an XNET Device I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

Form Factor

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Device

Short Name

FormFac

Description

Returns the XNET board physical form factor.

Enumeration	Value
PXI	0
PCI	1
C Series	2

Interfaces

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Device

Short Name

Intfs

Description

Returns an array of XNET Interface I/O names associated with this physical hardware device.

XNET Interface Details

The XNET Interface I/O Name represents a physical communication port on an XNET device. An XNET device may have one or more XNET Interface I/O names, depending on the number of physical connectors the board has.

You can pass the XNET Interface I/O name to the XNET Interface Property Node to retrieve hardware information about the interface. This XNET interface is the same I/O name used to create the session.

Displayed on the front panel, the XNET Interface I/O name displays the interface string name.

This string is used for:

- **XNET String to IO Name.vi** to retrieve the XNET Interface I/O name.
- Identification in Measurement & Automation Explorer (MAX).

Number of Ports

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Device

Short Name

NumPorts

Description

Returns the number of physical port connectors on the XNET board.

Remarks

For example, returns 2 for an NI PCI-8517 two-port FlexRay device.

Product Name

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Device

Short Name

ProductName

Description

Returns the XNET device product name.

Remarks

For example, returns NI PCI-8517 (2 ports) for an NI PCI-8517 device.

Product Number

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Device

Short Name

ProductNum

Description

Returns the numeric portion of the XNET device product name.

Remarks

For example, returns 8517 for an NI PCI-8517 two-port FlexRay device.

Serial Number

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Device

Short Name

SerNum

Description

Returns the serial number associated with the XNET device.

Remarks

The serial number is written in HEX on a label on the physical XNET board. Convert the return value from this property to HEX to match the label.

Slot Number

Data Type	Direction	Required?	Default
U32	Read Only	No	N/A

Property Class

XNET Device

Short Name

SlotNum

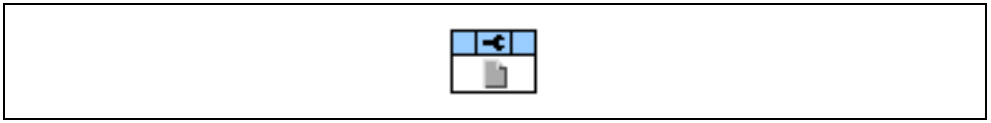
Description

Physical slot where the device (module) is located.

For PXI and C Series, this is the slot number within the chassis.

XNET Interface Property Node

Format



Description

Property node used to read/write properties for an XNET Interface I/O Name.

Pull down this node to add properties. Right-click to change direction between read and write. Right-click each property name to create a constant, control, or indicator.

For help on a specific property, open the LabVIEW context help window (<Ctrl-H>) and move your cursor over the property name.

For more information about LabVIEW property nodes, open the main LabVIEW help (select **Search the LabVIEW Help** from the **Help** menu) and look for the *Property Nodes* topic in the index.

CAN.Termination Capability

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Interface

Short Name

CAN.TermCap

Description

Returns an enumeration indicating whether the XNET interface can terminate the CAN bus.

Enumeration	Value
No	0
Yes	1

Remarks

Signal reflections on the CAN bus can cause communication failure. To prevent reflections, termination can be present as external resistance or resistance the XNET board applies internally. This enumeration determines whether the XNET board can add termination to the bus.

To select the CAN transceiver termination, refer to Interface:CAN:Termination.

CAN.Transceiver Capability

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Interface

Short Name

CAN.TcvrCap

Description

Returns an enumeration indicating the CAN bus physical transceiver support.

Enumeration	Value
High-Speed (HS)	0
Low-Speed (LS)	1
XS (HS, LS, SW, or External)	2

Remarks

The XS value in the enumeration indicates the board has the physical transceivers for High-Speed (HS), Low-Speed (LS), and Single Wire (SW), and can connect to an external transceiver. This value is switchable through the Interface:CAN:Transceiver Type property.

Device

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Interface

Short Name

Device

Description

From the XNET Interface I/O Name, this property returns the XNET device I/O name.

Remarks

The XNET device I/O name returned is the physical XNET board that contains the XNET interface. This property determines the physical XNET device through the XNET Device Serial Number property for a given XNET Interface I/O name.

Name

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Interface

Short Name

Name

Description

Returns the string name assigned to the XNET Interface I/O name.

Remarks

This string is used for:

- **XNET String to IO Name.vi**, to retrieve the XNET Interface I/O name.
- Identification in MAX.

Number

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Interface

Short Name

Number

Description

Returns unique number associated with the XNET interface.

Remarks

The XNET driver assigns each port connector in the system a unique number XNET driver. This number, plus its protocol name, is the XNET Interface I/O Name string name. For example:

XNET Interface String Name	Number
CAN1	1
FlexRay3	3

Port Number

Data Type	Direction	Required?	Default
U32	Read Only	No	N/A

Property Class

XNET Interface

Short Name

PortNum

Description

Returns the physical port number printed near the connector on the XNET device.

Remarks

The port numbers on an XNET board are physically identified with numbering. Use this property, along with the XNET Device Serial Number property, to associate an XNET interface with a physical (XNET board and port) combination.

Note It is easier to find the physical location of an XNET Interface with **XNET Blink.vi**.

Protocol

Data Type	Direction	Required?	Default
	Read Only	No	N/A

Property Class

XNET Interface

Short Name

Protocol

Description

Returns a protocol supported by the XNET Interface I/O Name as an enumeration.

Enumeration	Value
CAN	0
FlexRay	1
LIN	2

Remarks

The protocol enumeration matches the protocol part of the XNET Interface string name.

String Name	Protocol Enumeration
CAN1	0
FlexRay3	1

XNET Interface Constant

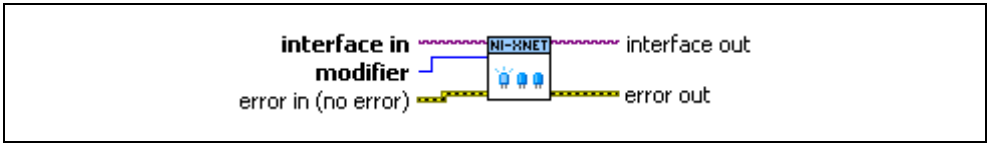
This constant provides the constant form of the XNET Interface I/O name. You drag a constant to the block diagram of your VI, then select an interface. You can change constants only during configuration, prior to running the VI. For a complete description, refer to *XNET Interface I/O Name*.

XNET Blink.vi

Purpose

Blinks LEDs for the XNET interface to identify its physical port in the system.

Format



Inputs

interface in is the XNET Interface I/O name.

modifier controls LED blinking:

Disable (0) Disable blinking for identification. This option turns off both LEDs for the port.

Enable (1) Enable blinking for identification. Both LEDs of the interface's physical port turn on and off. The hardware blinks the LEDs automatically until you disable, so there is no need to call the **XNET Blink** VI repetitively.

Both LEDs blink green (not red). The blinking rate is approximately three times per second.

error in is the error cluster input (refer to *Error Handling*).

Outputs

interface out is the same as interface in, provided for use with subsequent VIs.

error out is the error cluster output (refer to *Error Handling*).

Description

Each XNET device contains one or two physical ports. Each port is labeled on the hardware as *Port 1* or *Port 2*. The XNET device also provides two LEDs per port. For a two-port board, LEDs 1 and 2 are assigned to Port 1, and LEDs 3 and 4 are assigned to physical Port 2.

When your application uses multiple XNET devices, this VI helps to identify each interface to associate its software behavior (LabVIEW code) to its hardware connection (port). Prior to running your XNET sessions, you can call this VI to blink the interface LEDs.

For example, if you have a system with three PCI CAN cards, each with two ports, you can use this VI to blink the LEDs for interface CAN4, to identify it among the six CAN ports.

The LEDs of each port support two states:

- **Identification**: Blink LEDs to identify the physical port assigned to the interface.
- **In Use**: LED behavior that XNET sessions control.

Identification LED State

You can use the **XNET Blink** VI only in the Identification state. If you call this VI while one or more XNET sessions for the interface are open (created), it returns an error, because the port's LEDs are in the In Use state.

In Use LED State

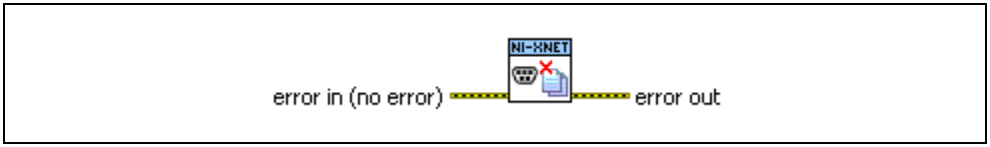
When you create an XNET session for the interface, the LEDs for that physical port transition to the In Use state. If you called the **XNET Blink** VI previously to enable blinking for identification, that LED behavior no longer applies. The In Use LED state remains until all XNET sessions are cleared. This typically occurs when all LabVIEW VIs are no longer running. The patterns that appear on the LEDs while In Use are documented in the *LEDs* section of Chapter 3, *NI-XNET Hardware Overview*.

XNET System Close.vi

Purpose

Closes the XNET system to refresh XNET hardware information.

Format



Inputs

error in is the error cluster input (refer to *Error Handling*).

Outputs

error out is the error cluster output (refer to *Error Handling*).

Description

When your VI first uses the XNET System Property Node, NI-XNET obtains information about all available devices and interfaces in the system. While using property nodes for the devices and interfaces, the hardware information maintains consistency. For example, if you physically add a new device (for example, a plug-in a CompactDAQ chassis), the new device does not appear in the system properties.

Use **XNET System Close.vi** to close the system and associated devices and interfaces. The next time your VI uses the XNET System property node, the hardware information is refreshed.

If you previously used **XNET Blink.vi** to blink a device's LEDs for identification, **XNET System Close.vi** disables blinking when it closes the associated device.

XNET String to IO Name.vi

Purpose

Converts a LabVIEW string to an XNET I/O Name.

Description

This polymorphic VI converts a LabVIEW string to an XNET I/O name.

This VI is not required for LabVIEW 2009 or newer. It is provided only for backward compatibility of VIs written in LabVIEW version prior to 2009. Currently supported versions of LabVIEW can now cast LabVIEW strings to XNET I/O names automatically.

XNET Convert.vi

Purpose

Converts between NI-XNET signal data and frame data or vice versa.

Description

The instances of this polymorphic VI specify the conversion direction and type of frame data.

There are two categories of **XNET Convert** instance VIs:

- **Frame to Signal**: Converts frame data to signal data. A stream of frames is read, and the signal values are filled with the values of the latest respective frames. Frames not matching any signals are ignored. If two or more frames with the same ID are present, the most recent (last) value is returned.
- **Signal to Frame**: Converts signal data to frame data. One frame for each ID involved in the signal list is returned. Data not occupied by the signals from the list is filled with the respective default values.

You can use both categories with the same conversion session mode.

The **XNET Convert** instance VIs are:

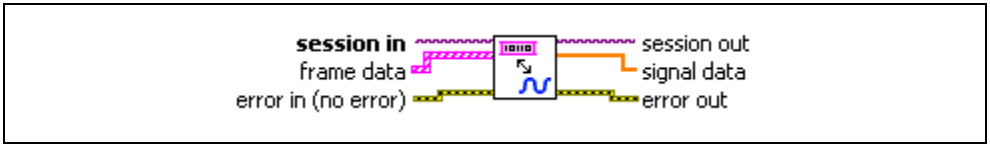
- **XNET Convert (Frame CAN to Signal).vi**: Reads a set of CAN frames and extracts the most recent signal values from them.
- **XNET Convert (Frame FlexRay to Signal).vi**: Reads a set of FlexRay frames and extracts the most recent signal values from them.
- **XNET Convert (Frame LIN to Signal).vi**: Reads a set of LIN frames and extracts the most recent signal values from them.
- **XNET Convert (Frame Raw to Signal).vi**: Reads a set of raw frames and extracts the most recent signal values from them.
- **XNET Convert (Signal to Frame CAN).vi**: Reads a set of signal values and creates CAN frames with their representation.
- **XNET Convert (Signal to Frame FlexRay).vi**: Reads a set of signal values and creates FlexRay frames with their representation.
- **XNET Convert (Signal to Frame LIN).vi**: Reads a set of signal values and creates LIN frames with their representation.
- **XNET Convert (Signal to Frame Raw).vi**: Reads a set of signal values and creates raw frames with their representation.

XNET Convert (Frame CAN to Signal).vi

Purpose

Converts between NI-XNET CAN frame data and signals.

Format



Inputs

session in is the session to read. This session is returned from **XNET Create Session.vi**. The session mode must be Conversion.

frame data provides the array of LabVIEW clusters.

Each array element corresponds to a frame value to convert.

The data you write is converted to signal values in the order you provide them. Only the latest signal value is returned.

For an example of how this data applies, refer to *Conversion Mode*.

The elements of each cluster are specific to the CAN protocol. For more information, refer to Appendix A, *Summary of the CAN Standard*, or the CAN protocol specification.

The cluster elements are:

identifier is the CAN frame arbitration identifier.

If **extended?** is false, the identifier uses standard format, so 11 bits of this identifier are valid.

If **extended?** is true, the identifier uses extended format, so 29 bits of this identifier are valid.

extended? is a Boolean value that determines whether the identifier uses extended format (true) or standard format (false).

echo? is not used for conversion. You must set this element to false.



type is the frame type (decimal value in parentheses):

- **CAN Data (0)**: The CAN data frame contains **payload** data. This is the most commonly used frame type for CAN.
- **CAN Remote (1)**: CAN remote frame. Your application transmits a CAN remote frame to request data for the corresponding **identifier**. A remote ECU should respond with a CAN data frame for the **identifier**, which you can obtain using **XNET Read.vi**. This value is not meaningful, as a remote frame does not contain any data to convert.



timestamp represents absolute time using the LabVIEW absolute timestamp type. **timestamp** is not used for conversion. You must set this element to the default value, invalid (0).



payload is the array of data bytes for a CAN data frame.

The array size indicates the payload length of the frame value to transmit. According to the CAN protocol, the payload length range is 0–8. For CAN FD, the range can be 0–8, 12, 16, 20, 24, 32, 48, or 64.

For more information, refer to the section for each mode.

For a transmitted remote frame (CAN Remote type), the payload length in the frame value specifies the number of payload bytes requested. Your application provides this payload length by filling **payload** with the requested number of bytes. This enables your application to specify the frame payload length, but the actual values in the payload bytes are ignored (not contained in the transmitted frame).



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



signal data returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data returns the most recent converted value for each signal. If multiple frames for a signal are input, only signal data from the most recent frame is returned. Here, most recent is defined by the order of the frames in the frame data array, not the timestamp.

If no frame is input for the corresponding signals, the XNET Signal Default Value is returned.

For an example of how this data applies, refer to *Conversion Mode*.



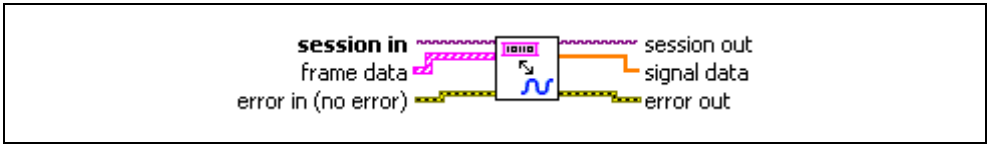
error out is the error cluster output (refer to *Error Handling*).

XNET Convert (Frame FlexRay to Signal).vi

Purpose

Converts between NI-XNET FlexRay frame data and signals.

Format



Inputs

session in is the session to read. This session is returned from **XNET Create Session.vi**. The session mode must be Conversion.

frame data provides the array of LabVIEW clusters.

Each array element corresponds to a frame value to convert.

The data you write is converted to signal values in the order you provide them. Only the latest signal value is returned.

For an example of how this data applies, refer to *Conversion Mode*.

The elements of each cluster are specific to the FlexRay protocol. For more information, refer to Appendix B, *Summary of the FlexRay Standard*, or the *FlexRay Protocol Specification*.

The cluster elements are:

slot specifies the slot number within the FlexRay cycle.

cycle count specifies the cycle number.

The FlexRay cycle count increments from 0 to 63, then rolls over back to 0.

startup? is a Boolean value that specifies whether the frame is a startup frame (true) or not (false). This field is ignored for conversion.

sync? is a Boolean value that specifies whether the frame is a sync frame (true) or not (false). This field is ignored for conversion.



preamble? is a Boolean value that specifies the value of the payload preamble indicator in the frame header.

If the frame is in the static segment, **preamble?** being true indicates the presence of a network management vector at the beginning of the payload. The XNET Cluster FlexRay:Network Management Vector Length property specifies the number of bytes at the beginning.

If the frame is in the dynamic segment, **preamble?** being true indicates the presence of a message ID at the beginning of the payload. The message ID is always 2 bytes in length.

If **preamble?** is false, the payload does not contain a network management vector or a message ID.

This field is ignored for conversion.



chA is a Boolean value that specifies whether to transmit the frame on channel A (true) or not (false). This field is ignored for conversion.



chB is a Boolean value that specifies whether to transmit the frame on channel B (true) or not (false). This field is ignored for conversion.



echo? is not used for conversion. You must set this element to false.



type is the frame type. **type** is not used for transmit, so you must leave this element uninitialized. All frame values are assumed to be the **FlexRay Data** type. Frames of **FlexRay Data** type contain payload data.

The **FlexRay Null** type is not transmitted based on this type. As specified in the XNET Frame FlexRay:Timing Type property, the FlexRay null frame is transmitted when a cyclically timed frame does not have new data.



timestamp represents absolute time using the LabVIEW absolute timestamp type. **timestamp** is not used for conversion. You must set this element to the default value, invalid (0).



payload is the array of data bytes for FlexRay frames of type **FlexRay Data**.

The array size indicates the payload length of the frame value to transmit. According to the FlexRay protocol, the length range is 0–254.

You can leave all other FlexRay frame cluster elements uninitialized. For more information, refer to the section for each mode.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



signal data returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data returns the most recent converted value for each signal. If multiple frames for a signal are input, only signal data from the most recent frame is returned. Here, most recent is defined by the order of the frames in the frame data array, not the timestamp.

If no frame is input for the corresponding signals, the XNET Signal Default Value is returned.

For an example of how this data applies, refer to *Conversion Mode*.



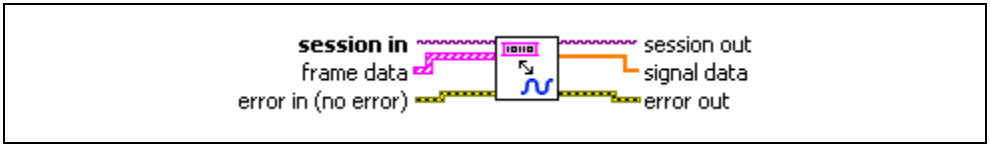
error out is the error cluster output (refer to *Error Handling*).

XNET Convert (Frame LIN to Signal).vi

Purpose

Converts between NI-XNET LIN frame data and signals.

Format



Inputs

session in is the session to read. This session is returned from **XNET Create Session.vi**. The session mode must be Conversion.

frame data provides the array of LabVIEW clusters.

Each array element corresponds to a frame value to convert.

The data you write is converted to signal values in the order you provide them. Only the latest signal value is returned.

For an example of how this data applies, refer to *Conversion Mode*.

The elements of each cluster are specific to the LIN protocol. For more information, refer to Appendix C, *Summary of the LIN Standard*, or the LIN protocol specification.

The cluster elements are:

identifier is not used for transmit. You must set this element to 0.

Each frame is identified based on the list of frames or signals provided for the session. The actual identifier to transmit is taken from the database (frame and schedule properties). Therefore, this identifier in the frame value is ignored.

event slot? is not used for transmit. You must set this element to false.

The currently running schedule is used to map the specific frame to a corresponding schedule entry (slot). The schedule entry itself determines whether the slot is unconditional, sporadic, or event triggered.



event ID is not used for transmit. You must set this element to 0.



echo? is not used for conversion. You must set this element to false.



type is the frame type (decimal value in parentheses):

LIN Data (64) The LIN data frame contains **payload** data. This is currently the only frame type for LIN.

This value is ignored for conversion.



timestamp represents absolute time using the LabVIEW absolute timestamp type. **timestamp** is not used for conversion. You must set this element to the default value, invalid (0).



payload is the array of data bytes for a LIN data frame.

The array size indicates the payload length of the frame value to transmit. According to the LIN protocol, the payload length range is 0–8.

For more information, refer to the section for each mode.



error in is the error cluster input (refer to *Error Handling*).

Outputs



session out is the same as **session in**, provided for use with subsequent VIs.



signal data returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data returns the most recent converted value for each signal. If multiple frames for a signal are input, only signal data from the most recent frame is returned. Here, most recent is defined by the order of the frames in the frame data array, not the timestamp.

If no frame is input for the corresponding signals, the XNET Signal Default Value is returned.

For an example of how this data applies, refer to *Conversion Mode*.



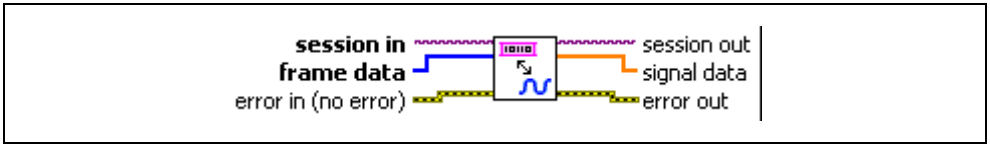
error out is the error cluster output (refer to *Error Handling*).

XNET Convert (Frame Raw to Signal).vi

Purpose

Converts between NI-XNET raw frame data and signals.

Format



Inputs

session in is the session to read. This session is returned from **XNET Create Session.vi**. The session mode must be Conversion.

frame data provides the array of bytes, representing frames to transmit.

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

This frame format is the same for read and write of raw data and also is used for log file examples.

For information about which elements of the raw frame are applicable, refer to the **XNET Convert.vi** instance for the protocol in use (**XNET Convert (Frame CAN to Signal).vi**, **XNET Convert (Frame FlexRay to Signal).vi**, or **XNET Convert (Frame LIN to Signal).vi**).

For an example of how this data applies, refer to *Conversion Mode*.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

signal data returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data returns the most recent converted value for each signal. If multiple frames for a signal are input, only signal data from the most recent frame is

returned. Here, most recent is defined by the order of the frames in the frame data array, not the timestamp.

If no frame is input for the corresponding signals, the XNET Signal Default Value is returned.

For an example of how this data applies, refer to *Conversion Mode*.



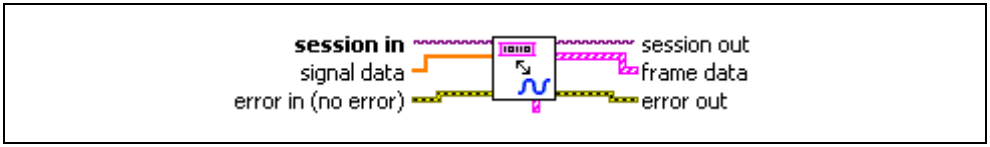
error out is the error cluster output (refer to *Error Handling*).

XNET Convert (Signal to Frame CAN).vi

Purpose

Converts between NI-XNET signals and CAN frame data.

Format



Inputs

session in is the session to read. This session is returned from **XNET Create Session.vi**. The session mode must be Conversion.

signal data returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data provides the value for the next conversion of each signal.

For an example of how this data applies, refer to *Conversion Mode*.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

frame data returns the array of LabVIEW clusters.

Each array element corresponds to a frame the session converted.

The elements of each cluster are specific to the CAN protocol. For more information, refer to Appendix A, *Summary of the CAN Standard*, or the CAN protocol specification.

The cluster elements are:

identifier is the CAN frame arbitration identifier.

If **extended?** is false, the identifier uses standard format, so 11 bits of this identifier are valid.

If **extended?** is true, the identifier uses extended format, so 29 bits of this identifier are valid.



extended? is a Boolean value that determines whether the identifier uses extended format (true) or standard format (false).



echo? is a Boolean value that determines whether the frame was an echo of a successful transmit (true), or received from the network (false). For conversion, it is always set to false.



type is the frame type (decimal value in parentheses):

- **CAN Data (0)**: The CAN data frame contains **payload** data. This is the most commonly used frame type for CAN.
- **CAN Remote (1)**: 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**. This value is not meaningful, as a remote frame does not contain any data to convert.



timestamp represents the absolute time when the XNET interface received the frame (end of frame), accurate to microseconds. The **timestamp** returned by the conversion is always invalid (0).



payload is the array of **data** bytes for the CAN data frame.

The array size indicates the received frame value payload length. According to the CAN protocol, this payload length range is 0–8. For CAN FD, the range can be 0–8, 12, 16, 20, 24, 32, 48, or 64.

For a received remote frame (**type** of **CAN Remote**), the payload length in the frame value specifies the number of **payload** bytes requested. This payload length is provided to your application by filling **payload** with the requested number of bytes. Your application can use the **payload** array size, but you must ignore the actual values in the **payload** bytes.



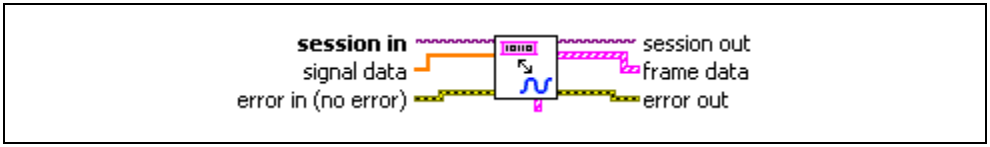
error out is the error cluster output (refer to *Error Handling*).

XNET Convert (Signal to Frame FlexRay).vi

Purpose

Converts between NI-XNET signals and FlexRay frame data.

Format



Inputs

session in is the session to read. This session is returned from **XNET Create Session.vi**. The session mode must be Conversion.

signal data returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data provides the value for the next conversion of each signal.

For an example of how this data applies, refer to *Conversion Mode*.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

frame data returns the array of LabVIEW clusters.

Each array element corresponds to a frame the session converted.

The elements of each cluster are specific to the FlexRay protocol. For more information, refer to Appendix B, *Summary of the FlexRay Standard*, or the *FlexRay Protocol Specification*.

The cluster elements are:

slot specifies the slot number within the FlexRay cycle.

cycle count specifies the cycle number.

The FlexRay cycle count increments from 0 to 63, then rolls over back to 0.

For conversion, this is always set to 0.



startup? is a Boolean value that specifies whether the frame is a startup frame (true) or not (false).

This field is set to false by conversion.



sync? is a Boolean value that specifies whether the frame is a sync frame (true) or not (false).

This field is set to false by conversion.



preamble? is a Boolean value that specifies the value of the payload preamble indicator in the frame header.

If the frame is in the static segment, **preamble?** being true indicates the presence of a network management vector at the beginning of the payload. The XNET Cluster FlexRay:Network Management Vector Length property specifies the number of bytes at the beginning.

If the frame is in the dynamic segment, **preamble?** being true indicates the presence of a message ID at the beginning of the payload. The message ID is always 2 bytes in length.

If **preamble?** is false, the payload does not contain a network management vector or a message ID.

This field is set to false by conversion.



chA is a Boolean value that specifies whether the frame was received on channel A (true) or not (false). This field is set to false by conversion.



chB is a Boolean value that specifies whether the frame was received on channel B (true) or not (false). This field is set to false by conversion.



echo? is a Boolean value that determines whether the frame was an echo of a successful transmit (true), or received from the network (false).



type is the frame type (decimal value in parentheses):

- **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.



timestamp represents the absolute time when the XNET interface received the frame (end of frame), accurate to microseconds. The timestamp returned by the conversion is always invalid (0).



payload is the array of data bytes for FlexRay frames of type **FlexRay Data** or **FlexRay Null.**

The array size indicates the received frame value payload length.

According to the FlexRay protocol, this length range is 0–254.



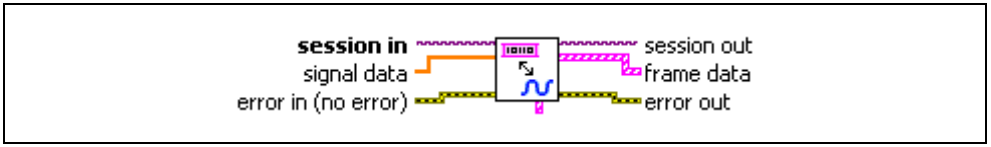
error out is the error cluster output (refer to *Error Handling*).

XNET Convert (Signal to Frame LIN).vi

Purpose

Converts between NI-XNET signals and LIN frame data.

Format



Inputs

session in is the session to read. This session is returned from **XNET Create Session.vi**. The session mode must be Conversion.

signal data returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data provides the value for the next conversion of each signal.

For an example of how this data applies, refer to *Conversion Mode*.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

frame data returns the array of LabVIEW clusters.

Each array element corresponds to a frame the session converted.

The elements of each cluster are specific to the LIN protocol. For more information, refer to Appendix C, *Summary of the LIN Standard*, or the LIN protocol specification.

The cluster elements are:



identifier is the identifier received within the frame's header.

The identifier is a number from 0 to 63.

If the schedule entry (slot) is unconditional or sporadic, this identifies the payload data (LIN frame). If the schedule entry is event triggered, this identifies the schedule entry itself, and the protected ID contained in the first payload byte identifies the payload.



event slot? is not used. This element is false.



event ID is not used. This element is 0.



echo? is a Boolean value that determines whether the frame was an echo of a successful transmit (true), or received from the network (false). For conversion, it is always set to false.



type is the frame type (decimal value in parentheses):

- **LIN Data (64)**: The LIN data frame contains payload data. This is currently the only frame type for LIN.

For conversion, this is always set to false.



timestamp represents the absolute time when the XNET interface received the frame (end of frame), accurate to microseconds. The **timestamp** returned by the conversion is always invalid (0).



payload is the array of data bytes for the LIN data frame.

The array size indicates the received frame's payload length.

According to the LIN protocol, this payload is 0–8 bytes in length.



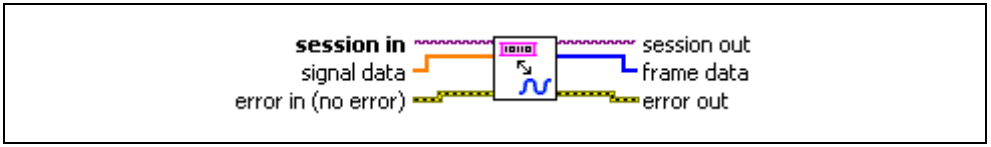
error out is the error cluster output (refer to *Error Handling*).

XNET Convert (Signal to Frame Raw).vi

Purpose

Converts between NI-XNET signals and raw frame data.

Format



Inputs

session in is the session to read. This session is returned from **XNET Create Session.vi**. The session mode must be Conversion.

signal data returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data provides the value for the next conversion of each signal.

For an example of how this data applies, refer to *Conversion Mode*.

error in is the error cluster input (refer to *Error Handling*).

Outputs

session out is the same as **session in**, provided for use with subsequent VIs.

frame data returns an array of bytes.

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

This frame format is the same for read and write of raw data, and it is also used for log file examples.

The data always returns complete frames.

For information about which elements of the raw frame are applicable, refer to the frame read for the protocol in use (**XNET Convert (Signal to Frame CAN).vi**, **XNET Convert (Signal to Frame FlexRay).vi**, or **XNET Convert (Signal to Frame LIN).vi**).

For an example of how this data applies, refer to *Conversion Mode*.



error out is the error cluster output (refer to *Error Handling*).

Controls Palette

This palette provides front panel controls for NI-XNET. You drag a control to the front panel of your VI.

Typically, you use I/O name controls to select a name during configuration, and the name 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 **XNET Create Session.vi**, followed by calls to **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 a name 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 **XNET Create Session.vi** and **XNET Read.vi** are called.

XNET Session Control

This control provides the control form of the XNET Session I/O name. You drag a control to the front panel of your VI, so that the user of the VI can select a name. For a complete description, refer to *XNET Session I/O Name*.

Database Controls

XNET Database Control

This control provides the control form of the XNET Database I/O name. You drag a control to the front panel of your VI, so that the user of the VI can select a name. For a complete description, refer to *XNET Database I/O Name*.

XNET Cluster Control

This control provides the control form of the XNET Cluster I/O name. You drag a control to the front panel of your VI, so that the user of the VI can select a name. For a complete description, refer to *XNET Cluster I/O Name*.

XNET ECU Control

This control provides the control form of the XNET ECU I/O name. You drag a control to the front panel of your VI, so that the user of the VI can select a name. For a complete description, refer to *XNET ECU I/O Name*.

XNET Frame Control

This control provides the control form of the XNET Frame I/O name. You drag a control to the front panel of your VI, so that the user of the VI can select a name. For a complete description, refer to *XNET Frame I/O Name*.

XNET Signal Control

This control provides the control form of the XNET Signal I/O name. You drag a control to the front panel of your VI, so that the user of the VI can select a name. For a complete description, refer to *XNET Signal I/O Name*.

XNET LIN Schedule Control

This control provides the control form of the XNET LIN Schedule I/O name. You drag a control to the front panel of your VI, so that the user of the VI can select a name. For a complete description, refer to *XNET LIN Schedule I/O Name*.

XNET LIN Schedule Entry Control

This control provides the control form of the XNET LIN Schedule Entry I/O name. You drag a control to the front panel of your VI, so that the user of the VI can select a name. For a complete description, refer to *XNET LIN Schedule Entry I/O Name*.

System Controls

XNET Interface Control

This control provides the control form of the XNET Interface I/O name. You drag a control to the front panel of your VI, so that the user of the VI can select a name. For a complete description, refer to *XNET Interface I/O Name*.

XNET Terminal Control

This control provides the control form of the XNET Terminal I/O name. You drag a control to the front panel of your VI, so that the user of the VI can select a name. For a complete description, refer to *XNET Terminal I/O Name*.

Additional Topics

This section includes additional CAN, FlexRay, and LIN-related information.

Overall

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 section 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 **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 **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).

Cyclic and Event Timing

For all embedded network protocols (for example, CAN, LIN, and FlexRay), 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 **XNET Write.vi**. The call to **XNET Write.vi** 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, **XNET Write.vi** 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* section, 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. Similar to database properties, you cannot change Run Mode after a session is created.

- **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, **XNET Write (State LIN Schedule Change).vi** 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, **XNET Write.vi** 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, **XNET Write.vi** 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.

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

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 **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 **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).

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 **XNET Write.vi**, in some cases the problem cannot be detected until the data is about to be transmitted. Because the transmission occurs after **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 **XNET Read (State CAN Comm).vi**, for FlexRay this is **XNET Read (State FlexRay Comm).vi**, and for LIN this is **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 **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.

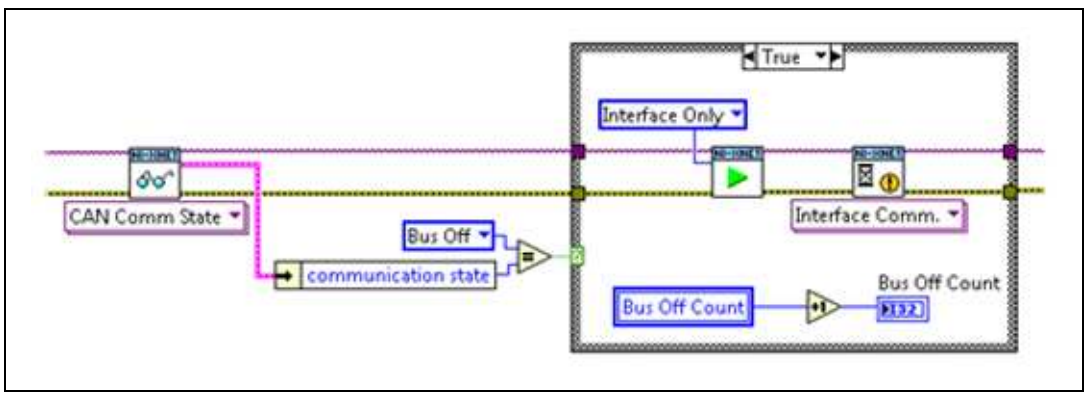


Figure 4-15. Restart on CAN Bus Off State

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.

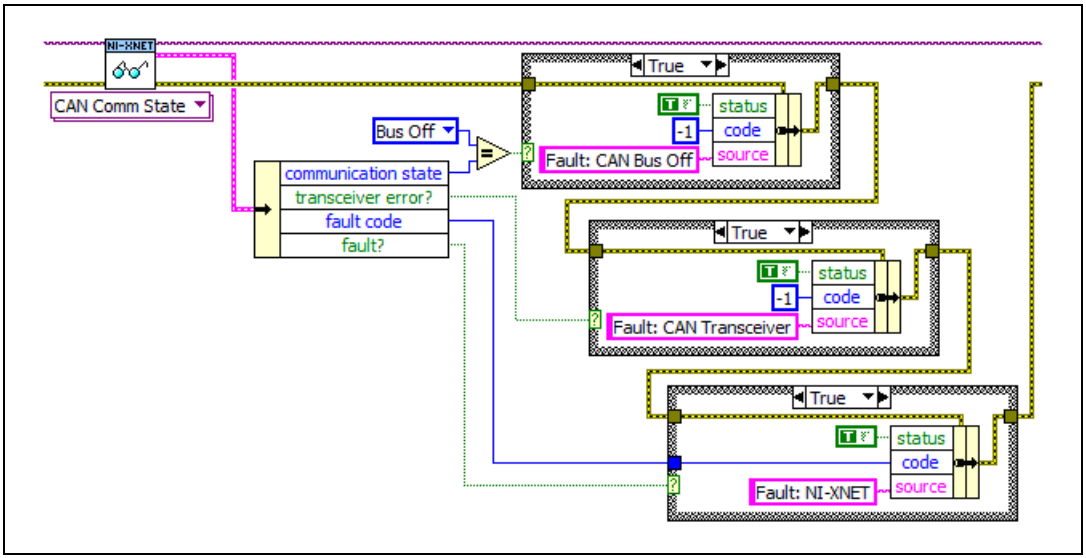


Figure 4-16. Propagating CAN Faults to an Error

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.

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 simply 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.

Raw Frame Format

This section describes the raw data format for frames. The TDMS file format, **XNET Read (Frame Raw).vi**, and **XNET Write (Frame Raw).vi** use this format.

The raw frame format is for examples that demonstrate access to log files. 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.

Refer to the NI-XNET logfile examples for VIs that convert raw frame data to/from LabVIEW clusters for CAN, FlexRay, or LIN frames.

The raw frame format consists of one or more frames encoded in a sequence of bytes. 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).

Table 4-3. Base Unit Elements

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 12:00 a.m. January 1 1601 Coordinated Universal Time (UTC). 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 big-endian computing platforms (for example, CompactRIO with a PowerPC), Byte Offset 0 is the most significant byte. The LabVIEW absolute timestamp data type is different than this U64 timestamp. NI-XNET includes a pair of VIs to convert between this U64 timestamp format and the LabVIEW timestamp format. The NI-XNET VIs handle all time format and byte order aspects. For more information, refer to the NI-XNET examples for logfile 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). All unused bits are 0.

Table 4-3. Base Unit Elements (Continued)

Element	Byte Offset	Description
Type	12	The frame type. This element specifies the fundamental frame type. The Identifier, Flag, and Info element interpretation is different for each type. The upper 4 bits of this element specify the protocol. The valid values in decimal are: 0 CAN 2 FlexRay 4 LIN 14 Special The lower 4 bits of this element contain the specific type. For information about the specific CAN Type values, refer to **XNET Read (Frame CAN).vi**. For information about the specific FlexRay Type values, refer to **XNET Read (Frame FlexRay).vi**. For information about the specific LIN Type values, refer to **XNET Read (Frame LIN).vi**. Special values specify features not related to the protocol or bus traffic. For more information about special frames, refer to *Special Frames*.

Table 4-3. Base Unit Elements (Continued)

Element	Byte Offset	Description
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 **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 • Bit 2 specifies the frame Preamble bit • Bit 4 specifies if the frame transfers on Channel A • Bit 5 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, if bit 0 of the Flags element is clear, the Info element is 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.

Table 4-3. Base Unit Elements (Continued)

Element	Byte Offset	Description
PayloadLength	15	The PayloadLength indicates the number of valid data bytes in Payload. For 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. 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 Payload, bytes 8–11 are in the first byte 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 logfile, but PayloadLength determines the number of valid bytes.

Payload Unit

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

Table 4-4. Payload Unit Elements

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

Special Frames

The NI-XNET driver offers a few special frames not directly used in bus communication.

Delay Frame

A Delay frame is used during replay. When a frame with a Delay frame type is in the stream output queue while the Interface:Output Stream Timing property is set to a replay mode, the hardware delays for the requested time. 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 a Frame Stream Input session can receive. 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 XNET Connect Terminals.vi to connect the external connection to the internal LogTrigger terminal. A Log

Trigger frame is applicable to CAN, LIN, and FlexRay. The Log Trigger Frame fields are as follows:

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

LIN Frame

Element	Description
identifier	0
event slot?	False
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

FlexRay Frame

Element	Description
slot	0
cycle count	0

Element	Description
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

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. (Refer to *Start Interface* for more information.) 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, LIN, and FlexRay. The fields of the Start Trigger frame are as follows:

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

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

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

Bus Error Frame

A Bus Error frame is a special frame that a Frame Stream Input session can receive. This frame is generated when a bus error is detected on the hardware 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:

CAN Frame

Element	Description
identifier	0
extended?	False
echo?	False
type	CAN Bus Error
timestamp	Time when the bus error was detected
payload length	5 (may increase in future)
payload	Byte 0: CAN Comm State 0 = Error Active 1 = Error Passive 2 = Bus Off Byte 1: TX Error Counter Byte 2: RX Error Counter Byte 3: Detected Bus Error 0 = None (never returned) 1 = Stuff 2 = Form 3 = Ack 4 = Bit 1 5 = Bit 0 6 = CRC Byte 4: Transceiver Error? 0 = no error 1 = error

LIN Frame

Element	Description
identifier	0
event slot?	False
event ID	0
echo?	False
type	LIN Bus Error
timestamp	Time when the bus error was detected
payload length	5 (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 6 = CRC Byte 2: Identifier on bus Byte 3: Received byte on bus Byte 4: Expected byte on bus

Required Properties

When you create a new object in a database, the object 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 **XNET Create Session.vi**. 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 section lists all required properties. Properties with a protocol prefix (for example, *FlexRay:*) in the property name apply only a session that uses the specified protocol.

The Cluster object class requires the following properties:

- Baud Rate[1]
- 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

[1] 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 Baud Rate property or XNET Session Interface:Baud Rate property. For CAN and LIN with other session modes, the XNET Cluster Baud Rate property is required.

- 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
- LIN: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 property:

- 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

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 can be accessed using the API. In other words, the session controls receive or transmit 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 **XNET Read.vi** or **XNET Write.vi**, you can perform a more specific transition using **XNET Start.vi** and **XNET Stop.vi**. 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*.

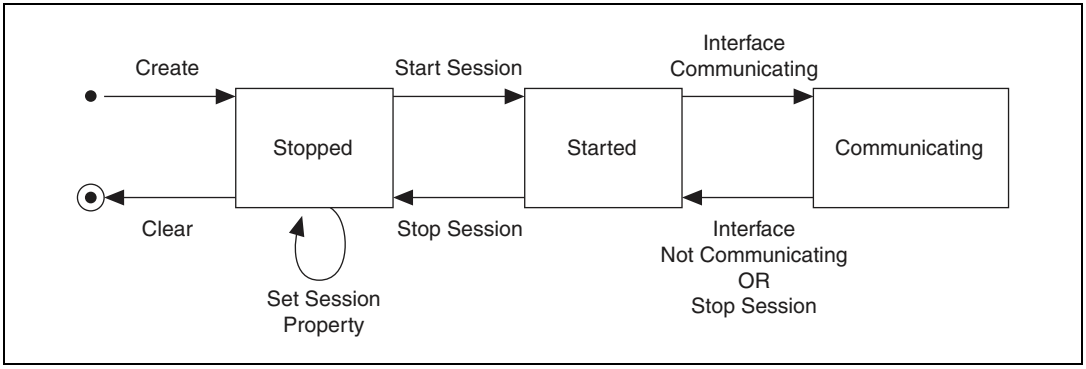


Figure 4-17. Session State Model

Interface State Model

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

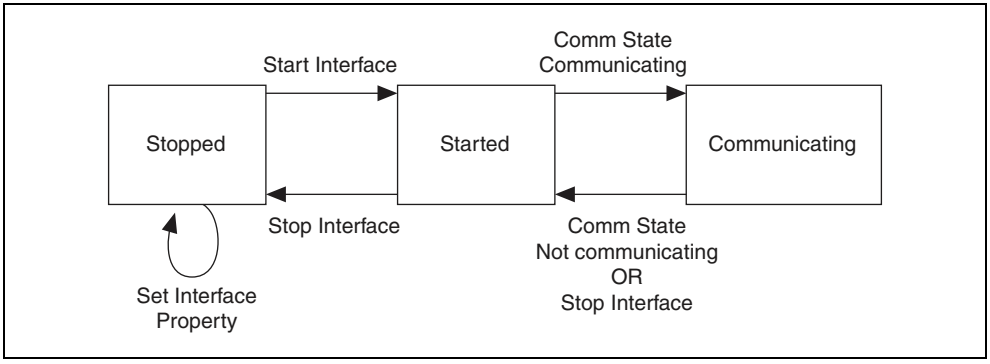


Figure 4-18. Interface State Model

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 property in the Session Property Node 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 **XNET Read.vi**, **XNET Write.vi**, or **XNET Start.vi** using defaults, 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 **XNET Start.vi** 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** 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 **XNET Read.vi** or **XNET Write.vi**.

Session Transitions

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:

- **XNET Create Session.vi method**: When your application calls **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 calls to **XNET Create Session.vi** in sequence prior to the first use of **XNET Read.vi** 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 **XNET Read.vi** 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 **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 Property 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 **XNET Read.vi**. To read received frames, **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 **XNET Write.vi**. The auto-start feature of **XNET Write.vi** performs a Start of scope Normal, which starts the session and interface.

To start the session prior to calling **XNET Read.vi** or **XNET Write.vi**, you can call **XNET Start.vi**. The **XNET Start.vi** default scope is Normal, which starts the session and interface. You also can use **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 **XNET Clear.vi** or **XNET Stop.vi**. **XNET Stop.vi** provides the same scope as **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, then you call **XNET Stop.vi**, you still can call **XNET Read.vi** to read the frame values from the queues. To flush the queues of a session, call **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 **XNET Clear.vi** or **XNET Stop.vi**.

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 Property Node Interface category. 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*.

Interface Transitions

Set Interface Property

While the interface is Stopped, you can change interface-specific properties. These properties are in the Session Property Node Interface category. 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 the 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.vi or XNET Write.vi 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 method**: If you call this VI 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** 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 (**XNET Read.vi**, **XNET Write.vi**, or **XNET Start.vi**).
- **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, **XNET**

Stop.vi), 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** 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 **XNET Stop.vi**. In other words, using **XNET Stop.vi** 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 (State CAN Comm).vi**.

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 (State FlexRay Comm).vi**.

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 (State LIN Comm).vi**.

Comm State Not Communicating

This transition occurs when the interface no longer is 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**.

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 FlexRay Comm).vi**.

For LIN, this occurs when communication enters the Idle state. For more information about the specific LIN interface communication states, refer to **XNET Read (State LIN Comm).vi**.

TDMS

This section 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 section 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. Table 4-5 lists the channel properties.

Table 4-5. Channel Properties

Name	Data Type	Permissions	Description
NI_network_protocol		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		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. 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.

Table 4-5. Channel Properties (Continued)

Name	Data Type	Permissions	Description
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 *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 like CANdb, LDF, 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 significant byte in lowest offset, Motorola byte order) If this property does not exist, the data is not interpreted as raw frames, but as plain U8 samples.

Table 4-5. Channel Properties (Continued)

Name	Data Type	Permissions	Description
NI_network_content		Optional	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 NI_network_protocol. For CAN, FlexRay, and LIN, the string encoding is: <*alias*>.<<*cluster*>> 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 c:\MyDatabases\CANdb\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, 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.

Table 4-5. Channel Properties (Continued)

Name	Data Type	Permissions	Description
			This property is optional. For applications that read the logfile, 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.

CAN

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.

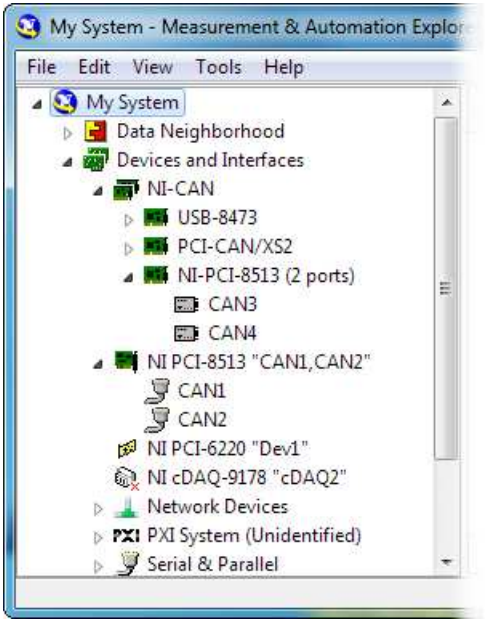
If you are starting a new application, you typically use NI-XNET (not NI-CAN).

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 Hardware and Software 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**.



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.

Table 4-6. NI-CAN and 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 format.
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.
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 section 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 **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 **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 **XNET Read.vi** returns its data.

If the CAN remote frame is received, a subsequent call to **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 **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 **XNET Write.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 **XNET Write.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 **XNET Write.vi**, 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 **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.

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 **XNET Write.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 **XNET Write.vi** has transmitted. Because the database-specified timing for the frame is event based, after the CAN data frames for **XNET Write.vi** have transmitted, the CAN data frame does not transmit again until a subsequent call to **XNET Write.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 **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 **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 **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 **XNET Write.vi** to provide the CAN data frames used for transmit. When you call **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 **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 **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 **XNET Read.vi** and **XNET Write.vi**. You must call **XNET Write.vi** to provide the event that triggers the CAN remote frame transmit. When you call **XNET Write.vi**, the data is ignored, and one CAN remote frame transmits as soon as possible. Each call to **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 **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

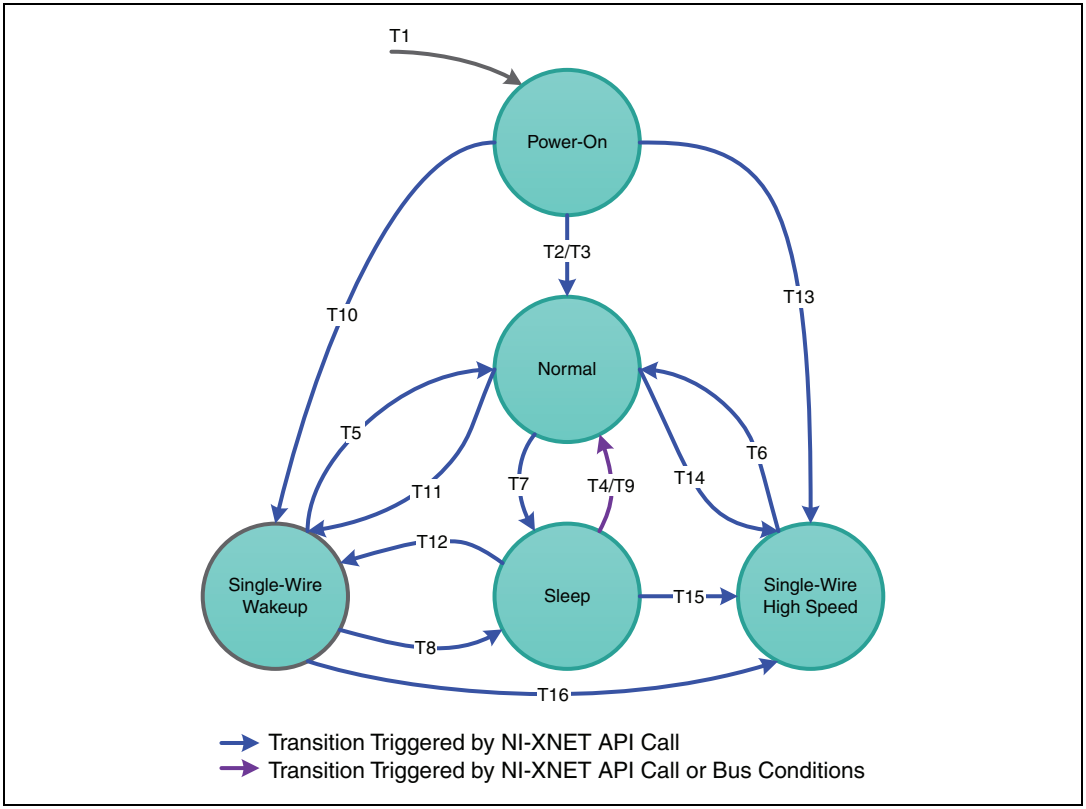
The behavior is the same as *Cyclic Remote*. When you write data using **XNET Write.vi**, the CAN data frame transmits onto the network when the associated CAN remote frame is received (same identifier). Unlike *Cyclic Data*, the remote ECU sends the associated CAN remote frame in an event-driven manner, but the behavior is the same regarding **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 transceriver by setting 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
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

T#	Condition	From	To
10	Interface:CAN:Transceiver State with value SW Wakeup	Power-on	SW Wakeup
11	Interface:CAN:Transceiver State with value SW Wakeup	Normal	SW Wakeup
12	Interface:CAN:Transceiver State with value SW Wakeup	Sleep	SW Wakeup
13	Interface:CAN:Transceiver State with value SW HighSpeed	Power-on	SW High Speed
14	Interface:CAN:Transceiver State with value SW HighSpeed	Normal	SW High Speed
15	Interface:CAN:Transceiver State with value SW HighSpeed	Sleep	SW High Speed
16	Interface:CAN:Transceiver State with value SW HighSpeed	SW Wakeup	SW High Speed

FlexRay

FlexRay Timing Type and Session Mode

For each XNET frame FlexRay:Timing Type property value, this section 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 **XNET Read.vi**.

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 **XNET Read.vi** 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 **XNET Read.vi**). 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 **XNET Read.vi** 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 **XNET Read.vi** 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 **XNET Read.vi** 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 **XNET Read (Frame FlexRay).vi**).

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 **XNET Write.vi**, 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 **XNET Write.vi** 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 **XNET Write.vi**, 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 **XNET Write.vi**. 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 **XNET Write.vi** has transmitted. Because the database-specified timing for the frame is event based, after the FlexRay data frames for **XNET Write.vi** have transmitted, the FlexRay data frame does not transmit again until a subsequent call to **XNET Write.vi**.

If the frame is contained in the static segment, a FlexRay null frame transmits when no new data is available (no new call to **XNET Write.vi**). If you pass the FlexRay null frame to **XNET Write.vi**, 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 **XNET Write.vi**.

Frame Output Stream Mode

This session mode is not supported for FlexRay.

Protocol Data Units (PDUs) in NI-XNET

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.

Relationship Between Frames, Signals, and PDUs

Frames and PDUs

The frame element contains an arbitrary number of nonoverlapping PDUs. A frame can have multiple PDUs, and the same PDU can exist in different frames. Figure 4-19 shows the one-to-*n* (one PDU in *n* number of frames) and *n*-to-one (*n* number of PDUs in one frame) relationships.

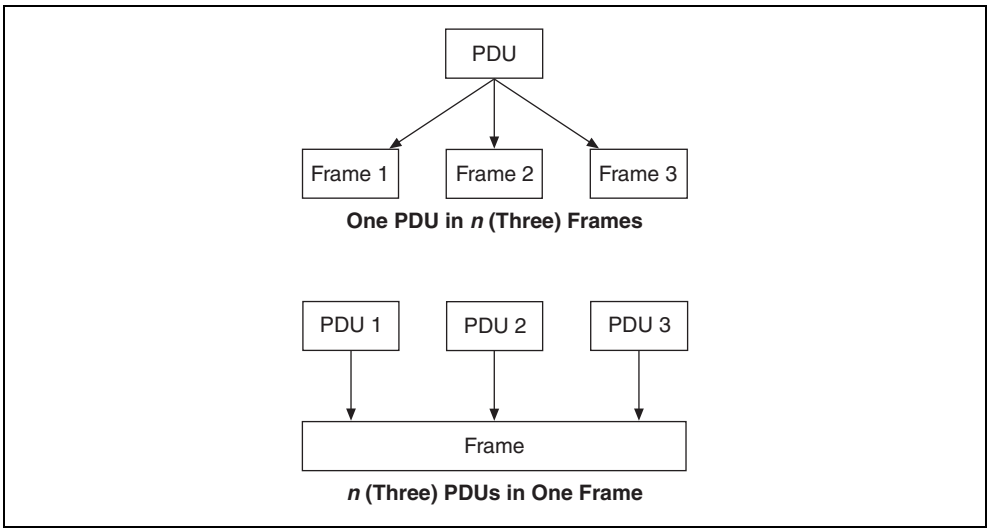


Figure 4-19. Relationships Between PDUs and Frames

Signals and PDUs

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

Figure 4-20 represents the relationship between frames, PDUs, and signals.

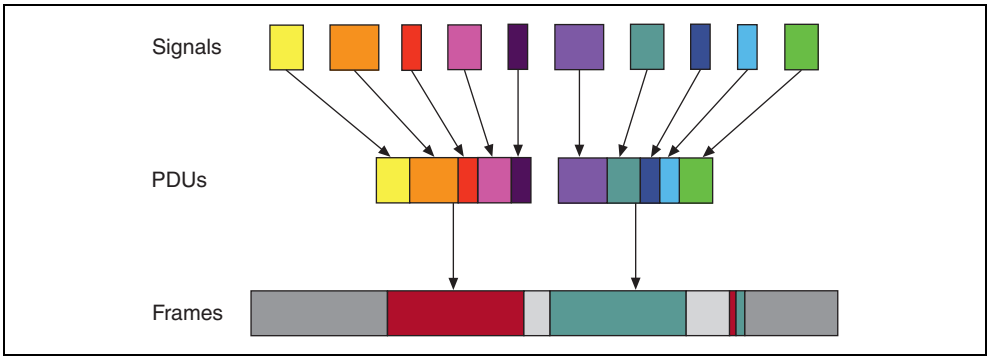


Figure 4-20. Relationships Between Frames, PDUs, and Signals

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.

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 Figure 4-21.)

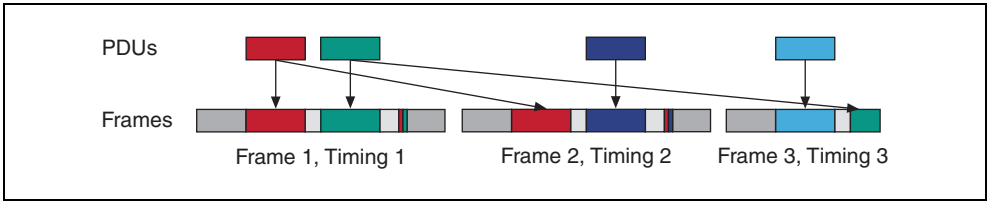


Figure 4-21. PDU Timing and Frame Timing

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, wire the PDU(s) to the special Create Session modes for PDU. (Refer to **XNET Create Session (PDU Input Queued).vi** for more information.) 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.

FlexRay Startup/Wakeup

Use the FlexRay Startup mechanism to take an idle interface and properly integrate into a FlexRay cluster.

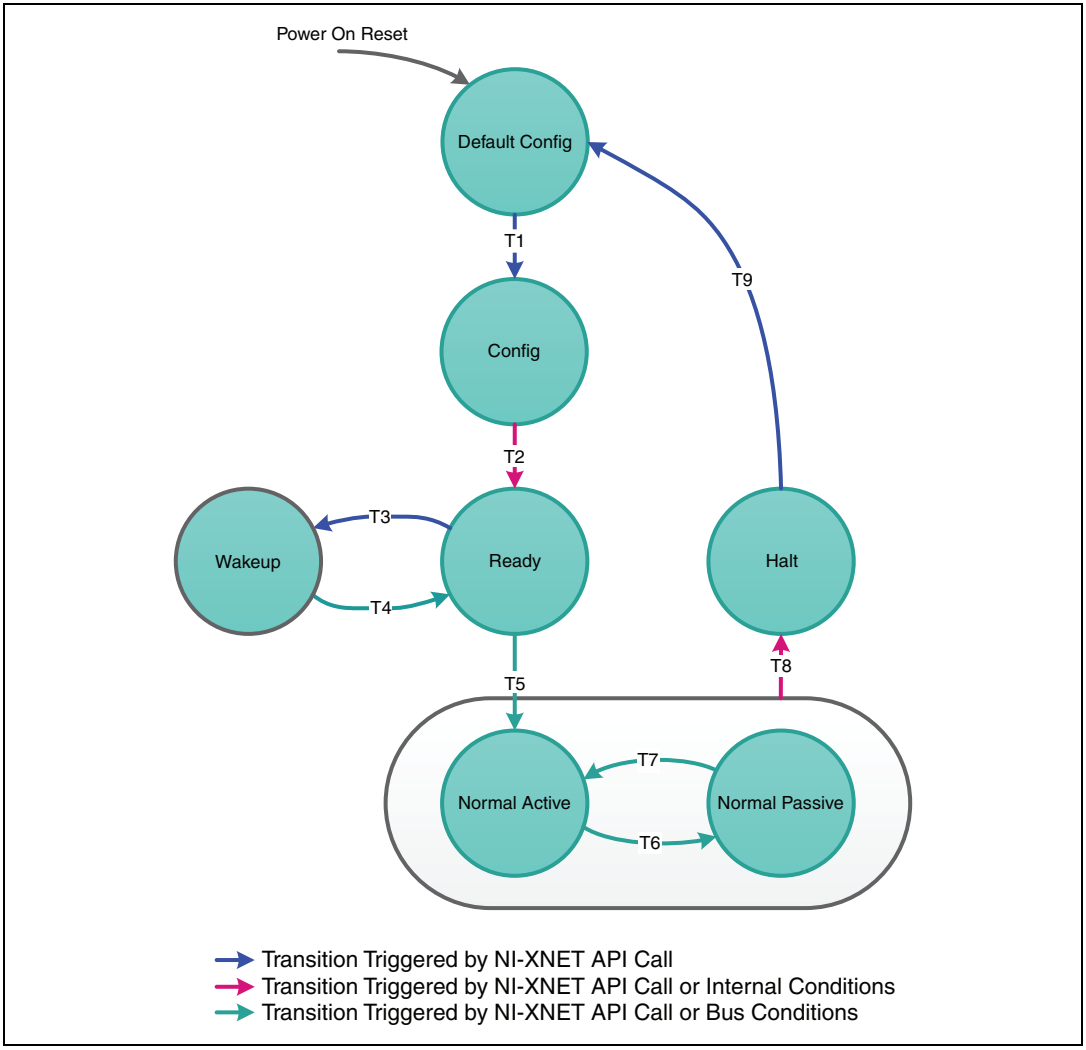
If your cluster does not support the wakeup mechanism, this process is straightforward. After creating your FlexRay session, call **XNET Start.vi**, which causes the interface to transition from **Default Config** to **Ready**, where it attempts to integrate with the FlexRay cluster. If your node is 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 **XNET Stop.vi**, 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 **XNET Start.vi**, 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 Local Sleep. 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 **XNET Start.vi** 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 Remote Wake after you put your FlexRay interface to sleep. When you invoke the **XNET Start.vi** 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 **XNET Start.vi** 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 Remote Wake, 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 (Interface:FlexRay:Sleep property set to Remote Wake)	Ready	Wakeup
4	Wakeup channel awake	Wakeup	Ready

T#	Condition	From	To
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 (**XNET Stop.vi**)		
9	Interface stopped (**XNET Stop.vi**)	Halt	Default Config

[1]If you are not using synchronization, the **XNET Start.vi** 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.

LIN

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 **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 **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 **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 **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 **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 **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 **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 **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 **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.

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 **XNET Create Session.vi**, followed by calls to **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 **XNET Create Session.vi** and **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 **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`) for CAN, FIBEX (`.xml`) for FlexRay, or LDF (`.ldf`) for LIN. 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.vi** 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.

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 *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`), LDF (`.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, uppercase letters, 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, uppercase letters, 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`) 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 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.

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, or FIBEX (`.xml`) for FlexRay. 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 *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`), LDF (`.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, uppercase letters, 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 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 Delete (LIN Schedule).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.

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).

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`), LDF (`.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, uppercase letters, 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`) 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 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.

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 *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`), LDF (`.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, uppercase letters, 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 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.

XNET Interface I/O Name

The XNET interface 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 have the NI interface connected to multiple clusters simultaneously.

The XNET interface I/O name is used to select an interface to pass to **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 *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 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, or LIN 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 either CAN for a CAN interface or FlexRay for a FlexRay 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 **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 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).

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 **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 **XNET Read.vi** or **XNET Write.vi**. For general information about I/O names, such as when to use them, refer to *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 **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 Property Node**: Use the 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.vi**, **XNET Stop.vi**, and **XNET Flush.vi**: Control the session and buffer states.
- **XNET Wait.vi** and **XNET Create Timing Source.vi**: Handle notification of events that occur in the session.
- **XNET Connect Terminals.vi** and **XNET Disconnect Terminals.vi**: 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).

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 *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`), LDF (`.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, uppercase letters, 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 **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 signal property node provides information such as the signal position and size in the payload, scaling formula to physical units, and so on.

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 **XNET Database Create (Signal).vi** using the parent XNET Frame (not the subframe).

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.vi** or **XNET Disconnect Terminals.vi**, both of which operate on the session. For general information about I/O names, such as when to use them, refer to *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 **XNET Connect Terminals.vi**.

Lowercase letters, uppercase letters, 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 **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.

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`), LDF (`.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, uppercase letters, 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 LDF (`.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`).

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 **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.

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.

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`), LDF (`.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, uppercase letters, 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.

5

NI-XNET API for C

This chapter explains how to use the NI-XNET API for C and describes the NI-XNET C functions and properties.

Getting Started

This section helps you get started using NI-XNET for C. It includes information about using NI-XNET within 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...**.

Examples

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.

A few examples are suggested to get started with NI-XNET.

For CAN (at **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.

For FlexRay (at **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.

For LIN (at **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.

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.

Visual C++

The NI-XNET software supports Microsoft Visual C/C++ version 6 or later.

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*.

Examples

NI-XNET includes C examples that demonstrate a wide variety of use cases.

You can find examples for the C language in the MS Visual C subfolder of the `\Users\Public\Public Documents\National Instruments\NI-XNET\Examples` directory on Windows 7 or Windows Vista and the `\Documents and Settings\All Users\Shared Documents\National Instruments\NI-XNET\Examples` directory on Windows XP. Each example is in a separate folder. A description of each example is in comments at the top of the `.c` file.

Interfaces

What Is an Interface?

The interface represents a single CAN, FlexRay, or LIN 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 either *CAN* for a CAN interface, *FlexRay* for a FlexRay interface, or *LIN* for a LIN 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, and a two-port LIN device in your system, the interface names are *CAN1*, *CAN2*, *FlexRay1*, *FlexRay2*, *LIN1*, and *LIN2*, respectively.

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. 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, programs written to use *CAN1* work with either hardware configuration without change.

How Do I View Available Interfaces?

Measurement and Automation Explorer (MAX)

Use 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 with the hardware product name, such as *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*. When you select a port, the right window shows a picture of the device with the port circled and the port LED blinking. The blinking LED 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).

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.

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.

Databases

What Is a Database?

For the 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`) for CAN, FIBEX (`.xml`) for FlexRay, or LIN Description File (`.ldf`) for LIN. 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.

What Is an 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.

Database Programming

The NI-XNET software provides various methods for creating your application database configuration. Figure 5-1 shows a process for deciding the database source. A description of each step in the process follows the flowchart.

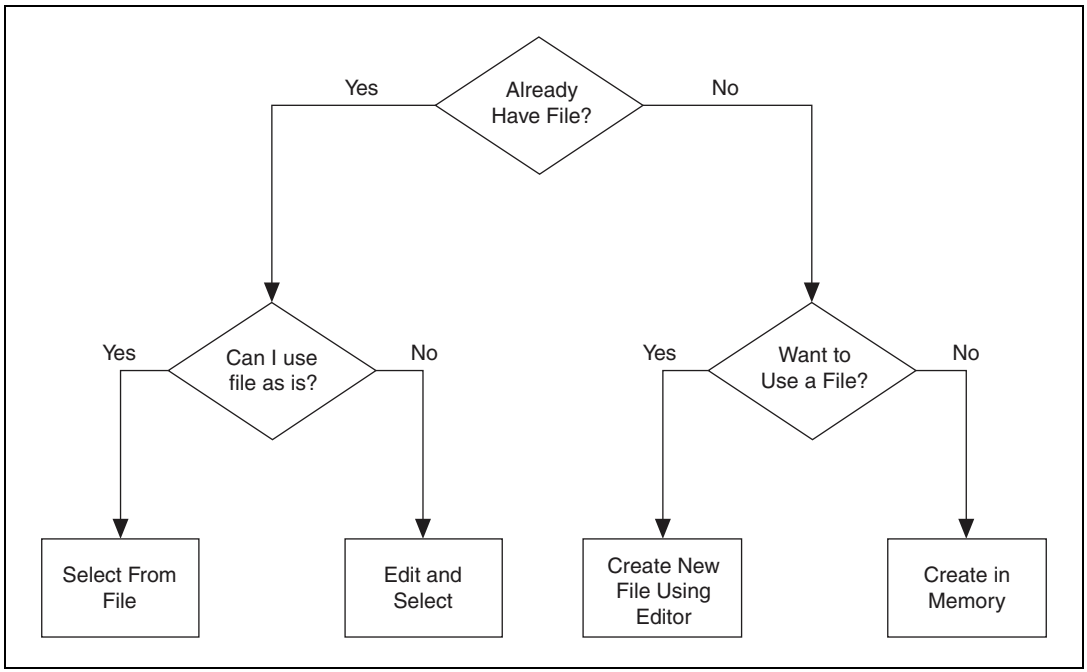


Figure 5-1. Decision Process for Choosing Database Source

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, 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 for the NI-XNET session: 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 the Database 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.

Sessions

What Is a Session?

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

Each session configuration includes:

- **Interface**: This specifies the National Instruments hardware to use. (Refer to *What Is an Interface?*)
- **Database objects**: These describe how external hardware communicates. (Refer to *What Is a Database?*)
- **Mode**: This specifies the direction and representation of I/O data. (Refer to *Session Modes*.)

The links above link to detailed information about configuration. The *Session Modes* section has additional links to sections 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.

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 Mode: 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 Mode: 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 Mode: 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 Mode: 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 Mode: 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 Mode: 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 Mode: 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 Mode: 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 Mode: 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 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.
- Frame Output Queued Mode: Provides a sequence of values for a single frame, for transmit using that frame's timing as the database specifies.

- Frame Output Single-Point Mode: 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 Mode: This mode does not use any hardware. It is used to convert data between the signal representation and frame representation.

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 `nxReadFrame` returns values for that frame only. If you need sequential data for multiple frames, create multiple sessions, one per frame.

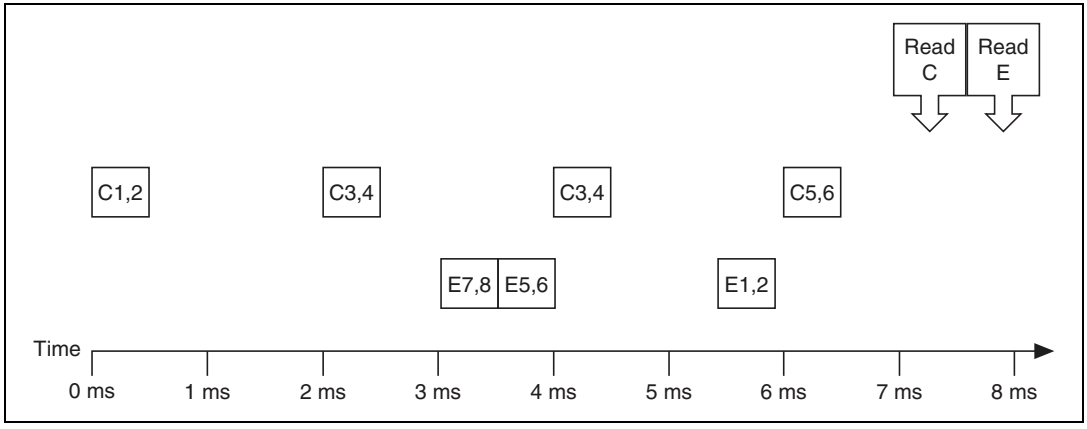
The input data is returned as an array of frame values. These values represent all values received for the frame since the previous call to `nxReadFrame`.

Example

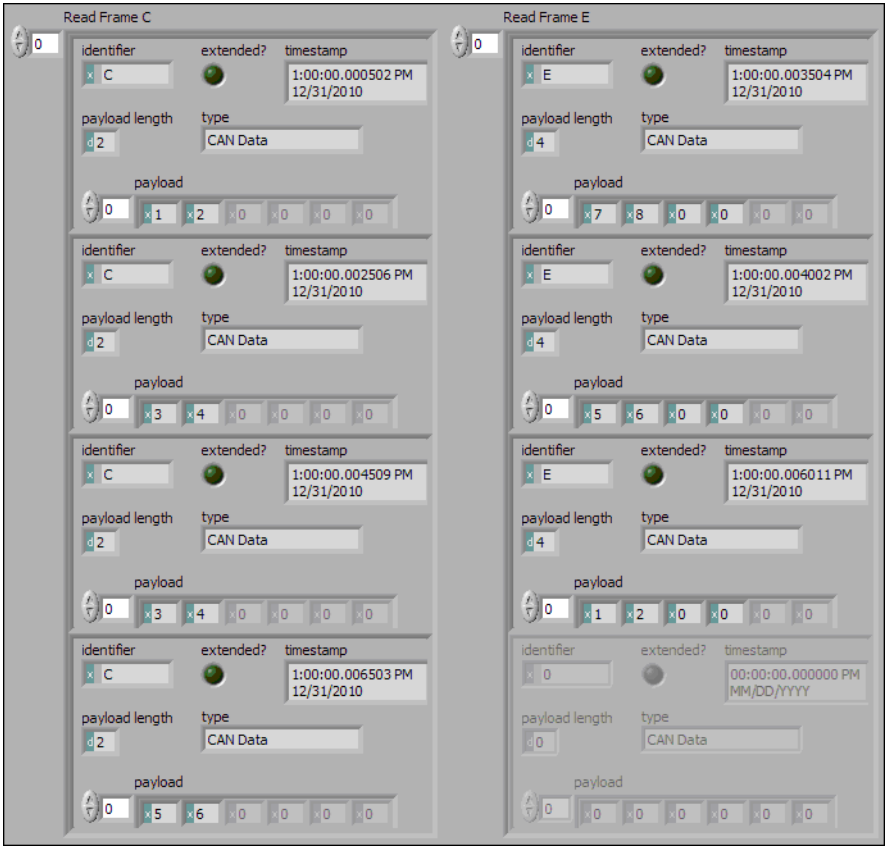
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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

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



The following figure shows the data returned from the two calls to `nxReadFrame` (two different sessions).



The first call to `nxReadFrame` returned an array of values for frame C, and the second call to `nxReadFrame` returns 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 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.

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.

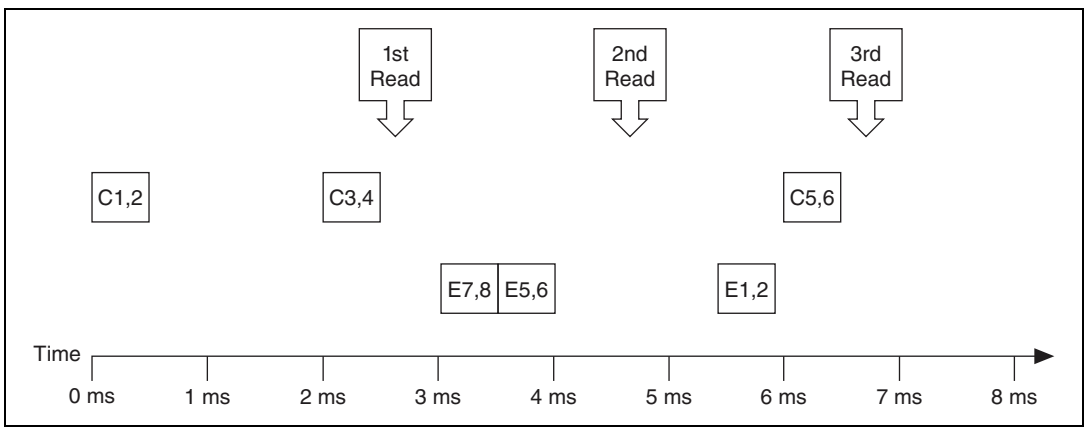
The input data is returned as an array of frames, one for each frame specified for the session.

Example

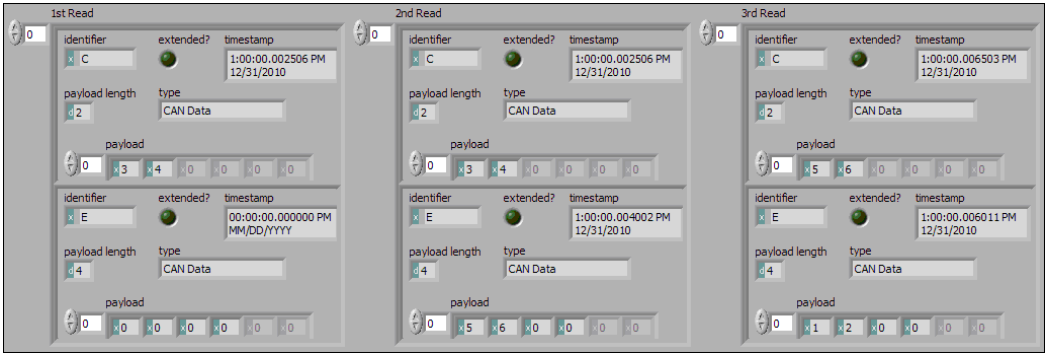
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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.

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.

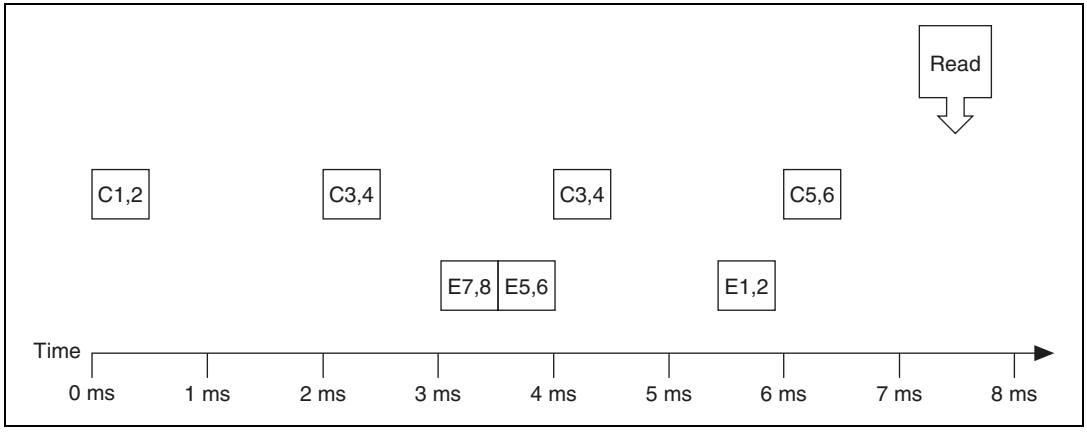
When used with a FlexRay interface, frames from both channels are returned. For example, if a frame is received in a static slot on both channel A and channel B, two frames are returned from `nxReadFrame`.

Example

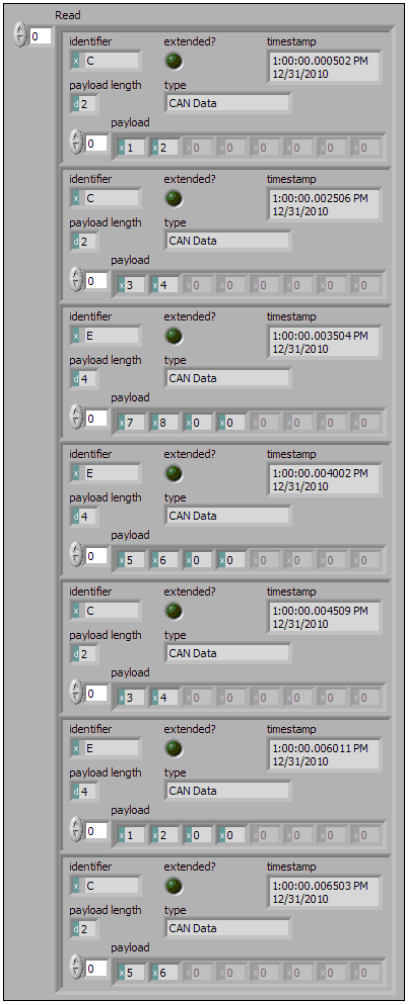
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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.

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.

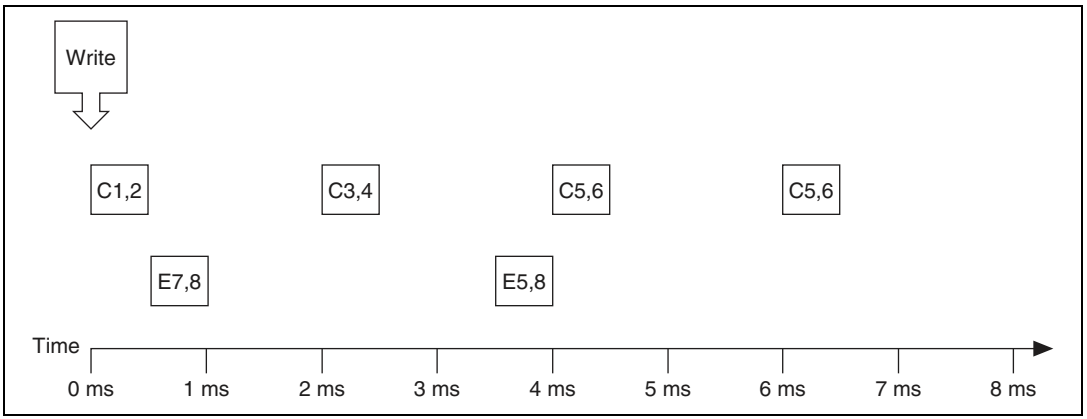
For this mode, NI-XNET transmits each frame according to its properties in the database. Therefore, when you call `nxWriteFrame`, the number of payload bytes in each frame value must match that frame's Payload Length property. The other frame value elements are ignored, so you can leave them uninitialized. 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

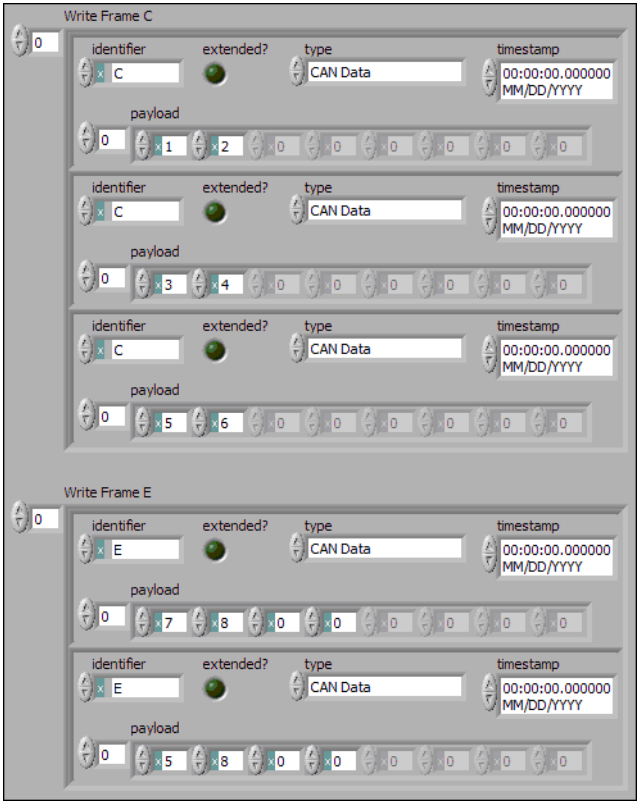
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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.

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.

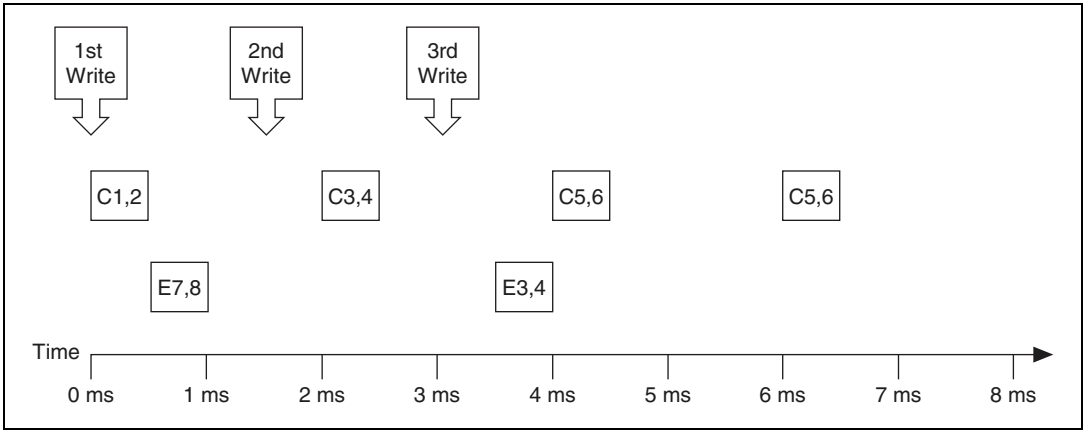
For this mode, NI-XNET transmits each frame according to its properties in the database. Therefore, when you call nxWriteFrame, the number of payload bytes in each frame value must match that frame's Payload Length property. The other frame value elements are ignored, so you can leave them uninitialized. 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

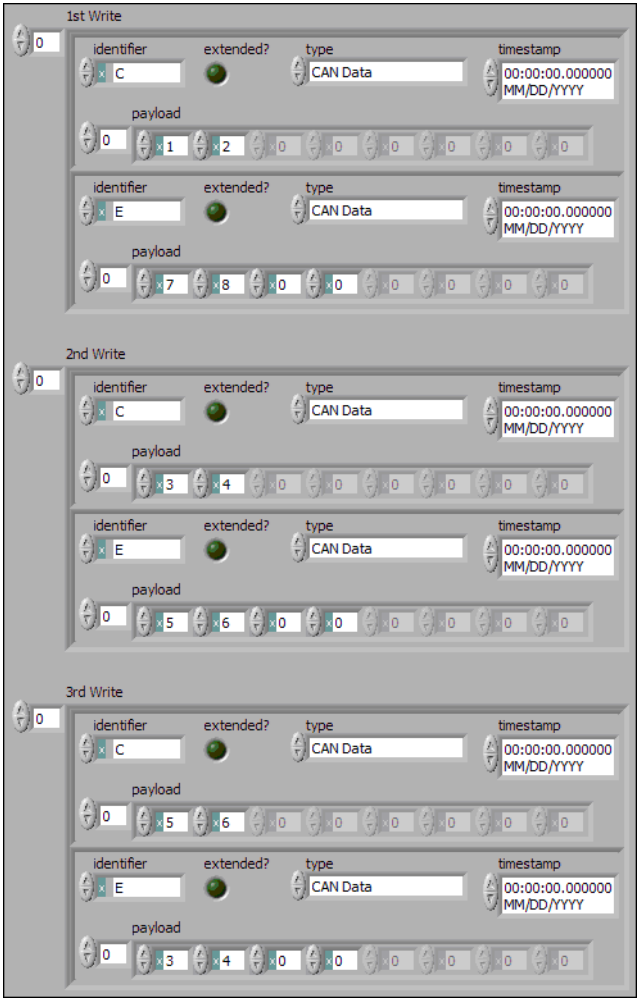
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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).

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 nonempty payload, the header + response for the frame is transmitted. If a frame for transmit is defined in the database (in-memory or otherwise), it is transmitted using its database checksum type. If the frame for transmit is not defined in the database, it is transmitted using enhanced checksum.

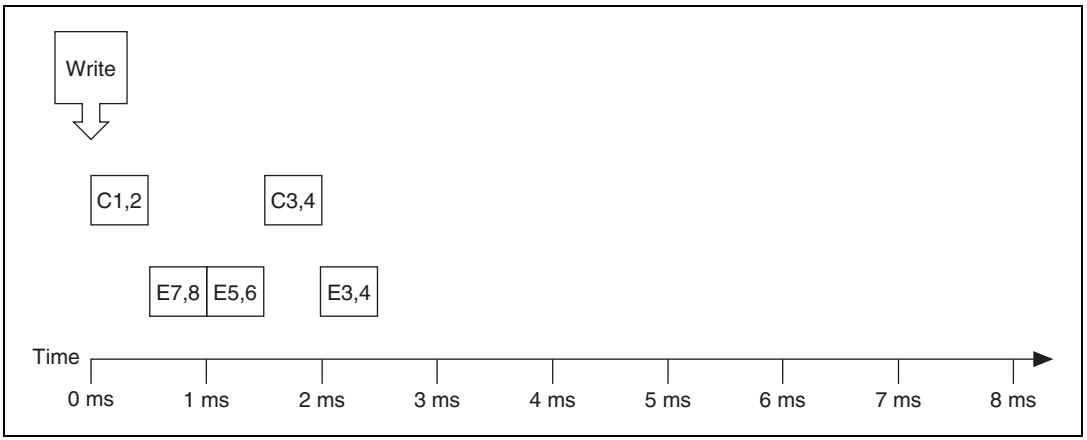
The frame values for this mode are stored in a queue, such that every value provided is transmitted.

Example

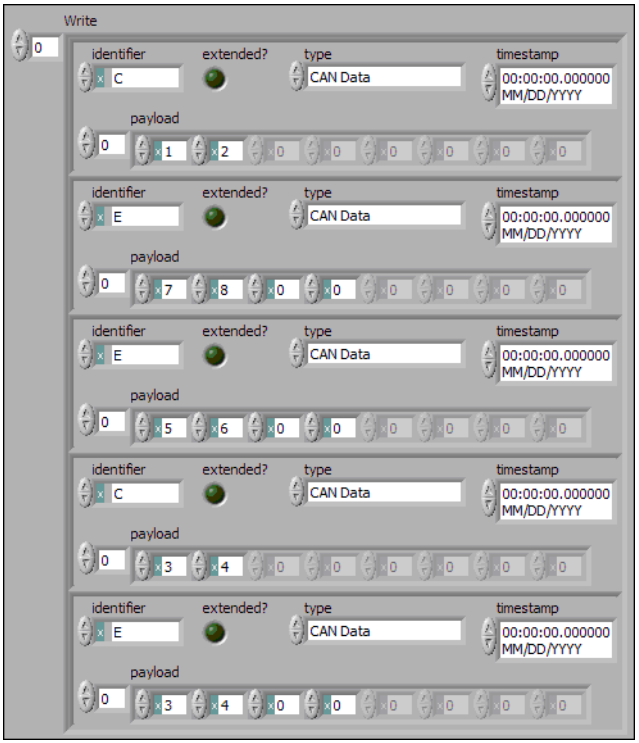
In this example CAN database, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven CAN frames, refer to *Cyclic and Event Timing*.

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.

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.

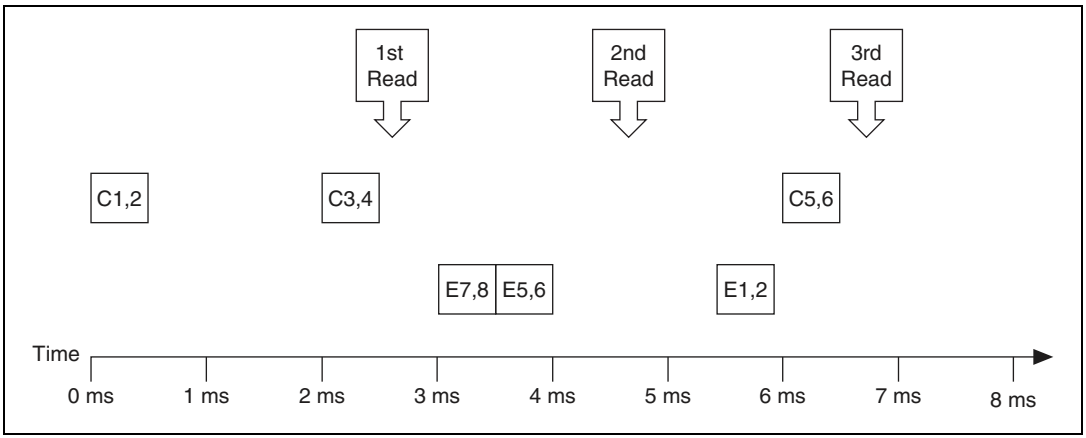
You also can specify a trigger signal for a frame. This signal name is *:trigger:.<frame name>*, and once it is specified in the `nxCreateSession` signal list, it returns a value of 0.0 if the frame did not arrive since the last Read (or Start), and 1.0 if at least one frame of this ID arrived. You can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

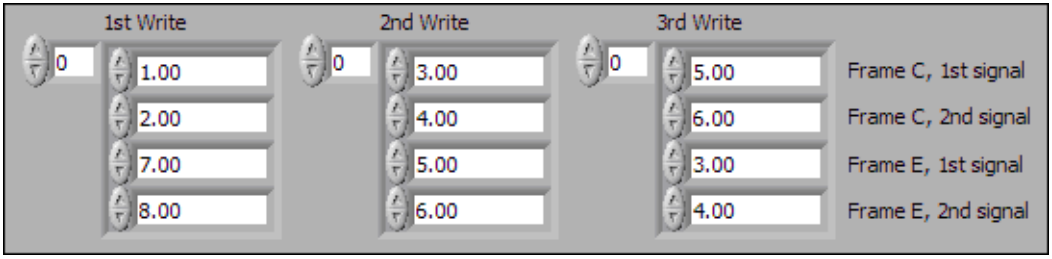
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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.

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.

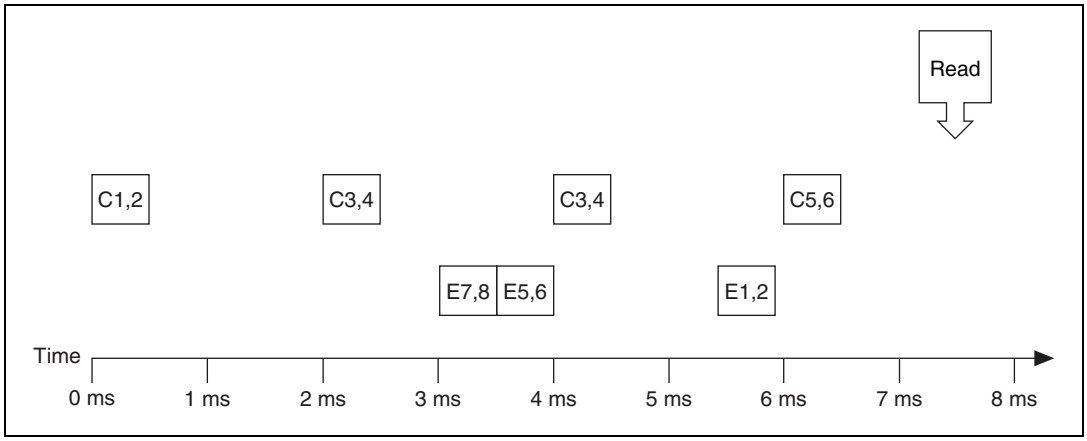
You specify the resample rate using the XNET Session Resample Rate property.

Example

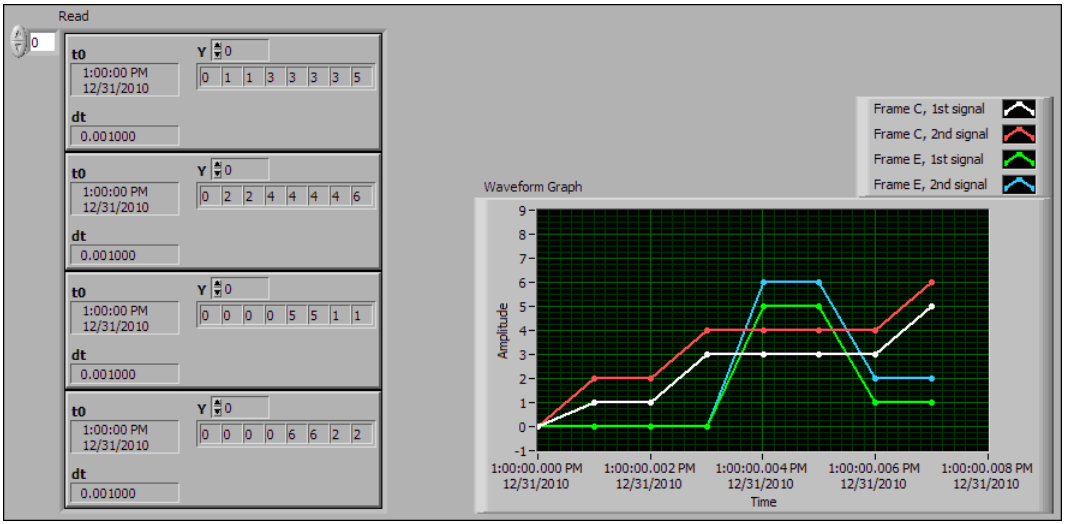
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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.

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.

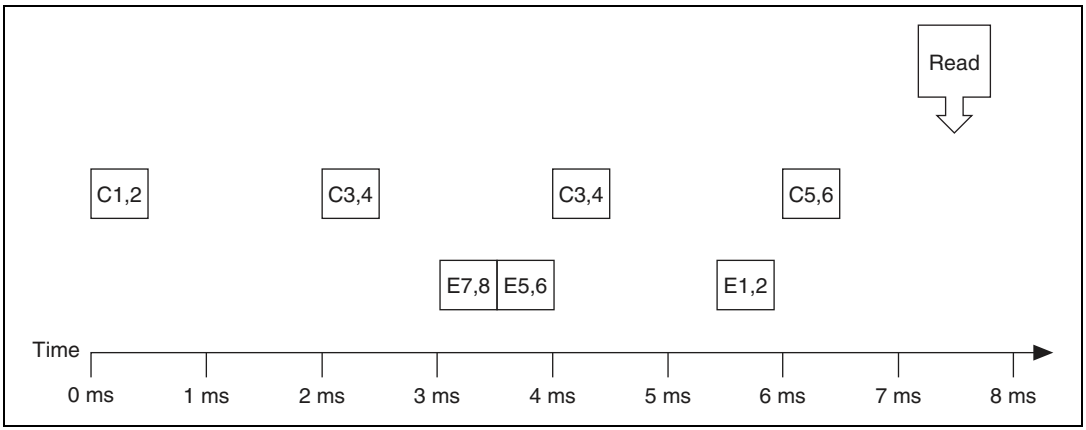
The received frames for this mode are stored in queues to avoid signal data loss.

Example

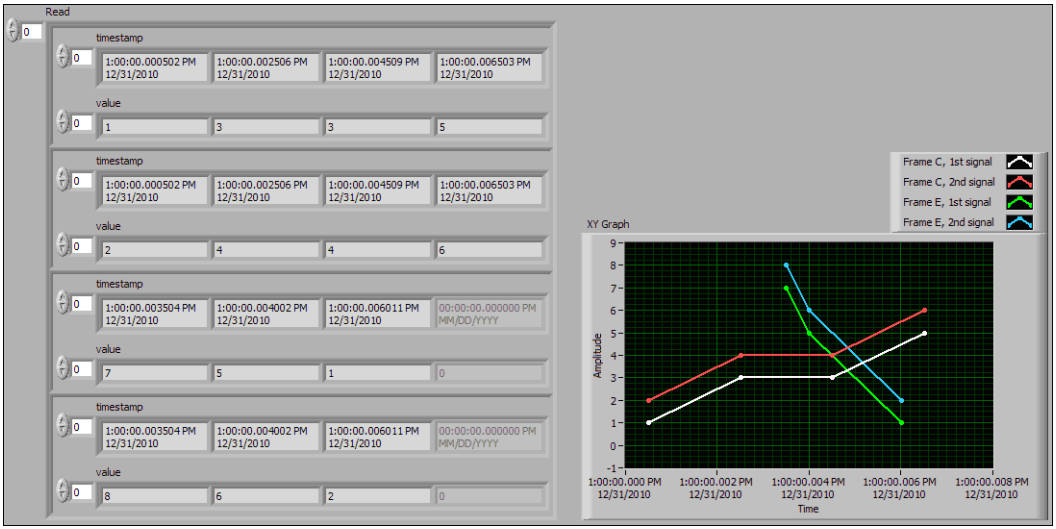
In this example network, frame C is a cyclic frame that transmits on the network once every 2 ms. Frame E is an event-driven frame. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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.

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.

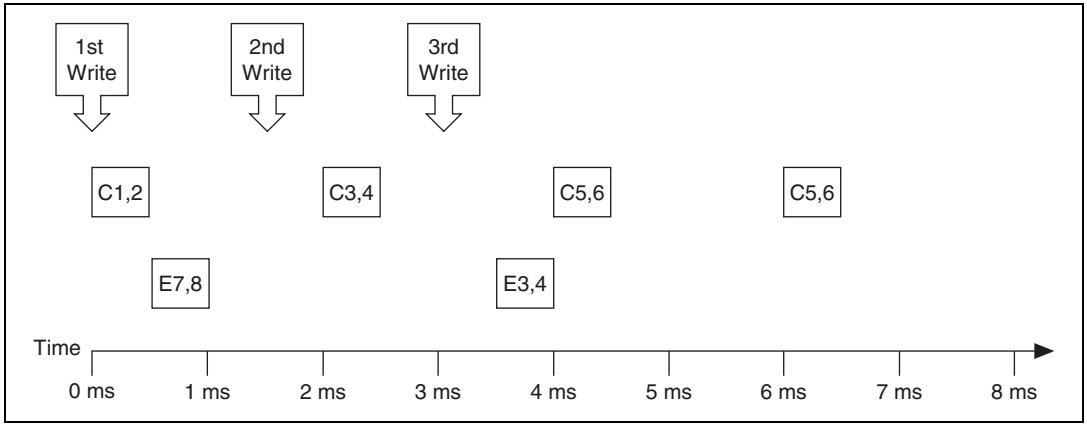
You also can specify a trigger signal for a frame. This signal name is *:trigger:.<frame name>*, and once it is specified in the `nxCreateSession` signal list, you can write a value of 0.0 to suppress writing of that frame, or any value not equal to 0.0 to write the frame. You can specify multiple trigger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

Example

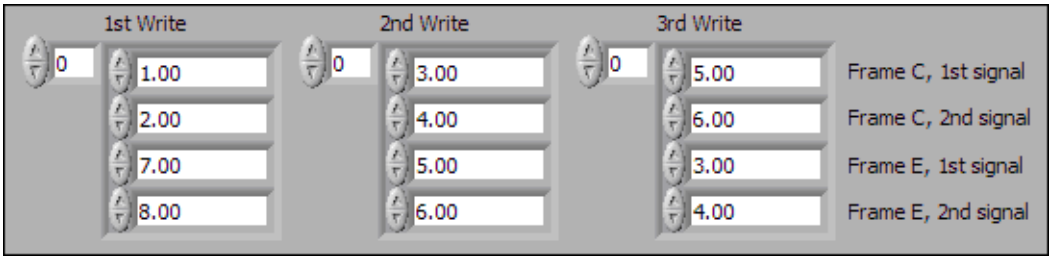
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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).

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.

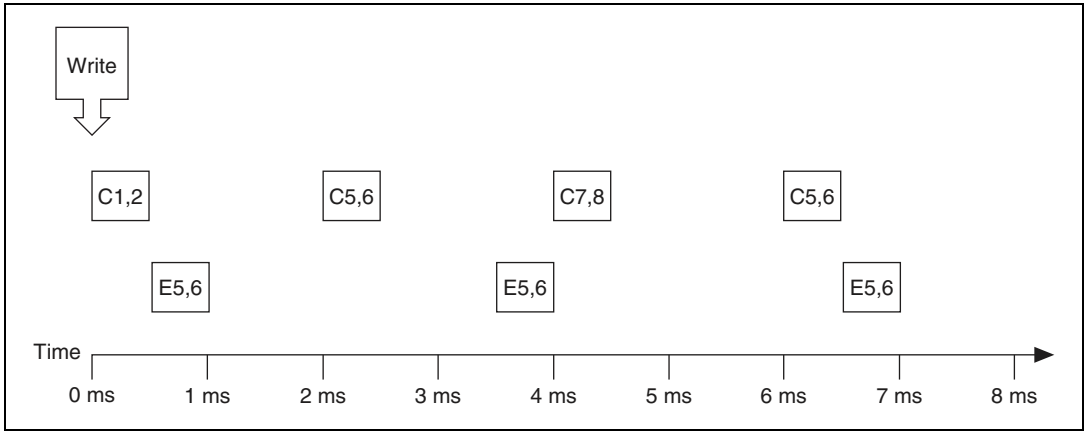
This mode is not supported for a LIN interface operating as slave. For more information, refer to *LIN Frame Timing and Session Mode*.

Example

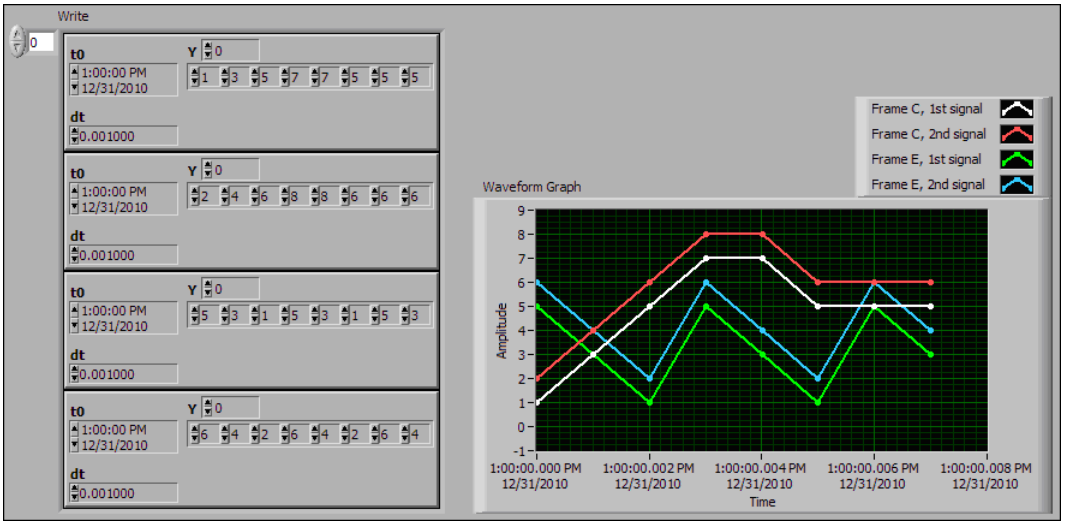
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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 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 Mode is the recommended mode for writing a sequence of signal values.

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.

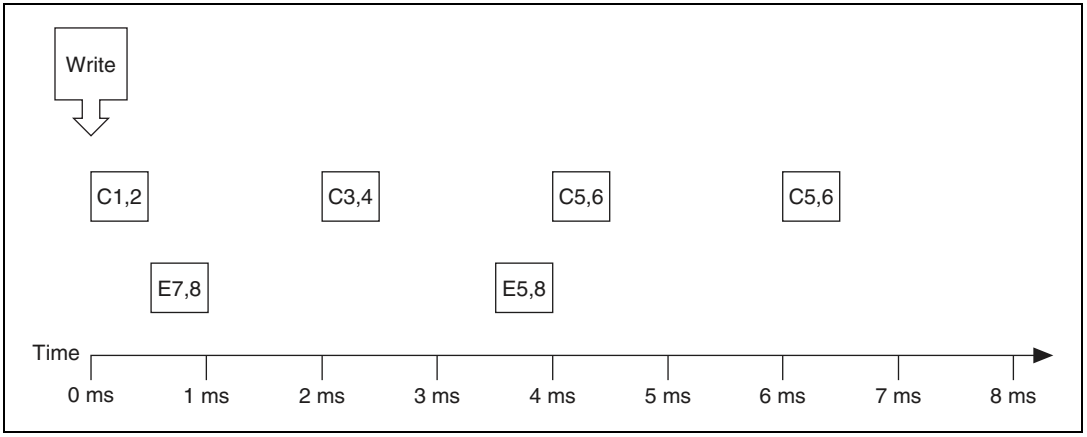
The frames for this mode are stored in queues, such that every signal provided is transmitted in a frame.

Examples

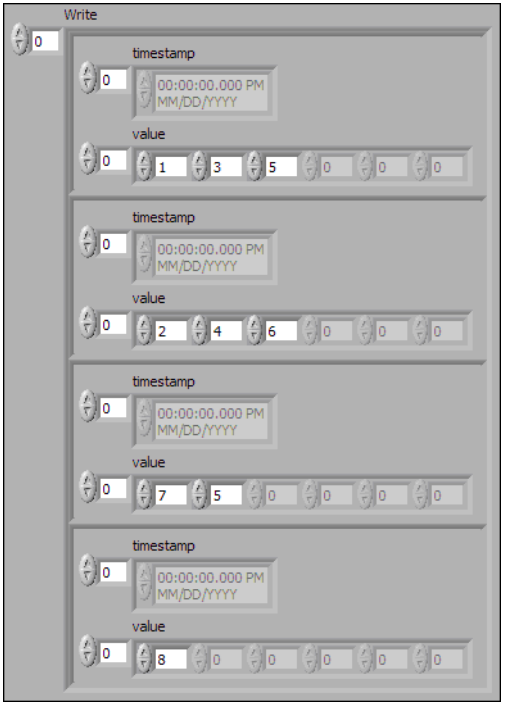
In this example network, frame C is a cyclic frame that transmits on the network once every 2.0 ms. Frame E is an event-driven frame that uses a transmit time (minimum interval) of 2.5 ms. For information about cyclic and event-driven frames, refer to *Cyclic and Event Timing*.

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.

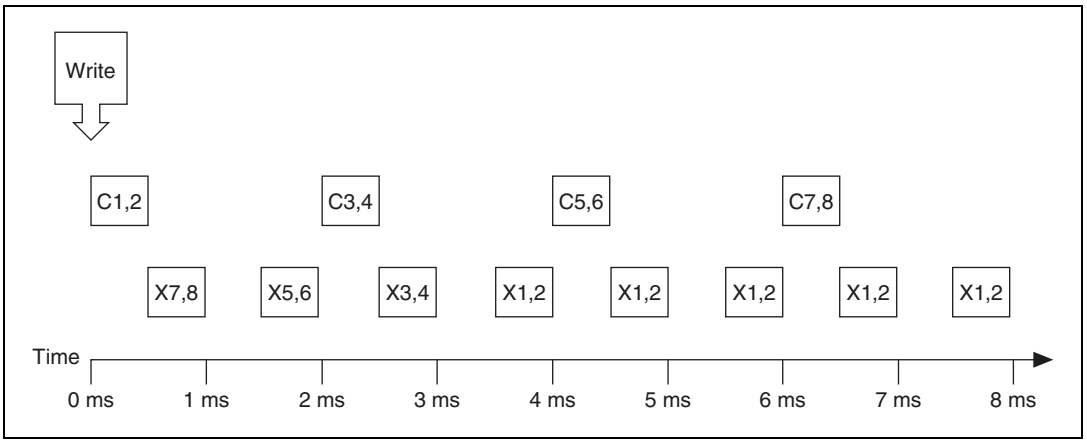
At 4.0 ms in the timeline, signal values 5 and 6 are taken from index 2 of the first two Y arrays and used for transmit of frame C.

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.

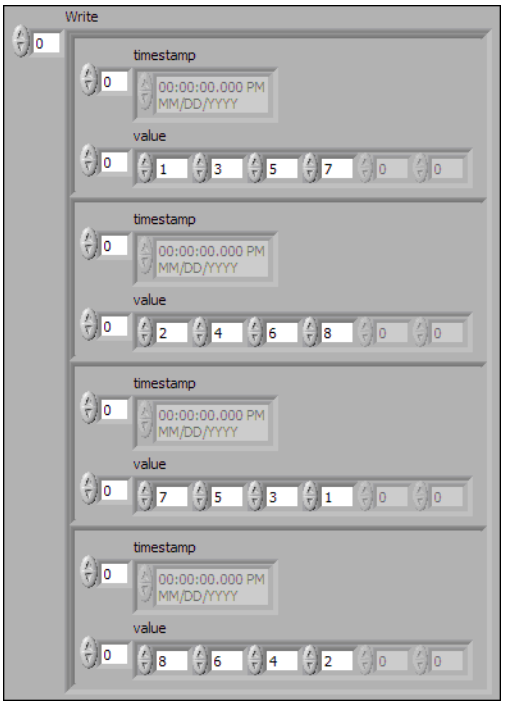
Because frame C is a cyclic frame, it transmits repeatedly. Although there are no more signal values for frame C, the values of the previous frame are used again at 6.0 ms in the timeline and every 2.0 ms thereafter. If `nxWriteSignalXY` is called again, the new signal values are used.

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.

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.

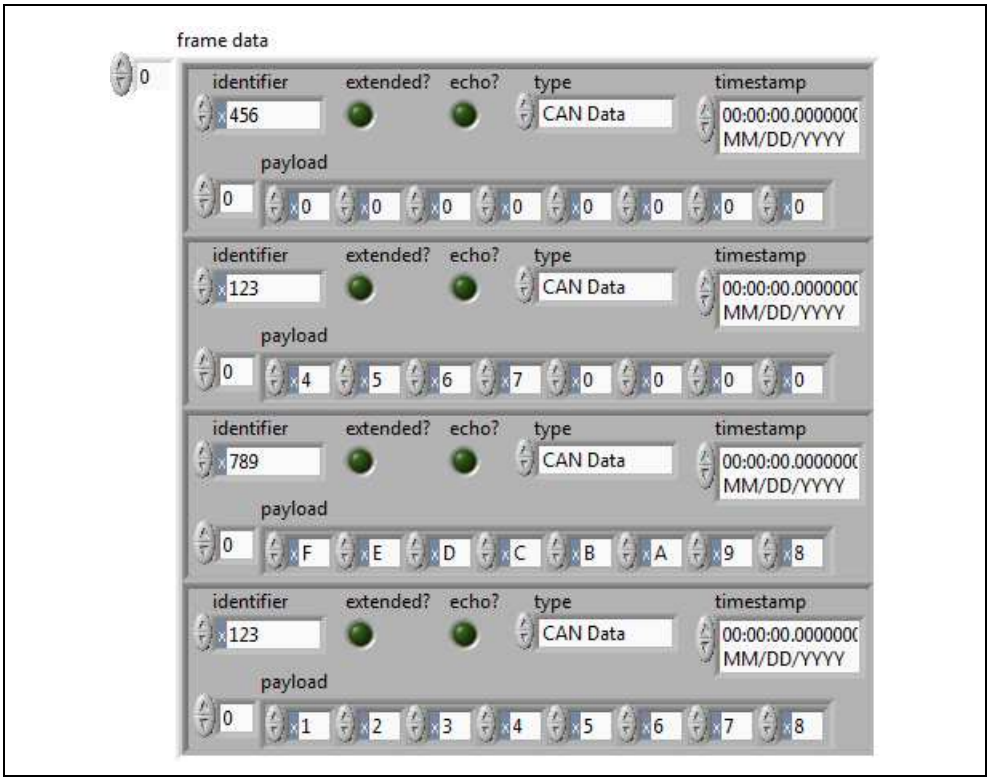
Conversion occurs with the `nxConvertFramesToSignalsSinglePoint` or `nxConvertSignalsToFramesSinglePoint` functions. None of the Read or Write functions work with this mode; they return an error because hardware I/O is not permitted.

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.

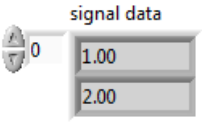
Example 1: Conversion of CAN Frames to Signals

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:

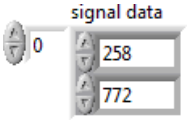


Explanation: The data are taken from frame 4. Frames 1 and 3 are ignored because they have a wrong (unmatched) ID. Frame 2 is ignored because its data are overwritten later with the values from frame 4, because frames are processed in the order of input.

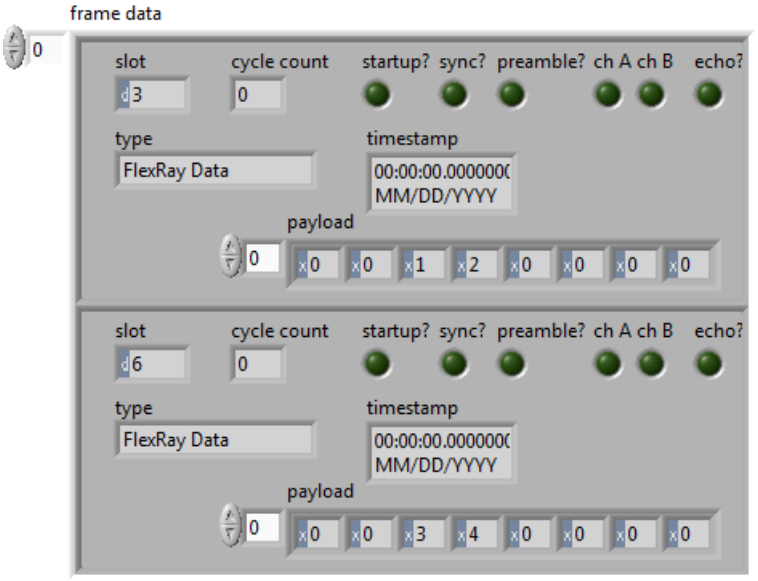
Example 2: Conversion of Signals to FlexRay Frames

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 × 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 × 256 + 4) and placed at byte 2 of the frame with slot ID 6. All other data are filled with the default values (0).

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.

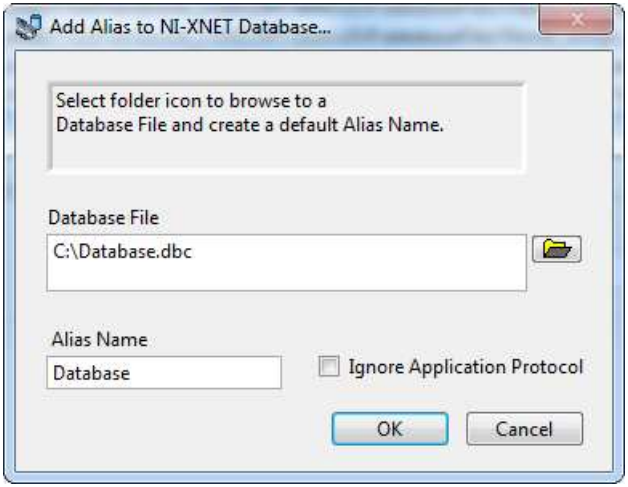
FIBEX databases do not define support for J1939 in the standard. If you save a J1939 database to FIBEX in the NI-XNET Database Editor or with the `nxdbSaveDatabase` API function, the J1939 properties are saved in a FIBEX extension defined by National Instruments in the FIBEX XML file.

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. 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.



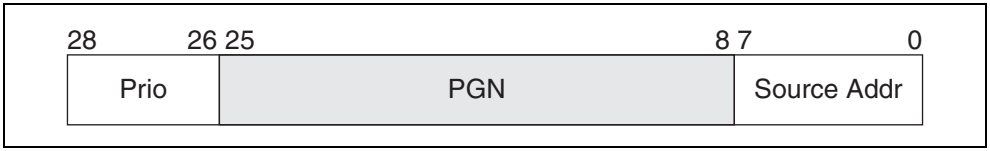
5. To have NI-XNET interpret the alias as an alias for a J1939 database, leave **Ignore Application Protocol** unchecked. To have NI-XNET interpret the alias as an alias for a plain CAN database, check **Ignore Application Protocol**.
6. Click **OK** to complete the alias addition.

J1939 Basics

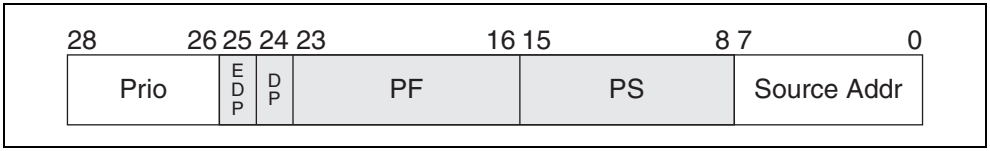
A J1939 network consists of ECUs connected by a CAN bus running at 250 k baud rate. Some newer networks might use a 500 k baud rate. A physical ECU can contain one or more logical ECUs called nodes or Controller Applications. This description refers to it as a node or ECU.

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 transmitting XNET session requires a node 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.

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.

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.

Transmitting Frames

When transmitting frames, the granted address of the node automatically replaces the source address part of the identifier.

Transmitting Frames without Granted Node Address

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. If a transmitting session without an address tries to send a frame without a valid address in the identifier, this returns an error.

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 with the `nxReadState` 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.

NI-XNET API for C Reference

This section describes the NI-XNET C functions and properties.

Functions

This section includes the NI-XNET functions.

nxBlink

Purpose

Blinks LEDs for the XNET interface to identify its physical port in the system.

Format

```
nxStatus_t _NXFUNC nxBlink (
    nxSessionRef_t InterfaceRef,
    u32 Modifier);
```

Inputs

`nxSessionRef_t InterfaceRef`

The XNET Interface I/O name.

`u32 Modifier`

Controls LED blinking:

`Disable (0)`

Disable blinking for identification. This option turns off both LEDs for the port.

`Enable (1)`

Enable blinking for identification. Both LEDs of the interface's physical port turn on and off. The hardware blinks the LEDs automatically until you disable, so there is no need to call the `nxBlink` function repetitively.

Both LEDs blink green (not red). The blinking rate is approximately three times per second.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

Each XNET device contains one or two physical ports. Each port is labeled on the hardware as *Port 1* or *Port 2*. The XNET device also provides two LEDs per port. For a two-port board, LEDs 1 and 2 are assigned to Port 1, and LEDs 3 and 4 are assigned to physical Port 2.

When your application uses multiple XNET devices, this function helps to identify each interface to associate its software behavior to its hardware connection (port). Prior to running your XNET sessions, you can call this function to blink the interface LEDs.

For example, if you have a system with three PCI CAN cards, each with two ports, you can use this function to blink the LEDs for interface CAN4, to identify it among the six CAN ports.

The LEDs of each port support two states:

- **Identification**: Blink LEDs to identify the physical port assigned to the interface.
- **In Use**: LED behavior that XNET sessions control.

Identification LED State

You can use the `nxBlink` function only in the Identification state. If you call this function while one or more XNET sessions for the interface are open (created), it returns an error, because the port's LEDs are in the In Use state.

In Use LED State

When you create an XNET session for the interface, the LEDs for that physical port transition to the In Use state. If you called the `nxBlink` function previously to enable blinking for identification, that LED behavior no longer applies. The In Use LED state remains until all XNET sessions are cleared. This typically occurs when the application terminates. The patterns that appear on the LEDs while In Use are documented in the *LEDs* section of Chapter 3, *NI-XNET Hardware Overview*.

nxClear

Purpose

Clears (closes) the XNET session.

Format

```
nxStatus_t nxClear (
    nxSessionRef_t SessionRef);
```

Inputs

`nxSessionRef_t SessionRef`

The reference to the session to clear. This session reference is returned from `nxCreateSession`.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function stops communication for the session and releases all resources the session uses. `nxClear` internally calls `nxStop` with normal scope, so if this is the last session using the interface, communication stops.

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

nxConnectTerminals

Purpose

Connects terminals on the XNET interface.

Format

```
nxStatus_t _NXFUNC nxConnectTerminals (
    nxSessionRef_t SessionRef,
    const char * source,
    const char * destination);
```

Inputs

`nxSessionRef_t SessionRef`

The reference to the session to use for the connection.

`const char * source terminal`

The connection source name.

`const char * destination terminal`

The connection destination name.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

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

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

Valid Combinations of Source/Destination

The following table lists all valid combinations of `source terminal` and `destination terminal`.

Source	Destination				
	PXI_Trig*x*	**FrontPanel0 FrontPanel1**	**Start Trigger**	**Master Timebase**	**Log Trigger**
PXI_Trig*x*	X	X	✓	✓	✓
FrontPanel0 FrontPanel1	X	X	✓	✓	✓
PXI_Star[1]	X	X	✓	X	X
PXI_Clk10[1]	X	X	X	✓	X
StartTrigger	✓	✓	X	X	X
CommTrigger	✓	✓	X	X	X
FlexRayStartCycle[2]	✓	✓	X	X	X
FlexRayMacrotick[2]	✓	✓	X	✓	X
1MHzTimebase	✓	✓	X	X	X
10MHzTimebase	✓	X	X	X	X

[1] Valid only on PXI hardware.

[2] Valid only on FlexRay hardware.

Source Terminals

The following table describes the valid `source terminals`.

Source Terminal	Description
PXI_Trig*x*	Selects a general-purpose trigger line as the connection source (input), where *x* is a number from 0 to 7. For PCI cards, these are the RTSI lines. For PXI cards, these are the PXI Trigger lines. For C Series modules in a CompactDAQ chassis, all modules in the chassis automatically share a common timebase. For information about routing the StartTrigger for CompactDAQ, refer to the XNET Session Interface:Source Terminal:Start Trigger property.
FrontPanel0 FrontPanel1	Selects a general-purpose Front Panel Trigger line as the connection source (input).
PXI_Star	Selects the PXI star trigger signal. Within a PXI chassis, some PXI products can source star trigger from Slot 2 to all higher-numbered slots. PXI_Star enables the PXI XNET hardware to receive the star trigger when it is in Slot 3 or higher. **Note**: You cannot use this terminal with a PCI device.
PXI_Clk10	Selects the PXI 10 MHz backplane clock. The only valid `destination terminal` for this source is MasterTimebase. This routes the 10 MHz PXI backplane clock for use as the XNET card timebase. When you use PXI_Clk10 as the XNET card timebase, you also must use PXI_Clk10 as the timebase for other PXI cards to perform synchronized input/output. **Note**: You cannot use this terminal with a PCI device.
StartTrigger	Selects the start trigger, which is the event set when the Start Interface transition occurs. The start trigger is the same for all sessions using a given interface. You can route the start trigger of this XNET card to the start trigger of other XNET or DAQ cards to ensure that sampling begins at the same time on both cards. For example, you can synchronize two XNET cards by routing StartTrigger as the `source terminal` on one XNET card and then routing StartTrigger as the `destination terminal` on the other XNET card, with both cards using the same PXI Trigger line for the connections.

Source Terminal	Description
CommTrigger	Selects the communicating trigger, which is the event set when the Comm State Running transition occurs. The communicating trigger is the same for all sessions using a given interface. You can route the communicating trigger of this XNET card to the start trigger of other XNET or DAQ cards to ensure that sampling begins at the same time on both cards. The communicating trigger is similar to a start trigger, but is used if your clock source is the FlexRayMacrotick, which is not available until the interface is properly integrated into the bus. Because you cannot generate a start trigger to another interface until the synchronization clock is also available, you can use this trigger to allow for the clock under this special circumstance.
FlexRayStartCycle	Selects the FlexRay Start of Cycles as an advanced trigger source. This generates a repeating pulse that external hardware can use to synchronize a measurement or other action with each FlexRay cycle. **Note**: You can use this terminal only with a FlexRay device.
FlexRayMacrotick	Selects the FlexRay Macrotick as a timing source. The FlexRay Macrotick is the basic unit of time in a FlexRay network. You can use this `source terminal` to synchronize other measurements to the actual time on the FlexRay bus. In this scenario, you would configure the FlexRayMacrotick as the `source terminal` and route it to a PXI Trigger or front panel terminal. After the interface is communicating to the FlexRay network, the Macrotick signal becomes available. You also can connect the FlexRayMacrotick to the MasterTimebase. This configures the counter that timestamps received frames to run synchronized to FlexRay time, and also allows you to read the FlexRay cycle macrotick to do additional synchronization with the FlexRay bus in your application. **Note**: You can use this terminal only with a FlexRay device.

Source Terminal	Description
1MHzTimebase	Selects the XNET card's local 1 MHz oscillator. The only valid `destination terminals` for this source are PXITrigger0–PXITrigger7. This **source terminal** routes the XNET card local 1 MHz clock so that other NI cards can use it as a timebase. For example, you can synchronize two XNET cards by connecting 1MHzTimebase to PXI_Trig*x* on one XNET card and then connecting PXI_Trig*x* to MasterTimebase on the other XNET card.
10MHzTimebase	Selects the XNET card's local 10 MHz oscillator. This routes the XNET card local 10 MHz clock for use as a timebase by other NI cards. For example, you can synchronize two XNET cards by connecting 10MHzTimebase to PXI_Trig*x* on one XNET card and then connecting PXI_Trig*x* to MasterTimebase on the other XNET card.

Destination Terminals

The following table describes the valid `destination terminals`.

Destination Terminal	Description
PXI_Trig*x*	Selects a general-purpose trigger line as the connection destination (output), where *x* is a number from 0 to 7. For PCI cards, these are the RTSI lines. For PXI cards, these are the PXI Trigger lines. For C Series modules in a CompactDAQ chassis, all modules in the chassis automatically share a common timebase. For information about routing the StartTrigger for CompactDAQ, refer to the XNET Session Interface:Source Terminal:Start Trigger property.
FrontPanel0 FrontPanel1	Selects a general-purpose Front Panel Trigger line as the connection destination (output).

Destination Terminal	Description
StartTrigger	Selects the start trigger, which is the event that allows the interface to begin communication. The start trigger occurs on the first `source terminal` low-to-high transition. The start trigger is the same for all sessions using a given interface. This causes the Start Interface transition to occur. You can route the start trigger of another XNET or DAQ card to ensure that sampling begins at the same time on both cards. For example, you can synchronize with an M-Series DAQ MIO card by routing the AI start trigger of the MIO card to a RTSI line and then routing the same PXI Trigger line with StartTrigger as the `destination terminal` on the XNET card. The default (disconnected) state of this destination means the start trigger occurs when `nxStart` is invoked with the scope set to either Normal or Interface Only. Alternately, if Auto Start? is enabled, reading or writing to a session may start the interface.
MasterTimebase	MasterTimebase instructs the XNET card to use the connection `source terminal` as the master timebase. The XNET card uses this master timebase for input sampling (including timestamps of received messages) as well as periodic output sampling. Your XNET hardware supports incoming frequencies of 1 MHz, 10 MHz, and 20 MHz, and automatically detects the frequency without any additional configuration. For example, you can synchronize a CAN and DAQ M Series MIO card by connecting the 10 MHz oscillator (board clock) of the DAQ card to a PXI_Trig line, and then connecting the same PXI_Trig line as the `source terminal`. For PXI form factor hardware, you also can use PXI_Clk10 as the `source terminal`. This receives the PXI 10 MHz backplane clock for use as the master timebase. MasterTimebase applies separately to each port of a multiport XNET card, meaning you could run each port off of a separate incoming (or onboard) timebase signal.

Destination Terminal	Description
	If you are using a PCI board, the default connection to the Master Timebase is the local oscillator. If you are using a PXI board, the default connection to the MasterTimebase is the PXI_Clk10 signal, if it is available. Some chassis allow PXI_Clk10 to be turned off. In this case, the hardware automatically uses the local oscillator as the default MasterTimebase.
Log Trigger	The Log Trigger terminal generates a frame when it detects a rising edge. When connected, this frame is transferred into the queue of the Frame Stream Input session if the session is started. For information about this frame, including the interpretation of the frame payload, refer to *Special Frames*.

nxConvertFramesToSignalsSinglePoint

Purpose

Converts between NI-XNET frames and signals using a session of Conversion Mode.

Format

```
nxStatus_t nxConvertFramesToSignalsSinglePoint (
    nxSessionRef_t SessionRef,
    void * FrameBuffer,
    u32 NumberOfBytesForFrames,
    f64 * ValueBuffer,
    u32 SizeOfValueBuffer,
    nxTimestamp_t * TimestampBuffer,
    u32 SizeOfTimestampBuffer);
```

Inputs

`nxSessionRef_t SessionRef`

The session to convert. This session is returned from `nxCreateSession`. The session mode must be Conversion.

`void * FrameBuffer`

Provides the array of bytes, representing frames to convert.

The raw bytes encode one or more frames using the Raw Frame Format. This frame format is the same for read and write of raw data and also is used for log file examples.

For information about which elements of the raw frame are applicable, refer to *Raw Frame Format*.

The data you write is queued for transmit on the network. Using the default queue configuration for this mode, you can safely write 1536 frames if you have a sufficiently long timeout. To write more data, refer to the XNET Session Number of Values Unused property to determine the actual amount of queue space available for writing.

`u32 NumberOfBytesForFrames`

The size (in bytes) of the buffer passed to `FrameBuffer`. This is used to calculate the number of frames to convert.

`u32 SizeOfValueBuffer`

You should set this to the size (in bytes) of the array passed to `ValueBuffer`. If this is too small to fit one element for each signal in the session, an error is returned.

`u32 SizeOfTimestampBuffer`

You should set this to the size (in bytes) of the array passed to `TimestampBuffer`. If `TimestampBuffer` is not NULL, and this is too small to fit one element for each signal in the session, an error is returned.

Outputs

`f64* ValueBuffer`

Returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data returns the most recent value received for each signal. If multiple frames for a signal are received since the previous call to `nxReadSignalSinglePoint` (or session start), only signal data from the most recent frame is returned.

If no frame is received for the corresponding signals since you started the session, the XNET Signal Default Value is returned.

`nxTimestamp_t* TimestampBuffer`

Optionally returns a one-dimensional array of timestamp values of the times when the corresponding signal values arrived. Each timestamp value is the number of 100 ns increments since Jan 1, 1601 12:00 AM UTC.

You can pass `TimestampBuffer` as NULL; in this case, no timestamps are returned. You also should pass 0 to `SizeOfTimeStampBuffer` in this case.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

The frames passed into the `FrameBuffer` array are read one by one, and the signal values found are written to internal buffers for each signal. Frames are identified by their identifier (FlexRay: slot) field. Frames unknown to the session are silently ignored. After all frames in the `FrameBuffer` array are processed, the internal signal buffers' status is returned in the `ValueBuffer` array, and optionally, the corresponding timestamps from the frames where a signal value was found are returned in the `TimestampBuffer` array. The signal internal buffers' status is being preserved over multiple calls to this function.

This way, for example, data returned from multiple calls of `nxFrameRead` for a Frame Input Stream Mode session (or any other Frame Input session) can be passed to this function directly.

nxConvertSignalsToFramesSinglePoint

Purpose

Converts between NI-XNET signals and frames using a session of Conversion Mode.

Format

```
nxStatus_t nxConvertSignalsToFramesSinglePoint (
    nxSessionRef_t SessionRef,
    f64 * ValueBuffer,
    u32 SizeOfValueBuffer
    void * Buffer,
    u32 SizeOfBuffer,
    u32 * NumberOfBytesReturned);
```

Inputs

`nxSessionRef_t SessionRef`

The session to convert. This session is returned from `nxCreateSession`. The session mode must be Conversion.

`f64 * ValueBuffer`

Provides a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data provides the value for the conversion of each signal.

`u32 SizeOfValueBuffer`

You should set this to the size (in bytes) of the array passed to `ValueBuffer`. If this is too small to fit one element for each signal in the session, an error is returned.

`u32 SizeOfBuffer`

You should set this to the size (in bytes) of the array passed to `Buffer`.

This number does not represent the number of frames to convert. As encoded in raw data, each frame can vary in length. Therefore, the number represents the maximum raw bytes to be converted, not the number of frames.

For each frame used in the session, you must provide buffer space in the array passed to `Buffer`.

CAN and LIN frames are always 24 bytes in length. To convert a specific number of frames, multiply that number by 24.

FlexRay frames vary in length. For example, if you pass `SizeOfBuffer` of 91, the buffer may return 80 bytes, within which the first 24 bytes encode the first frame, and the next 56 bytes encode the second frame.

If `SizeOfBuffer` is positive, the data array size is no greater than this number. The minimum size for a single frame is 24 bytes, so you must use at least that number.

Outputs

`void * Buffer`

Returns an array of bytes.

The raw bytes encode one or more frames using the Raw Frame Format. This frame format is the same for read and write of raw data, and it is also used for log file examples.

The data always returns complete frames.

For each frame that appears in the session, exactly one frame is returned. If the buffer is not large enough to hold all the data, an error is returned.

`u32 * NumberOfBytesReturned`

Returns the number of valid bytes in the Buffer array.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

The signal values written to the `ValueBuffer` array are written to a raw frame buffer array. For each frame included in the session, one frame is generated in the array that contains the signal values. Signals not present in the session are written as their respective default values; empty space in the frames that signals do not occupy is written with the frame's default payload.

The frame header values are filled with appropriate values so that this function's output can be directly written to a Frame Output session.

nxCreateSession

Purpose

Creates an XNET session at run time using strings.

Format

```
nxStatus_t nxCreateSession (
    const char * DatabaseName,
    const char * ClusterName,
    const char * List,
    const char * Interface,
    u32 Mode,
    nxSessionRef_t * SessionRef);
```

Inputs

`const char * DatabaseName`

The XNET database to use for interface configuration. The database name must use the <*alias*> or <*filepath*> syntax (refer to *Databases*).

Three special values for this parameter exist:

- **:memory:**—This is the default in-memory database. You can create and manipulate it using the `nxdb...` functions. As long as you do not save its content to a real database file using `nxdbSaveDatabase`, its content is available to `nxCreateSession` with this special parameter. After you create the session, you must set the XNET Session Interface:Baud Rate property prior to starting the session.
- **:can_fd: or :can_fd_brs:**—These databases are similar to the default in-memory database, but configure the cluster in either CAN FD or CAN FD+BRS mode, respectively. After you create the session, you must set the XNET Session Interface:Baud Rate and Interface:CAN:FD Baud Rate properties prior to starting the session.
- **:can_j1939:**—This database is similar to the empty in-memory database (:memory:), but configures the cluster in CAN SAE J1939 application protocol mode. After you create the session, you must set the XNET Session Interface:Baud Rate property using a Session node. You must set this baud rate prior to starting the session.
- **:subordinate:**—This “database” is available only for a mode of `nxMode_FrameInStream`. A subordinate session uses the cluster and interface configuration from other sessions. For example, you may have a test application with which the end user specifies the database file, cluster, and signals to read/write. You also have a second application with which you want to log all received frames (input

stream), but that application does not specify a database. You run this second application using a subordinate session, meaning it does not configure or start the interface, but depends on the primary test application. For a subordinate session, start and stop of the interface (using the nxStart/nxStop functions) is ignored. The subordinate session reads frames only when another nonsubordinate session starts the interface.

`const char * ClusterName`

The XNET cluster to use for interface configuration. The name must specify a cluster from the database given in the `DatabaseName` parameter. If it is left blank, the cluster is extracted from the `List` parameter; this is not allowed for modes of `nxMode_FrameInStream` or `nxMode_FrameOutStream`.

`const char * List`

Provides the list of signals or frames for the session.

The `List` syntax depends on the mode:

Mode	List Syntax
`nxMode_SignalInSinglePoint`, `nxMode_SignalOutSinglePoint`	`List` contains one or more XNET Signal names. If more than one name is provided, a comma must separate each name. Each name must be one of the following options, whichever uniquely identifies a signal within the database given in the `DatabaseName` parameter: • *<Signal>* • *<Frame>.<Signal>* • *<Cluster>.<Frame>.<Signal>* • *<PDU>.<Signal>* • *<Cluster>.<PDU>.<Signal>* `List` may also contain one or more trigger signals. For information about trigger signals, refer to *Signal Output Single-Point Mode* or *Signal Input Single-Point Mode*.

Mode	List Syntax
nxMode_SignalInWaveform, nxMode_SignalOutWaveform	List contains one or more XNET Signal names. If more than one name is provided, a comma must separate each name. Each name must be one of the following options, whichever uniquely identifies a signal within the database given in the DatabaseName parameter: • <*Signal*> • <*Frame*>.<*Signal*> • <*Cluster*>.<*Frame*>.<*Signal*> • <*PDU*>.<*Signal*> • <*Cluster*>.<*PDU*>.<*Signal*>
nxMode_SignalInXY, nxMode_SignalOutXY	List contains one or more XNET Signal names. If more than one name is provided, a comma must separate each name. Each name must be one of the following uptions, whichever uniquely identifies a signal within the database given in the DatabaseName parameter: • <*Signal*> • <*Frame*>.<*Signal*> • <*Cluster*>.<*Frame*>.<*Signal*> • <*PDU*>.<*Signal*> • <*Cluster*>.<*PDU*>.<*Signal*>
nxMode_FrameInStream, nxMode_FrameOutStream	List is empty (“ ”).
nxMode_FrameInQueued, nxMode_FrameOutQueued	List contains only one XNET Frame or PDU name. Only one name is supported. Each name must be one of the following options, whichever uniquely identifies a frame within the database given in the DatabaseName parameter: • <*Frame*> • <*Cluster*>.<*Frame*> • <*PDU*> • <*Cluster*>.<*PDU*>

Mode	List Syntax
nxMode_FrameInSinglePoint, nxMode_FrameOutSinglePoint	List contains one or more XNET Frame or PDU names. If more than one name is provided, a comma must separate each name. Each name must be one of the following options, whichever uniquely identifies a frame within the database given in the DatabaseName parameter: • <*Frame*> • <*Cluster*>.<*Frame*> • <*PDU*> • <*Cluster*>.<*PDU*>
nxMode_SignalConversionSinglePoint	List contains one or more XNET Signal names. If more than one name is provided, a comma must separate each name. Each name must be one of the following options, whichever uniquely identifies a signal within the database given in the DatabaseName parameter: • <*Signal*> • <*Frame*>.<*Signal*> • <*Cluster*>.<*Frame*>.<*Signal*> • <*PDU*>.<*Signal*> • <*Cluster*>.<*PDU*>.<*Signal*>

`const char * Interface`

The XNET Interface to use for this session. If `Mode` is `nxMode_SignalConversionSinglePoint`, this input is ignored. You can set it to an empty string.

`u32 Mode`

The session mode. It can be one of the following constants defined in `nixnet.h`:

nxMode_SignalInSinglePoint	0
nxMode_SignalInWaveform	1
nxMode_SignalInXY	2
nxMode_SignalOutSinglePoint	3
nxMode_SignalOutWaveform	4
nxMode_SignalOutXY	5
nxMode_FrameInStream	6

nxMode_FrameInQueued	7
nxMode_FrameInSinglePoint	8
nxMode_FrameOutStream	9
nxMode_FrameOutQueued	10
nxMode_FrameOutSinglePoint	11
nxMode_SignalConversionSinglePoint	12

Outputs

nxSessionRef_t* SessionRef

Returns the handle to the session created. Pass this value to any other NI-XNET API functions.

Return Value

nxStatus_t

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function creates a session using the named database objects specified in List from the database named in DatabaseName.

nxCreateSessionByRef

Purpose

Creates an XNET session at run time using database references.

Format

```
nxStatus_t nxCreateSessionByRef (
    u32 NumberOfDatabaseRef,
    nxDatabaseRef_t * ArrayOfDatabaseRef,
    const char * Interface,
    u32 Mode,
    nxSessionRef_t * SessionRef);
```

Inputs

`u32 NumberOfDatabaseRef`

The number of references passed in `ArrayOfDatabaseRef`.

`nxDatabaseRef_t *ArrayOfDatabaseRef`

The array of database objects to be used in the session. This can be an array of signal references, an array of frame references, or a single cluster reference, depending on the mode:

Mode	ArrayOfDatabaseRef Syntax
`nxMode_SignalInSinglePoint`, `nxMode_SignalOutSinglePoint`	`ArrayOfDatabaseRef` contains one or more XNET Signal refs.
`nxMode_SignalInWaveform`, `nxMode_SignalOutWaveform`	`ArrayOfDatabaseRef` contains one or more XNET Signal refs.
`nxMode_SignalInXY`, `nxMode_SignalOutXY`	`ArrayOfDatabaseRef` contains one or more XNET Signal refs.
`nxMode_FrameInStream`, `nxMode_FrameOutStream`	`ArrayOfDatabaseRef` contains only one XNET Cluster ref.
`nxMode_FrameInQueued`, `nxMode_FrameOutQueued`	`ArrayOfDatabaseRef` contains only one XNET Frame or PDU ref.
`nxMode_FrameInSinglePoint`, `nxMode_FrameOutSinglePoint`	`ArrayOfDatabaseRef` contains one or more XNET Frame or PDU refs.

`const char * Interface`

The XNET Interface to use for this session.

`u32 Mode`

The session mode. It can be one of the following constants defined in `nixnet.h`:

`nxMode_SignalInSinglePoint`	0
`nxMode_SignalInWaveform`	1
`nxMode_SignalInXY`	2
`nxMode_SignalOutSinglePoint`	3
`nxMode_SignalOutWaveform`	4
`nxMode_SignalOutXY`	5
`nxMode_FrameInStream`	6
`nxMode_FrameInQueued`	7
`nxMode_FrameInSinglePoint`	8
`nxMode_FrameOutStream`	9
`nxMode_FrameOutQueued`	10
`nxMode_FrameOutSinglePoint`	11



Note You can use the `nxMode_FrameInQueued`, `nxMode_FrameInSinglePoint`, `nxMode_FrameOutQueued`, and `nxMode_FrameOutSinglePoint` modes for PDUs also.

Outputs

`nxSessionRef_t* SessionRef`

Returns the handle to the session created. Pass this value to any other NI-XNET API functions.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function creates a session using the referenced database objects from an open database specified in `ArrayOfDatabaseRef`.

nxdbAddAlias

Purpose

Adds a new alias to a database file.

Format

```
nxStatus_t _NXFUNC nxdbAddAlias (
    const char * DatabaseAlias,
    const char * DatabaseFilepath,
    u32        DefaultBaudRate);
```

Inputs

`const char * DatabaseAlias`

Provides the desired alias name. Unlike the names of other XNET database objects, the alias name can use special characters such as space and dash. If the alias name already exists, this function changes the previous filepath to the specified filepath.

`const char * DatabaseFilepath`

Provides the path to the CANdb, FIBEX, or LDF file.

`u32 DefaultBaudRate`

Provides the default baud rate, used when filepath refers to a CANdb database (`.dbc`) or an NI-CAN database (`.ncd`). These database formats are specific to CAN and do not specify a cluster baud rate. Use this default baud rate to specify a default CAN baud rate to use with this alias. If `Filepath` refers to a FIBEX database (`.xml`) or LIN LDF file, the `DefaultBaudRate` parameter is ignored. The FIBEX and LDF database formats require a valid baud rate for every cluster, and NI-XNET uses that baud rate as the default.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

NI-XNET uses alias names for database files. The alias names provide a shorter name for display, allow for changes to the file system without changing the application, and enable efficient deployment to LabVIEW Real-Time (RT) targets.

This function is supported on Windows only. For RT targets, you can pass the new alias to `nxdbDeploy` to transfer an optimized binary image of the database to the RT target. After deploying the database, you can use the alias name in any application for the Windows host and RT target.

nxdbCloseDatabase

Purpose

Closes the database.

Format

```
nxStatus_t _NXFUNC nxdbCloseDatabase (
    nxDatabaseRef_t DatabaseRef,
    u32 CloseAllRefs);
```

Inputs

`nxDatabaseRef_t DatabaseRef`

The reference to the database to close.

`u32 CloseAllRefs`

Indicates that a database open multiple times (refer to `nxdbOpenDatabase`) should be closed completely (`CloseAllRefs` = 1), or just the reference counter should be decremented (`CloseAllRefs` = 0), and the database remains open. When the database is closed completely, all references to objects in this database become invalid.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function closes a database. For the case that different threads of an application are using the same database, `nxdbOpenDatabase` and `nxdbCloseDatabase` maintain a reference counter indicating how many times the database is open. Every thread can open the database, work with it, and close the database independently using `CloseAllRefs` = 0. Only the last call to `nxdbCloseDatabase` actually closes access to the database.

Another option is that only one thread executes `nxdbCloseDatabase` once, using `CloseAllRefs` = 1, which closes access for all other threads. This may be convenient when, for example, the main program needs to stop all running threads and be sure the database is closed properly, even if some threads could not execute `nxdbCloseDatabase`.

nxdbCreateObject

Purpose

Creates a new XNET cluster.

Format

```
nxStatus_t _NXFUNC nxdbCreateObject (
    nxDatabaseRef_t ParentObjectRef,
    u32 ObjectClass,
    const char * ObjectName,
    nxDatabaseRef_t * DbObjectRef);
```

Inputs

`nxDatabaseRef_t ParentObjectRef`

The reference to the parent database object. You first must open a database file using nxdbOpenDatabase.

`u32 ObjectClass`

The class of object to be created.

`const char * ObjectName`

The name of the database object to create. The name must be unique for all database objects of the same class in a database. Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the name. The space ( ), period (.), and other special characters are not supported within the name. The name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The name is limited to 128 characters.

Outputs

`nxDatabaseRef_t * DbObjectRef`

The reference to the newly created database object.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function creates an XNET database object. You can create the following objects using this function:

- nxClass_Cluster; parent is nxClass_Database object
- nxClass_Frame; parent is nxClass_Cluster object
- nxClass_PDU; parent is nxClass_Cluster
- nxClass_Subframe; parent is nxClass_PDU or nxClass_Frame[1]
- nxClass_Signal; parent is nxClass_PDU or nxClass_Frame[1]
- nxClass_ECU; parent is nxClass_Cluster
- nxClass_LINSched; parent is nxClass_Cluster
- nxClass_LINSchedEntry; parent is nxClass_LINSched

The `ObjectName` input becomes the `nxProp...Name` property of the created object

The database object is created and remains in memory until the database is closed. This function does not change the open database file on disk. To save the newly created object to the file, use `nxdbSaveDatabase`.

[1] You can create a subframe or signal on a frame object only if there is a one-to-one relationship between frames and PDUs, or PDUs are not used (for example, in DBC files).

nxdbDeleteObject

Purpose

Deletes an XNET database object and all its child objects.

Format

```
nxStatus_t _NXFUNC nxdbDeleteObject (
    nxDatabaseRef_t DbObjectRef);
```

Inputs

`nxDatabaseRef_t DbObjectRef`

References the database object to delete.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function deletes an XNET database object with all its child objects. When deleting a frame, it also deletes PDUs mapped to the frame and all signals and subframes defined in those PDUs. To delete a frame without PDUs, unmap the PDUs by setting the XNET Frame PDU References property to an empty array before deleting the frame object.

Upon deletion, the references to all deleted objects are closed and no longer can be used.

The objects are deleted from a database in memory. The change is in force until the database is closed. This function does not change the open database file on disk. To save the changed database to the file, use `nxdbSaveDatabase`.

nxdbDeploy

Purpose

Deploys a database to a remote Real-Time (RT) target.

Format

```
nxStatus_t _NXFUNC nxdbDeploy (
    const char * IPAddress,
    const char * DatabaseAlias,
    u32 WaitForComplete,
    u32 * PercentComplete);
```

Inputs

`const char * IPAddress`

The target IP address.

`const char * DatabaseAlias`

Provides the database alias name. To deploy a database text file, first add an alias using `nxdbAddAlias`.

`u32 WaitForComplete`

Determines whether the function returns directly or waits until the entire transmission is completed.

Outputs

`u32 * PercentComplete`

Indicates the deployment progress.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function transfers an optimized binary image of the database to the RT target. After deploying the database, you can use the alias name in any application for the Windows host and the LabVIEW RT target.

This function is supported on Windows only. RT database deployments are managed remotely from Windows.

This function must access the remote RT target from Windows, so `IPAddress` must specify a valid IP address for the RT target. You can find this IP address using MAX.

If the RT target access is password protected, use the following syntax for the IP address to deploy an alias: *[user:password@]IPaddress*.

Remote file transfer can take a few seconds, especially when the RT target is far away.

If `WaitForComplete` is true, this function waits for the entire transfer to complete, then returns. The return value reflects the deployment status, and `PercentComplete` is 100.

If `WaitForComplete` is false, this function transfers a portion of the database and returns before it is complete. For an incomplete transfer, the return value returns success, and `PercentComplete` is less than 100. You can use `PercentComplete` to display transfer progress on your front panel. You must call `nxdbDeploy` in a loop until `PercentComplete` is returned as 100, at which time the return value reflects the entire deployment status.

nxdbFindObject

Purpose

Finds an object in the database.

Format

```
nxStatus_t _NXFUNC nxdbFindObject (
    nxDatabaseRef_t ParentObjectRef,
    u32 ObjectClass,
    const char * ObjectName,
    nxDatabaseRef_t * DbObjectRef);
```

Inputs

`nxDatabaseRef_t ParentObjectRef`

The reference to the parent object.

`u32 ObjectClass`

The class of the object to find.

`const char * ObjectName`

The name of the object to find.

Outputs

`nxDatabaseRef_t * DbObjectRef`

A reference to the found object that you can use in subsequent function calls to reference the object.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function finds an object relative to a parent object.

Unlike `nxdbCreateObject`, this function allows `ParentObjectRef` to be a grandparent or great-grandparent.

If `ParentObjectRef` is a direct parent (for example, frame for signal), the `ObjectName` to search for can be short, and the search proceeds quickly.

If `ParentObjectRef` is not a direct parent (for example, database for signal), the `ObjectName` to search for must be qualified such that it is unique within the scope of `ParentObjectRef`.

For example, if the class of `ParentObjectRef` is nxClass_Cluster, and `ObjectClass` is nxClass_Signal, you can specify `ObjectName` of *mySignal*, assuming that signal name is unique to the cluster. If not, you must include the frame name as a prefix, such as *myFrameA.mySignal*.

You must call this function to get a reference to a database object before you can access it.

NI-XNET supports the following database classes:

- nxClass_Cluster
- nxClass_Frame
- nxClass_PDU
- nxClass_Signal
- nxClass_Subframe
- nxClass_ECU
- nxClass_LINSched
- nxClass_LINSchedEntry

nxdbGetDatabaseList

Purpose

Gets the current list of databases on a system.

Format

```
nxStatus_t _NXFUNC nxdbGetDatabaseList (
    const char * IPAddress,
    u32 SizeofAliasBuffer,
    char * AliasBuffer,
    u32 SizeofFilepathBuffer,
    char * FilepathBuffer,
    u32 * NumberOfDatabases);
```

Inputs

`const char * IPAddress`

The target IP address.

If `IPAddress` is an empty string, this function retrieves aliases and file paths for the local Windows system.

If `IPAddress` is a valid IP address, this function retrieves aliases and file paths for the remote RT target. You can find this IP address using MAX.

`u32 SizeofAliasBuffer`

The size of the buffer provided to take the list of alias names.

`u32 SizeofFilepathBuffer`

The size of the buffer provided to take the list of filepaths of the database files.

Outputs

`char * AliasBuffer`

Returns a comma-separated list of strings, one for every alias registered in the system. If no aliases are registered, the list is empty.

`char * FilepathBuffer`

Returns a comma-separated list of strings that contain the file paths and filenames of the databases assigned to the aliases, one for every alias registered in the system.

If no aliases are registered, the list is empty. This parameter applies to Windows targets only; on RT targets, this list always is empty.

`u32 * NumberOfDatabases`

Returns the number of databases registered on the system.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

For a local Windows call (IP address empty), `FilepathBuffer` returns a comma-separated list of file paths. The number of elements in this list is the same as in `AliasBuffer`. It provides the Windows file path for each corresponding alias.

For a remote call to RT, `FilepathBuffer` is empty. NI-XNET handles the file system on the RT target automatically, so that only the alias is needed.

This function is supported on Windows only. RT database deployments are managed remotely from Windows.

This call checks for the existence of the database file and removes any aliases that are no longer valid.

nxdbGetDatabaseListSizes

Purpose

Gets the buffer sizes required to read the current list of databases on a system.

Format

```
nxStatus_t _NXFUNC nxdbGetDatabaseListSizes (
    const char * IPAddress,
    u32 * SizeofAliasBuffer,
    u32 * SizeofFilepathBuffer);
```

Inputs

`const char * IPAddress`

The target IP address.

If `IPAddress` is an empty string, this function retrieves aliases and file paths for the local Windows system.

If `IPAddress` is a valid IP address, this function retrieves aliases and file paths for the remote RT target. You can find this IP address using MAX.

`u32 SizeofAliasBuffer`

Size of the buffer provided to take the list of alias names.

`u32 SizeofFilepathBuffer`

Size of the buffer provided to take the list of file paths of the database files.

Outputs

`u32 SizeofAliasBuffer`

Size of the buffer needed to take the list of alias names.

`u32 SizeofFilepathBuffer`

Size of the buffer needed to take the list of file paths of the database files.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

For a local Windows call (IP address empty), `SizeofFilepathBuffer` returns the size of a buffer needed to query the list of file paths.

For a remote call to RT, `SizeofFilepathBuffer` is empty. NI-XNET handles the file system on the RT target automatically, so that only the alias is needed.

This function is supported on Windows only. RT database deployments are managed remotely from Windows.

nxdbGetDBCAttribute

Purpose

Reads an attribute value, attribute enumeration, defined attributes, or signal value table from a DBC file.

Format

```
nxStatus_t nxdbGetDBCAttribute (
   nxDatabaseRef_t DbObjectRef,
   const u32 Mode,
   const char* AttributeName,
   const u32 AttributeTextSize,
   char* AttributeText,
   u32* IsDefault);
```

Inputs

`nxDatabaseRef_t DbObjectRef`

The reference to the database object for which to get the attribute.

`const u32 Mode`

The mode specification of this function. Depending on this value, the function returns the following data:

- **Mode 0: Get Attribute Value**: For a given object (for example, a signal), the function returns the attribute value assigned to the object. The attribute values always are returned as text in `AttributeText`. The DBC specification also allows defining other data types, such as integer or float. If necessary, you can convert the value to a number by using, for example, the `atoi()` function. If the attribute is defined as an enumeration of text strings, the attribute value returned here is the index to the enumeration list, which you can retrieve using Mode 1 of this function.
- **Mode 1: Get Enumeration**: For a given attribute name, the function returns the enumeration text table as a comma-separated string in `AttributeText`. Because the enumeration for a given attribute name is the same for all objects of the same type, `ObjectRef` can point to any object with the given class (`ObjectRef` specifies the class). If no enumeration is defined for an attribute, the function returns an empty string.
- **Mode 2: Get Attribute Name List**: Returns all attribute names defined for the given object type as a comma-separated string. `ObjectRef` can point to any object in the database of the given class (`ObjectRef` specifies the object class). `AttributeName` is ignored (it should be set to empty string or NULL).
- **Mode 3: Get Signal Value Table**: This is valid only when `ObjectRef` points to a signal. `AttributeName` is ignored (it should be set to empty string or NULL). If the

given signal contains a value table, the function returns a comma-separated list in the form *{[value,string],<value>,<string>}*. The list contains any number of corresponding *value,string* pairs. If no value table is defined for the signal, the result is an empty string.

`const char* AttributeName`

The attribute name as defined in the DBC file.

`u32 AttributeTextSize`

The size in bytes for the `AttributeText` buffer passed to this function, including `\0` for the end of string mark.

`char* AttributeText`

The buffer in which the attribute value is returned. You can use the `nxdbGetDBCAttributeSize` function to determine the minimum buffer size for the given attribute.

`u32* IsDefault`

Indicates that a default value is used instead of specific value for this object. DBC files define a default value for an attribute with the given name, and then specific values for particular objects. If the specific value for an object is not defined, the default value is returned. If the value returned in `IsDefault` is 0 (false), the attribute value is specific for this object; otherwise, it is a default. `IsDefault` has no meaning if the `Mode` parameter is not 0 (refer to the Mode description above).

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

Depending on the `Mode` parameter, this function reads an attribute value, attribute enumeration, list of existing attributes, or value table of a signal from a DBC file. Refer to the `Mode` parameter description above for details.

Attributes are supported for the following object types:

- Cluster (DBC file: Network attribute)
- Frame (DBC file: Message attribute)
- Signal (DBC file: Signal attribute)
- ECU (DBC file: Node attribute)

Databases other than DBC do not support attributes. Attributes are not saved to a FIBEX file when you open and save a DBC file.

nxdbGetDBCAttributeSize

Purpose

Retrieves the minimum size of the buffer required by the nxdbGetDBCAttribute function.

Format

```
nxStatus_t nxdbGetDBCAttributeSize (
    nxDatabaseRef_t DbObjectRef,
    const int Mode,
    const char* AttributeName,
    u32* AttributeTextSize);
```

Inputs

`nxDatabaseRef_t DbObjectRef`

The reference to the database object for which to get the attribute size.

`const u32 Mode`

The mode specification of this function. Refer to nxdbGetDBCAttribute for details.

`const char* AttributeName`

The attribute name as defined in the DBC file.

`u32* AttributeTextSize`

Returns the required buffer size in bytes for the attribute value, including `\0` for the end of string mark.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

You can use `nxdbGetDBCAttributeSize` prior to calling the nxdbGetDBCAttribute function to retrieve the required buffer size. Using this size, you can allocate memory for a buffer large enough to hold the attribute value.

nxdbGetProperty

Purpose

Reads properties for an XNET Database object.

Format

```
nxStatus_t _NXFUNC nxdbGetProperty (
    nxDatabaseRef_t DbObjectRef,
    u32 PropertyID,
    u32 PropertySize,
    void * PropertyValue);
```

Inputs

`nxDatabaseRef_t DbObjectRef`

The reference to the database object for which to get the property value.

`u32 PropertyID`

Specifies the ID of the property to get.

`u32 PropertySize`

The size of the property to get.

Outputs

`void * PropertyValue`

A void pointer to a buffer that receives the property value.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function is used to read properties for an XNET Database object. Refer to the following sections for information about properties you can use with this function:

- *XNET Cluster Properties*
- *XNET Database Properties*
- *XNET ECU Properties*
- *XNET Frame Properties*
- *XNET Signal Properties*
- *XNET Subframe Properties*

nxdbGetPropertySize

Purpose

Gets a property value size in bytes.

Format

```
nxStatus_t _NXFUNC nxdbGetPropertySize (
    nxDatabaseRef_t DbObjectRef,
    u32 PropertyID,
    u32 * PropertySize);
```

Inputs

`nxDatabaseRef_t DbObjectRef`

The reference to the database object for which to get the property value size.

`u32 PropertyID`

Specifies the ID of the property for which to get the size.

`u32 PropertySize`

The size of the property to get.

Outputs

`u32 PropertySize`

The size of the property value in bytes.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

Use this function to get a property value size in bytes. Refer to the following sections for information about properties you can use with this function:

- *XNET Cluster Properties*
- *XNET Database Properties*
- *XNET ECU Properties*
- *XNET Frame Properties*
- *XNET Signal Properties*
- *XNET Subframe Properties*

nxdbMerge

Purpose

Merges database objects and related subobjects from the source to the destination cluster.

Format

```
nxStatus_t _NXFUNC nxdbMerge (
    nxDatabaseRef_t TargetClusterRef,
    nxDatabaseRef_t SourceObjRef,
    u32 CopyMode,
    const char * Prefix,
    u32 WaitForComplete,
    u32 *PercentComplete);
```

Inputs

`nxDatabaseRef_t TargetClusterRef`

References the cluster object where the source object is merged.

`nxDatabaseRef_t SourceObjRef`

References the object to be merged into the target cluster.

`u32 CopyMode`

Defines the merging behavior if the target cluster already contains an object with the same name.

`Ccnst char * Prefix`

The prefix to be added to the source object name if an abject with the same name and type exists in the target cluster.

`U32 WaitForComplete`

Determines whether the function returns directly or waits until the entire transmission is completed.

Outputs

`u32 * PercentComplete`

Indicates the merging progress.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function merges a database object with all dependent child objects into the target cluster. This function works with the following objects: Frame, PDU, ECU, LIN Schedule, or a cluster. All listed objects must have unique names in the cluster. They are referenced here as objects, as opposed to child objects (for example, a signal is a child of a frame).

If the source object name is not used in the target cluster, this function copies the source objects with the child objects to the target. If an object with the same name exists in the target cluster, you can avoid name collisions by specifying the prefix to be added to the name.

If an object with the same name exists in the target cluster, the merge behavior depends on the `CopyMode` input:

- **nxdbMerge_CopyUseSource**: The target object with all dependent child objects is removed from the target cluster and replaced by the source objects.
- **nxdbMerge_CopyUseTarget**: The source object is ignored (the target cluster object with child objects remains unchanged).
- **nxdbMerge_MergeUseSource**: This adds child objects from the source object to child objects from the destination object. If target object contains a child object with the same name, the child object from the source frame replaces it. The source object properties (for example, payload length of the frame) replace the target properties.
- **nxdbMerge_MergeUseTarget**: This adds child objects from the source object to child objects from the destination object. If the target object contains a child object with the same name, it remains unchanged. The target object properties remain unchanged (for example, payload length).

Example

Target frame F1(v1) has signals S1 and S2(v1). Source frame F1(v2) has signals S2(v2) and S3.

(v1) and (v2) are two versions of one object with same name, but with different properties.

- Result of **nxdbMerge_CopyUseSource**: F1(v2), S2(v2), S3.
- Result of **nxdbMerge_CopyUseTarget**: F1(v1), S1, S2(v1).
- Result of **nxdbMerge_MergeUseSource**: F1(v2), S1, S2(v2), S3.
- Result of **nxdbMerge_MergeUseTarget**: F1(v1), S1, S2(v1), S3.

If the source object is a cluster, this function copies all contained PDUs, ECUs, and LIN schedules with their child objects to the destination cluster.

Depending on the number of contained objects in the source and destination clusters, the execution can take a longer time. If `WaitForComplete` is true, this function waits until the

merging process gets completed. If the execution completes without errors, `PercentComplete` returns 100. If `WaitForComplete` is false, the function returns quickly, and `PercentComplete` returns values less than 100. You must call `nxdbMerge` repeatedly until `PercentComplete` returns 100. You can use the time between calls to update a progress bar.

nxdbOpenDatabase

Purpose

Opens a database file.

Format

```
nxStatus_t _NXFUNC nxdbOpenDatabase (
    const char * DatabaseName,
    nxDatabaseRef_t * DatabaseRef);
```

Inputs

`const char * DatabaseName`

The cluster to open.

Outputs

`nxDatabaseRef_t * DatabaseRef`

A reference to the database that you can use in subsequent function calls to reference the database.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function opens a database. When an already open database is opened, this function grants access to the same database and increases an internal reference counter. A multiple referenced (open) database must be closed as many times as it has been opened. Until it is completely closed, the access to this database remains granted, and the database uses computer resources (memory and handles). For more information, refer to `nxdbCloseDatabase`.

nxdbRemoveAlias

Purpose

Removes a database alias from the system.

Format

```
nxStatus_t _NXFUNC nxdbRemoveAlias (
    const char * DatabaseAlias);
```

Inputs

`const char * DatabaseAlias`

The name of the alias to delete.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function removes the alias from NI-XNET, but does not affect the database text file. It just removes the alias association to the database file path.

This function is supported on Windows only, and the alias is removed from Windows only (not RT targets). Use `nxdbUndeploy` to remove an alias from a Real-Time (RT) target.

nxdbSaveDatabase

Purpose

Saves the open database to a FIBEX 3.1.0 file file or exports a cluster from a database to a specific file format.

Format

```
nxStatus_t _NXFUNC nxdbSaveDatabase (
    nxDatabaseRef_t DatabaseRef,
    const char * DbFilepath);
```

Inputs

`nxDatabaseRef_t DatabaseRef`

References the database to be saved or the database cluster to be exported.

`const char * DbFilepath`

Contains the pathname to the database file or is empty (saves to the original file path).

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

If the `DatabaseRef` parameter is a database reference, this function saves the XNET database current state to a FIBEX 3.1.0 file. The file extension must be `.xml`. If the target file exists, it is overwritten.

If the `DatabaseRef` parameter is a cluster reference, this function exports the cluster in a specific file format. A CAN cluster is exported as a CANdb++ database file (`.dbc`). A LIN cluster is exported as a LIN database file (`.ldf`). A FlexRay cluster cannot be exported, and the function returns an appropriate error. If the target file exists, it is overwritten.

XNET saves to the FIBEX file only features that XNET sessions use to communicate on the network. If the original file was created using non-XNET software, the target file may be missing details from the original file. For example, NI-XNET supports only linear scaling. If the original FIBEX file used a rational equation that cannot be expressed as a linear scaling, XNET converts this to a linear scaling with factor 1.0 and offset 0.0.

If `DbFilepath` is empty, the file is saved to the same FIBEX file specified when opened. If opened as a file path, it uses that file path. If opened as an alias, it uses the file path registered for that alias. In the case of a cluster export, the filepath must not be empty.

Saving a database is not supported under Real-Time (RT), but you can deploy and use a database saved on Windows on a Real-Time (RT) target (refer to `nxdbDeploy`).

nxdbSetProperty

Purpose

Writes properties for an XNET Database object.

Format

```
nxStatus_t _NXFUNC nxdbSetProperty (
    nxDatabaseRef_t DbObjectRef,
    u32 PropertyID,
    u32 PropertySize,
    void * PropertyValue);
```

Inputs

`nxDatabaseRef_t DbObjectRef`

The reference to the database object for which to get the property value.

`u32 PropertyID`

Specifies the ID of the property to set.

`u32 PropertySize`

The size of the property to set.

Outputs

`void * PropertyValue`

A void pointer to a buffer that contains the property value to set.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

Use this function to write properties for an XNET Database object. Refer to the following sections for information about properties you can use with this function:

- *XNET Cluster Properties*
- *XNET Database Properties*
- *XNET ECU Properties*
- *XNET Frame Properties*
- *XNET Signal Properties*
- *XNET Subframe Properties*

nxdbUndeploy

Purpose

Undeploys a database from a remote LabVIEW Real-Time (RT) target.

Format

```
nxStatus_t _NXFUNC nxdbUndeploy (
    const char * IPAddress,
    const char * DatabaseAlias);
```

Inputs

`const char * IPAddress`

The target IP address.

`const char * DatabaseAlias`

Provides the database alias name.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function completely deletes the database file and its alias from the RT target.

This function is supported on Windows only. RT database deployments are managed remotely from Windows.

This function must access the remote RT target from Windows, so `IPAddress` must specify a valid IP address for the RT target. You can find this IP address using MAX.

If the RT target access is password protected, you can use the following syntax for the IP address to undeploy an alias: *[user:password@]IPaddress*.

nxDisconnectTerminals

Purpose

Disconnects terminals on the XNET interface.

Format

```
nxStatus_t _NXFUNC nxDisconnectTerminals (
    nxSessionRef_t SessionRef,
    const char * source,
    const char * destination);
```

Inputs

`nxSessionRef_t SessionRef`

The reference to the session to use for the connection.

`const char * source terminal`

The connection source name.

`const char * destination terminal`

The connection destination name.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function disconnects a specific pair of source/destination terminals previously connected with `nxConnectTerminals`.

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

This function typically is used to change terminal connections dynamically while an application is running. To disconnect a terminal, you first must stop the interface using `nxStop` with the Interface Only scope. Then you can call `nxDisconnectTerminals` and

nxConnectTerminals to adjust terminal connections. Finally, you can call nxStart with the Interface Only scope to restart the interface.

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

nxFlush

Purpose

Flushes (empties) all XNET session queues.

Format

```
nxStatus_t _NXFUNC nxFlush (
    nxSessionRef_t SessionRef);
```

Inputs

`nxSessionRef_t SessionRef`

The reference to the session to flush. This session is from `nxCreateSession`.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

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

`nxStart` and `nxStop` have no effect on these queues. Use `nxFlush` to discard all values in the session's queues.

For example, if you call `nxWrite` to write three frames, then immediately call `nxStop`, then call `nxStart` a few seconds later, the three frames transmit. If you call `nxFlush` between `nxStop` and `nxStart`, no frames transmit.

As another example, if you receive three frames, then call `nxStop`, the three frames remains in the queue. If you call `nxStart` a few seconds later, then call `nxRead`, you obtain the three frames received earlier, potentially followed by other frames received after calling `nxStart`. If you call `nxFlush` between `nxStop` and `nxStart`, `nxRead` returns only frames received after the calling `nxStart`.

nxGetProperty

Purpose

Retrieves an XNET session property.

Format

```
nxStatus_t nxGetProperty (
    nxSessionRef_t SessionRef,
    u32 PropertyID,
    u32 PropertySize,
    void * PropertyValue);
```

Inputs

`nxSessionRef_t SessionRef`

The session to get the property from. This session is returned from `nxCreateSession`.

`u32 PropertyID`

The ID of the property desired. The appropriate constants are listed in the Properties section and defined in `nixnet.h`.

`u32 PropertySize`

The number of bytes provided for the buffer passed to `PropertyValue`. This can be a fixed-size (for example, 4 bytes for a u32 property) or variable-sized buffer. If the property has variable size (for example, a string property whose size is determined at runtime), call `nxGetPropertySize` to retrieve the necessary size of the buffer beforehand.

Outputs

`void * PropertyValue`

Returns the value of the desired property.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

Refer to the following sections for information about properties you can use with this function:

- *XNET Device Properties*
- *XNET Interface Properties*
- *XNET Session Properties*
- *XNET System Properties*

nxGetPropertySize

Purpose

Retrieves the data size of an XNET session property.

Format

```
nxStatus_t nxGetPropertySize (
    nxSessionRef_t SessionRef,
    u32 PropertyID,
    u32 * PropertySize);
```

Inputs

`nxSessionRef_t SessionRef`

The session to get the property from. This session is returned from `nxCreateSession`.

`u32 PropertyID`

The ID of the property desired. The appropriate constants are listed in the Properties section and defined in `nixnet.h`.

Outputs

`u32 * PropertySize`

Returns the number of bytes to be provided for the buffer to retrieve the property. Pass a buffer of that size to `nxGetProperty`.

Note For string properties, the property size returned includes the space for the terminating NULL byte.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

Refer to the following sections for information about properties you can use with this function:

- *XNET Device Properties*
- *XNET Interface Properties*
- *XNET Session Properties*
- *XNET System Properties*

nxGetSubProperty

Purpose

Retrieves a property of a frame or signal within an XNET session.

Format

```
nxStatus_t nxGetSubProperty (
    nxSessionRef_t SessionRef,
    u32 ActiveIndex,
    u32 PropertyID,
    u32 PropertySize,
    void * PropertyValue);
```

Inputs

`nxSessionRef_t SessionRef`

The session to get the property from. This session is returned from `nxCreateSession`.

`u32 ActiveIndex`

Identifies the frame or signal within the session. It is the index to the list given in `nxCreateSession`.

`u32 PropertyID`

The ID of the property desired. The properties to use with this function are listed in the *Frame Properties* section for the session. Within your code, applicable `PropertyID` values begin with the prefix `nxProp_SessionSub`.

`u32 PropertySize`

The number of bytes provided for the buffer passed to `PropertyValue`. This can be a fixed-size (for example, 4 bytes for a u32 property) or variable-sized buffer. If the property has variable size (for example, a string property whose size is determined at runtime), call `nxGetSubPropertySize` to retrieve the necessary size of the buffer beforehand.

Outputs

`void * PropertyValue`

Returns the value of the desired property.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

nxGetSubPropertySize

Purpose

Retrieves the data size of a property of a frame or signal within an XNET session.

Format

```
nxStatus_t nxGetSubPropertySize (
    nxSessionRef_t SessionRef,
    u32 ActiveIndex,
    u32 PropertyID,
    u32 * PropertySize);
```

Inputs

`nxSessionRef_t SessionRef`

The session to get the property from. This session is returned from `nxCreateSession`.

`u32 ActiveIndex`

Identifies the frame or signal within the session. It is the index to the list given in `nxCreateSession`.

`u32 PropertyID`

The ID of the property desired. The properties to use with this function are listed in the *Frame Properties* section for the session. Within your code, applicable `PropertyID` values begin with the prefix `nxProp_SessionSub`.

Outputs

`u32 * PropertySize`

Returns the number of bytes to be provided for the buffer to retrieve the property. Pass a buffer of that size to `nxGetSubProperty`.

Note For string properties, the property size returned includes the space for the terminating NULL byte.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

nxReadFrame

Purpose

Reads data from a session as an array of raw bytes.

Format

```
nxStatus_t nxReadFrame (
    nxSessionRef_t SessionRef,
    void * Buffer,
    u32 SizeOfBuffer,
    f64 Timeout,
    u32 * NumberOfBytesReturned);
```

Inputs

`nxSessionRef_t SessionRef`

The session to read. This session is returned from `nxCreateSession`. The session mode must be Frame Input Stream Mode, Frame Input Queued Mode, or Frame Input Single-Point Mode.

`u32 SizeOfBuffer`

The number of bytes desired.

This number does not represent the number of frames to read. As encoded in raw data, each frame can vary in length. Therefore, the number represents the maximum raw bytes to read, not the number of frames.

Standard CAN and LIN frames are always 24 bytes in length. If you want to read a specific number of frames, multiply that number by 24.

CAN FD and FlexRay frames vary in length. For example, if you pass `SizeOfBuffer` of 91, the buffer might return 80 bytes, within which the first 24 bytes encode the first frame, and the next 56 bytes encode the second frame.

If `SizeOfBuffer` is positive, the data array size is no greater than this number. The minimum size for a single frame is 24 bytes, so you must use at least that number.

`f64 Timeout`

The time to wait for number to read frame bytes to become available; the timeout is represented as 64-bit floating-point in units of seconds.

To avoid returning a partial frame, even when `SizeOfBuffer` bytes are available from the hardware, this read may return fewer bytes in `Buffer`. For example, assume you pass `SizeOfBuffer` of 70 bytes and `Timeout` of 10 seconds. During the read, two frames are received, the first 24 bytes in size, and the second 56 bytes in size, for a total of 80 bytes. The read returns after the two frames are received, but only the first frame is copied to data. If the read copied 46 bytes of the second frame (up to the limit of 70), that frame

would be incomplete and therefore difficult to interpret. To avoid this problem, the read always returns complete frames in `Buffer`.

If `Timeout` is positive, `nxReadFrame` waits for `SizeOfBuffer` frame bytes to be received, then returns complete frames up to that number. If the bytes do not arrive prior to the timeout, an error is returned.

If `Timeout` is negative, `nxReadFrame` waits indefinitely for `SizeOfBuffer` frame bytes.

If `Timeout` is zero, `nxReadFrame` does not wait and immediately returns all available frame bytes up to the limit `SizeOfBuffer` specifies.

This input is optional. The default value is 0.0.

If the session mode is Frame Input Single-Point, you must set `Timeout` to 0.0. Because this mode reads the most recent value of each frame, `Timeout` does not apply.

Outputs

`void * Buffer`

Returns an array of bytes.

The raw bytes encode one or more frames using the Raw Frame Format. This frame format is the same for read and write of raw data, and it is also used for log file examples.

The data always returns complete frames.

Note For PDU sessions, only the payload for the specified PDU is returned in the array of bytes.

For an example of how this data applies to network traffic, refer to *Frame Input Stream Mode*, *Frame Input Queued Mode*, or *Frame Input Single-Point Mode*.

`u32 * NumberOfBytesReturned`

Returns the number of valid bytes in the `Buffer` array.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

The raw bytes encode one or more frames using the Raw Frame Format. The session must use Frame Input Stream Mode, Frame Input Queued Mode, or Frame Input Single-Point Mode.

The raw frame format is protocol independent, so the session can use either a CAN, FlexRay, or LIN interface.

The raw frames are associated to the session's list of frames as follows:

Frame Input Stream Mode: Array of all frame values received (list ignored).

Frame Input Queued Mode: Array of frame values received for the single frame specified in the list.

Frame Input Single-Point Mode: Array of single frame values, one for each frame specified in the list.

nxReadSignalSinglePoint

Purpose

Reads data from a session of Signal Input Single-Point Mode.

Format

```
nxStatus_t nxReadSignalSinglePoint (
    nxSessionRef_t SessionRef,
    f64 * ValueBuffer,
    u32 SizeOfValueBuffer,
    nxTimestamp_t * TimestampBuffer,
    u32 SizeOfTimestampBuffer);
```

Inputs

`nxSessionRef_t SessionRef`

The session to read. This session is returned from `nxCreateSession`. The session mode must be a Signal Input Single-Point Mode.

`u32 SizeOfValueBuffer`

Should be set to the size (in bytes) of the array passed to `ValueBuffer`. If this is too small to fit one element for each signal in the session, an error is returned.

`u32 SizeOfTimestampBuffer`

Should be set to the size (in bytes) of the array passed to `TimestampBuffer`. If `TimestampBuffer` is not NULL, and this is too small to fit one element for each signal in the session, an error is returned.

Outputs

`f64* ValueBuffer`

Returns a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data returns the most recent value received for each signal. If multiple frames for a signal are received since the previous call to `nxReadSignalSinglePoint` (or session start), only signal data from the most recent frame is returned.

If no frame is received for the corresponding signals since you started the session, the XNET Signal Default Value is returned.

For an example of how this data applies to network traffic, refer to *Signal Input Single-Point Mode*.

A trigger signal returns a value of 1.0 or 0.0, depending on whether its frame arrived since the last Read (or Start) or not. For more information about trigger signals, refer to *Signal Input Single-Point Mode*.

`nxTimestamp_t* TimestampBuffer`

Optionally returns a one-dimensional array of timestamp values of the times when the corresponding signal values arrived. Each timestamp value is the number of 100 ns increments since Jan 1, 1601 12:00 AM UTC.

`TimestampBuffer`

Can be passed as NULL; then no timestamps are returned. `SizeOfTimeStampBuffer` also should be passed 0 in this case.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

nxReadSignalWaveform

Purpose

Reads data from a session of Signal Input Waveform Mode.

The data represents a waveform of resampled values for each signal in the session.

Format

```
nxStatus_t nxReadSignalWaveform (
    cnxSessionRef_t SessionRef,
    f64 Timeout,
    nxTimestamp_t * StartTime,
    f64 * DeltaTime,
    f64 * ValueBuffer,
    u32 SizeOfValueBuffer,
    u32 * NumberOfValuesReturned);
```

Inputs



Note In the following, *N* means the maximum number of samples to read. It is calculated from `SizeOfValueBuffer`.

`nxSessionRef_t SessionRef`

The session to read. This session is returned from `nxCreateSession`. The session mode must be Signal Input Waveform.

`f64 Timeout`

The time to wait for *N* samples to become available.

The timeout is represented as 64-bit floating-point in units of seconds.

If `Timeout` is positive, `nxReadSignalWaveform` waits for *N* samples, then returns that number. If the samples do not arrive prior to the timeout, an error is returned.

If `Timeout` is negative, `nxReadSignalWaveform` waits indefinitely for *N* samples.

If `Timeout` is zero, `nxReadSignalWaveform` does not wait and immediately returns all available samples up to the limit *N* specifies.

Because time determines sample availability, typical values for this timeout are 0 (return available) or a large positive value such as 100.0 (wait for a specific *N*).

`u32 SizeOfValueBuffer`

The size (in bytes) of the array passed to `ValueBuffer`. It is used to calculate *N* = trunc (`SizeOfValueBuffer` / (sizeof (f64) * (number of signals in the session))). There always is a maximum of *N* samples per waveform returned, even if `SizeOfValueBuffer` is not a multiple of (sizeof (f64) * (number of signals in the session)).

Outputs

`nxTimestamp_t* StartTime`

Optionally returns the start time of the waveform returned in `ValueBuffer`. It is the absolute time of the first sample, given in 100 ns increments since Jan 1, 1601, 12:00 AM UTC.

`StartTime` can be passed as NULL; in this case, no value is returned.

`f64* DeltaTime`

Optionally returns the time increment between successive values of the waveform returned in `ValueBuffer`. The value returned is 1.0/Resample Rate.

`DeltaTime` can be passed as NULL; in this case, no value is returned.

`f64* ValueBuffer`

Returns a two-dimensional array of f64 samples. First, *N* samples are reserved for the first signal in the session, then *N* samples for the second, and so on. *N* * (number of signals in the session) * sizeof (f64) should be passed in `SizeOfValueBuffer` to recalculate *N*.

For an example of how this data applies to network traffic, refer to *Signal Input Waveform Mode*.

`u32* NumberOfValuesReturned`

The number of waveform samples per signal that have been returned in `ValueBuffer`. This is always less than or equal to *N*.

`NumberOfValuesReturned` can be passed as NULL; in this case, no value is returned.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

The data represents a waveform for each signal in the session.

nxReadSignalXY

Purpose

Reads data from a session of Signal Input XY Mode.

Format

```
nxStatus_t nxReadSignalXY (
    nxSessionRef_t SessionRef,
    nxTimestamp_t * TimeLimit,
    f64 * ValueBuffer,
    u32 SizeOfValueBuffer,
    nxTimestamp_t * TimestampBuffer,
    u32 SizeOfTimestampBuffer,
    u32 * NumPairsBuffer,
    u32 SizeOfNumPairsBuffer);
```

Inputs

Note In the following, *N* means the maximum number of samples to read per signal. It is calculated from `SizeOfValueBuffer` and `SizeOfTimestampBuffer`.

`nxSessionRef_t SessionRef`

The session to read. This session is returned from `nxCreateSession`. The session mode must be Signal Input XY.

`nxTimestamp_t* TimeLimit`

The timestamp to wait for before returning signal values. It is the absolute time, given in 100 ns increments since Jan 1, 1601, 12:00 AM UTC.

If `TimeLimit` is valid, `nxReadSignalXY` waits for the timestamp to occur, then returns available values (up to number to read). If you increment `TimeLimit` by a fixed number of seconds for each call to `nxReadSignalXY`, you effectively obtain a moving window of signal values.

The `Timeout` of other `nxRead` functions specifies the maximum amount time to wait for a specific (number to read) values. The `TimeLimit` of `nxReadSignalXY` does not specify a worst-case timeout value, but rather a specific absolute timestamp to wait for.

`u32 SizeOfValueBuffer`

The size (in bytes) of the array passed to `ValueBuffer`. *N* is calculated from this as: *N* = trunc (`SizeOfValueBuffer` / (sizeof (f64) * (number of signals in the session))). If both `SizeOfValueBuffer` and `SizeOfTimestampBuffer` deliver a valid *N* value (*N* > 0), the smaller of the two values is used to avoid buffer overflows.

`u32 SizeOfTimestampBuffer`

The size (in bytes) of the array passed to `TimestampBuffer`. *N* is calculated from this as: *N* = trunc (`SizeOfTimestampBuffer` / (sizeof (f64) * (number of signals in the session))). If both `SizeOfValueBuffer` and `SizeOfTimestampBuffer` deliver a valid *N* value (*N* > 0), the smaller of the two values is used to avoid buffer overflows.

`u32 SizeOfNumPairsBuffer`

The size (in bytes) of the array passed to `NumPairsBuffer`. For each signal in the session, an array element should be provided. If the buffer is too small, an error is returned.

Outputs

`f64* ValueBuffer`

Returns a two-dimensional array of f64 samples. First, *N* samples are reserved for the first signal in the session, then *N* samples for the second, and so on. *N* * (number of signals in the session) * sizeof (f64) should be passed in `SizeOfValueBuffer` to recalculate *N*.

For an example of how this data applies to network traffic, refer to *Signal Input XY Mode*.

`nxTimestamp_t* TimestampBuffer`

Returns a two-dimensional array of timestamps. First, *N* timestamps are reserved for the first signal in the session, then *N* timestamps for the second, and so on. *N* * (number of signals in the session) * sizeof (f64) should be passed in `SizeOfTimestampBuffer` to recalculate *N*.

The timestamps are given in 100 ns increments since Jan 1, 1601, 12:00 AM UTC.

`u32* NumPairsBuffer`

Returns a one-dimensional array of signal/timestamp pair counts, one for each signal in the session. Upon output, the samples and timestamps for signal #(i) in the preceding arrays are valid up to, but not including, index `NumPairsBuffer`[i] (zero based).

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

The data represents an XY plot of timestamp/value pairs for each signal in the session.

nxReadState

Purpose

Reads communication states of an XNET session.

Format

```
nxStatus_t nxReadState (
    nxSessionRef_t SessionRef,
    u32 StateID,
    u32 StateSize,
    void * StateValue,
    nxStatus_t * Fault);
```

Inputs

`nxSessionRef_t SessionRef`

The session to read. This session is returned from `nxCreateSession`.

`u32 StateID`

Indicates the state to be read. Possible values are:

`nxState_TimeCurrent`	Current interface time
`nxState_TimeCommunicating`	Time interface started communicating
`nxState_TimeStart`	Time interface was started
`nxState_CANComm`	CAN communication state
`nxState_FlexRayComm`	FlexRay communication state
`nxState_FlexRayStats`	FlexRay statistics
`nxState_LINComm`	LIN communication state
`nxState_SessionInfo`	Session running state

The value determines the format output as `StateValue`.

`u32 StateSize`

Indicates the size of the buffer provided for `StateValue`.

Outputs

`void* StateValue`

Returns the desired state. Formats and values are:

`StateID` = `nxState_TimeCurrent`:

`StateValue` must point to an `nxTimestamp_t` buffer. It is filled with the current interface time in 100 ns increments since Jan 1, 1601 12:00 AM UTC.

`StateID` = `nxState_TimeCommunicating`:

`StateValue` must point to an `nxTimestamp_t` buffer. It is filled with the time the interface started communicating in 100 ns increments since Jan 1, 1601 12:00 AM UTC. This time is usually later than the interface start time (`StateID` = `nxState_TimeStart`), because the interface must undergo a communication startup procedure.

If the interface is not communicating when this read is called, an invalid time is returned (0).

`StateID = nxState_TimeStart`:

`StateValue` must point to an `nxTimestamp_t` buffer. It is filled with the time the interface was started in 100 ns increments since Jan 1, 1601 12:00 AM UTC.

If the interface is not started when this read is called, an invalid time is returned (0).

`StateID` = `nxState_CANComm`:

`StateValue` must point to a u32 buffer. It is filled with a communication state DWORD, which is comprised of several bitfields. You can use macros in `nixnet.h` to access these bitfields.

Bit	Meaning	
0–3	Communication State	
	Error Active (0)	This state reflects normal communication, with few errors detected. The CAN interface remains in this state as long as receive error counter and transmit error counter are both below 128.
	Error Passive (1)	If either the receive error counter or transmit error counter increment above 127, the CAN interface transitions into this state. Although communication proceeds, the CAN device generally is assumed to have problems with receiving frames. When a CAN interface is in error passive state, acknowledgement errors do not increment the transmit error counter. Therefore, if the CAN interface transmits a frame with no other device (ECU) connected, it eventually enters error passive state due to retransmissions, but does not enter bus off state.

Bit	Meaning	
	Bus Off (2)	If the transmit error counter increments above 255, the CAN interface transitions into this state. Communication immediately stops under the assumption that the CAN interface must be isolated from other devices. When a CAN interface transitions to the bus off state, communication stops for the interface. All NI-XNET sessions for the interface no longer receive or transmit frame values. To restart the CAN interface and all its sessions, call `nxStart`.
	Init (3)	This is the CAN interface initial state on power-up. The interface is essentially off, in that it is not attempting to communicate with other nodes (ECUs). When the start trigger occurs for the CAN interface, it transitions from the Init state to the Error Active state. When the interface stops due to a call to `nxStop`, the CAN interface transitions from either Error Active or Error Passive to the Init state. When the interface stops due to the Bus Off state, it remains in that state until you restart.
4	Transceiver Error Transceiver error indicates whether an error condition exists on the physical transceiver. This is typically referred to as the transceiver chip NERR pin. False indicates normal operation (no error), and true indicates an error.	
5	Sleep Sleep indicates whether the transceiver and communication controller are in their sleep state. False indicates normal operation (awake), and true indicates sleep.	
8–11	Last Error Last error specifies the status of the last attempt to receive or transmit a frame (decimal value in parentheses): None (0) The last receive or transmit was successful. Stuff (1) More than 5 equal bits have occurred in sequence, which the CAN specification does not allow. Form (2) A fixed format part of the received frame used the wrong format.	

Bit	Meaning
	Ack (3) Another node (ECU) did not acknowledge the frame transmit. If you call the appropriate `nxWrite` function and do not have a cable connected, or the cable is connected to a node that is not communicating, you see this error repeatedly. The CAN communication state eventually transitions to Error Passive, and the frame transmit retries indefinitely. Bit 1 (4) During a frame transmit (with the exception of the arbitration ID field), the interface wanted to send a recessive bit (logical 1), but the monitored bus value was dominant (logical 0). Bit 0 (5) During a frame transmit (with the exception of the arbitration ID field), the interface wanted to send a dominant bit (logical 0), but the monitored bus value was recessive (logical 1). CRC (6) The CRC contained within a received frame does not match the CRC calculated for the incoming bits.
16–23	Transmit Error Counter The transmit error counter begins at 0 when communication starts on the CAN interface. The counter increments when an error is detected for a transmitted frame and decrements when a frame transmits successfully. The counter increases more for an error than it is decreased for success. This ensures that the counter generally increases when a certain ratio of frames (roughly 1/8) encounter errors. When communication state transitions to Bus Off, the transmit error counter no longer is valid.
24–31	Receive Error Counter The receive error counter begins at 0 when communication starts on the CAN interface. The counter increments when an error is detected for a received frame and decrements when a frame is received successfully. The counter increases more for an error than it is decreased for success. This ensures that the counter generally increases when a certain ratio of frames (roughly 1/8) encounter errors.

`StateID` = `nxState_FlexRayComm`:

`StateValue` must point to a u32 buffer. It is filled with a communication state DWORD, which is comprised of several bitfields. You can use macros in `nixnet.h` to access these bitfields.

Bit	Meaning	
0–3	POC State	
	POC state specifies the FlexRay interface state (decimal value in parentheses):	
	Default Config (0)	This is the FlexRay interface initial state on power-up. The interface is essentially off, in that it is not configured and is not attempting to communicate with other nodes (ECUs).
	Ready (1)	When the interface starts, it first enters Config state to validate the FlexRay cluster and interface properties. Assuming the properties are valid, the interface transitions to this Ready state. In the Ready state, the FlexRay interface attempts to integrate (synchronize) with other nodes in the network cluster. This integration process can take several FlexRay cycles, up to 200 ms. If the integration succeeds, the interface transitions to Normal Active. You can use `nxReadState` to read the time when the FlexRay interface entered Ready. If integration succeeds, you can use `nxReadState` to read the time when the FlexRay entered Normal Active.
	Normal Active (2)	This is the normal operation state. The NI-XNET interface is adequately synchronized to the cluster to allow continued frame transmission without disrupting the transmissions of other nodes (ECUs). If synchronization problems occur, the interface can transition from this state to Normal Passive.
	Normal Passive (3)	Frame reception is allowed, but frame transmission is disabled due to degraded synchronization with the cluster remainder. If synchronization improves, the interface can transition to Normal Active. If synchronization continues to degrade, the interface transitions to Halt.
	Halt (4)	Communication halted due to synchronization problems. When the FlexRay interface is in Halt state, all NI-XNET sessions for the interface stop, and no frame values are received or transmitted. To restart the FlexRay interface, you must restart the NI-XNET sessions. If you clear (close) all NI-XNET sessions for the interface, it transitions from Halt to Default Config state.

Bit	Meaning	
	Config (15)	This state is transitional when configuration is valid. If you detect this state after starting the interface, it typically indicates a problem with the configuration. Check the `fault?` output for a fault. If no fault is returned, check your FlexRay cluster and interface properties. You can check the validity of these properties using the NI-XNET Database Editor, which displays invalid configuration properties. In the FlexRay specification, this value is referred to as the Protocol Operation Control (POC) state. For more information about the FlexRay POC state, refer to Appendix B, *Summary of the FlexRay Standard*.
4–7	Clock Correction Failed Clock correction failed returns the number of consecutive even/odd cycle pairs that have occurred without successful clock synchronization. If this count reaches the value in the XNET Cluster FlexRay:Max Without Clock Correction Passive property, the FlexRay interface POC state transitions from Normal Active to Normal Passive state. If this count reaches the value in the XNET Cluster FlexRay:Max Without Clock Correction Fatal property, the FlexRay interface POC state transitions from Normal Passive to Halt state. In the FlexRay specification, this value is referred to as `vClockCorrectionFailed`.	
8-12	Passive to Active Count Passive to active count returns the number of consecutive even/odd cycle pairs that have occurred with successful clock synchronization. This count increments while the FlexRay interface is in POC state Error Passive. If the count reaches the value in the XNET Session Interface:FlexRay:Allow Passive to Active property, the interface POC state transitions to Normal Active. In the FlexRay specification, this value is referred to as `vAllowPassiveToActive`.	
13	Channel A Sleep? Indicates whether channel A currently is asleep.	
14	Channel B Sleep? Indicates whether channel B currently is asleep.	

`StateID` = `nxState_FlexRayStats`:

`StateValue` must point to an `nxFlexRayStats_t` buffer (defined in `nixnet.h`). It is filled with communication statistics values. The values are:

`u32 NumSyntaxErrorChA`

The number of syntax errors that have occurred on channel A since communication started.

A syntax error occurs if:

- A node starts transmitting while the channel is not in the idle state.
- There is a decoding error.
- A frame is decoded in the symbol window or in the network idle time.
- A symbol is decoded in the static segment, dynamic segment, or network idle time.
- A frame is received within the slot after reception of a semantically correct frame (two frames in one slot).
- Two or more symbols are received within the symbol window.

`u32 NumSyntaxErrorChB`

The number of syntax errors that have occurred on channel B since communication started.

`u32 NumContentErrorChA`

The number of content errors that have occurred on channel A since communication started.

A content error occurs if:

- In a static segment, a frame payload length does not match the global cluster property.
- In a static segment, the Startup indicator (bit) is 1 while the Sync indicator is 0.
- The frame ID encoded in the frame header does not match the current slot.
- The cycle count encoded in the frame header does not match the current cycle count.
- In a dynamic segment, the Sync indicator is 1.
- In a dynamic segment, the Startup indicator is 1.
- In a dynamic segment, the Null indicator is 0.

`u32 NumContentErrorChB`

The number of content errors that have occurred on channel B since communication started.

`u32 NumSlotBoundaryViolationChA`

The number of slot boundary violations that have occurred on channel A since communication started.

A slot boundary violation error occurs if the interface does not consider the channel to be idle at the boundary of a slot (either beginning or end).

`u32 NumSlotBoundaryViolationChB`

The number of slot boundary violations that have occurred on channel B since communication started.

For more information about these statistics, refer to Appendix B, *Summary of the FlexRay Standard*.

`StateID = nxState_LINComm:`

`StateValue` must point to a u32 array buffer. It is filled with a communication state DWORD, which is comprised of several bitfields, and a schedule DWORD, which is comprised of a single bitfield. You can use macros in `nixnet.h` to access these bitfields.

Communication State DWORD

Bit	Meaning
0	Reserved
1	Sleep Indicates whether the transceiver and communication controller are in their sleep state. False (0) indicates normal operation (awake), and true (1) indicates sleep. This value changes from 0 to 1 only when you set the XNET Session Interface:LIN:Sleep property to `nxLINSleep_RemoteSleep` or `nxLINSleep_LocalSleep`. This value changes from 1 to 0 when one of the following occurs: • You set the XNET Session Interface:LIN:Sleep property to `nxLINSleep_RemoteWake` or `nxLINSleep_LocalWake`. • The interface receives a remote wakeup pattern (break). In addition to this `nxReadState` function, you can wait for a remote wakeup event using the `nxWait` function with the `nxCondition_IntfCommunicating` condition.

Bit	Meaning
2–3	Communication State Idle (0) — This is the LIN interface initial state on power-up. The interface is essentially off, in that it is not attempting to communicate with other nodes (ECUs). When the start trigger occurs for the LIN interface, it transitions from the Idle state to the Active state. When the interface stops due to a call to XNET Stop, the LIN interface transitions from either Active or Inactive to the Idle state. Active (1) — This state reflects normal communication. The LIN interface remains in this state as long as bus activity is detected (frame headers received or transmitted). Inactive (2) — This state indicates that no bus activity has been detected in the past four seconds. Regardless of whether the interface acts as a master or slave, it transitions to this state after four seconds of bus inactivity. As soon as bus activity is detected (break or frame header), the interface transitions to the Active state. The LIN interface does not go to sleep automatically when it transitions to Inactive. To place the interface into sleep mode, set the XNET Session Interface:LIN:Sleep property when you detect the Inactive state.
4–7	Last Error Specifies the status of the last attempt to receive or transmit a frame. It is an enumeration (ring data type). For a table of all values for last error, refer to *Last Error Table*.
8–15	Last Error Received Returns the value received from the network when last error occurred. For a table that describes how this field is populated based on the last error, refer to *Last Error Table*.
16–23	Last Error Expected Returns the value that the LIN interface expected to see (instead of last received). For a table that describes how this field is populated based on the last error, refer to *Last Error Table*.
24–29	Last Error ID Returns the frame identifier in which the last error occurred. For a table that describes how this field is populated based on the last error, refer to *Last Error Table*.

Bit	Meaning
30	Reserved
31	Transceiver Ready Indicates whether the LIN transceiver is powered from the bus. True (1) indicates the bus power exists, so it is safe to start communication on the LIN interface. If this value is false (0), you cannot start communication successfully. Wire power to the LIN transceiver and run your application again.

Schedule DWORD

Bit	Meaning
0–7	Schedule Index Indicates the LIN schedule that the interface currently is running. This index refers to a LIN schedule that you requested using the `nxWriteState` function. It indexes the array of schedules represented in the XNET Session Interface:LIN:Schedule Names property. This index applies only when the LIN interface is running as a master. If the LIN interface is running as a slave only, this element should be ignored.
8–31	Reserved

Last Error Table

The following table lists each value for **last error**, along with a description, and applicable use of **last received**, **last expected**, and **last identifier**. In the **last error** column, the decimal value is shown in parentheses after the string name.

Last Error	Description	Last Received	Last Expected	Last Identifier
None (0)	No bus error has occurred since the previous communication state read.	0 (N/A)	0 (N/A)	0 (N/A)
Unknown ID (1)	Received a frame identifier that is not valid (0–63).	0 (N/A)	0 (N/A)	0 (N/A)
Form (2)	The form of a received frame is incorrect. For example, the database specifies 8 bytes of payload, but you receive only 4 bytes.	0 (N/A)	0 (N/A)	Received frame ID
Framing (3)	The byte framing is incorrect (for example, a missing stop bit).	0 (N/A)	0 (N/A)	Received frame ID
Readback (4)	The interface transmitted a byte, but the value read back from the transceiver was different. This often is caused by a cabling problem, such as noise.	Value read back	Value transmitted	Received frame ID
Timeout (5)	Receiving the frame took longer than the LIN-specified timeout.	0 (N/A)	0 (N/A)	Received frame ID
Checksum (6)	The received checksum was different than the expected checksum.	Received checksum	Calculated checksum	Received frame ID

`StateID = nxState_SessionInfo:`

`StateValue` must point to a u32. It contains the current session running state. The running states are:

`nxSessionInfoState_Stopped (0)`

All frames in the session are stopped.

`nxSessionInfoState_Started (1)`

All frames in the session are started.

`nxSessionInfoState_Mix (2)`

Some frames in the session are started while other frames are stopped. This state may occur when using `nxStart` or `nxStop` with the Session Only option.

`nxStatus_t* Fault`

Returns a numeric code you can use to obtain a description of the fault. If no fault occurred, the fault code is 0.

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

To obtain a fault description, pass the fault code to `nxStatusToString`.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

You can use `nxReadState` with any XNET session mode.

Your application can use `nxReadState` to check for problems on the network independently from other aspects of your application. For example, you intentionally may introduce noise into the CAN cables to test how your ECU behaves under these conditions. When you do this, you do not want the status of NI-XNET functions to return errors, because this may cause your application to stop. Your application can use `nxReadState` to read the network state quickly as data, so that it does not introduce errors into the flow of your code.

Alternately, to log bus errors, you can set the Interface:Bus Error Frames to Input Stream? property to cause CAN and LIN bus errors to be logged as a special frame (refer to *Special Frames* for more information) into a Frame Stream Input queue.

nxSetProperty

Purpose

Sets an XNET session property.

Format

```
nxStatus_t nxSetProperty (
    nxSessionRef_t SessionRef,
    u32 PropertyID,
    u32 PropertySize,
    void * PropertyValue);
```

Inputs

`nxSessionRef_t SessionRef`

The session to set the property for. This session is returned from `nxCreateSession`.

`u32 PropertyID`

The ID of the property to set. The appropriate constants are listed in the Properties section and defined in `nixnet.h`.

`u32 PropertySize`

The number of bytes provided for the buffer passed to `PropertyValue`. This can be a fixed-size (for example, 4 bytes for a u32 property) or variable-sized buffer (for example, for a string property).

`void * PropertyValue`

Contains the value to set for the desired property.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

Refer to the following sections for information about properties you can use with this function:

- *XNET Device Properties*
- *XNET Interface Properties*
- *XNET Session Properties*
- *XNET System Properties*

nxSetSubProperty

Purpose

Sets a property of a frame or signal within an XNET session.

Format

```
nxStatus_t nxSetSubProperty (
    nxSessionRef_t SessionRef,
    u32 ActiveIndex,
    u32 PropertyID,
    u32 PropertySize,
    void * PropertyValue);
```

Inputs

`nxSessionRef_t SessionRef`

The session to set the property for. This session is returned from `nxCreateSession`.

`u32 ActiveIndex`

Identifies the frame or signal within the session. It is the index to the list given in `nxCreateSession`.

`u32 PropertyID`

The ID of the property to set. The properties to use with this function are listed in the *Frame Properties* section for the session. Within your code, applicable `PropertyID` values begin with the prefix `nxProp_SessionSub`.

`u32 PropertySize`

The number of bytes provided for the buffer passed to `PropertyValue`. This can be a fixed-size (for example, 4 bytes for a u32 property) or variable-sized buffer (for example, for a string property).

`void * PropertyValue`

Contains the value to set for the desired property.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

nxStart

Purpose

Starts communication for the specified XNET session.

Format

```
nxStatus_t nxStart (
    nxSessionRef_t SessionRef,
    u32 Scope);
```

Inputs

`nxSessionRef_t SessionRef`

The session to start. This session is returned from `nxCreateSession`.

`u32 Scope`

Describes the impact of this operation on the underlying state models for the session and its interface.

Normal (0) The session is started followed by starting the interface. This is equivalent to calling `nxStart` with the Session Only `Scope` followed by calling `nxStart` with the Interface Only `Scope`.

Session Only (1) The session is placed into the Started state (refer to *State Models*). If the interface is in the Stopped state before this function runs, the interface remains in the Stopped state, and no communication occurs with the bus. To have multiple sessions start at exactly the same time, start each session with the Session Only `Scope`. When you are ready for all sessions to start communicating on the associated interface, call `nxStart` with the Interface Only `Scope`. Starting a previously started session is considered a no-op. This operation sends the command to start the session, but does not wait for the session to be started. It is ideal for a real-time application where performance is critical.

Interface Only (2) If the underlying interface is not previously started, the interface is placed into the Started state (refer to *State Models*). After the interface starts communicating, all previously started sessions can transfer data to and from the bus. Starting a previously started interface is considered a no-op.

Session Only Blocking (3) The session is placed in the Started state (refer to *State Models*). If the interface is in the Stopped state before this function runs, the interface remains in the Stopped state, and no communication occurs with the bus. To have

multiple sessions start at exactly the same time, start each session with the Session Only `Scope`. When you are ready for all sessions to start communicating on the associated interface, call `nxStart` with the Interface Only `Scope`. Starting a previously started session is considered a no-op. This operation waits for the session to start before completing.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

Because the session is started automatically by default, this function is optional. This function is for more advanced applications to start multiple sessions in a specific order. For more information about the automatic start feature, refer to the Auto Start? property.

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

- **Sessions**: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.
- **Interface**: The interface physically connects to the bus and transmits (or receives) data for the sessions.

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

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

nxStatusToString

Purpose

Converts a status code returned from a function into a descriptive string.

Format

```
void _NXFUNC nxStatusToString (
    nxStatus_t Status,
    u32 SizeofString,
    char * StatusDescription);
```

Inputs

`nxStatus_t Status`

The status code to be explained.

`u32 SizeofString`

The size of the string provided to store the explanation of the status code.

Outputs

`char * StatusDescription`

The string in which the explanation of the status code will be stored.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function converts a status code returned from a function into a descriptive string.

`SizeofString` is the size allocated for the string. The description is truncated to size `SizeofString` if needed, but a size of 2048 characters is large enough to hold any description. The text returned in `StatusDescription` is null-terminated, so it can be used with ANSI C functions such as `printf`.

nxStop

Purpose

Stops communication for the specified XNET session.

Format

```
nxStatus_t nxStop (
    nxSessionRef_t SessionRef,
    u32 Scope);
```

Inputs

`nxSessionRef_t SessionRef`

The session to stop. This session is returned from `nxCreateSession`.

`u32 Scope`

Describes the impact of this operation on the underlying state models for the session and its interface.

Normal (0) The session is stopped. If this is the last session stopped on the interface, the interface is also stopped. If any other sessions are running on the interface, this call is treated just like the Session Only `Scope`, to avoid disruption of communication on the other sessions.

Session Only (1) The session is placed in the Stopped state (refer to *State Models*). If the interface was in the Started or Running state before this function is called, the interface remains in that state and communication continues, but data from this session does not transfer. This `Scope` generally is not necessary, as the Normal `Scope` only stops the interface if there are no other running sessions. This operation sends the command to stop the session, but does not wait for the session to be stopped. It is ideal for a real-time application where performance is critical.

Interface Only (2) The underlying interface is placed in the Stopped state (refer to *State Models*). This prevents all communication on the bus, for all sessions. This allows you modify certain properties that require the interface to be stopped (for example, CAN baud rate). All sessions remain in the Started state. To have multiple sessions stop at exactly the same time, first stop the interface with the Interface Only `Scope` and then stop each session with either the Normal or Session Only `Scope`.

Session Only Blocking (3) The session is placed in the Stopped state (refer to *State Models*). If the interface was in the Started or Running state before this function is called, the interface remains in that state and communication continues, but data from this session does not transfer. This `Scope` generally is not necessary, as the Normal `Scope` stops the interface only if there are no other running sessions. This operation waits for the session to stop before completing.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

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

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

- **Sessions**: You can create one or more sessions, each of which contains frames or signals to be transmitted (or received) on the bus.
- **Interface**: The interface physically connects to the bus and transmits (or receives) data for the sessions.

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

nxSystemClose

Purpose

Closes a system session.

Format

```
nxStatus_t _NXFUNC nxSystemClose (
    nxSessionRef_t SystemRef);
```

Inputs

`nxSessionRef_t SystemRef`

The reference to the system session to close.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function is used to close a system session.

nxSystemOpen

Purpose

Opens a special system session.

Format

```
nxStatus_t _NXFUNC nxSystemOpen (
    nxSessionRef_t * SystemRef);
```

Outputs

`nxSessionRef_t * SystemRef`

The reference to the opened system session.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function opens a special system session.

The system session is not used to read/write on the network (as with sessions created using `nxCreateSession`). Use the system session to interact with the NI driver and interface hardware.

For example, you can traverse through properties to find all NI-XNET interfaces in your system.

The following functions are supported for the system session:

- `nxGetProperty`: Get a property with prefix `nxPropSys_`, `nxPropDev_`, or `nxPropIntf_`.
- `nxGetPropertySize`: Get a string property size.
- `nxBlink`: Blink LED(s) on the interface.

nxWait

Purpose

Waits for a certain condition to occur.

Format

```
nxStatus_t _NXFUNC nxWait (
    nxSessionRef_t SessionRef,
    u32 Condition,
    u32 ParamIn,
    f64 Timeout,
    u32 * ParamOut);
```

Inputs

`nxSessionRef_t SessionRef`

The session to which the wait is applied.

`u32 Condition`

Specifies the condition to wait for.

`u32 ParamIn`

An optional parameter that provides simple data to qualify the condition.

`f64 Timeout`

Specifies the maximum amount of time in seconds to wait.

Outputs

`u32 * ParamOut`

An optional parameter that provides simple data to qualify the condition that occurred.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

This function waits for a condition to occur for the session.

The `Condition` parameter specifies to wait for one of the following conditions.

nxCondition_TransmitComplete

All frames written for the session have been transmitted on the bus. This condition applies to CAN, LIN, and FlexRay. This condition is state based, and the state is Boolean (true/false). The `ParamIn` and `ParamOut` parameters are ignored for this condition because `nxWait` simply waits for the state to become true.

nxCondition_IntfCommunicating

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

After this wait succeeds, calls to `nxReadState` will return:

- `nxState_CANComm`: `nxCANCommState_ErrorActive`
- `nxState_CANComm`: `nxCANCommState_ErrorPassive`
- `nxState_TimeCommunicating`: Valid time for communication (invalid time of 0 prior)

This condition is state based. The `ParamIn` and `ParamOut` parameters are ignored for this condition because `nxWait` simply waits for a communicating state.

nxCondition_IntfRemoteWakeup

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

This wait is used for LIN when you set the XNET Session Interface:LIN:Sleep property to `nxLINSleep_RemoteSleep` or `nxLINSleep_LocalSleep`. When asleep, if a remote LIN ECU transmits the wakeup pattern (break), the XNET LIN interface detects this transmission and wakes up. This wait detects that remote wakeup.

This condition is state based. The `ParamIn` and `ParamOut` parameters are ignored for this condition, because `nxWait` simply waits for the remote wakeup to occur.

nxWriteFrame

Purpose

Writes data to a session as an array of raw bytes.

Format

```
nxStatus_t nxWriteFrame (
    nxSessionRef_t SessionRef,
    void * Buffer,
    u32 NumberOfBytesForFrames,
    f64 Timeout);
```

Inputs

`nxSessionRef_t SessionRef`

The session to write. This session is returned from `nxCreateSession`. The session mode must be Frame Output Stream Mode, Frame Output Queued Mode, or Frame Output Single-Point Mode.

`void * Buffer`

Provides the array of bytes, representing frames to transmit.

The raw bytes encode one or more frames using the Raw Frame Format. This frame format is the same for read and write of raw data and also is used for log file examples.

If needed, you can write data for a partial frame. For example, if a complete raw frame is 24 bytes, you can write 12 bytes, then write the next 12 bytes. You typically do this when you are reading raw frame data from a logfile and want to avoid iterating through the data to detect the start and end of each frame.

Note For PDU sessions, the array of bytes represents the payload of the specified PDU only, not that of the entire frame.

For information about which elements of the raw frame are applicable, refer to *Raw Frame Format*.

The data you write is queued up for transmit on the network. Using the default queue configuration for this mode, you can safely write 1536 frames if you have a sufficiently long timeout. To write more data, refer to the XNET Session Number of Values Unused property to determine the actual amount of queue space available for writing.

For an example of how this data applies to network traffic, refer to *Frame Output Stream Mode*, *Frame Output Queued Mode*, or *Frame Output Single-Point Mode*.

Additionally, you can use `nxWriteFrame` on any signal or frame input session if it contains CAN Event Remote frames (refer to CAN:Timing Type). In this case, it signals

an event to transmit those remote frames. The `Buffer` parameter is ignored, and you can set it to NULL in that case.

`u32 NumberOfBytesForFrames`

The size (in bytes) of the buffer passed to `Buffer`. This is used to calculate the number of frames to transmit.

`f64 Timeout`

The time to wait for the raw data to be queued up for transmit.

The timeout is represented as 64-bit floating-point in units of seconds.

If `Timeout` is positive, `nxWriteFrame` waits up to that timeout for space to become available in queues. If the space is not available prior to the timeout, a timeout error is returned.

If `Timeout` is negative, `nxWriteFrame` waits indefinitely for space to become available in queues.

If `Timeout` is 0, `nxWriteFrame` does not wait and immediately returns with a timeout error if all data cannot be queued. Regardless of the timeout used, if a timeout error occurs, none of the data is queued, so you can attempt to call `nxWriteFrame` again at a later time with the same data.

If the session mode is Frame Output Single-Point, you must set `Timeout` to 0.0. Because this mode writes the most recent value of each frame, `Timeout` does not apply.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

The raw bytes encode one or more frames using the Raw Frame Format. The session must use a mode of Frame Output Stream, Frame Output Queued, or Frame Output Single-Point. The raw frame format is protocol independent.

The raw frames are associated to the session's list of frames as follows:

- Frame Output Stream Mode: Array of all frame values for transmit (list ignored). For LIN, if the payload length is 0, only the header part of the LIN frame is transmitted. If the payload length is nonzero, the header and response parts of the LIN frame are transmitted.
- Frame Output Queued Mode: Array of frame values to transmit for the single frame specified in the list.
- Frame Output Single-Point Mode: Array of single frame values, one for each frame specified in the list.

- Any signal or frame input mode: The `Buffer` parameter is ignored, and you can set it to NULL. The function transmits an event remote frame.

nxWriteSignalSinglePoint

Purpose

Writes data to a session of Signal Output Single-Point Mode.

Format

```
nxStatus_t nxWriteSignalSinglePoint (
    nxSessionRef_t SessionRef,
    f64 * ValueBuffer,
    u32 SizeOfValueBuffer);
```

Inputs

`nxSessionRef_t SessionRef`

The session to write. This session is returned from `nxCreateSession`. The session mode must be Signal Output Single-Point.

`f64 * ValueBuffer`

Provides a one-dimensional array of signal values. Each signal value is scaled, 64-bit floating point.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

The data provides the value for the next transmit of each signal. If `nxWriteSignalSinglePoint` is called twice before the next transmit, the transmitted frame uses signal values from the second call to `nxWriteSignalSinglePoint`.

For an example of how this data applies to network traffic, refer to *Signal Output Single-Point Mode*.

A trigger signal written a value of 0.0 suppresses writing of its frame's data; writing a value not equal to 0.0 enables it. For more information about trigger signals, refer to *Signal Output Single-Point Mode*.

`u32 SizeOfValueBuffer`

Should be set to the size (in bytes) of the array passed to `ValueBuffer`. If this is too small to fit one element for each signal in the session, an error is returned.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

nxWriteSignalWaveform

Purpose

Writes data to a session of Signal Output Waveform Mode. The data represents a waveform of resampled values for each signal in the session.

Format

```
nxStatus_t nxWriteSignalWaveform (
    nxSessionRef_t SessionRef,
    f64 Timeout,
    f64 * ValueBuffer,
    u32 SizeOfValueBuffer);
```

Inputs

`nxSessionRef_t SessionRef`

The session to write. This session is returned from `nxCreateSession`. The session mode must be Signal Output Waveform.

`f64 Timeout`

The time to wait for the data to be queued for transmit. The timeout does not wait for frames to be transmitted on the network (refer to `nxWait`).

The timeout is represented as 64-bit floating-point in units of seconds.

If `Timeout` is positive, `nxWriteSignalWaveform` waits up to that timeout for space to become available in queues. If the space is not available prior to the timeout, a timeout error is returned.

If `Timeout` is negative, `nxWriteSignalWaveform` waits indefinitely for space to become available in queues.

If `Timeout` is 0, `nxWriteSignalWaveform` does not wait and immediately returns an error if all data cannot be queued. Regardless of the timeout used, if a timeout error occurs, none of the data is queued, so you can attempt to call `nxWriteSignalWaveform` again at a later time with the same data.

`f64* ValueBuffer`

Provides a two-dimensional array of f64 samples. First, *N* samples are reserved for the first signal in the session, then *N* samples for the second, and so on. *N* * (number of signals in the session) * sizeof (f64) should be passed in `SizeOfValueBuffer` to recalculate *N*.

The data you write is queued for transmit on the network. Using the default queue configuration for this mode, and assuming a 1000 Hz resample rate, you can safely write 64 elements if you have a sufficiently long timeout. To write more data, refer to the

XNET Session Number of Values Unused property to determine the actual amount of queue space available for writing.

For an example of how this data applies to network traffic, refer to *Signal Output Waveform Mode*.

Each array element corresponds to a signal configured for the session. The order of signals in the array corresponds to the order in the session list.

`u32 SizeOfValueBuffer`

Should be set to the size (in bytes) of the array passed to `ValueBuffer`. The number of samples to be written (*N*) per signal is calculated from this size. Set this to (*N*) * (number of signals in the session) * sizeof (f64).

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

The data represents a waveform for each signal in the session.

nxWriteSignalXY

Purpose

Writes data to a session of Signal Output XY Mode. The data represents a sequence of signal values for transmit using each frame's timing as the database specifies.

Format

```
nxStatus_t nxWriteSignalXY (
    nxSessionRef_t SessionRef,
    f64 Timeout,
    f64 * ValueBuffer,
    u32 SizeOfValueBuffer,
    nxTimestamp_t * TimestampBuffer,
    u32 SizeOfTimestampBuffer,
    u32 * NumPairsBuffer,
    u32 SizeOfNumPairsBuffer);
```

Inputs

`nxSessionRef_t SessionRef`

The session to write. This session is returned from `nxCreateSession`. The session mode must be Signal Output XY.

`f64 Timeout`

The time to wait for the data to be queued for transmit. The timeout does not wait for frames to be transmitted on the network (refer to `nxWait`).

The timeout is represented as 64-bit floating-point in units of seconds.

If `Timeout` is positive, `nxWriteSignalXY` waits up to that timeout for space to become available in queues. If the space is not available prior to the timeout, a timeout error is returned.

If `Timeout` is negative, `nxWriteSignalXY` waits indefinitely for space to become available in queues.

If `Timeout` is 0, `nxWriteSignalXY` does not wait and immediately returns with a timeout error if all data cannot be queued. Regardless of the timeout used, if a timeout error occurs, none of the data is queued, so you can attempt to call `nxWriteSignalXY` again at a later time with the same data.

`f64* ValueBuffer`

Provides a two-dimensional array of f64 samples. First, *N* samples are reserved for the first signal in the session, then *N* samples for the second, and so on. *N* * (number of signals in the session) * sizeof (f64) should be passed in `SizeOfValueBuffer` to recalculate *N*.

The data you write is queued for transmit on the network. Using the default queue configuration for this mode, you can safely write 64 elements if you have a sufficiently long timeout. To write more data, refer to the XNET Session Number of Values Unused property to determine the actual amount of queue space available for writing.

For an example of how this data applies to network traffic, refer to *Signal Output XY Mode*.

`u32 SizeOfValueBuffer`

The size (in bytes) of the array passed to `ValueBuffer`.

`nxTimestamp_t* TimestampBuffer`

Provides a two-dimensional array of timestamps. First, *N* timestamps are reserved for the first signal in the session, then *N* timestamps for the second and so on. *N* * (number of signals in the session) * sizeof (f64) should be passed in `SizeOfTimestampBuffer` to recalculate *N*.

The timestamps are given in 100 ns increments since Jan 1, 1601, 12:00 AM UTC.

This array is for future expansion; it is not used in the current implementation of NI-XNET. Pass NULL on input.

`u32 SizeOfTimestampBuffer`

The size (in bytes) of the array passed to `TimestampBuffer`.

This value is for future expansion; it is not used in the current implementation of NI-XNET. Pass 0 on input.

`u32* NumPairsBuffer`

Provides an one-dimensional array of signal/timestamp pair counts, one for each signal in the session. Upon input, the samples and timestamps for signal #(i) in the preceding arrays are valid up to, but not including, index `NumPairsBuffer`[i] (zero based) and are written up to that point.

`u32 SizeOfNumPairsBuffer`

The size (in bytes) of the array passed to `NumPairsBuffer`. For each signal in the session, an array element should be provided. If the buffer is too small, an error is returned.

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

The data represents an XY plot of timestamp/value pairs for each signal in the session.

nxWriteState

Purpose

Writes communication states of an XNET session.

Format

```
nxStatus_t nxWriteState (
    nxSessionRef_t SessionRef,
    u32 StateID,
    u32 StateSize,
    void * StateValue);
```

Inputs

`nxSessionRef_t SessionRef`

The session to write. This session is returned from `nxCreateSession`. The session protocol must be LIN.

`u32 StateID`

Indicates the state to be written. Possible values are:

`nxState_LINScheduleChange`

Changes the LIN schedule.

`nxState_FlexRaySymbol`

Transmits a FlexRay symbol.

`nxState_LINDiagnosticScheduleChange`

Changes the LIN diagnostic schedule.

The value determines the format to be written to `StateValue`.

`u32 StateSize`

Indicates the size of the buffer provided for `StateValue`.

`void* StateValue`

Writes the desired state. Formats and values are:

`StateID = nxState_LINScheduleChange`

`StateValue` must point to a u32 buffer that contains the index to the schedule table that the LIN master executes. The schedule tables are sorted the way they are returned from the database with the XNET Cluster Schedules property.

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

`StateID = nxState_FlexRaySymbol`

`StateValue` must point to a u32 buffer that contains the value 0.

`StateID = nxState_LINDiagnosticScheduleChange`

`StateValue` must point to a u32 buffer that contains the diagnostic schedule that the LIN master executes. Possible values are:

- `nxLINDiagnosticSchedule_NULL`: The master does not execute any diagnostic schedule. No master request or slave response headers are transmitted on the LIN.
- `nxLINDiagnosticSchedule_MasterReq`: The master executes a diagnostic master request schedule (transmits a master request header onto the LIN) if it can. First, a master request schedule must be defined for the LIN cluster in the imported or in-memory database. Otherwise, error nxErrDiagnosticScheduleNotDefined is returned when attempting to set this value. Second, the master must have a frame output queued session created for the master request frame, and there must be one or more new master request frames pending in the queue. If no new frames are pending in the output queue, no master request header is transmitted. This allows the timing of master request header transmission to be controlled by the timing of master request frame writes to the output queue.

  If there are no normal schedules pending, the master is effectively in diagnostics-only mode, and master request headers are transmitted at a rate determined by the slot delay defined for the master request frame slot in the master request schedule or the `nxPropSession_IntfLINDiagSTmin` time, whichever is greater, and the state of the master request frame output queue as described above.

  If there are normal schedules pending, the master is effectively in diagnostics-interleaved mode, and a master request header transmission is inserted between each complete execution of a run-once or run-continuous schedule, as long as the `nxPropSession_IntfLINDiagSTmin` time has been met, and there are one or more new master request frames pending in the master request frame output queue.
- `nxLINDiagnosticSchedule_SlaveResp`: The master executes a diagnostic slave response schedule (transmits a slave response header onto the LIN) if it is able to. A slave response schedule must be defined for the LIN cluster in the imported or in-memory database. Otherwise, error nxErrDiagnosticScheduleNotDefined is returned when attempting to set this value.

  If there are no normal schedules pending, the master is effectively in diagnostics-only mode, and slave response headers are transmitted at the rate of the slot delay defined for the slave response frame slot in the slave response

schedule. The addressed slave may or may not respond to each header, depending on its specified P2min and STmin timings.

If there are normal schedules pending, the master is effectively in diagnostics-interleaved mode, and a slave response header transmission is inserted between each complete execution of a run-once or run-continuous schedule. Here again, the addressed slave may or may not respond to each header, depending on its specified P2min and STmin timings.

Outputs

Return Value

`nxStatus_t`

The error code the function returns in the event of an error or warning. A value of 0 indicates success. A positive value indicates a warning. A negative value indicates an error.

Description

You can use `nxWriteState` with an XNET LIN master session to set the schedule that the LIN master executes.

You also can use `nxWriteState` with an XNET FlexRay session to transmit a symbol on the FlexRay bus.

Executing this function on any other type of session causes an error.

You can use `nxWriteState` with an XNET LIN master session to set the diagnostic schedule that the LIN master executes. Use this state to transmit master request messages and query for slave response messages after node configuration has been performed. Node configuration should be handled using `nxState_LINScheduleChange`. Write the node configuration schedule defined for the LIN cluster using `nxState_LINScheduleChange`, so that it is the first schedule executed for the LIN, with a run mode of once. The data for each node configuration service request entry in the node configuration schedule is automatically transmitted by the master. After the node configuration schedule has completed, use `nxState_LINDiagnosticScheduleChange` to run diagnostic schedules, or `nxState_LINScheduleChange` to run normal schedules.

Properties

This section includes the XNET properties.

XNET Cluster Properties

This section includes the XNET Cluster properties.

Baud Rate

Data Type	Direction	Required?	Default
u32	Read/Write	No	0

Property Class

XNET Cluster

Property ID

`nxPropClst_BaudRate`

Description

The Baud Rate property sets the baud rate all cluster nodes use. This baud rate represents the rate from the database, so it is read-only from the session. Use a session interface property (for example, XNET Session Interface:Baud Rate) to override the database baud rate with an application-specific baud rate.

CAN

For CAN, this rate can be 33333, 40000, 50000, 62500, 80000, 83333, 100000, 125000, 160000, 200000, 250000, 400000, 500000, 800000, or 1000000. Some transceivers may support only a subset of these values.

If you need values other than these, use the custom settings as described in the XNET Session Interface:Baud Rate property.

FlexRay

For FlexRay, this rate can be 2500000, 5000000, or 10000000.

LIN

For LIN, this rate can be 2400–20000 inclusive.

If you need values other than these, use the custom settings as described in the XNET Session Interface:Baud Rate property.

CAN:FD Baud Rate

Data Type	Direction	Required?	Default
u32	Read/Write	No	0

Property Class

XNET Cluster

Property ID

`nxPropClst_CanFdBaudRate`

Description

The FD Baud Rate property sets the fast data baud rate for the CAN FD + BRS CAN:I/O Mode property. This property represents the database fast data baud rate for the CAN FD + BRS I/O Mode. Refer to the CAN:I/O Mode property for a description of this mode. Use a session interface property (for example, Interface:CAN:FD Baud Rate) to override the database fast baud rate with an application-specific fast baud rate.

NI-XNET CAN hardware currently accepts the following numeric baud rates: 200000, 250000, 400000, 500000, 800000, 1000000, 1250000, 1600000, 2000000, 2500000, 4000000, 5000000, and 8000000. Some transceivers may support only a subset of these values.

If you need values other than these, use the custom settings as described in the Interface:CAN:FD Baud Rate property.

CAN:I/O Mode

Data Type	Direction	Required?	Default
u32	Read/Write	No	0

Property Class

XNET Cluster

Property ID

`nxPropClst_CanIoMode`

Description

This property specifies the CAN I/O Mode of the cluster. It is a ring of three values:

Enumeration	Value	Meaning
nxCANioMode_CAN	0	This is the default CAN 2.0 A/B standard I/O mode as defined in ISO 11898-1:2003. A fixed baud rate is used for transfer, and the payload length is limited to 8 bytes.
nxCANioMode_CAN_FD	1	This is the CAN FD mode as specified in the *CAN with Flexible Data-Rate* specification, version 1.0. Payload lengths up to 64 are allowed, but they are transmitted at a single fixed baud rate (defined by the XNET Cluster Baud Rate or XNET Session Interface:Baud Rate properties).
nxCANioMode_CAN_FD_BRS	2	This is the CAN FD as specified in the *CAN with Flexible Data-Rate* specification, version 1.0, with the optional Baud Rate Switching enabled. The same payload lengths as CAN FD mode are allowed; additionally, the data portion of the CAN frame is transferred at a different (higher) baudrate (defined by the CAN:FD Baud Rate or XNET Session Interface:CAN:FD Baud Rate properties).

Comment

Data Type	Direction	Required?	Default
char *	Read/Write	No	Empty String

Property Class

XNET Cluster

Property ID

`nxPropClst_Comment`

Description

A comment describing the cluster object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
i32	Read Only	N/A	N/A

Property Class

XNET Cluster

Property ID

`nxPropClst_ConfigStatus`

Description

The cluster object configuration status.

Configuration Status returns an NI-XNET error code. You can pass the value to the error code input of `nxStatusToString` to convert it to a text description of the configuration problem.

By default, incorrectly configured clusters in the database are not returned from the XNET Database Clusters property because they cannot be used in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When the configuration status of a cluster becomes invalid after the database has been opened, the cluster still is returned from the Clusters property even if ShowInvalidFromOpen? is false.

Database

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET Cluster

Property ID

`nxPropClst_DatabaseRef`

Description

Refnum to the cluster parent database.

The parent database is defined when the cluster object is created. You cannot change it afterwards.

ECUs

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET Cluster

Property ID

`nxPropClst_ECURefs`

Description

ECUs in this cluster.

Returns an array of references to all ECUs defined in this cluster. An ECU is assigned to a cluster when the ECU object is created. You cannot change this assignment afterwards.

To add an ECU to a cluster, use `nxdbCreateObject`. To remove an ECU from the cluster, use `nxdbDeleteObject`.

FlexRay:Action Point Offset

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayActPtOff`

Description

This property specifies the number of macroticks (MT) that the action point is offset from the beginning of a static slot or symbol window.

This property corresponds to the global cluster parameter `gdActionPointOffset` in the *FlexRay Protocol Specification*.

The action point is that point within a given slot where the actual transmission of a frame starts. This is slightly later than the start of the slot, to allow for a clock drift between the network nodes.

The range for this property is 1–63 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:CAS Rx Low Max

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayCASRxLMax`

Description

This property specifies the upper limit of the collision avoidance symbol (CAS) acceptance window. The CAS symbol is transmitted by the FlexRay interface (node) during the symbol window within the communication cycle. A receiving FlexRay interface considers the CAS to be valid if the pattern's low level is within 29 gdBit (`cdCASRxLowMin`) and CAS Rx Low Max.

This property corresponds to the global cluster parameter `gdCASRxLowMax` in the *FlexRay Protocol Specification*.

The values for this property are in the range 67–99 gdBit.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Channels

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayChannels`

Description

This property specifies the FlexRay channels used in the cluster. Frames defined in this cluster are expected to use the channels this property specifies. Refer to the XNET Frame FlexRay:Channel Assignment property.

This property corresponds to the global cluster parameter `gChannels` in the *FlexRay Protocol Specification*.

A FlexRay cluster supports two independent network wires (channels A and B). You can choose to use both or only one in your cluster.

The values (enumeration) for this property are:

1 Channel A only

2 Channel B only

3 Channels A and B

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Cluster Drift Damping

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayClstDriftDmp`

Description

This property specifies the cluster drift damping factor, based on the longest microtick used in the cluster. Use this global FlexRay parameter to compute the local cluster drift damping factor for each cluster node. You can access the local cluster drift for the XNET FlexRay interface from the XNET Session Interface:FlexRay:Cluster Drift Damping property.

This property corresponds to the global cluster parameter `gdClusterDriftDamping` in the *FlexRay Protocol Specification*.

The values for this property are in the range 0–5 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Cold Start Attempts

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayColdStAts`

Description

This property specifies the maximum number of times a node in this cluster can start the cluster by initiating schedule synchronization. This global cluster parameter is applicable to all cluster notes that can perform a coldstart (send startup frames).

This property corresponds to the global cluster parameter `gColdStartAttempts` in the *FlexRay Protocol Specification*.

The values for this property are in the range 2–31.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Cycle

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayCycle`

Description

This property specifies the duration of one FlexRay communication cycle, expressed in microseconds.

This property corresponds to the global cluster parameter `gdCycle` in the *FlexRay Protocol Specification*.

All frame transmissions complete within a cycle. After this time, the frame transmissions restart with the first frame in the next cycle. The communication cycle counts increment from 0–63, after which the cycle count resets back to 0.

The range for this property is 10–16000 µs.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Dynamic Segment Start

Data Type	Direction	Required?	Default
u32	Read Only	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayDynSegStart`

Description

This property specifies the start of the dynamic segment, expressed as the number of macroticks (MT) from the start of the cycle.

The range for this property is 8–15998 MT.

This property is calculated from other cluster properties. It is based on the total static segment size. It is set to 0 if the FlexRay:Number of Minislots property is 0 (no dynamic segment exists).

FlexRay:Dynamic Slot Idle Phase

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayDynSlotIdlPh`

Description

This property specifies the dynamic slot idle phase duration.

This property corresponds to the global cluster parameter `gdDynamicSlotIdlePhase` in the *FlexRay Protocol Specification*.

The values for this property are in the range 0–2 minislots.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Latest Guaranteed Dynamic Slot

Data Type	Direction	Required?	Default
u32	Read Only	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayLatestGuarDyn`

Description

This property specifies the highest slot ID in the dynamic segment that still can transmit a full-length (for example, Payload Length Dynamic Maximum) frame, provided all previous slots in the dynamic segment have transmitted full-length frames also.

A larger slot ID cannot be guaranteed to transmit a full-length frame in each cycle (although a frame might go out depending on the dynamic segment load).

The range for this property is 2–2047 slots.

This read-only property is calculated from other cluster properties. If the Number of Minislots is zero, no dynamic slots exist, and this property returns 0. Otherwise, the Number of Minislots is used along with Payload Length Dynamic Maximum to determine the latest dynamic slot guaranteed to transmit in the next cycle. In other words, when all preceding dynamic slots transmit with Payload Length Dynamic Maximum, this dynamic slot also can transmit with Payload Length Dynamic Maximum, and its frame ends prior to the end of the dynamic segment.

FlexRay:Latest Usable Dynamic Slot

Data Type	Direction	Required?	Default
u32	Read Only	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayLatestUsableDyn`

Description

This property specifies the highest slot ID in the dynamic segment that can still transmit a full-length (that is, Payload Length Dynamic Maximum) frame, provided no other frames have been sent in the dynamic segment.

A larger slot ID cannot transmit a full-length frame (but could probably still transmit a shorter frame).

The range for this property is 2–2047.

This read-only property is calculated from other cluster properties. If the Number of Minislots is zero, no dynamic slots exist, and this property returns 0. Otherwise, Number of Minislots is used along with Payload Length Dynamic Maximum to determine the latest dynamic slot that can be used when all preceding dynamic slots are empty (zero payload length). In other words, this property is calculated under the assumption that all other dynamic slots use only one minislot, and this dynamic slot uses the number of minislots required to deliver the maximum payload. The frame for this dynamic slot must end prior to the end of the dynamic segment. Any frame transmitted in a preceding dynamic slot is likely to preclude this slot's frame.

FlexRay:Listen Noise

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayLisNoise`

Description

This property specifies the upper limit for the startup and wakeup listen timeout in the presence of noise. It is used as a multiplier for the Interface:FlexRay:Listen Timeout property.

This property corresponds to the global cluster parameter `gListenNoise` in the *FlexRay Protocol Specification*.

The values for this property are in the range 2–16.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Macro Per Cycle

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayMacroPerCycle`

Description

This property specifies the number of macroticks in a communication cycle. For example, if the FlexRay cycle has a duration of 5 ms (5000 µs), and the duration of a macrotick is 1 µs, the XNET Cluster FlexRay:Macro Per Cycle property is 5000.

This property corresponds to the global cluster parameter `gMacroPerCycle` in the *FlexRay Protocol Specification*.

The macrotick (MT) is the basic timing unit in the FlexRay cluster. Nearly all timing-dependent properties are expressed in terms of macroticks.

The range for this property is 10–16000 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Macrotick

Data Type	Direction	Required?	Default
f64	Read Only	N/A	Calculated from Other Cluster Parameters

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayMacrotick`

Description

This property specifies the duration of the clusterwide nominal macrotick, expressed in microseconds.

This property corresponds to the global cluster parameter `gdMacrotick` in the *FlexRay Protocol Specification*.

The macrotick (MT) is the basic timing unit in the FlexRay cluster. Nearly all timing-dependent properties are expressed in terms of macroticks.

The range for this property is 1–6 µs.

This property is calculated from the XNET Cluster FlexRay:Cycle and FlexRay:Macro Per Cycle properties and rounded to the nearest permitted value.

FlexRay:Max Without Clock Correction Fatal

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayMaxWoClkCorFat`

Description

This property defines the number of consecutive even/odd cycle pairs with missing clock correction terms that cause the controller to transition from the Protocol Operation Control status of Normal Active or Normal Passive to the Halt state. Use this global parameter as a threshold for testing the clock correction failure counter.

This property corresponds to the global cluster parameter `gMaxWithoutClockCorrectionFatal` in the *FlexRay Protocol Specification*.

The values for this property are in the range 1–15 even/odd cycle pairs.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Max Without Clock Correction Passive

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayMaxWoClkCorPas`

Description

This property defines the number of consecutive even/odd cycle pairs with missing clock correction terms that cause the controller to transition from the Protocol Operation Control status of Normal Active to Normal Passive. Use this global parameter as a threshold for testing the clock correction failure counter.

Note This property, Max Without Clock Correction Passive, <= Max Without Clock Correction Fatal <= 15.

This property corresponds to the global cluster parameter `gMaxWithoutClockCorrectionPassive` in the *FlexRay Protocol Specification*.

The values for this property are in the range 1–15 even/odd cycle pairs.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Minislot

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayMinislot`

Description

This property specifies the duration of a minislot, expressed in macroticks (MT).

This property corresponds to the global cluster parameter `gdMinislot` in the *FlexRay Protocol Specification*.

In the dynamic segment of the FlexRay cycle, frames can have variable payload length.

Minislots are the dynamic segment time increments. In a minislot, a dynamic frame can start transmission, but it usually spans several minislots. If no frame transmits, the slot counter (slot ID) is incremented to allow for the next frame.

The total dynamic segment length is determined by multiplying this property by the FlexRay:Number of Minislots property. The total dynamic segment length must be shorter than the Macro Per Cycle property minus the total static segment length.

The range for this property is 2–63 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Minislot Action Point Offset

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayMinislotActPt`

Description

This property specifies the number of macroticks (MT) the minislot action point is offset from the beginning of a minislot.

This property corresponds to the global cluster parameter `gdMinislotActionPointOffset` in the *FlexRay Protocol Specification*.

The action point is that point within a given slot where the actual transmission of a frame starts. This is slightly later than the start of the slot to allow for a clock drift between the network nodes.

The range for this property is 1–31 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Network Management Vector Length

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayNMVecLen`

Description

This property specifies the length of the Network Management vector (NMVector) in a cluster.

Only frames transmitted in the static segment of the communication cycle use the NMVector. The NMVector length specifies the number of bytes in the payload segment of the FlexRay frame transmitted in the status segment that can be used as the NMVector.

This property corresponds to the global cluster parameter `gNetworkManagementVectorLength` in the *FlexRay Protocol Specification*.

The range for this property is 0–12 bytes.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:NIT

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayNIT`

Description

This property is the Network Idle Time (NIT) duration, expressed in macroticks (MT).

This property corresponds to the global cluster parameter `gdNIT` in the *FlexRay Protocol Specification*.

The NIT is a period at the end of a FlexRay communication cycle where no frames are transmitted. The network nodes use it to re-sync their clocks to the common network time.

Configure the NIT to be the Macro Per Cycle property minus the total static and dynamic segment lengths minus the optional symbol window duration.

The range for this property is 2–805 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:NIT Start

Data Type	Direction	Required?	Default
u32	Read Only	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayNITStart`

Description

This property specifies the start of the Network Idle Time (NIT), expressed as the number of macroticks (MT) from the start of the cycle.

The NIT is a period at the end of a FlexRay communication cycle where no frames are transmitted. The network nodes use it to re-sync their clocks to the common network time.

The range for this property is 8–15998 MT.

This property is calculated from other cluster properties. It is the total size of the static and dynamic segments plus the symbol window length, which is optional in a FlexRay communication cycle.

FlexRay:Number of Minislots

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayNumMinislt`

Description

This property specifies the number of minislots in the dynamic segment.

This property corresponds to the global cluster parameter `gNumberOfMinislots` in the *FlexRay Protocol Specification*.

In the FlexRay cycle dynamic segment, frames can have variable payload lengths.

Minislots are the dynamic segment time increments. In a minislot, a dynamic frame can start transmission, but it usually spans several minislots. If no frame transmits, the slot counter (slot ID) is incremented to allow for the next frame.

The total dynamic segment length is determined by multiplying this property by the Minislot property. The total dynamic segment length must be shorter than the Macro Per Cycle property minus the total static segment length.

The range for this property is 0–7986.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Number of Static Slots

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayNumStatSlt`

Description

This property specifies the number of static slots in the static segment.

This property corresponds to the global cluster parameter `gNumberOfStaticSlots` in the *FlexRay Protocol Specification*.

Each static slot is used to transmit one (static) frame on the bus.

The total static segment length is determined by multiplying this property by the Static Slot property. The total static segment length must be shorter than the Macro Per Cycle property.

The range for this property is 2–1023.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Offset Correction Start

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayOffCorSt`

Description

This property specifies the start of the offset correction phase within the Network Idle Time (NIT), expressed as the number of macroticks (MT) from the start of the cycle.

This property corresponds to the global cluster parameter `gOffsetCorrectionStart` in the *FlexRay Protocol Specification*.

The NIT is a period at the end of a FlexRay communication cycle where no frames are transmitted. The network nodes use it to re-sync their clocks to the common network time.

The Offset Correction Start is usually configured to be NIT Start + 1, but can deviate from that value. The range for this property is 9–15999 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Payload Length Dynamic Maximum

Data Type	Direction	Required?	Default
u32	Read/Write	N/A	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayPayldLenDynMax`

Description

This property specifies the maximum of the payload lengths of all dynamic frames.

In the FlexRay cycle dynamic segment, frames can have variable payload length.

The range for this property is 0–254 bytes (even numbers only).

The value returned for this property is the maximum of the payload lengths of all frames defined for the dynamic segment in the database.

Use this property to calculate the XNET Cluster FlexRay:Latest Usable Dynamic Slot and FlexRay:Latest Guaranteed Dynamic Slot properties.

You may temporarily set this to a larger value (if it is not yet the maximum), and then this value is returned for this property. But this setting is lost once the database is closed, and after a reopen, the maximum of the frames is returned again. The changed value is returned from the FlexRay:Payload Length Dynamic Maximum property until the database is closed.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Payload Length Maximum

Data Type	Direction	Required?	Default
u32	Read Only	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayPayldLenMax`

Description

This property returns the payload length of any frame (static or dynamic) in this cluster with the longest payload. The payload specifies that the frame transfers the data.

The range for this property is 0–254 bytes (even numbers only).

FlexRay:Payload Length Static

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayPayldLenSt`

Description

This property specifies the payload length of a static frame. All static frames in a cluster have the same payload length.

This property corresponds to the global cluster parameter `gPayloadLengthStatic` in the *FlexRay Protocol Specification*.

The range for this property is 0–254 bytes (even numbers only).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Static Slot

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayStatSlot`

Description

This property specifies the duration of a slot in the static segment in macroticks (MT).

This property corresponds to the global cluster parameter `gdStaticSlot` in the *FlexRay Protocol Specification*.

Each static slot is used to transmit one (static) frame on the bus.

The static slot duration takes into account the XNET Cluster FlexRay:Payload Length Static and FlexRay:Action Point Offset properties, as well as maximum propagation delay.

In the FlexRay cycle static segment, all frames must have the same payload length; therefore, the duration of a static frame is the same.

The total static segment length is determined by multiplying this property by the FlexRay:Number of Static Slots property. The total static segment length must be shorter than the FlexRay:Macro Per Cycle property.

The range for this property is 4–661 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Symbol Window

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRaySymWin`

Description

This property specifies the symbol window duration, expressed in macroticks (MT).

This property corresponds to the global cluster parameter `gdSymbolWindow` in the *FlexRay Protocol Specification*.

The symbol window is a slot after the static and dynamic segment, and is used to transmit Collision Avoidance symbols (CAS) and/or Media Access Test symbols (MTS). The symbol window is optional for a given cluster (the Symbol Window property can be zero). A symbol transmission starts at the action point offset within the symbol window.

The range for this property is 0–142 MT.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Symbol Window Start

Data Type	Direction	Required?	Default
u32	Read Only	N/A	Calculated from Other Cluster Properties

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRaySymWinStart`

Description

This property specifies the macrotick offset at which the symbol window begins from the start of the cycle. During the symbol window, a channel sends a single Media Test Access Symbol (MTS).

The range for this property is 8–15998 MT.

This property is calculated from other cluster properties. It is based on the total static and dynamic segment size. It is set to zero if the Symbol Window property is 0 (no symbol window exists).

FlexRay:Sync Node Max

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRaySyncNodeMax`

Description

This property specifies the maximum number of nodes that may send frames with the sync frame indicator bit set to one.

This property corresponds to the global cluster parameter `gSyncNodeMax` in the *FlexRay Protocol Specification*.

Sync frames define the zero points for the clock drift measurement. Startup frames are special sync frames transmitted first after a network startup. There must be at least two startup nodes in a network.

The range for this property is 2–15.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:TSS Transmitter

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayTSSTx`

Description

This property specifies the number of bits in the Transmission Start Sequence (TSS). A frame transmission may be truncated at the beginning. The amount of truncation depends on the nodes involved and the channel topology layout. For example, the purpose of the TSS is to "open the gates" of an active star (that is, to cause the star to properly set up input and output connections). During this setup, an active star truncates a number of bits at the beginning of a communication element. The TSS prevents the frame or symbol content from being truncated. You must set this property to be greater than the expected worst case truncation of a frame.

This property corresponds to the global cluster parameter `gdTSSTransmitter` in the *FlexRay Protocol Specification*.

The range for this property is 3–15 bit.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Use Wakeup

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayUseWakeup`

Description

This property indicates whether the FlexRay cluster supports wakeup. This value is set to True if the WAKE-UP tree is present in the FIBEX file. This value is set to False if the WAKE-UP tree is not present in the FIBEX file.

When this property is True, the FlexRay cluster uses wakeup functionality; otherwise, the FlexRay cluster does not use wakeup functionality.

When creating a new database, the default value of this property is false. However, if you set any wakeup parameter (for example, `nxPropClst_FlexRayWakeSymRxIdl`), this property automatically is set to True, and the WAKE-UP tree is saved in the FIBEX file when saved.

FlexRay:Wakeup Symbol Rx Idle

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayWakeSymRxIdl`

Description

This property specifies the number of bits the node uses to test the idle portion duration of a received wakeup symbol. Collisions, clock differences, and other effects can deform the transmitted wakeup pattern.

This property corresponds to the global cluster parameter `gdWakeupSymbolRxIdle` in the *FlexRay Protocol Specification*.

The range for this property is 14–59 gdBit (bit duration).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Wakeup Symbol Rx Low

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayWakeSymRxLow`

Description

This property specifies the number of bits the node uses to test the low portion duration of a received wakeup symbol. This lower limit of zero bits must be received for the receiver to detect the low portion. Active starts, clock differences, and other effects can deform the transmitted wakeup pattern.

This property corresponds to the global cluster parameter `gdWakeupSymbolRxLow` in the *FlexRay Protocol Specification*.

The range for this property is 10–55 gdBit (bit duration).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Wakeup Symbol Rx Window

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayWakeSymRxWin`

Description

This property specifies the size of the window used to detect wakeups. Detection of a wakeup requires a low and idle period from one WUS (wakeup symbol) and a low period from another WUS, to be detected entirely within a window of this size. Clock differences and other effects can deform the transmitted wakeup pattern.

This property corresponds to the global cluster parameter `gdWakeupSymbolRxWindow` in the *FlexRay Protocol Specification*.

The range for this property is 76–301 gdBit (bit duration).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Wakeup Symbol Tx Idle

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayWakeSymTxIdl`

Description

This property specifies the number of bits the node uses to transmit the wakeup symbol idle portion.

This property corresponds to the global cluster parameter `gdWakeupSymbolTxIdle` in the *FlexRay Protocol Specification*.

The range for this property is 45–180 gdBit (bit duration).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Wakeup Symbol Tx Low

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	Read from Database

Property Class

XNET Cluster

Property ID

`nxPropClst_FlexRayWakeSymTxLow`

Description

This property specifies the number of bits the node uses to transmit the wakeup symbol low phase.

This property corresponds to the global cluster parameter `gdWakeupSymbolTxLow` in the *FlexRay Protocol Specification*.

The range for this property is 15–60 gdBit (bit duration).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this cluster, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Frames

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET Cluster

Property ID

`nxPropClst_FrmRefs`

Description

Frames in this cluster.

Returns an array of refnums to all frames defined in this cluster. A frame is assigned to a cluster when the frame object is created. You cannot change this assignment afterwards.

To add a frame to a cluster, use `nxdbCreateObject`. To remove a frame from a cluster, use `nxdbDeleteObject`.

Name (Short)

Data Type	Direction	Required?	Default
char *	Read/Write	Yes	Defined in Create Object

Property Class

XNET Cluster

Property ID

`nxPropClst_Name`

Description

String identifying the cluster object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

If you use a FIBEX file, the short name comes from the file. If you use a CANdb (`.dbc`), LDF (`.ldf`), or NI-CAN (`.ncd`) file, no cluster name is stored in the file, so NI-XNET uses the name *Cluster*. If you create the cluster yourself, it comes from the `Name` input of `nxdbCreateObject`.

A cluster name must be unique for all clusters in a database.

This short name does not include qualifiers to ensure that it is unique, such as the database name. It is for display purposes.

You can write this property to change the cluster's short name.

PDUs

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET Cluster

Property ID

`nxPropClst_PDURefs`

Description

PDUs in this cluster.

Returns an array of database references (`nxDatabaseRef_t`) of all PDUs defined in this cluster. A PDU is assigned to a cluster when the PDU object is created. You cannot change this assignment afterwards.

To add a PDU to a cluster, use `nxdbCreateObject`. To remove a PDU from a cluster, use `nxdbDeleteObject`.

PDUs Required?

Data Type	Direction	Required?	Default
Boolean	Read Only	N/A	N/A

Property Class

XNET Cluster

Property ID

`nxPropClst_PDUsReqd`

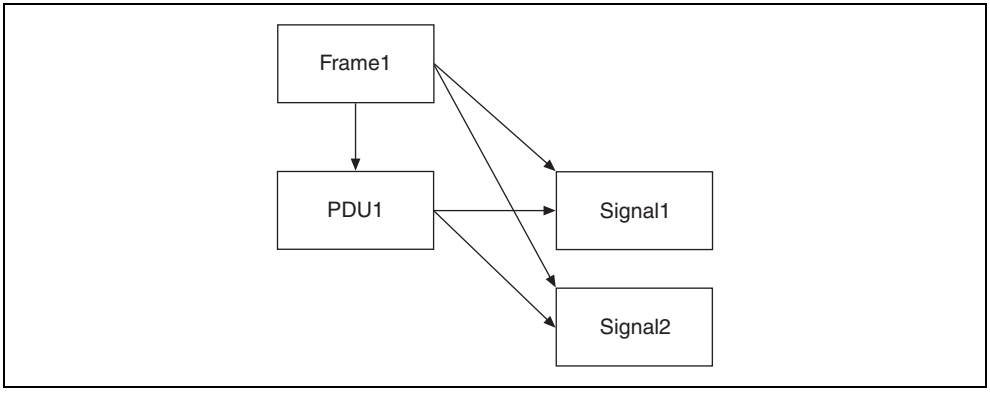
Description

Determines whether using PDUs in the database API is required for this cluster.

If this property returns false, it is safe to use signals as child objects of a frame without PDUs. This behavior is compatible with NI-XNET 1.1 or earlier. Clusters from `.dbc`, `.ncd`, or FIBEX 2 files always return false for this property, so using PDUs from those files is not required.

If this property returns true, the cluster contains PDU configuration, which requires reading the PDUs as frame child objects and then signals as PDU child objects, as shown in the following figure.

Internally, the database always uses PDUs, but shows the same signal objects also as children of a frame.



The following conditions must be fulfilled for all frames in the cluster to return false from the PDUs Required? property:

- Only one PDU is mapped to the frame.
- This PDU is not mapped to other frames.
- The PDU Start Bit in the frame is 0.
- The PDU Update Bit is not used.

If the conditions are not fulfilled for a given frame, signals from the frame are still returned, but reading the property returns a warning.

The NI-XNET session supports frames requiring PDUs only for FlexRay. For frames requiring PDUs on a CAN or LIN cluster, the XNET Frame Configuration Status property and `nxCreateSession` return an error.

Protocol

Data Type	Direction	Required?	Default
u32	Read/Write	No	CAN

Property Class

XNET Cluster

Property ID

`nxPropClst_Protocol`

Description

Determines the cluster protocol.

The values (enumeration) for this property are:

0 CAN

1 FlexRay

2 LIN

Schedules

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET Cluster

Property ID

`nxPropClst_LINSchedules`

Description

An array of LIN schedules defined in this cluster. You assign a LIN schedule to a cluster when you create the LIN schedule object. You cannot change this assignment afterwards. The schedules in this array are sorted alphabetically by schedule name.

Signals

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET Cluster

Property ID

`nxPropClst_SigRefs`

Description

This property returns refnums to all XNET Signals defined in this cluster.

A signal is assigned to a cluster when the signal object is created. You cannot change this assignment afterwards.

To add a signal to a cluster, use `nxdbCreateObject`. To remove a signal from a cluster, use `nxdbDeleteObject`.

Tick

Data Type	Direction	Required?	Default
f64	Read Only	N/A	N/A

Property Class

XNET Cluster

Property ID

`nxPropClst_LINTick`

Description

Relative time between LIN ticks (relative f64 in seconds). The LIN Schedule Entry Delay property must be a multiple of this tick.

This tick is referred to as the “timebase” in the LIN specification.

The XNET ECU LIN Master property defines the Tick property in this cluster. You cannot use the Tick property when there is no LIN Master property defined in this cluster.

Application Protocol

Data Type	Direction	Required?	Default
u32	Read/Write	No	Read from Database

Property Class

XNET Cluster

Short Name

`nxPropClst_ApplicationProtocol`

Description

This property specifies the application protocol. It is a ring of two values:

Enumeration	Value	Meaning
None	0	The default application protocol.
J1939	1	Indicates J1939 clusters. The value enables the following features: • Sending/receiving long frames as the SAE J1939 specification specifies, using the J1939 transport protocol. • Using a special notation for J1939 identifiers. • Using J1939 address claiming.

XNET Database Properties

This section includes the XNET Database properties.

Clusters

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET Database

Property ID

`nxPropDatabase_ClstRefs`

Description

Returns an array of refnums to XNET Clusters in this database.

A cluster is assigned to a database when the cluster object is created. You cannot change this assignment afterwards.

FIBEX files can contain any number of clusters, and each cluster uses a unique name.

For CANdb (`.dbc`), LDF (`.ldf`), or NI-CAN (`.ncd`) files, the file contains only one cluster, and no cluster name is stored in the file. For these database formats, NI-XNET uses the name *Cluster* for the single cluster.

ShowInvalidFromOpen?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Database

Property ID

`nxPropDatabase_ShowInvalidFromOpen`

Description

Shows frames and signals that are invalid at database open time.

After opening a database, this property always is set to false, meaning that invalid clusters, frames, and signals are not returned in properties that return XNET I/O Names for the database (for example, XNET Cluster Frames and XNET Frame Signals). Invalid clusters, frames, and signals are incorrectly defined and therefore cannot be used in the bus communication. The false setting is recommended when you use the database to create XNET sessions.

In case the database was opened to correct invalid configuration (for example, in a database editor), you must set the property to true prior to reading properties that return XNET I/O Names for the database (for example, XNET Cluster Frames and XNET Frame Signals).

For invalid objects, the XNET Cluster Configuration Status, XNET Frame Configuration Status, and XNET Signal Configuration Status properties return an error code that explains the problem. For valid objects, Configuration Status returns success (no error).

Clusters, frames, and signals that became invalid after the database is opened are still returned from the XNET Database Clusters, XNET Cluster Frames, and XNET Frame Signals properties, even if ShowInvalidFromOpen? is false and Configuration Status returns an error code. For example, if you open the frame with valid properties, then you set the Start Bit beyond the payload length, the Configuration Status returns an error, but the frame is returned from XNET Cluster Frames.

XNET Device Properties

The XNET Device properties provide information about a specific NI-XNET hardware device. Within NI-XNET, the term *device* refers to your National Instruments CAN/FlexRay/LIN hardware product, such as a PXI or PCI card.

You obtain the handle to a specific device using the XNET System Properties.

Form Factor

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Device

Property ID

`nxPropDev_FormFac`

Description

Returns the XNET board physical form factor.

Enumeration	Value	Define
PXI	0	`nxDevForm_PXI`
PCI	1	`nxDevForm_PCI`
C Series	2	`nxDevForm_cSeries`

Interfaces

Data Type	Direction	Required?	Default
u32[]	Read Only	No	N/A

Property Class

XNET Device

Property ID

`nxPropDev_IntfRefs`

Description

Returns an array of handles to all interfaces associated with this physical hardware device.

Number of Ports

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Device

Property ID

`nxPropDev_NumPorts`

Description

Returns the number of physical port connectors on the XNET board.

Remarks

For example, returns 2 for an NI PCI-8517 two-port FlexRay device.

Product Name

Data Type	Direction	Required?	Default
cstr	Read Only	No	N/A

Property Class

XNET Device

Property ID

`nxPropDev_Name`

Description

Returns the XNET device product name.

Remarks

For example, returns NI PCI-8517 (2 ports) for an NI PCI-8517 device.

Product Number

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Device

Property ID

`nxPropDev_ProductNum`

Description

Returns the numeric portion of the XNET device product name.

Remarks

For example, returns 8517 for an NI PCI-8517 two-port FlexRay device.

Serial Number

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Device

Property ID

`nxPropDev_SerNum`

Description

Returns the serial number associated with the XNET device.

Remarks

The serial number is written in hex on a label on the physical XNET board. Convert the return value from this property to hex to match the label.

Slot Number

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Device

Property ID

`nxPropDev_SlotNum`

Description

Physical slot where the device (module) is located.

For PXI and C Series, this is the slot number within the chassis.

XNET ECU Properties

This section includes the XNET ECU properties.

Cluster

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_ClstRef`

Description

Refnum to the parent cluster to which the ECU is connected.

The parent cluster is determined when the ECU object is created. You cannot change it afterwards.

Comment

Data Type	Direction	Required?	Default
char *	Read/Write	No	Empty String

Property Class

XNET ECU

Property ID

`nxPropECU_Comment`

Description

Comment describing the ECU object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
i32	Read Only	N/A	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_ConfigStatus`

Description

The ECU object configuration status.

Configuration Status returns an NI-XNET error code. You can pass the value to the `nxStatusToString` error code input to convert the value to a text description of the configuration problem.

By default, incorrectly configured ECUs in the database are not returned from the XNET Cluster ECUs property because they cannot be used in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When the configuration status of an ECU became invalid after the database is opened, the ECU still is returned from the ECUs property even if ShowInvalidFromOpen? is false.

FlexRay:Coldstart?

Data Type	Direction	Required?	Default
Boolean	Read Only	N/A	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_FlexRayIsColdstart`

Description

Indicates that the ECU is sending a startup frame.

This property is valid only for ECUs connected to a FlexRay bus. It returns true when one of the frames this ECU transmits (refer to the XNET ECU Frames Transmitted property) has the XNET Frame FlexRay:Startup? property set to true. You can determine the frame transmitting the startup using the XNET ECU FlexRay:Startup Frame property. An ECU can send only one startup frame on the FlexRay bus.

FlexRay:Connected Channels

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET ECU

Property ID

`nxPropECU_FlexRayConnectedChs`

Description

This property specifies the channel(s) that the FlexRay ECU (node) is physically connected to. The default value of this property is connected to all channels available on the cluster.

This property corresponds to the `pChannels` node parameter in the *FlexRay Protocol Specification*.

The values supported for this property (enumeration) are A = 1, B = 2, and A and B = 3.

FlexRay:Startup Frame

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_FlexRayStartupFrameRef`

Description

Returns the refnum to the startup frame the ECU sends.

This property is valid only for ECUs connected to a FlexRay bus. If the ECU transmits a frame (refer to the XNET ECU Frames Transmitted property) with the XNET Frame FlexRay:Startup? property set to true, this property returns this frame. Otherwise, it is empty.

FlexRay:Wakeup Channels

Data Type	Direction	Required?	Default
u32	Read/Write	No	None

Property Class

XNET ECU

Property ID

`nxPropECU_FlexRayWakeupChs`

Description

This property specifies the channel(s) on which the FlexRay ECU (node) is allowed to generate the wakeup pattern. The default value of this property is not to be a wakeup node.

When importing from a FIBEX file, this parameter corresponds to a WAKE-UP-CHANNEL being set to True for each connected channel.

The values supported for this property (enumeration) are A = 1, B = 2, A and B = 3, and None = 4.

FlexRay:Wakeup Pattern

Data Type	Direction	Required?	Default
u32	Read/Write	No	2

Property Class

XNET ECU

Property ID

`nxPropECU_FlexRayWakeupPtrn`

Description

This property specifies the number of repetitions of the wakeup symbol that are combined to form a wakeup pattern when the FlexRay ECU (node) enters the POC:WAKEUP_SEND state. The POC:WAKEUP_SEND state is one of the FlexRay controller state transitions during the wakeup process. In this state, the controller sends the wakeup pattern on the specified Wakeup Channel and checks for collisions on the bus.

This property is used when FlexRay:Wakeup Channels is set to a value other than None and FlexRay:Use Wakeup is True.

This property corresponds to the `pWakeupPattern` node parameter in the *FlexRay Protocol Specification*.

The supported values for this property are 2–63.

Frames Received

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read/Write	No	Empty Array

Property Class

XNET ECU

Property ID

`nxPropECU_RxFrmRefs`

Description

Returns an array of refnums to frames the ECU receives.

This property defines all frames the ECU receives. All frames an ECU receives in a given cluster must be defined in the same cluster.

Frames Transmitted

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read/Write	No	Empty Array

Property Class

XNET ECU

Property ID

`nxPropECU_FrmsTx`

Description

Returns an array of refnums to frames the ECU transmits.

This property defines all frames the ECU transmits. All frames an ECU transmits in a given cluster must be defined in the same cluster.

LIN Master

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET ECU

Property ID

`nxPropECU_LINMaster`

Description

Determines whether the ECU is a LIN master (true) or LIN slave (false).

LIN Version

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_LINProtocolVer`

Description

Version of the LIN standard this ECU uses. The values (enumeration) for this property are:

- `nxLINProtocolVer_1_2`
- `nxLINProtocolVer_1_3`
- `nxLINProtocolVer_2_0`
- `nxLINProtocolVer_2_1`

LIN:Initial NAD

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_LINInitialNAD`

Description

Initial NAD of a LIN slave node. NAD is the address of a slave node and is used in diagnostic services. Initial NAD is replaced by configured NAD with node configuration services.



Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

LIN:Configured NAD

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_LINConfigNAD`

Description

Configured NAD of a LIN slave node. NAD is the address of a slave node and is used in diagnostic services. Initial NAD is replaced by configured NAD with node configuration services.

Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

LIN:Supplier ID

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_LINSupplierID`

Description

Supplier ID is a 16-bit value identifying the supplier of the LIN node (ECU).

Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

LIN:Function ID

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_LINFunctionID`

Description

Function ID is a 16-bit value identifying the function of the LIN node (ECU).

Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

LIN:P2min

Data Type	Direction	Required?	Default
Double	Read Only	N/A	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_LINP2min`

Description

The minimum time in seconds between reception of the last frame of the diagnostic request and the response sent by the node.

Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

LIN:STmin

Data Type	Direction	Required?	Default
Double	Read Only	N/A	N/A

Property Class

XNET ECU

Property ID

`nxPropECU_LINSTmin`

Description

The minimum time in seconds the node requires to prepare for the next frame of the diagnostic service.



Caution This property is not saved in the FIBEX database. You can import it only from an LDF file.

Name (Short)

Data Type	Direction	Required?	Default
char *	Read/Write	Yes	Defined in Create Object

Property Class

XNET ECU

Property ID

`nxPropECU_Name`

Description

String identifying the ECU object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

An ECU name must be unique for all ECUs in a cluster.

This short name does not include qualifiers to ensure that it is unique, such as the database and cluster name. It is for display purposes.

You can write this property to change the ECU's short name.

XNET Frame Properties

This section includes the XNET Frame properties.

CAN:Extended Identifier?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Frame

Property ID

`nxPropFrm_CANExtID`

Description

This property determines whether the XNET Frame Identifier property in a CAN cluster represents a standard 11-bit (false) or extended 29-bit (true) arbitration ID.

CAN:Timing Type

Data Type	Direction	Required?	Default
u32	Read/Write	No	Event Data (If Not in Database)

Property Class

XNET Frame

Property ID

`nxPropFrm_CANTimingType`

Description

Specifies the CAN frame timing.

Because this property specifies the behavior of the frame's transfer within the embedded system (for example, a vehicle), it describes the transfer between ECUs in the network. In the following description, *transmitting ECU* refers to the ECU that transmits the CAN data frame (and possibly receives the associated CAN remote frame). *Receiving ECU* refers to an ECU that receives the CAN data frame (and possibly transmits the associated CAN remote frame).

When you use the frame within an NI-XNET session, an output session acts as the transmitting ECU, and an input session acts as a receiving ECU. For a description of how these CAN timing types apply to the NI-XNET session mode, refer to *CAN Timing Type and Session Mode*.

The CAN timing types (decimal value in parentheses) are:

`nxFrmCANTiming_CyclicData (0)`

The transmitting ECU transmits the CAN data frame in a cyclic (periodic) manner. The XNET Frame CAN:Transmit Time property defines the time between cycles. The transmitting ECU ignores CAN remote frames received for this frame.

`nxFrmCANTiming_EventData (1)`

The transmitting ECU transmits the CAN data frame in an event-driven manner. The XNET Frame CAN:Transmit Time property defines the minimum interval. For NI-XNET, the event occurs when you call `nxWrite`. The transmitting ECU ignores CAN remote frames received for this frame.

`nxFrmCANTiming_CyclicRemote (2)`

The receiving ECU transmits the CAN remote frame in a cyclic (periodic) manner. The XNET Frame CAN:Transmit Time property defines the time between cycles. The transmitting ECU responds to each CAN remote frame by transmitting the associated CAN data frame.

`nxFrmCANTiming_EventRemote (3)`

The receiving ECU transmits the CAN remote frame in an event-driven manner. The XNET Frame CAN:Transmit Time property defines the minimum interval. For NI-XNET, the event occurs when you call `nxWriteFrame`. The transmitting ECU responds to each CAN remote frame by transmitting the associated CAN data frame.

If you are using a FIBEX database, this property is a required part of the XML schema for a frame, so the default (initial) value is obtained from the file.

If you are using a CANdb (`.dbc`) database, this property is an optional attribute in the file. If NI-XNET finds an attribute named `GenMsgSendType`, that attribute is the default value of this property. If the `GenMsgSendType` attribute begins with cyclic, this property's default value is Cyclic Data; otherwise, it is Event Data. If the CANdb file does not use the `GenMsgSendType` attribute, this property uses a default value of Event Data, which you can change in your application.

If you are using an `.ncd` database or an in-memory database (XNET Create Frame), this property uses a default value of Event Data. Within your application, change this property to the desired timing type.

CAN:Transmit Time

Data Type	Direction	Required?	Default
Double	Read/Write	No	0.1 (If Not in Database)

Property Class

XNET Frame

Property ID

`nxPropFrm_CANTxTime`

Description

Specifies the time between consecutive frames from the transmitting ECU.

The data type is 64-bit floating point (DBL). The units are in seconds.

Although the fractional part of the DBL data type can provide resolution of picoseconds, the NI-XNET CAN transmit supports an accuracy of 500 µs. Therefore, when used within an NI-XNET output session, this property is rounded to the nearest 500 µs increment (0.0005).

For an XNET Frame CAN:Timing Type of Cyclic Data or Cyclic Remote, this property specifies the time between consecutive data/remote frames. A time of 0.0 is invalid.

For an XNET Frame CAN:Timing Type of Event Data or Event Remote, this property specifies the minimum time between consecutive data/remote frames when the event occurs quickly. This is also known as the debounce time or minimum interval. The time is measured from the end of previous frame (acknowledgment) to the start of the next frame. A time of 0.0 specifies no minimum (back to back frames allowed).

If you are using a FIBEX database, this property is a required part of the XML schema for a frame, so the default (initial) value is obtained from the file.

If you are using a CANdb (`.dbc`) database, this property is an optional attribute in the file. If NI-XNET finds an attribute named `GenMsgCycleTime`, that attribute is interpreted as a number of milliseconds and used as the default value of this property. If the CANdb file does not use the `GenMsgCycleTime` attribute, this property uses a default value of 0.1 (100 ms), which you can change in your application.

If you are using a `.ncd` database or an in-memory database (XNET Create Frame), this property uses a default value of 0.1 (100 ms). Within your application, change this property to the desired time.

Cluster

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_ClusterRef`

Description

This property returns the refnum to the parent cluster in which the frame has been created. You cannot change the parent cluster after the frame object has been created.

Comment

Data Type	Direction	Required?	Default
char *	Read/Write	No	Empty String

Property Class

XNET Frame

Property ID

`nxPropFrm_Comment`

Description

Comment describing the frame object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
i32	Read Only	N/A	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_ConfigStatus`

Description

The frame object configuration status.

Configuration Status returns an NI-XNET error code. You can pass the value to the `nxStatusToString` error code input to convert the value to a text description of the configuration problem.

By default, incorrectly configured frames in the database are not returned from the XNET Cluster Frames property because they cannot be used in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When a frame configuration status became invalid after the database is opened, the frame still is returned from the XNET Cluster Frames property even if ShowInvalidFromOpen? is false.

Examples of invalid frame configuration:

- A required property of the frame or an object contained in this frame has not been defined. For example, Frame Payload Length.
- The number of bytes specified for this frame is incorrect. CAN frames must use 0 to 8 bytes. FlexRay frames must use 0 to 254 bytes (even numbers only).
- The CAN arbitration ID is invalid. The standard ID is greater than 0x7FF (11 bits) or the extended ID is greater than 0x1FFFFFFF (29 bits).
- The FlexRay frame is specified to use channels not defined in the cluster. For example, the XNET Cluster FlexRay:Channels property is set to Channel A only, but the XNET Frame FlexRay:Channel Assignment property is set to Channel A and B.
- The XNET Frame FlexRay:Channel Assignment property in this dynamic FlexRay frame is set to Channel A and B, but dynamic frames can be sent on only one channel (A or B).

Default Payload

Data Type	Direction	Required?	Default
u8 *	Read/Write	No	Array of All 0

Property Class

XNET Frame

Property ID

`nxPropFrm_DefaultPayload`

Description

The frame default payload, specified as an array of bytes (U8).

The number of bytes in the array must match the XNET Frame Payload Length property.

This property's initial value is an array of all 0. For the database formats NI-XNET supports, this property is not provided in the database file.

When you use this frame within an NI-XNET session, this property's use varies depending on the session mode. The following sections describe this property's behavior for each session mode.

Frame Output Single-Point and Frame Output Queued Modes

Use this property when a frame transmits prior to a call to `nxWrite`. This can occur when you set the XNET Session Auto Start? property to false and call `nxStart` prior to `nxWrite`. When Auto Start? is true (default), the first call to `nxWrite` also starts frame transmit, so this property is not used.

The following frame configurations potentially can transmit prior to a call to `nxWrite`:

- XNET Frame CAN:Timing Type of Cyclic Data.
- XNET Frame CAN:Timing Type of Cyclic Remote (for example, a remote frame received prior to a call to `nxWrite`).
- XNET Frame CAN:Timing Type of Event Remote (for example, a remote frame received prior to a call to `nxWrite`).
- XNET Frame CAN:Timing Type of Cyclic.
- LIN frame in a schedule entry of Type unconditional

The following frame configurations cannot transmit prior to a call to `nxWrite`, so this property is not used:

- XNET Frame CAN:Timing Type of Event Data.
- XNET Frame FlexRay:Timing Type of Event.
- LIN frame in a schedule entry of Type sporadic or event triggered

Frame Output Stream Mode

This property is not used. Transmit is limited to frames provided to `nxWrite`.

Signal Output Single-Point, Signal Output Waveform, and Signal Output XY Modes

Use this property when a frame transmits prior to a call to `nxWrite`. Refer to *Frame Output Single-Point and Frame Output Queued Modes* for a list of applicable frame configurations.

This property is used as the initial payload, then each XNET Signal Default Value is mapped into that payload, and the result is used for the frame transmit.

Frame Input Stream and Frame Input Queued Modes

This property is not used. These modes do not return data prior to receiving frames.

Frame Input Single-Point Mode

This property is used for frames `nxRead` returns prior to receiving the first frame.

Signal Input Single-Point, Signal Input Waveform, and Signal Input XY Modes

This property is not used. Each XNET Signal Default Value is used when `nxRead` is called prior to receiving the first frame.

FlexRay:Base Cycle

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_FlexRayBaseCycle`

Description

The first communication cycle in which a frame is sent.

In FlexRay, a communication cycle contains a number of slots in which a frame can be sent. Every node on the bus provides a 6-bit cycle counter that counts the cycles from 0 to 63 and then restarts at 0. The cycle number is common for all nodes on the bus.

NI-XNET has two mechanisms for changing the frame sending frequency:

- If the frame should be sent faster than the cycle period, use In-Cycle Repetition (refer to the XNET Frame FlexRay:In Cycle Repetitions:Identifiers property).
- If the frame should be sent slower than the cycle period, use this property and the XNET Frame FlexRay:Cycle Repetition property.

The second method is called cycle multiplexing. It allows sending multiple frames in the same slot, but on different cycle counters.

If a frame should be sent in every cycle, set this property to 0 and the XNET Frame FlexRay:Cycle Repetition property to 1. For cycle multiplexing, set the FlexRay:Cycle Repetition property to 2, 4, 8, 16, 32, or 64.

Example:

- FrameA and FrameB are both sent in slot 12.
- **FrameA**: The FlexRay:Base Cycle property is 0 and XNET Frame FlexRay:Cycle Repetition property is 2. This frame is sent when the cycle counter has the value 0, 2, 4, 6, ...
- **FrameB**: The FlexRay:Base Cycle property is 1 and XNET Frame FlexRay:Cycle Repetition property is 2. This frame is sent when the cycle counter has the value 1, 3, 5, 7, ...

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Channel Assignment

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_FlexRayChAssign`

Description

This property determines on which FlexRay channels the frame must be transmitted. A frame can be transmitted only on existing FlexRay channels, configured in the XNET Cluster FlexRay:Channels property.

Frames in the dynamic FlexRay segment cannot be sent on both channels; they must use either channel A or B. Frames in the dynamic segment use slot IDs greater than the `number of static slots` cluster parameter.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:Cycle Repetition

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_FlexRayCycleRep`

Description

The number of cycles after which a frame is sent again.

In FlexRay, a communication cycle contains a number of slots in which a frame can be sent. Every node on the bus provides a 6-bit cycle counter that counts the cycles from 0 to 63 and then restarts at 0. The cycle number is common for all nodes on the bus.

NI-XNET has two mechanisms for changing the frame sending frequency:

- If the frame should be sent faster than the cycle period, use In-Cycle Repetition (refer to the XNET Frame FlexRay:In Cycle Repetitions:Identifiers property).
- If the frame should be sent slower than the cycle period, use the XNET Frame FlexRay:Base Cycle property and this property.

The second method is called cycle multiplexing. It allows sending multiple frames in the same slot, but on different cycle counters.

If a frame should be sent in every cycle, set the XNET Frame FlexRay:Base Cycle property property to 0 and this property to 1. For cycle multiplexing, set this property to 2, 4, 8, 16, 32, or 64.

Examples:

- FrameA and FrameB are both sent in slot 12.
- **FrameA**: The XNET Frame FlexRay:Base Cycle property is set to 0 and FlexRay:Cycle Repetition property is set to 2. This frame is sent when the cycle counter has the value 0, 2, 4, 6, ...
- **FrameB**: The XNET Frame FlexRay:Base Cycle property is set to 1 and FlexRay:Cycle Repetition property is set to 2. This frame is sent when the cycle counter has the value 1, 3, 5, 7, ...

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

FlexRay:In Cycle Repetitions:Channel Assignments

Data Type	Direction	Required?	Default
u32 *	Read/Write	No	Empty Array

Property Class

XNET Frame

Property ID

`nxPropFrm_FlexRayInCycRepChAssigns`

Description

FlexRay channels for in-cycle frame repetition.

A FlexRay frame can be sent multiple times per cycle. The XNET Frame FlexRay:Channel Assignment property defines the first channel assignment in the cycle. This property defines subsequent channel assignments. The XNET Frame FlexRay:In Cycle Repetitions:Identifiers property defines the corresponding slot IDs. Both properties are arrays of maximum three values, determining the slot ID and channel assignments for the frame. Values at the same array position are corresponding; therefore, both arrays must have the same size.

You must set the XNET Frame FlexRay:Channel Assignment property before setting this property. The FlexRay:Channel Assignment is a required property that is undefined when a new frame is created. When FlexRay:Channel Assignment is undefined, setting FlexRay:In Cycle Repetitions:Channel Assignments returns an error.

FlexRay:In Cycle Repetitions:Enabled?

Data Type	Direction	Required?	Default
Boolean	Read Only	No	False

Property Class

XNET Frame

Property ID

`nxPropFrm_FlexRayInCycRepEnabled`

Description

FlexRay in-cycle frame repetition is enabled.

A FlexRay frame can be sent multiple times per cycle. The XNET Frame Identifier property defines the first slot ID in the cycle. The XNET Frame FlexRay:In Cycle Repetitions:Identifiers property can define the subsequent slot IDs, and the FlexRay:In Cycle Repetitions:Channel Assignments property defines the corresponding FlexRay channels. Both properties are arrays of maximum three values determining the slot ID and FlexRay channels for the frame. Values at the same array position are corresponding; therefore, both arrays must have the same size.

This property returns true when at least one in-cycle repetition has been defined, which means that both the FlexRay:In Cycle Repetitions:Identifiers and XNET Frame FlexRay:In Cycle Repetitions:Channel Assignments arrays are not empty.

This property returns false when at least one of the previously mentioned arrays is empty. In this case, in-cycle-repetition is not used.

FlexRay:In Cycle Repetitions:Identifiers

Data Type	Direction	Required?	Default
u32 *	Read/Write	No	Empty Array

Property Class

XNET Frame

Property ID

`nxPropFrm_FlexRayInCycRepIDs`

Description

FlexRay in-cycle repetition slot IDs.

A FlexRay frame can be sent multiple times per cycle. The XNET Frame Identifier property defines the first slot ID in the cycle. The FlexRay:In Cycle Repetitions:Identifiers property defines subsequent slot IDs. The XNET Frame FlexRay:In Cycle Repetitions:Channel Assignments property defines the corresponding FlexRay channel assignments. Both properties are arrays of maximum three values, determining the subsequent slot IDs and channel assignments for the frame. Values at the same array position are corresponding; therefore, both arrays must have the same size.

You must set the XNET Frame Identifier property before setting the FlexRay:In Cycle Repetitions:Identifiers property. Identifier is a required property that is undefined when a new frame is created. When Identifier is undefined, setting in-cycle repetition slot IDs returns an error.

FlexRay:Payload Preamble?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Frame

Property ID

`nxPropFrm_FlexRayPreamble`

Description

This property determines whether payload preamble is used in a FlexRay frame:

- For frames in the static segment, it indicates that the network management vector is transmitted at the beginning of the payload.
- For frames in the dynamic segment, it indicates that the message ID is transmitted at the beginning of the payload.

FlexRay:Startup?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Frame

Property ID

`nxPropFrm_FlexRayStartup`

Description

This property determines whether the frame is a FlexRay startup frame. FlexRay startup frames always are FlexRay sync frames also:

- When this property is set to true, the XNET Frame FlexRay:Sync? property automatically is set to true.
- When this property is set to false, the XNET Frame FlexRay:Sync? property is not changed.
- When the XNET Frame FlexRay:Sync? property is set to false, this property automatically is set to false.
- When the XNET Frame FlexRay:Sync? property is set to true, this property is not changed.

An ECU can send only one startup frame. The startup frame, if an ECU transmits it, is returned from the XNET ECU FlexRay:Startup Frame property.

FlexRay:Sync?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Frame

Property ID

`nxPropFrm_FlexRayStartup`

Description

This property determines whether the frame is a FlexRay sync frame. FlexRay startup frames always are FlexRay sync frames also:

- When this property is set to false, the XNET Frame FlexRay:Startup? property is automatically set to false.
- When this property is set to true, the XNET Frame FlexRay:Startup? property is not changed.
- When the XNET Frame FlexRay:Startup? property is set to true, this property is set to true.
- When the XNET Frame FlexRay:Startup? property is set to false, this property is not changed.

An ECU can send only one sync frame.

FlexRay:Timing Type

Data Type	Direction	Required?	Default
u32	Read/Write	No	Cyclic in Static Segment, Event in Dynamic Segment

Property Class

XNET Frame

Property ID

`nxPropFrm_FlexRayTimingType`

Description

Specifies the FlexRay frame timing (decimal value in parentheses):

`nxFrmFlexRayTiming_Cyclic (0)`

Payload data transmits on every occurrence of the frame's slot.

`nxFrmFlexRayTiming_Event (1)`

Payload data transmits in an event-driven manner. Within the ECU that transmits the frame, the event typically is associated with the availability of new data.

This property's behavior depends on the FlexRay segment where the frame is located: static or dynamic. If the frame's Identifier (slot) is less than or equal to the cluster's Number Of Static Slots, the frame is static.

Static

Cyclic means no null frame is transmitted. If new data is not provided for the cycle, the previous payload data transmits again.

Event means a null frame is transmitted when no event is pending for the cycle.

This property's default value for the static segment is Cyclic.

Dynamic

Cyclic means the frame transmits in its minislot on every cycle.

Event means the frame transmits in the minislot when the event is pending for the cycle.

This property's default value for the dynamic segment is Event.

For a description of how these FlexRay timing types apply to the NI-XNET session mode, refer to *FlexRay Timing Type and Session Mode*.

Identifier

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_ID`

Description

Determines the frame identifier.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value in LabVIEW using the property node.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

CAN

For CAN frames, this is the Arbitration ID.

When the XNET Frame CAN:Extended Identifier? property is set to false, this is the standard CAN identifier with a size of 11 bits, which results in allowed range of 0–2047. However, the CAN standard disallows identifiers in which the first 7 bits are all recessive, so the working range of identifiers is 0–2031.

When the XNET Frame CAN:Extended Identifier? property is set to true, this is the extended CAN identifier with a size of 29 bits, which results in allowed range of 0–536870911.

FlexRay

For FlexRay frames, this is the Slot ID in which the frame is sent. The valid value range for a FlexRay Slot ID is 1–2047.

You also can send a FlexRay frame in multiple slots per cycle. You can define subsequent slot IDs for the frame in the XNET Frame FlexRay:In Cycle Repetitions:Identifiers property. Use this concept to increase a frame's sending frequency. To decrease a frame's sending frequency and share the same slot for different frames depending on the cycle counter, refer to the XNET Frame FlexRay:Base Cycle and XNET Frame FlexRay:Cycle Repetition properties.

The slot ID determines whether a FlexRay frame is sent in a static or dynamic segment. If the slot ID is less than or equal to the XNET Cluster FlexRay:Number of Static Slots property, the frame is sent in the communication cycle static segment; otherwise, it is sent in the dynamic segment.

If the frame identifier is not in the allowed range, this is reported as an error in the XNET Cluster Configuration Status property.

LIN

For LIN frames, this is the frame's ID (unprotected). The valid range for a LIN frame ID is 0–63 (inclusive).

LIN:Checksum

Data Type	Direction	Required?	Default
u32	Read Only	N/A	`nxFrmLINChecksum_Enhanced`

Property Class

XNET Frame

Property ID

`nxPropFrm_LINChecksum`

Description

Determines whether the LIN frame transmitted checksum is classic or enhanced. The enhanced checksum considers the protected identifier when it is generated.

The values (enumeration) for this property are:

`nxFrmLINChecksum_Classic`	0
`nxFrmLINChecksum_Enhanced`	1

The checksum is determined from the LIN version of ECUs transmitting and receiving the frame. The lower version of both ECUs is significant. If the LIN version of both ECUs is 2.0 or higher, the checksum type is enhanced; otherwise, the checksum type is classic.

Diagnostic frames (with decimal identifier 60 or 61) always use classic checksum, even on LIN 2.*x*.

Mux:Data Multiplexer Signal

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_MuxDataMuxSigRef`

Description

Data multiplexer signal in the frame.

This property returns a refnum to the data multiplexer signal. If the data multiplexer is not defined in the frame, the property returns 0. Use the XNET Frame Mux:Is Data Multiplexed? property to determine whether the frame contains a multiplexer signal.

You can create a data multiplexer signal by creating a signal and then setting the XNET Signal Mux:Data Multiplexer? property to true.

A frame can contain only one data multiplexer signal.

Mux:Is Data Multiplexed?

Data Type	Direction	Required?	Default
Boolean	Read Only	No	False

Property Class

XNET Frame

Property ID

`nxPropFrm_MuxIsMuxed`

Description

Frame is data multiplexed.

This property returns true if the frame contains a multiplexer signal. Frames containing a multiplexer contain subframes that allow using bits of the frame payload for different information (signals) depending on the multiplexer value.

Mux:Static Signals

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_MuxStaticSigRefs`

Description

Static signals in the frame.

Returns an array of refnums to signals in the frame that do not depend on the multiplexer value. Static signals are contained in every frame transmitted, as opposed to dynamic signals, which are transmitted depending on the multiplexer value.

You can create static signals by specifying the frame as the parent object. You can create dynamic signals by specifying a subframe as the parent.

If the frame is not multiplexed, this property returns the same array as the XNET Frame Signals property.

Mux:Subframes

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_MuxSubframeRefs`

Description

Returns an array of references to subframes in the frame. A subframe defines a group of signals transmitted using the same multiplexer value. Only one subframe at a time is transmitted in the frame.

A subframe is defined by creating a subframe object as a child of a frame.

Name (Short)

Data Type	Direction	Required?	Default
char *	Read/Write	Yes	Defined in Create Object

Property Class

XNET Frame

Property ID

`nxPropFrm_Name`

Description

String identifying a frame object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A frame name must be unique for all frames in a cluster.

This short name does not include qualifiers to ensure that it is unique, such as the database and cluster name. It is for display purposes.

You can write this property to change the frame's short name.

Payload Length

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_PayloadLen`

Description

Number of bytes of data in the payload.

For CAN and LIN, this is 0–8.

For FlexRay, this is 0–254. As encoded on the FlexRay bus, all frames use an even payload (16-bit words), and the payload of all static slots must be the same. Nevertheless, this property specifies the number of payload bytes used within the frame, so its value can be odd. For example, if a FlexRay cluster uses static slots of 18 bytes, it is valid for this property to be 15, which specifies that the last 3 bytes are unused.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this frame, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

PDU References

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read/Write	No	Empty Array

Property Class

XNET Frame

Property ID

`nxPropFrm_PDURefs`

Description

This property maps existing PDUs to a frame. A mapped PDU is transmitted inside the frame payload when the frame is transmitted. You can map one or more PDUs to a frame and one PDU to multiple frames.

Mapping PDUs to a frame requires setting three frame properties. All three properties are arrays of values:

- **PDU References**—Set this property first to define the sequence of values for the other two properties.
- **PDU Start Bits**—Defines the start bit of the PDU inside the frame.
- **PDU Update Bits**—Defines the update bit for the PDU inside the frame. If the update bit is not used, set the value to –1. (Refer to *Update Bit* for more information.)

Values on the same array position are corresponding. For example, PDUs[0], StartBits[0], and UpdateBits[0] define the mapping for the first PDU in the frame.

Databases imported from FIBEX prior to version 3.0, from DBC, NCD, or LDF files have a strong one-to-one relationship between frames and PDUs. Every frame has exactly one PDU mapped, and every PDU is mapped to exactly one frame.

To unmap PDUs from a frame, set this property to an empty array. A frame without mapped PDUs contains no signals.

NI-XNET supports advanced PDU configuration (multiple PDUs in one frame or one PDU used in multiple frames) only for FlexRay. Refer to the XNET Cluster PDUs Required? property.

For CAN and LIN, NI-XNET supports only a one-to-one relationship between frames and PDUs. For those interfaces, advanced PDU configuration returns an error from the XNET Frame Configuration Status property and `nxCreateSession`. If you do not use advanced PDU configuration, you can avoid using PDUs in the database API and create signals and subframes directly on a frame.

PDU Start Bits

Data Type	Direction	Required?	Default
u32 *	Read/Write	No	Empty Array

Property Class

XNET Frame

Property ID

`nxPropFrm_PDUStartBits`

Description

This property defines the start bits of PDUs mapped to a frame. A mapped PDU is transmitted inside the frame payload when the frame is transmitted. You can map one or more PDUs to a frame and one PDU to multiple frames.

Mapping PDUs to a frame requires setting of three frame properties. All three properties are arrays of values:

- **PDU References**—Set this property first to define the sequence of values for the other two properties.
- **PDU Start Bits**—This property defines the start bit of the PDU inside the frame.
- **PDU Update Bits**—Defines the update bit for the PDU inside the frame. If the update bit is not used, set the value to –1. (Refer to *Update Bit* for more information.)

Values on the same array position are corresponding. For example, PDUs[0], StartBits[0], and UpdateBits[0] define the mapping for the first PDU in the frame.

PDU Update Bits

Data Type	Direction	Required?	Default
u32 *	Read/Write	No	Empty Array

Property Class

XNET Frame

Property ID

`nxPropFrm_PDUUpdateBits`

Description

This property defines update bits of PDUs mapped to a frame. If the update bit is not used for the PDU, set the value to –1. (Refer to *Update Bit* for more information.)

Mapping PDUs to a frame requires setting three frame properties. All three properties are arrays of values:

- **PDU References**: Set this property first to define the sequence of values for the other two properties.
- **PDU Start Bits**: Defines the start bit of the PDU inside the frame.
- **PDU Update Bits**: This property defines the update bit for the PDU inside the frame. If the update bit is not used, set the value to –1.

Values on the same array position are corresponding. For example, PDUs[0], StartBits[0], and UpdateBits[0] define the mapping for the first PDU in the frame.

Signals

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET Frame

Property ID

`nxPropFrm_SigRefs`

Description

Refnums to all signals in the frame.

This property returns an array with references to all signals in the frame, including static and dynamic signals and the multiplexer signal.

This property is read only. You can add signals to a frame using `nxdbCreateObject` and remove them using `nxdbDeleteObject`.

Application Protocol

Data Type	Direction	Required?	Default
u32	Read/Write	No	Read from Database

Property Class

XNET Frame

Short Name

`nxPropFrm_ApplicationProtocol`

Description

This property specifies the frame's application protocol. It is a ring of two values:

Enumeration	Value	Meaning
None	0	The default application protocol.
J1939	1	Indicates J1939 frames. The value enables the following features: • Sending/receiving long frames as the SAE J1939 specification specifies, using the J1939 transport protocol. • Using a special notation for J1939 identifiers.

XNET Interface Properties

The XNET Interface properties provide information about a specific NI-XNET interface. The NI-XNET interface represents a single CAN, FlexRay, or LIN connector (port) on the device.

You obtain the handle to a specific interface using the XNET System Properties.

CAN.Termination Capability

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Interface

Property ID

`nxPropIntf_CANTermCap`

Description

Returns an enumeration indicating whether the XNET interface can terminate the CAN bus.

Enumeration	Value
No	0
Yes	1

Remarks

Signal reflections on the CAN bus can cause communication failure. To prevent reflections, termination can be present as external resistance or resistance the XNET board applies internally. This enumeration determines whether the XNET board can add termination to the bus.

To select the CAN transceiver termination, refer to XNET Session Interface:CAN:Termination.

CAN.Transceiver Capability

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Interface

Property ID

`nxPropIntf_CANTcvrCap`

Description

Returns an enumeration indicating the CAN bus physical transceiver support.

Enumeration	Value
High-Speed (HS)	0
Low-Speed (LS)	1
XS (HS, LS, SW, or External)	2

Remarks

The XS value in the enumeration indicates the board has the physical transceivers for High-Speed (HS), Low-Speed (LS), and Single Wire (SW), and can connect to an external transceiver. This value is switchable through the XNET Session Interface:CAN:Transceiver Type property.

Device

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Interface

Property ID

`nxPropIntf_DevRef`

Description

From the XNET Interface handle, this property returns the XNET device handle.

Remarks

The XNET device handle returned is the physical XNET board that contains the XNET interface. This property determines the physical XNET device through the XNET Device Serial Number property for a given XNET Interface handle.

Name

Data Type	Direction	Required?	Default
cstr	Read Only	No	N/A

Property Class

XNET Interface

Property ID

`nxPropIntf_Name`

Description

Returns the string name assigned to the XNET interface handle.

Remarks

This string is used for identification in MAX.

Number

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Interface

Property ID

`nxPropIntf_Num`

Description

Returns unique number associated with the XNET interface.

Remarks

The XNET driver assigns each port connector in the system a unique number XNET driver. This number, plus its protocol name, is the interface name string. For example:

XNET Interface String Name	Number
CAN1	1
FlexRay3	3

Port Number

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Interface

Property ID

`nxPropIntf_PortNum`

Description

Returns the physical port number printed near the connector on the XNET device.

Remarks

The port numbers on an XNET board are physically identified with numbering. Use this property, along with the XNET Device Serial Number property, to associate an XNET interface with a physical (XNET board and port) combination.



Note It is easier to find the physical location of an XNET interface with `nxBlink`.

Protocol

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Interface

Property ID

`nxPropIntf_Protocol`

Description

Returns the protocol supported by the interface as an enumeration.

Enumeration	Value
CAN	0
FlexRay	1
LIN	2

Remarks

The protocol enumeration will match the protocol portion of the XNET interface name string:

XNET Interface String Name	Protocol
CAN1	0
FlexRay3	1

XNET LIN Schedule Properties

Cluster

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET LIN Schedule

Property ID

`nxPropLINSched_ClstRef`

Description

This property returns the reference to the parent cluster in which the you created the schedule. You cannot change the parent cluster after creating the schedule object.

Comment

Data Type	Direction	Required?	Default
cstr	Read/Write	No	Empty String

Property Class

XNET LIN Schedule

Property ID

`nxPropLINSched_Comment`

Description

A comment describing the schedule object. A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
nxStatus_t	Read Only	N/A	N/A

Property Class

XNET LIN Schedule

Property ID

`nxPropLINSched_ConfigStatus`

Description

The LIN schedule object configuration status.

Configuration Status returns an NI-XNET error code. You can pass the value to the `Status` parameter of the `nxStatusToString` function to convert the value to a text description of the configuration problem.

By default, incorrect configured schedules in the database are not returned from the Cluster `Schedules` property because they cannot be used in the bus communication. You can change this behavior by setting the Database `ShowInvalidFromOpen?` property to true. When the configuration status of a schedule becomes invalid after opening the database, the schedule still is returned from the Cluster `Schedules` property even if `ShowInvalidFromOpen?` is false.

An example of invalid schedule configuration is when a required schedule property has not been defined. For example, a schedule entry within this schedule has an undefined delay time.

Entries

Data Type	Direction	Required?	Default
nxDatabaseRef_t[]	Read Only	N/A	N/A

Property Class

XNET LIN Schedule

Property ID

`nxPropLINSched_Entries`

Description

The array of entries for this LIN schedule.

The position of each entry in this array specifies the position in the schedule. The database file and/or the order that you create entries at runtime determine the position.

Name

Data Type	Direction	Required?	Default
cstr	Read/Write	Yes	Defined in nxdbCreateObject

Property Class

XNET LIN Schedule

Property ID

`nxPropLINSched_Name`

Description

String identifying the LIN schedule object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A schedule name must be unique for all schedules in a cluster.

Priority

Data Type	Direction	Required?	Default
u32	Read/Write	No	42

Property Class

XNET LIN Schedule

Property ID

`nxPropLINSched_Priority`

Description

Priority of this run-once LIN schedule when multiple run-once schedules are pending for execution.

The valid range for this property is 1–254. Lower values correspond to higher priority.

This property applies only when the Run Mode property is Once. Run-once schedule requests are queued for execution based on this property. When all run-once schedules have completed, the master returns to the previously running continuous schedule (or null).

Run-continuous schedule requests are not queued. Only the most recent run-continuous schedule is used, and it executes only if no run-once schedule is pending. Therefore, a run-continuous schedule has an effective priority of 255, but this property is not used.

Null schedule requests take effect immediately and supercede any running run-once or run-continuous schedule. The queue of pending run-once schedule requests is flushed (emptied without running them). Therefore, a null schedule has an effective priority of 0, but this property is not used.

This property is not read from the database, but is handled like a database property. After opening the database, the default value is returned, and you can change the property. But similar to database properties, you cannot change it after a session is created.

Run Mode

Data Type	Direction	Required?	Default
u32	Read/Write	No	See Description

Property Class

XNET LIN Schedule

Property ID

`nxPropLINSched_RunMode`

Description

This property is a ring (enumerated list) with the following values:

String	Value
Continuous	0
Once	1
Null	2

This property specifies how the master runs this schedule:

- **Continuous**: The master runs the schedule continuously. When the last entry executes, the schedule starts again with the first entry.
- **Once**: The master runs the schedule once (all entries), then returns to the previously running continuous schedule (or null). If requests are submitted for multiple run-once schedules, each run-once executes in succession based on its Priority, then the master returns to the continuous schedule (or null).
- **Null**: All communication stops immediately. A schedule with this run mode is called a null schedule.

This property is not read from the database, but is handled like a database property. After opening the database, the default value is returned, and you can change the property. But similar to database properties, you cannot change it after a session is created.

Usually, the default value for the run mode is Continuous. If the schedule is configured to be a collision resolving table for an event-triggered entry, the default is Once.

XNET LIN Schedule Entry Properties

Collision Resolving Schedule

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read/Write	No	Null

Property Class

XNET LIN Schedule Entry

Property ID

`nxPropLINSchedEntry_CollisionResSched`

Description

A LIN schedule that resolves a collision for this event-triggered entry.

This property applies only when the entry type is event triggered. When a collision occurs for the event-triggered entry in this schedule, the master must switch to the collision resolving schedule to transfer the unconditional frames successfully.

When the entry type is any value other than event triggered, this property returns Null (invalid).

Delay

Data Type	Direction	Required?	Default
f64	Read/Write	Yes	N/A

Property Class

XNET LIN Schedule Entry

Property ID

`nxPropLINSchedEntry_Delay`

Description

The time from the start of this entry (slot) to the start of the next entry. (The property uses a double value in seconds, with the fractional part used for milliseconds or microseconds.)

Event Identifier

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET LIN Schedule Entry

Property ID

`nxPropLINSchedEntry_EventID`

Description

The event-triggered entry identifier. This identifier is unprotected (NI-XNET handles the protection).

This property applies only when the entry type is event triggered. This identifier is for the event triggered entry itself, and the first payload byte is for the protected identifier of the contained unconditional frame.

Frames

Data Type	Direction	Required?	Default
nxDatabaseRef_t[]	Read/Write	No	Empty Array

Property Class

XNET LIN Schedule Entry

Property ID

`nxPropLINSchedEntry_Frames`

Description

The array of frames for this LIN schedule entry.

If the entry Type is unconditional, this array contains one element, which is the single unconditional frame for this entry.

If the entry Type is sporadic, this array contains one or more unconditional frames for this entry. When multiple frames are pending for this entry, the order in the array determines the priority to transmit.

If the entry Type is event triggered, this array contains one or more unconditional frames for this entry. When multiple frames for this entry are pending to be sent by distinct slaves, this property uses the Collision Resolving Schedule to process the frames.

Name

Data Type	Direction	Required?	Default
cstr	Read/Write	Yes	Defined in nxdbCreateObject

Property Class

XNET LIN Schedule Entry

Property ID

nxPropLINSchedEntry_Name

Description

String identifying the LIN schedule entry object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A schedule entry name must be unique for all entries in the same schedule.

Name Unique to Cluster

Data Type	Direction	Required?	Default
cstr	Read Only	N/A	N/A

Property Class

XNET LIN Schedule Entry

Property ID

`nxPropLINSchedEntry_NameUniqueToCluster`

Description

This property returns a LIN schedule entry name unique to the cluster that contains the object. If the single name is not unique within the cluster, the name is <*schedule-name*>.<*schedule-entry-name*>.

You can pass the name to the `nxdbFindObject` function to retrieve the reference to the object, while the single name is not guaranteed success in `nxdbFindObject` because it may be not unique in the cluster.

Node Configuration:Free Format:Data Bytes

Data Type	Direction	Required?	Default
u8*	Read/Write	Yes	N/A

Property Class

XNET LIN Schedule Entry

Property ID

`nxPropLINSchedEntry_NC_FF_DataBytes`

Description

An array of 8 bytes containing raw data for LIN node configuration.

Node configuration defines a set of services used to configure slave nodes in the cluster. Every service has a specific set of parameters coded in this byte array. In the LDF, file those parameters are stored, for example, in the node (ECU) or the frame object. NI-XNET LDF reader composes those parameters to the byte values like they are sent on the bus. The LIN specification document describes the node configuration services and the mapping of the parameters to the free format bytes.

The node configuration service is executed only if the Schedule Entry Type property is set to Node Configuration.

Caution This property is not saved to the FIBEX file. If you write this property, save the database, and reopen it, the node configuration services are not contained in the database. Writing this property is useful only in the NI-XNET session immediately following.

Schedule

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET LIN Schedule Entry

Property ID

`nxPropLINSchedEntry_Sched`

Description

The LIN schedule that uses this entry.

This LIN schedule is considered this entry's parent. You define the parent schedule when you create the entry object. You cannot change it afterwards.

Type

Data Type	Direction	Required?	Default
u32	Read/Write	No	Unconditional

Property Class

XNET LIN Schedule Entry

Property ID

`nxPropLINSchedEntry_Type`

Description

All frames that contain a payload are unconditional. The LIN schedule entry type determines the mechanism for transferring frames in this entry (slot):

0 **Unconditional**: A single frame transfers in this slot.

1 **Sporadic**: The master transmits in this slot. The master can select from multiple frames to transmit. Only updated frames are transmitted. When more than one frame is updated, the master decides by priority which frame to send. The other updated frame remains pending and can be sent when this schedule entry is processed the following time. The order of unconditional frames in the LIN Schedule Entry Frames property (the first frame has the highest priority) determines the frame priority.

2 **Event triggered**: Multiple slaves can transmit an unconditional frame in this slot. The slave transmits the frame only if at least one frame signal has been updated. When a collision occurs (multiple slaves try to transmit in the same slot), this is detected and resolved using a different schedule specified in the XNET LIN Schedule Collision Resolving Schedule property. The resolving schedule runs once, starting in the subsequent slot after the collision, and automatically returns to the previous schedule at the subsequent position where the collision occurred.

3 **Node configuration**: The schedule entry contains a node configuration service. The node configuration service is defined as raw data bytes in the XNET LIN Schedule Entry Node Configuration:Free Format:Data Bytes property.

XNET PDU Properties

This section includes the XNET PDU properties. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.)

Cluster

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET PDU

Property ID

`nxPropPDU_ClusterRef`

Description

This property returns the reference (`nxDatabaseRef_t`) to the parent cluster in which the PDU has been created. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.) You cannot change the parent cluster after creating the PDU object.

Comment

Data Type	Direction	Required?	Default
char *	Read/Write	No	Empty String

Property Class

XNET PDU

Property ID

`nxPropPDU_Comment`

Description

Comment describing the PDU object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
i32	Read Only	N/A	N/A

Property Class

XNET PDU

Property ID

`nxPropPDU_ConfigStatus`

Description

The PDU object's configuration status. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.)

Configuration Status returns an NI-XNET error code. The value can be passed to the error code input of `nxStatusToString` to convert it to a text description of the configuration problem.

By default, incorrectly configured PDUs in the database are not returned from the XNET Cluster PDUs property because they cannot be used in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When a PDU's configuration status became invalid after the database has been opened, the PDU still is returned from the XNET Cluster PDUs property even if ShowInvalidFromOpen? is false.

Examples of invalid PDU configuration:

- You have not defined a required property of the PDU (for example, PDU Payload Length).
- The number of bytes specified for this PDU is incorrect. CAN PDUs must use 0 to 8 bytes. FlexRay PDUs must use 0 to 254 bytes (PDUs payload must fit into a frame).

Frames

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET PDU

Property ID

`nxPropPDU_FrmRefs`

Description

References of all frames to which the PDU is mapped. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.) A PDU is transmitted within the frames to which it is mapped.

To map a PDU to a frame, use the XNET Frame PDU References, XNET Frame PDU Start Bits, and XNET Frame PDU Update Bits properties. You can map one PDU to multiple frames.

Mux:Data Multiplexer Signal

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET PDU

Property ID

`nxPropPDU_MuxDataMuxSigRef`

Description

Data multiplexer signal in the PDU. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.)

This property returns the reference to the data multiplexer signal. If data multiplexer is not defined in the PDU, the property returns 0. Use the XNET PDU Mux:Is Data Multiplexed? property to determine whether the PDU contains a multiplexer signal.

You can create a data multiplexer signal by creating a signal and then setting the XNET Signal Mux:Data Multiplexer? property to true.

A PDU can contain only one data multiplexer signal.

Mux:Is Data Multiplexed?

Data Type	Direction	Required?	Default
Boolean	Read Only	No	False

Property Class

XNET PDU

Property ID

`nxPropPDU_MuxIsMuxed`

Description

PDU is data multiplexed. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.)

This property returns true if the PDU contains a multiplexer signal. PDUs containing a multiplexer contain subframes that allow using bits of the payload for different information (signals), depending on the multiplexer value.

Mux:Static Signals

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET PDU

Property ID

`nxPropPDU_MuxStaticSigRefs`

Description

Static signals in the PDU. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.)

Returns an array of references to signals in the PDU that do not depend on the multiplexer value. Static signals are contained in every PDU transmitted, as opposed to dynamic signals, which are transmitted depending on the multiplexer value.

You can create static signals by specifying the PDU as the parent object. You can create dynamic signals by specifying a subframe as the parent.

If the PDU is not multiplexed, this property returns the same array as the XNET PDU Signals property.

Mux:Subframes

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET PDU

Property ID

`nxPropPDU_MuxSubframeRefs`

Description

Returns an array of references to subframes in the PDU. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.) A subframe defines a group of signals transmitted using the same multiplexer value. Only one subframe is transmitted in the PDU at a time.

You can define a subframe by creating a subframe object as a child of a PDU.

Name (Short)

Data Type	Direction	Required?	Default
char *	Read/Write	Yes	Defined in `nxdbCreateObject`

Property Class

XNET PDU

Property ID

`nxPropPDU_Name`

Description

String identifying a PDU object. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.)

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A PDU name must be unique for all PDUs in a cluster.

You can write this property to change the PDU's short name.

Payload Length

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET PDU

Property ID

`nxPropPDU_PayloadLen`

Description

Determines the size of the PDU data in bytes. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.)

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this PDU, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Signals

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET PDU

Property ID

`nxPropPDU_SigRefs`

Description

References to all signals in the PDU. (For more information about PDUs, refer to *Protocol Data Units (PDUs) in NI-XNET*.)

This property returns an array referencing to all signals in the PDU, including static and dynamic signals and the multiplexer signal.

This property is read only. You can add signals to a PDU using `nxdbCreateObject` and remove them using `nxdbDeleteObject`.

XNET Session Properties

This section includes the XNET Session properties.

Interface Properties

Properties in the Interface category apply to the interface and not the session. If more than one session exists for the interface, changing an interface property affects all the sessions.

CAN Interface Properties

This category includes CAN-specific interface properties.

Properties in the Interface category apply to the interface and not the session. If more than one session exists for the interface, changing an interface property affects all the sessions.

Interface:CAN:External Transceiver Config

Data Type	Direction	Required?	Default
u32	Write Only	No	0x00000007

Property Class

XNET Session

Property ID

`nxPropSession_IntfCANExtTcvrConfig`

Description

This property allows you to configure XS series CAN hardware to communicate properly with your external transceiver. The connector on your XS series CAN hardware has five lines for communicating with your transceiver.

Line	Direction	Purpose
Ext_RX	In	Data received from the CAN bus.
Ext_TX	Out	Data to transmit on the CAN bus.
Output0	Out	Generic output used to configure the transceiver mode.

Line	Direction	Purpose
Output1	Out	Generic output used to configure the transceiver mode.
NERR	In	Input to connect to the nERR pin of your transceiver to route status back from the transceiver to the hardware.

The Ext_RX and Ext_TX lines are self explanatory and provide for the transfer of CAN data to and from the transceiver. The remaining three lines are for configuring the transceiver and retrieving status from the transceivers. Not all transceivers use all pins. Typically, a transceiver has one or two lines that can configure the transceiver mode. The NI-XNET driver natively supports five transceiver modes: Normal, Sleep, Single Wire Wakeup, Single Wire High Speed, and Power-On. This property configures how the NI-XNET driver sets the outputs of your external transceiver for each mode.

The configuration is in the form of a u32 written as a bitmask. The u32 bitmask is defined as:

31	30..15	14..12	11..9	8..6	5..3	2..0
nERR Connected	Reserved	PowerOn Configuration	SWHighSpeed Configuration	SWWakeup Configuration	Sleep Configuration	Normal Configuration

Where each configuration is a 3-bit value defined as:

2	1	0
State Supported	Output1 Value	Output0 Value

The Interface:CAN:Transceiver State property changes the transceiver state. Based on the transceiver configuration, if the state is supported, the configuration determines how the two pins are set. If the state is not supported, an error is returned, because you tried to set an invalid configuration. Note that all transceivers must support a Normal state, so the State Supported bit for that configuration is ignored.

Other internal state changes may occur. For example, if you put the transceiver to sleep and a remote wakeup occurs, the transceiver automatically is changed to the normal state. For information about the state machine for the transceiver state, refer to *CAN Transceiver State Machine* in *Additional Topics*.

If nERR Connected is set, the nERR pin into the connector determines a transceiver error. It is active low, meaning a value of 0 on this pin indicates an error. A value of 1 indicates no

error. If this line is connected, the NI-XNET driver monitors this line and reports its status via the **Transceiver Error** field of `nxReadState (StateID = nxState_CANComm)`.

Examples

TJA1041 (HS): To connect to the TJA1041 transceiver, connect Output0 to the nSTB pin and Output1 to the EN pin. The TJA1041 does have an nERR pin, so that should be connected to the nERR input. The TJA1041 supports a power-on state, a sleep state, and a normal state. As this is not a single wire transceiver, it does not support any single wire state. For normal operation, the TJA1041 uses a 1 for both nSTB and EN. For sleep, the TJA1041 uses the standby mode, which uses a 0 for both nSTB and EN. For power-on, the TJA1041 uses a 1 for nSTB and a 0 for EN. The final configuration is 0x80005027.

TJA1054 (LS): You can connect and configure the TJA1054 identically to the TJA1041.

AU5790 (SW): To connect to the AU5790 transceiver, connect Output0 to the nSTB pin and Output1 to the EN pin. The AU5790 does not support any transceiver status, so you do not need to connect the nERR pin. The AU5790 supports all states. For normal operation, the AU5790 uses a 1 for both nSTB and EN. For sleep, the AU5790 uses a 0 for both nSTB and EN. For Single Wire Wakeup, the AU5790 requires nSTB to be a 0 and EN to be a 1. For Single Wire High-Speed, the AU5790 requires nSTB to be a 1, and EN to be a 0. For power-on, the sleep state is used so there is less interference on the bus.The final configuration is 0x00004DA7.

Interface:CAN:FD Baud Rate

Data Type	Direction	Required?	Default
u32	Read/Write	No	0

Property Class

XNET Session

Property ID

`nxPropSession_IntfCanFdBaudRate`

Description

Note You can modify this property only when the interface is stopped.

The Interface:CAN:FD Baud Rate property sets the fast data baud rate for CAN FD + BRS CAN:I/O Mode. The default value for this interface property is the same as the cluster's FD baud rate in the database. Your application can set this interface FD baud rate to override the value in the database.

When the upper nibble (0xF0000000) is clear, this is a numeric baud rate (for example, 500000).

NI-XNET CAN hardware currently accepts the following numeric baud rates: 200000, 250000, 400000, 500000, 800000, 1000000, 1250000, 1600000, 2000000, 2500000, 4000000, 5000000, and 8000000.

Note Not all CAN transceivers are rated to transmit at the requested rate. If you attempt to use a rate that exceeds the transceiver's qualified rate, XNET Start returns a warning. Chapter 3, *NI-XNET Hardware Overview*, describes the CAN transceivers' limitations.

When the upper nibble is set to 0x8 (that is, 0x80000000), the remaining bits provide fields for more custom CAN communication baud rate programming. The fields are shown in the following table:

	31..28	27..26	25..24	23..20	19..16	15...10	9..8	7..0
Normal	b0000	Baud Rate (200 k–8 M)						
Custom	b1000	Res	SJW (0–3)	TSEG2 (0–7)	TSEG1 (1–15)	Res	Tq (25–800)	

- (Re-)Synchronization Jump Width (SJW)
  - Valid programmed values are 0–3.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time Segment 2 (TSEG2) is the time segment after the sample point.
  - Valid programmed values are 0–7.
  - This is the Phase_Seg2(D) from *Bosch's CAN with Flexible Data-Rate* specification, version 1.0.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time Segment 1 (TSEG1) is the time segment before the sample point.
  - Valid programmed values are 1–15.
  - This is the combination of Prop_Seg(D) and Phase_Seg1(D) from Bosch's *CAN with Flexible Data-Rate* specification, version 1.0.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time quantum (Tq) is used to program the baud rate prescaler.
  - Valid programmed values are 25–800, in increments of 25 ns.

Interface:CAN:I/O Mode

Data Type	Direction	Required?	Default
u32	Read Only	—	Same as XNET Cluster CAN:I/O Mode

Property Class

XNET Session

Property ID

`nxPropSession_IntfCanIoMode`

Description

This property indicates the I/O Mode the interface is using. It is a ring of three values, as described in the following table:

Enumeration	Value	Meaning
CAN	0	This is the default CAN 2.0 A/B standard I/O mode as defined in ISO 11898-1:2003. A fixed baud rate is used for transfer, and the payload length is limited to 8 bytes.
CAN FD	1	This is the CAN FD mode as specified in the *CAN with Flexible Data-Rate* specification, version 1.0. Payload lengths are allowed up to 64 bytes, but they are transmitted at a single fixed baud rate (defined by the XNET Cluster Baud Rate or XNET Session Interface:Baud Rate properties.
CAN FD + BRS	2	This is the CAN FD mode as specified in the *CAN with Flexible Data-Rate* specification, version 1.0, with the optional Baud Rate Switching enabled. The same payload lengths as CAN FD mode are allowed; additionally, the data portion of the CAN frame is transferred at a different (higher) baud rate (defined by the XNET Cluster CAN:FD Baud Rate or XNET Session Interface:CAN:FD Baud Rate properties).

The value is initialized from the database cluster when the session is created and cannot be changed later. However, you can transmit standard CAN frames on a CAN FD network. Refer to the Interface:CAN:Transmit I/O Mode property.

Interface:CAN:Listen Only?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfCANLstnOnly`

Description



Note You can modify this property only when the interface is stopped.

The Listen Only? property configures whether the CAN interface transmits any information to the CAN bus.

When this property is false, the interface can transmit CAN frames and acknowledge received CAN frames.

When this property is true, the interface can neither transmit CAN frames nor acknowledge a received CAN frame. The true value enables passive monitoring of network traffic, which can be useful for debugging scenarios when you do not want to interfere with a communicating network cluster.

Interface:CAN:Pending Transmit Order

Data Type	Direction	Required?	Default
u32[]	Read/Write	No	As Submitted

Property Class

XNET Session

Property ID

`nxPropSession_IntfCANPendTxOrder`

Description



Note You can modify this property only when the interface is stopped.



Note Setting this property causes the internal queue to be flushed. If you start a session, queue frames, and then stop the session and change this mode, some frames may be lost. Set this property to the desired value once; do not constantly change modes.

The Pending Transmit Order property configures how the CAN interface manages the internal queue of frames. More than one frame may desire to transmit at the same time. NI-XNET stores the frames in an internal queue and transmits them onto the CAN bus when the bus is idle.

This property modifies how NI-XNET handles this queue of frames. The following table lists the accepted values:

Enumeration	Value
`nxCANPendTxOrder_AsSubmitted`	0
`nxCANPendTxOrder_ByIdentifier`	1

When you configure this property to be `nxCANPendTxOrder_AsSubmitted`, frames are transmitted in the order that they were submitted into the queue. There is no reordering of any frames, and a higher priority frame may be delayed due to the transmission or retransmission of a previously submitted frame. However, this mode has the highest performance.

When you configure this property to be `nxCANPendTxOrder_ByIdentifier`, frames with the highest priority identifier (lower CAN ID value) transmit first. The frames are stored in a priority queue sorted by ID. If a frame currently being transmitted requires retransmission (for example, it lost arbitration or failed with a bus error), and a higher priority frame is queued in

the meantime, the lower priority frame is not immediately retried, but the higher priority frame is transmitted instead. In this mode, you can emulate multiple ECUs and still see a behavior similar to a real bus in that the highest priority message is transmitted on the bus. This mode may be slower in performance (possible delays between transmissions as the queue is re-evaluated), and lower priority messages may be delayed indefinitely due to frequent high-priority messages.

Interface:CAN:Single Shot Transmit?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfCANSingShot`

Description

Note You can modify this property only when the interface is stopped.

Note Setting this property causes the internal queue to be flushed. If you start a session, queue frames, and then stop the session and change this mode, some frames may be lost. Set this property to the desired value once; do not constantly change modes.

The Single Shot Transmit? property configures whether the CAN interface retries failed transmissions.

When this property is false, failed transmissions retry as specified by the CAN protocol (ISO 11898-1, *6.11 Automatic Retransmission*). If a CAN frame is not transmitted successfully, the interface attempts to retransmit the frame as soon as the bus is idle again. This retransmit process continues until the frame is successfully transmitted.

When this property is true, failed transmissions do not retry. If a CAN frame is not transmitted successfully, no further transmissions are attempted.

Interface:CAN:Termination

Data Type	Direction	Required?	Default
u32	Read/Write	No	Off (0)

Property Class

XNET Session

Property ID

`nxPropSession_IntfCANTerm`

Description

Note You can modify this property only when the interface is stopped.

This property does not take effect until the interface is started.

The Termination property configures the onboard termination of the NI-XNET interface CAN connector (port). The enumeration is generic and supports two values: Off and On. However, different CAN hardware has different termination requirements, and the Off and On values have different meanings, as described below.

High-Speed CAN

High-Speed CAN networks are typically terminated on the bus itself instead of within a node. However, NI-XNET allows you to configure termination within the node to simplify testing. If your bus already has the correct amount of termination, leave this property in the default state of Off. However, if you require termination, set this property to On.

Value	Meaning	Description
Off	Disabled	Termination is disabled.
On	Enabled	Termination (120 Ω) is enabled.

Low-Speed/Fault-Tolerant CAN

Every node on a Low-Speed CAN network requires termination for each CAN data line (CAN_H and CAN_L). This configuration allows the Low-Speed/Fault-Tolerant CAN port to provide fault detection and recovery. Refer to *Termination* for more information about low-speed termination. In general, if the existing network has an overall network termination of 125 Ω or less, turn on termination to enable the 4.99 kΩ option. Otherwise, you should select the default 1.11 kΩ option.

Value	Meaning	Description
Off	1.11 kΩ	Termination is set to 1.11 kΩ.
On	4.99 kΩ	Termination is set to 4.99 kΩ.

Single Wire CAN

The ISO standard requires single wire transceivers to have a 9.09 kΩ resistor, and no additional configuration is supported.

Interface:CAN:Transceiver State

Data Type	Direction	Required?	Default
u32	Read/Write	No	Normal (0)

Property Class

XNET Session

Property ID

`nxPropSession_IntfCANTcvrState`

Description

The Transceiver State property configures the CAN transceiver and CAN controller modes. The transceiver state controls whether the transceiver is asleep or communicating, as well as configuring other special modes. The following table lists the accepted values.

Enumeration	Value
Normal	0
Sleep	1
Single Wire Wakeup	2
Single Wire High-Speed	3

Normal

This state sets the transceiver to normal communication mode. If the transceiver is in the Sleep mode, this performs a local wakeup of the transceiver and CAN controller chip.

Sleep

This state sets the transceiver and CAN controller chip to Sleep (or standby) mode. You can set the interface to Sleep mode only while the interface is communicating. If the interface has not been started, setting the transceiver to Sleep mode returns an error.

Before going to sleep, all pending transmissions are transmitted onto the CAN bus. Once all pending frames have been transmitted, the interface and transceiver go into Sleep (or standby) mode. Once the interface enters Sleep mode, further communication is not possible until a wakeup occurs. The transceiver and CAN controller wake from Sleep mode when either a local wakeup or remote wakeup occurs.

A local wakeup occurs when the application sets the transceiver state to either Normal or Single Wire Wakeup.

A remote wakeup occurs when a remote node transmits a CAN frame (referred to as the wakeup frame). The wakeup frame wakes up the NI-XNET interface transceiver and CAN controller chip. The CAN controller chip does not receive or acknowledge the wakeup frame. After detecting the wakeup frame and idle bus, the CAN interface enters Normal mode.

When the local or remote wakeup occurs, frame transmissions resume from the point at which the original Sleep mode was set.

You can use `nxReadState` to detect when a wakeup occurs. To suspend the application while waiting for the remote wakeup, use `nxWait`.

Single Wire Wakeup

For a remote wakeup to occur for Single Wire transceivers, the node that transmits the wakeup frame first must place the network into the Single Wire Wakeup Transmission mode by asserting a higher voltage.

This state sets a Single Wire transceiver into the Single Wire Wakeup Transmission mode, which forces the Single Wire transceiver to drive a higher voltage level on the network to wake up all sleeping nodes. Other than this higher voltage, this mode is similar to Normal mode. CAN frames can be received and transmitted normally.

If you are not using a Single Wire transceiver, setting this state returns an error. If your current mode is Single Wire High-Speed, setting this mode returns an error because you are not allowed to wake up the bus in high-speed mode.

The application controls the timing of how long the wakeup voltage is driven. The application typically changes to Single Wire Wakeup mode, transmits a single wakeup frame, and then returns to Normal mode.

Single Wire High-Speed

This state sets a Single Wire transceiver into Single Wire High-Speed Communication mode. If you are not using a Single Wire transceiver, setting this state returns an error.

Single Wire High-Speed Communication mode disables the transceiver's internal waveshaping function, allowing the SAE J2411 High-Speed baud rate of 83.333 kbytes/s to be used. The disadvantage versus Single Wire Normal Communication mode, which allows only the SAE J2411 baud rate of 33.333 kbytes/s, is degraded EMC performance. Other than the disabled waveshaping, this mode is similar to Normal mode. CAN frames can be received and transmitted normally.

This mode has no relationship to High-Speed transceivers. It is merely a higher speed mode of the Single Wire transceiver, typically used to download data when the onboard network is attached to an offboard tester ECU.

The Single Wire transceiver does not support use of this mode in conjunction with Sleep mode. For example, a remote wakeup cannot transition from sleep to this Single Wire High-Speed mode. Therefore, setting the mode to Sleep from Single Wire High-Speed mode returns an error.

Interface:CAN:Transceiver Type

Data Type	Direction	Required?	Default
u32	Read/Write	No	High-Speed (0) for High-Speed and XS Hardware; Low-Speed (1) for Low-Speed Hardware

Property Class

XNET Session

Property ID

`nxPropSession_IntfCANTcvrType`

Description

Note You can modify this property only when the interface is stopped.

For XNET hardware that provides a software-selectable transceiver, the Transceiver Type property allows you to set the transceiver type. Use the XNET Interface CAN.Transceiver Capability property to determine whether your hardware supports a software-selectable transceiver.

You also can use this property to determine the currently configured transceiver type.

The following table lists the accepted values:

Enumeration	Value
High-Speed (HS)	0
Low-Speed (LS)	1
Single Wire (SW)	2
External (Ext)	3
Disconnected (Disc)	4

The default value for this property depends on your type of hardware. If you have fixed-personality hardware, the default value is the hardware value. If you have hardware that supports software-selectable transceivers, the default is High-Speed.

This attribute uses the following values:

High-Speed

This configuration enables the High-Speed transceiver. This transceiver supports baud rates of 40 kbaud to 1 Mbaud. When using a High-Speed transceiver, you also can communicate with a CAN FD bus. Refer to Chapter 3, *NI-XNET Hardware Overview*, to determine which CAN FD baud rates are supported.

Low-Speed/Fault-Tolerant

This configuration enables the Low-Speed/Fault-Tolerant transceiver. This transceiver supports baud rates of 40–125 kbaud.

Single Wire

This configuration enables the Single Wire transceiver. This transceiver supports baud rates of 33.333 kbaud and 83.333 kbaud.

External

This configuration allows you to use an external transceiver to connect to your CAN bus. Refer to the XNET Session Interface:CAN:External Transceiver Config property for more information.

Disconnect

This configuration allows you to disconnect the CAN controller chip from the connector. You can use this value when you physically change the external transceiver.

Interface:CAN:Transmit I/O Mode

Data Type	Direction	Required?	Default
u32	Read/Write	No	Same as Interface:CAN:I/O Mode

Property Class

XNET Session

Property ID

`nxPropSession_IntfCanTxIoMode`

Description

This property specifies the I/O Mode the interface uses when transmitting a CAN frame. By default, it is the same as the XNET Cluster CAN:I/O Mode property. However, even if the interface is in CAN FD (+ BRS) mode, you can force it to transmit frames in the standard CAN format. For this purpose, set this property to CAN.

Note This property affects only the transmission of frames. Even if you set the Transmit I/O mode to CAN, the interface still can receive frames in FD modes (if the XNET Cluster CAN:I/O Mode property is configured in an FD mode).

The Transmit I/O mode may not exceed the mode set by the XNET Cluster CAN:I/O Mode property.

FlexRay Interface Properties

This category includes FlexRay-specific interface properties.

Properties in the Interface category apply to the interface and not the session. If more than one session exists for the interface, changing an interface property affects all the sessions.

These properties are calculated based on constraints in the *FlexRay Protocol Specification*. To calculate these properties, the constraints use cluster settings and knowledge of the oscillator that the FlexRay interface uses.

At Create Session time, the XNET driver automatically calculates these properties, and they are passed down to the hardware. However, you can use the XNET property node to change these settings.



Note Changing the interface properties can affect the integration and communication of the XNET FlexRay interface with the cluster.

Interface:FlexRay:Accepted Startup Range

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayAccStartRng`

Description

Range of measure clock deviation allowed for startup frames during node integration. This property corresponds to the `pdAcceptedStartupRange` node parameter in the *FlexRay Protocol Specification*.

The range for this property is 0–1875 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Allow Halt Due To Clock?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayAlwHltClk`

Description

Controls the FlexRay interface transition to the POC: halt state due to clock synchronization errors. If set to true, the node can transition to the POC: halt state. If set to false, the node does not transition to the POC: halt state and remains in the POC: normal passive state, allowing for self recovery.

This property corresponds to the `pAllowHaltDueToClock` node parameter in the *FlexRay Protocol Specification*.

The property is a Boolean flag.

The default value of this property is false.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Refer to `nxReadState` for more information about the POC: halt and POC: normal passive states.

Interface:FlexRay:Allow Passive to Active

Data Type	Direction	Required?	Default
u32	Read/Write	No	0

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayAlwPassAct`

Description

Number of consecutive even/odd cycle pairs that must have valid clock correction terms before the FlexRay node can transition from the POC: normal-passive to the POC: normal-active state. If set to zero, the node cannot transition from POC: normal-passive to POC: normal-active.

This property corresponds to the `pAllowPassiveToActive` node parameter in the *FlexRay Protocol Specification*.

The property is expressed as the number of even/odd cycle pairs, with values of 0–31.

The default value of this property is zero.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Refer to `nxReadState` for more information about the POC: normal-active and POC: normal-passive states.

Interface:FlexRay:AutoAsleepWhenStopped

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayAutoAslpWhnStp`

Description

This property indicates whether the FlexRay interface (node) automatically places the FlexRay transceiver and controller into sleep when the interface is stopped. The default value of this property is False, and you must handle the wakeup/sleep processing manually using `nxSetProperty` with the property ID of `nxPropSession_IntfFlexRaySleep`.

When this property is called with the value True while the interface is asleep, the interface is put to sleep immediately. When this property is called with the value False, the interface is set to a local awake state immediately.

If the interface is asleep when `nxStart` is called, the FlexRay interface waits for a wakeup pattern on the bus before transitioning out of the POC:READY state. To initiate a bus wakeup, set `nxSetProperty` with the property ID of `nxPropSession_IntfFlexRaySleep` and a value of `nxFlexRaySleep_RemoteWake`.

After `nxStop` is called, if this property is True, the FlexRay interface automatically goes back to sleep to be ready to handle the wakeup on subsequent `nxStart` calls. When this property is False when `nxStop` is called, the FlexRay interface remains in the sleep state it was in prior to the `nxStop` call.

You can overwrite the default value by writing this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Cluster Drift Damping

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayClstDriftDmp`

Description

Local cluster drift damping factor used for rate correction.

This property corresponds to the `pAllowPassiveToActive` node parameter in the *FlexRay Protocol Specification*. The range for the property is 0–20 MT.

The cluster drift damping property should be configured in such a way that the damping values in all nodes within the same cluster have approximately the same duration.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Coldstart?

Data Type	Direction	Required?	Default
Boolean	Read Only	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayColdstart`

Description

This property specifies whether the FlexRay interface operates as a coldstart node on the cluster. This property is read only and calculated from the XNET Session Interface:FlexRay:Key Slot Identifier property. If the KeySlot Identifier is 0 (invalid slot identifier), the XNET FlexRay interface does not act as a coldstart node, and this property is false. If the KeySlot Identifier is 1 or more, the XNET FlexRay interface transmits a startup frame from that slot, and the Coldstart? property is true.

This property returns a Boolean flag (true/false).

The default value of this property is false.

Interface:FlexRay:Connected Channels

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayConnectedChs`

Description

This property specifies the channel(s) that the FlexRay interface (node) is physically connected to. The default value of this property is connected to all channels available on the cluster. However, if you are using a node connected to only one channel of a multichannel cluster that uses wakeup, you must set the value properly. If you do not, your node may not wake up, as the wakeup pattern cannot be received on a channel that is not physically connected.

This property corresponds to the `pChannels` node parameter in the *FlexRay Protocol Specification*.

The values supported for this property (enumeration) are A = 1, B = 2, and A and B = 3.

You can overwrite the default value by writing this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Decoding Correction

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayDecCorr`

Description

This property specifies the value that the receiving FlexRay node uses to calculate the difference between the primary time reference point and secondary reference point. The clock synchronization algorithm uses the primary time reference and the sync frame's expected arrival time to calculate and compensate for the node's local clock deviation.

This property corresponds to the `pDecodingCorrection` node parameter in the *FlexRay Protocol Specification*.

The range for the property is 14–143 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Delay Compensation Ch A

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayDelayCompA`

Description

This property specifies the value that the XNET FlexRay interface (node) uses to compensate for reception delays on channel A. This takes into account the assumed propagation delay up to the maximum allowed propagation delay (`cPropagationDelayMax`) for microticks in the 0.0125–0.05 range. In practice, you should apply the minimum of the propagation delays of all sync nodes.

This property corresponds to the `pDelayCompensation[A]` node parameter in the *FlexRay Protocol Specification*.

The property range is 0–200 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Delay Compensation Ch B

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayDelayCompB`

Description

This property specifies the value that the XNET FlexRay interface (node) uses to compensate for reception delays on channel B. This takes into account the assumed propagation delay up to the maximum allowed propagation delay (`Propagation Delay Max`) for microticks in the 0.0125–0.05 range. In practice, you should apply the minimum of the propagation delays of all sync nodes.

This property corresponds to the `pDelayCompensation[B]` node parameter in the *FlexRay Protocol Specification*.

The property range is 0–200 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Key Slot Identifier

Data Type	Direction	Required?	Default
u32	Read/Write	No	0

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayKeySlotID`

Description

This property specifies the FlexRay slot number from which the XNET FlexRay interface transmits a startup frame, during the process of integration with other cluster nodes.

For a network (cluster) of FlexRay nodes to start up for communication, at least two nodes must transmit startup frames. If your application is designed to test only one external ECU, you must configure the XNET FlexRay interface to transmit a startup frame. If the one external ECU does not transmit a startup frame itself, you must use two XNET FlexRay interfaces for the test, each of which must transmit a startup frame.

There are two methods for configuring the XNET FlexRay interface as a coldstart node (transmit startup frame).

Output Session with Startup Frame

Create an output session that contains a startup frame (or one of its signals). The XNET Frame FlexRay:Startup? property is true for a startup frame. If you use this method, this Key Slot Identifier property contains the identifier property of that startup frame. You do not write this property.

Write this Key Slot Identifier Property

This interface uses the identifier (slot) you write to transmit a startup frame using that slot.

Note If you create an output session that contains the startup frame, with the same identifier as that specified in the Key Slot Identifier property, the data you write to the session transmits in the frame. If you do not create an output session that contains the startup frame, the interface transmits a null frame for startup purposes.

If you create an output session that contains a startup frame with an identifier that does not match the Key Slot Identifier property, an error is returned.

The default value of this property is 0 (no startup frame).

You can overwrite the default value by writing an identifier that corresponds to the identifier of a startup frame prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Latest Tx

Data Type	Direction	Required?	Default
u32	Read Only	No	0

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayLatestTx`

Description

This property specifies the number of the last minislot in which a frame transmission can start in the dynamic segment. This is a read-only property, as the FlexRay controller evaluates it based on the configuration of the frames in the dynamic segment.

This property corresponds to the `pLatestTx` node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 0–7981 minislots.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Listen Timeout

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayListTimo`

Description

This property specifies the upper limit for the startup listen timeout and wakeup listen timeout.

Refer to Appendix B, *Summary of the FlexRay Standard* for more information about startup and wakeup procedures within the FlexRay protocol.

This property corresponds to the `pdListenTimeout` node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 1284–1283846 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Macro Initial Offset Ch A

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayMacInitOffA`

Description

This property specifies the integer number of macroticks between the static slot boundary and the following macrotick boundary of the secondary time reference point based on the nominal macrotick duration. This property applies only to Channel A.

This property corresponds to the `pMacroInitialOffset[A]` node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 2–72 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Macro Initial Offset Ch B

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayMacInitOffB`

Description

This property specifies the integer number of macroticks between the static slot boundary and the following macrotick boundary of the secondary time reference point based on the nominal macrotick duration. This property applies only to Channel B.

This property corresponds to the `pMacroInitialOffset[B]` node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 2–72 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Max Drift

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayMaxDrift`

Description

This property specifies the maximum drift offset between two nodes that operate with unsynchronized clocks over one communication cycle.

This property corresponds to the `pdMaxDrift` node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 2–1923 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Micro Initial Offset Ch A

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayMicInitOffA`

Description

This property specifies the number of microticks between the closest macrotick boundary described by the Macro Initial Offset Ch A property and the secondary time reference point. This parameter depends on the Delay Compensation property for Channel A, and therefore you must set it independently for each channel.

This property corresponds to the `pMicroInitialOffset[A]` node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 0–240 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Micro Initial Offset Ch B

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayMicInitOffB`

Description

This property specifies the number of microticks between the closest macrotick boundary described by the Macro Initial Offset Ch B property and the secondary time reference point. This parameter depends on the Delay Compensation property for Channel B, and therefore you must set it independently for each channel.

This property corresponds to the `pMicroInitialOffset[B]` node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 0–240 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Microtick

Data Type	Direction	Required?	Default
u32	Read Only	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayMicrotick`

Description

This property specifies the duration of a microtick. This property is calculated based on the product of the Interface:FlexRay:Samples Per Microtick and Baud Rate properties. This is a read-only property.

This property corresponds to the `pdMicrotick` node parameter in the *FlexRay Protocol Specification*.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Null Frames To Input Stream?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayNullToInStrm`

Description

This property indicates whether the Frame Input Stream Mode session should return FlexRay null frames from `nxReadFrame`.

When this property uses the default value of false, FlexRay null frames are not returned for a Frame Input Stream Mode session. This behavior is consistent with the other two frame input modes (Frame Input Single-Point Mode and Frame Input Queued Mode), which never return FlexRay null frames from `nxReadFrame`.

When you set this property to true for a Frame Input Stream Mode session, `nxReadFrame` returns all FlexRay null frames that are received by the interface. This feature is used to monitor all frames that occur on the network, regardless of whether new payload is available or not. When you use `nxReadFrame`, each frame's type field indicates a null frame.

You can overwrite the default value prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Offset Correction

Data Type	Direction	Required?	Default
i32	Read Only	No	N/A

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayOffCorr`

Description

This property provides the maximum permissible offset correction value, expressed in microticks. The offset correction synchronizes the cycle start time. The value indicates the number of microticks added or subtracted to the offset correction portion of the network idle time, to synchronize the interface to the FlexRay network. The value is returned as a signed 32-bit integer (i32). The offset correction value calculation takes place every cycle, but the correction is applied only at the end of odd cycles. This is a read-only property.

This property can be read anytime prior to closing the FlexRay interface.

Interface:FlexRay:Offset Correction Out

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayOffCorrOut`

Description

This property specifies the magnitude of the maximum permissible offset correction value. This node parameter is based on the value of the maximum offset correction for the specific cluster.

This property corresponds to the `pOffsetCorrectionOut` node parameter in the *FlexRay Protocol Specification*.

The value range for this property is 5–15266 MT.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Rate Correction

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayRateCorr`

Description

Read-only property that provides the rate correction value, expressed in microticks. The rate correction synchronizes frequency. The value indicates the number of microticks added to or subtracted from the configured number of microticks in a cycle, to synchronize the interface to the FlexRay network.

The value is returned as a signed 32-bit integer (i32). The rate correction value calculation takes place in the static segment of an odd cycle, based on values measured in an even-odd double cycle.

This property can be read prior to closing the FlexRay interface.

Interface:FlexRay:Rate Correction Out

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayRateCorrOut`

Description

This property specifies the magnitude of the maximum permissible rate correction value. This node parameter is based on the value of the maximum rate correction for the specific cluster.

This property corresponds to the `pRateCorrectionOut` node parameter in the *FlexRay Protocol Specification*.

The range of values for this property is 2–1923 MT.

This property is calculated from the microticks per cycle and clock accuracy.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Samples Per Microtick

Data Type	Direction	Required?	Default
u32	Read/Write	No	Calculated from Cluster Settings

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRaySampPerMicro`

Description

This property specifies the number of samples per microtick.

There is a defined relationship between the “ticks” of the microtick timebase and the sample ticks of bit sampling. Specifically, a microtick consists of an integral number of samples.

As a result, there is a fixed phase relationship between the microtick timebase and the sample clock ticks.

This property corresponds to the `pSamplesPerMicrotick` node parameter in the *FlexRay Protocol Specification*.

The supported values for this property are 1, 2, and 4 samples.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Single Slot Enabled?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRaySingSlotEn`

Description

This property serves as a flag to indicate whether the FlexRay interface (node) should enter single slot mode following startup.

This Boolean property supports a strategy to limit frame transmissions following startup to a single frame (designated by the XNET Session Interface:FlexRay:Key Slot Identifier property). If you leave this property false prior to start (default), all configured output frames transmit. If you set this property to true prior to start, only the key slot transmits. After the interface is communicating (integrated), you can set this property to false at runtime to enable the remaining transmissions (the protocol's ALL_SLOTS command). After the interface is communicating, you cannot set this property from false to true.

This property corresponds to the `pSingleSlotEnabled` node parameter in the *FlexRay Protocol Specification*.

You can overwrite the default value prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Sleep

Data Type	Direction	Required?	Default
u32	Write Only	No	N/A

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRaySleep`

Description

Use the Sleep property to change the NI-XNET FlexRay interface sleep/awake state and optionally to initiate a wakeup on the FlexRay cluster.

The following table lists the accepted values:

String	Value	Description
`nxFlexRaySleep_LocalSleep`	0	Set interface and transceiver(s) to sleep
`nxFlexRaySleep_LocalWake`	1	Set interface and transceiver(s) to awake
`nxFlexRaySleep_RemoteWake`	2	Set interface and transceivers to awake and attempt to wake up the FlexRay bus by sending the wakeup pattern on the configured wakeup channel

This property is write only. Setting a new value is effectively a request, and the property node returns before the request is complete. To detect the current interface sleep/wake state, use `nxReadState`.

The FlexRay interface maintains a state machine to determine the action to perform when this property is set (request). The following table specifies the sleep/wake action on the FlexRay interface.

Request	Current Local State	
	Sleep	Awake
nxFlexRaySleep_LocalSleep	No action	Change local state
nxFlexRaySleep_LocalWake	Attempt to integrate with the bus (move from POC:READY to POC:NORMAL)	No action
nxFlexRaySleep_RemoteWake	Attempt to wake up the bus followed by an attempt to integrate with the bus (move from POC:READY to POC:NORMAL ACTIVE). If the interface is not yet started, setting nxFlexRaySleep_RemoteWake schedules a remote wake to be generated once the interface has started.	No action

Interface:FlexRay:Statistics Enabled?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayStatisticsEn`

Description

This XNET Boolean property enables reporting FlexRay error statistics. When this property is false (default), calls to `nxReadState` always return zero for each statistic. To enable FlexRay statistics, set this property to true in your application.

You can overwrite the default value prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Symbol Frames To Input Stream?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRaySymToInStrm`

Description

This property indicates whether the Frame Input Stream Mode session should return FlexRay symbols from `nxReadFrame`.

When this property uses the default value of false, FlexRay symbols are not returned for a Frame Input Stream Mode session. This behavior is consistent with the other two frame input modes (Frame Input Single-Point Mode and Frame Input Queued Mode), which never return FlexRay symbols from `nxReadFrame`.

When you set this property to True for a Frame Input Stream Mode session, `nxReadFrame` returns all FlexRay symbols the interface receives. This feature detects wakeup symbols and Media Access Test Symbols (MTS). When you use `nxReadFrame`, each frame's type field indicates a symbol.

When the frame type is FlexRay Symbol, the 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 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).

You can overwrite the default value prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Sync Frame Status

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRaySyncStatus`

Description

This property returns the status of sync frames since the interface (enumeration) start. *Within Limits* means the number of sync frames is within the protocol's limits since the interface start. *Below Minimum* means that in at least one cycle, the number of sync frames was below the limit the protocol requires (2 or 3, depending on number of nodes). *Overflow* means that in at least one cycle, the number of sync frames was above the limit set by the XNET Cluster FlexRay:Sync Node Max property. *Both Min and Max* means that both minimum and overflow errors have occurred (this is unlikely).

If the interface is not started, this property returns Within Limits. If you start the interface, but it fails to communicate (integrate), this property may be helpful in diagnosing the problem.

Refer to Appendix B, *Summary of the FlexRay Standard*, for more information about the FlexRay protocol startup and cluster integration procedure.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Sync Frames Channel A Even

Data Type	Direction	Required?	Default
u32 [16]	Read Only	No	N/A

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRaySyncChAEven`

Description

This property returns an array of sync frames (slot IDs) transmitted or received on channel A during the last even cycle. This read-only property returns an array in which each element holds the slot ID of a sync frame. If the interface is not started, this returns an empty array. If you start the interface, but it fails to communicate (integrate), this property may be helpful in diagnosing the problem.

Refer to Appendix B, *Summary of the FlexRay Standard*, for more information about the FlexRay protocol startup procedure.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Sync Frames Channel A Odd

Data Type	Direction	Required?	Default
u32 [16]	Read Only	No	N/A

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRaySyncChAOdd`

Description

This property returns an array of sync frames (slot IDs) transmitted or received on channel A during the last odd cycle. This read-only property returns an array in which each element holds the slot ID of a sync frame. If the interface is not started, this returns an empty array. If you start the interface, but it fails to communicate (integrate), this property may be helpful in diagnosing the problem.

Refer to Appendix B, *Summary of the FlexRay Standard*, for more information about the FlexRay protocol startup procedure.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Sync Frames Channel B Even

Data Type	Direction	Required?	Default
u32 [16]	Read Only	No	N/A

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRaySyncChBEven`

Description

This property returns an array of sync frames (slot IDs) transmitted or received on channel B during the last even cycle. This read-only property returns an array in which each element holds the slot ID of a sync frame. If the interface is not started, this returns an empty array. If you start the interface, but it fails to communicate (integrate), this property may be helpful in diagnosing the problem.

Refer to Appendix B, *Summary of the FlexRay Standard*, for more information about the FlexRay protocol startup procedure.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Sync Frames Channel B Odd

Data Type	Direction	Required?	Default
u32 [16]	Read Only	No	N/A

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRaySyncChBOdd`

Description

This property returns an array of sync frames (slot IDs) transmitted or received on channel B during the last odd cycle. This read-only property returns an array in which each element holds the slot ID of a sync frame. If the interface is not started, this returns an empty array. If you start the interface, but it fails to communicate (integrate), this property may be helpful in diagnosing the problem.

Refer to Appendix B, *Summary of the FlexRay Standard*, for more information about the FlexRay protocol startup procedure.

This property can be read any time prior to closing the FlexRay interface.

Interface:FlexRay:Termination

Data Type	Direction	Required?	Default
u32	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayTerm`

Description

This property controls termination at the NI-XNET interface (enumeration) connector (port). This applies to both channels (A and B) on each FlexRay interface. False means the interface is not terminated (default). True means the interface is terminated.

You can overwrite the default value by writing this property prior to starting the FlexRay interface (refer to *Session States* for more information). You can start the FlexRay interface by calling `nxStart` with `scope` set to either Normal or Interface Only on the session.

Interface:FlexRay:Wakeup Channel

Data Type	Direction	Required?	Default
u32	Read/Write	No	A

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayWakeupCh`

Description

This property specifies the channel the FlexRay interface (node) uses to send a wakeup pattern. This property is used only when the XNET Session Interface:FlexRay:Sleep property is set to `nxFlexRaySleep_RemoteWake`.

This property corresponds to the `pWakeupChannel` node parameter in the *FlexRay Protocol Specification*.

The values supported for this property (enumeration) are A = 0 and B = 1.

You can overwrite the default value by writing this property prior to starting the FlexRay interface (refer to *Session States* for more information).

Interface:FlexRay:Wakeup Pattern

Data Type	Direction	Required?	Default
u32	Read/Write	No	2

Property Class

XNET Session

Property ID

`nxPropSession_IntfFlexRayWakeupPtrn`

Description

This property specifies the number of repetitions of the wakeup symbol that are combined to form a wakeup pattern when the FlexRay interface (node) enters the POC:wakeup send state. The POC:wakeup send state is one of the FlexRay controller state transitions during the wakeup process. In this state, the controller sends the wakeup pattern on the specified Wakeup Channel and checks for collisions on the bus.

This property corresponds to the `pWakeupPattern` node parameter in the *FlexRay Protocol Specification*.

The supported values for this property are 2–63.

You can overwrite the default value by writing a value within the specified range to this property prior to starting the FlexRay interface (refer to *Session States* for more information).

LIN Interface Properties

This category includes LIN-specific interface properties.

Properties in the Interface category apply to the interface and not the session. If more than one session exists for the interface, changing an interface property affects all the sessions.

Interface:LIN:Break Length

Data Type	Direction	Required?	Default
u32	Read/Write	No	13

Property Class

XNET Session

Property ID

`nxPropSession_IntfLINBreakLength`

Description

This property determines the length of the serial break used at the start of a frame header (schedule entry). The value is specified in bit-times.

The valid range is 13–36 (inclusive). The default value is 13, which is the value the LIN standard specifies.

At baud rates below 9600, the upper limit may be lower than 36 to avoid violating hold times for the bus. For example, at 2400 baud, the valid range is 13–14.

This property is applicable only when the interface is the master.

Interface:LIN:DiagP2min

Data Type	Direction	Required?	Default
Double	Read/Write	No	0.05

Property Class

XNET Session

Property ID

`nxPropSession_IntfLINDiagP2min`

Description

When the interface is the slave, this is the minimum time in seconds between reception of the last frame of the diagnostic request message and transmission of the response for the first frame in the diagnostic response message by the slave.

This property applies only to the interface as slave. An attempt to write the property for interface as master results in error nxErrInvalidPropertyValue being reported.

Interface:LIN:DiagSTmin

Data Type	Direction	Required?	Default
Double	Read/Write	No	0

Property Class

XNET Session

Property ID

`nxPropSession_IntfLINDiagSTmin`

Description

When the interface is the slave, this property sets the minimum time in seconds it places between the end of transmission of a frame in a diagnostic response message and the start of transmission of the response for the next frame in the diagnostic response message.

When the interface is the master, this property sets the minimum time in seconds it places between the end of transmission of a frame in a diagnostic request message and the start of transmission of the next frame in the diagnostic request message.

Interface:LIN:Master?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfLINMaster`

Description

Note You can set this property only when the interface is stopped.

This Boolean property specifies the NI-XNET LIN interface role on the network: master (true) or slave (false).

In a LIN network (cluster), there always is a single ECU in the system called the master. The master transmits a schedule of frame headers. Each frame header is a remote request for a specific frame ID. For each header, typically a single ECU in the network (slave) responds by transmitting the requested ID payload. The master ECU can respond to a specific header as well, and thus the master can transmit payload data for the slave ECUs to receive. For more information, refer to Appendix C, *Summary of the LIN Standard*.

The default value for this property is false (slave). This means that by default, the interface does not transmit frame headers onto the network. When you use input sessions, you read frames that other ECUs transmit. When you use output sessions, the NI-XNET interface waits for the remote master to send a header for a frame in the output sessions, then the interface responds with data for the requested frame.

If you call the `nxWriteState` function to request execution of a schedule, that implicitly sets this property to true (master). You also can set this property to true using `nxSetProperty`, but no schedule is active by default, so you still must call the `nxWriteState` function at some point to request a specific schedule.

Regardless of this property's value, you use can input and output sessions. This property specifies which hardware transmits the scheduled frame headers: NI-XNET (true) or a remote master ECU (false).

Interface:LIN:Output Stream Slave Response List By NAD

Data Type	Direction	Required?	Default
u32[]	Read/Write	No	Empty Array

Property Class

XNET Session

Property ID

`nxPropSession_IntfLINOStrSlvRspLstByNAD`

Description

The Output Stream Slave Response List by NAD property provides a list of NADs for use with the replay feature (Interface:Output Stream Timing property set to Replay Exclusive or Replay Inclusive).

For LIN, the array of frames to replay might contain multiple slave response frames, each with the same slave response identifier, but each having been transmitted by a different slave (per the NAD value in the data payload). This means that processing slave response frames for replay requires two levels of filtering. First, you can include or exclude the slave response frame or ID for replay using Interface:Output Stream List or Interface:Output Stream List By ID. If you do not include the slave response frame or ID for replay, no slave responses are transmitted. If you do include the slave response frame or ID for replay, you can use the Output Stream Slave Response List by NAD property to filter which slave responses (per the NAD values in the array) are transmitted. This property is always inclusive, regardless of the replay mode (inclusive or exclusive). If the NAD is in the list and the response frame or ID has been enabled for replay, any slave response for that NAD is transmitted. If the NAD is not in the list, no slave response for that NAD is transmitted. The property's data type is an array of unsigned 32-bit integer (u32). Currently, only byte 0 is required to hold the NAD value. The remaining bits are reserved for future use.

Interface:LIN:Schedule Names

Data Type	Direction	Required?	Default
cstr	Read Only	No	N/A

Property Class

XNET Session

Property ID

`nxPropSession_IntfLINSchedNames`

Description

This property returns a comma-separated list of schedules for use when the NI-XNET LIN interface acts as a master (Interface:LIN:Master? is true). When the interface is master, you can pass the index of one of these schedules to the `nxWriteState` function to request a schedule change.

When the interface does not act as a master, you cannot control the schedule, and the `nxWriteState` function returns an error if it cannot set the interface into master mode (for example, if the interface already is started).

This list of schedules is the same list the XNET Cluster Schedules property used to configure the session.

Interface:LIN:Sleep

Data Type	Direction	Required?	Default
u32	Write Only	No	N/A

Property Class

XNET Session

Property ID

`nxPropSession_IntfLINSleep`

Description

Use the Sleep property to change the NI-XNET LIN interface sleep/awake state and optionally to change remote node (ECU) sleep/awake states.

The following table lists the accepted values:

String	Value	Description
nxLINSleep_RemoteSleep	0	Set interface to sleep locally and transmit sleep requests to remote nodes
nxLINSleep_RemoteWake	1	Set interface to awake locally and transmit wakeup requests to remote nodes
nxLINSleep_LocalSleep	2	Set interface to sleep locally and not to interact with the network
nxLINSleep_LocalWake	3	Set interface to awake locally and not to interact with the network

The property is write only. Setting a new value is effectively a request, and the property node returns before the request is complete. To detect the current interface sleep/wake state, use `nxReadState`.

The LIN interface maintains a state machine to determine the action to perform when this property is set (request). The following sections specify the action when the interface is master and slave.

Table 5-1. Sleep/Wake Action for Master

Request	Current Local State	
	Sleep	Awake
nxLINSleep_RemoteSleep	No action	Change local state; pause scheduler; transmit go-to-sleep request frame
nxLINSleep_RemoteWake	Change local state; transmit master wakeup pattern (serial break); resume scheduler	No action
nxLINSleep_LocalSleep	No action	Change local state
nxLINSleep_LocalWake	Change local state; resume scheduler	No action

When the master's scheduler pauses, it finishes the pending entry (slot) and saves its current position. When the master's scheduler resumes, it continues with the schedule where it left off (entry after the pause).

The go-to-sleep request is frame ID 63, payload length 8, payload byte 0 has the value 0, and the remaining bytes have the value 0xFF.

If the master is in the Sleep state, and a remote slave (ECU) transmits the slave wakeup pattern, this is equivalent to setting this property to Local Wake. In addition, a pending `nxWait` for `nxCondition_IntfRemoteWakeup` returns. This `nxWait` does not apply to setting this property, because you know when you set it.

Table 5-2. Sleep/Wake Action for Slave

Request	Current Local State	
	Sleep	Awake
nxLINSleep_RemoteSleep	Error	Error
nxLINSleep_RemoteWake	Change local state; transmit slave wakeup pattern	No action
nxLINSleep_LocalSleep	No action	Change local state
nxLINSleep_LocalWake	Change local state	No action

According to the LIN protocol standard, Remote Sleep is not supported for slave mode, so that request returns an error.

If the slave is in Sleep state, and a remote master (ECU) transmits the master wakeup pattern, this is equivalent to setting this property to Local Wake. In addition, a pending `nxWait` for `nxCondition_IntfRemoteWakeup` returns. This `nxWait` does not apply to setting this property, because you know when you set it.

Interface:LIN:Start Allowed without Bus Power?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfLINAlwStartWoBusPwr`

Description

Note You can modify this property only when the interface is stopped.

The Start Allowed Without Bus Power? property configures whether the LIN interface does not check for bus power present at interface start, or checks and reports an error if bus power is missing.

When this property is true, the LIN interface does not check for bus power present at start, so no error is reported if the interface is started without bus power.

When this property is false, the LIN interface checks for bus power present at start, and nxErrMissingBusPower is reported if the interface is started without bus power.

Interface:LIN:Termination

Data Type	Direction	Required?	Default
u32	Read/Write	No	Off (0)

Property Class

XNET Session

Property ID

`nxPropSession_IntfLINTerm`

Description

Notes You can modify this property only when the interface is stopped.

This property does not take effect until the interface is started.

The Termination property configures the NI-XNET interface LIN connector (port) onboard termination. The enumeration is generic and supports two values: Off (disabled) and On (enabled).

The following table lists the accepted values:

String	Value
nxLINTerm_Off	0
nxLINTerm_On	1

Per the LIN 2.1 standard, the Master ECU has a ~1 kΩ termination resistor between Vbat and Vbus. Therefore, use this property only if you are using your interface as the master and do not already have external termination.

For more information about LIN cabling and termination, refer to *NI-XNET LIN Hardware*.

Source Terminal Interface Properties

This category includes properties to route trigger signals between multiple DAQmx and XNET devices.

Interface:Source Terminal:Start Trigger

Data Type	Direction	Required?	Default
cstr	Read/Write	No	(Disconnected)

Property Class

XNET Session

Property ID

`nxPropSession_IntfSrcTermStartTrigger`

Description

This property specifies the name of the internal terminal to use as the interface Start Trigger. The data type is NI Terminal (DAQmx terminal), represented as a string.

This property is supported for C Series modules in a CompactDAQ chassis. It is not supported for CompactRIO, PXI, or PCI (refer to `nxConnectTerminals` for those platforms).

The digital trigger signal at this terminal is for the Start Interface transition, to begin communication for all sessions that use the interface. This property routes the start trigger, but not the timebase (used for timestamp of received frames and cyclic transmit of frames). Routing the timebase is not required for CompactDAQ, because all modules in the chassis automatically use a shared timebase.

Use this property to connect the interface Start Trigger to triggers in other modules and/or interfaces. When you read this property, you specify the interface Start Trigger as the source of a connection. When you write this property, you specify the interface Start Trigger as the destination of a connection, and the value you write represents the source. For examples that demonstrate use of this property to synchronize NI-XNET and NI-DAQmx hardware, refer to the **Synchronization** category within the NI-XNET examples.

The connection this property creates is disconnected when you clear (close) all sessions that use the interface.

Interface:Baud Rate

Data Type	Direction	Required?	Default
u32	Read/Write	Yes (If Not in Database)	0 (If Not in Database)

Property Class

XNET Session

Property ID

`nxPropSession_IntfBaudRate`

Description

Note You can modify this property only when the interface is stopped.

The Interface:Baud Rate property sets the CAN, FlexRay, or LIN interface baud rate. The default value for this interface property is the same as the cluster's baud rate in the database. Your application can set this interface baud rate to override the value in the database, or when no database is used.

CAN

When the upper nibble (0xF0000000) is clear, this is a numeric baud rate (for example, 500000).

NI-XNET CAN hardware currently accepts the following numeric baud rates: 33333, 40000, 50000, 62500, 80000, 83333, 100000, 125000, 160000, 200000, 250000, 400000, 500000, 800000, and 1000000.

Note The 33333 baud rate is supported with single-wire transceivers only.

Note Baud rates greater than 125000 are supported with high-speed transceivers only.

When the upper nibble is set to 0x8 (that is, 0x80000000), the remaining bits provide fields for more custom CAN communication baud rate programming. Additionally, if the upper nibble is set to 0xC (that is, 0xC0000000), the remaining bits provide fields for higher-precision custom CAN communication baud rate programming. The higher-precision

bit timings facilitate connectivity to a CAN FD cluster. The baud rate models are shown in the following table:

	31..28	27..26	25..24	23	22..20	19..16	15..14	13..12	11..8	7..4	3..0
Normal	b0000	Baud Rate (33.3 k–1 M)									
Custom	b1000	Res	SJW (0–3)	TSEG2 (0–7)		TSEG1 (1–15)	Res	Tq (125–0x3200)			
High Precision	b1100	SJW (0–15)		TSEG2 (0–15)		TSEG1 (1–63)		Tq (25–0x3200)			

The baud rate format in advanced mode is 0x8ABCDDDD, where A, B, C, and DDDD are defined as follows:

- (Re-)Synchronization Jump Width (SJW)
  - Valid programmed values are 0–3 in normal custom mode and 0–15 in high-precision custom mode.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time Segment 2 (TSEG2), which is the time segment after the sample point
  - Valid programmed values are 0–7 in normal custom mode and 0–15 in high-precision custom mode.
  - This is the Phase_Seg2 time from ISO 11898–1, 12.4.1 *Bit Encoding/Decoding*.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time Segment 1 (TSEG1), which is the time segment before the sample point
  - Valid programmed values are 1–0xF (1–15 decimal) in normal custom mode and 1–0x3F (1–63 decimal) in high-precision custom mode.
  - This is the combination of the Prop_Seg and Phase_Seg1 time from ISO 11898–1, 12.4.1 *Bit Encoding/Decoding*.
  - The actual hardware interpretation of this value is one more than the programmed value.
- Time quantum (Tq), which is used to program the baud rate prescaler
  - Valid programmed values are 125–12800, in increments of 0x7D (125 decimal) ns for normal custom mode and 25–12800, in increments of 0x19 (25 decimal) ns for high-precision custom mode.
  - This is the time quantum from ISO 11898–1, 12.4.1 *Bit Encoding/Decoding*.

An advanced baud rate example is 0x8014007D. This example breaks down into the following values:

- SJW = 0x0 (0x01 in hardware, due to the + 1)
- TSEG2 = 0x1 (0x02 in hardware, due to the + 1)
- TSEG 1 = 0x4 (0x05 in hardware, due to the + 1)
- Tq = 0x7D (125 ns in hardware)

Each time quanta is 125 ns. From IS0 11898–1, 12.4.1.2 *Programming of Bit Time*, the nominal time segments length is Sync_Seg(Fixed at 1) + (Prop_Seg + Phase_Seg1)(B) + Phase_Seg2(C) = 1 + 2 + 5 = 8. So, the total time for a bit in this example is 8 * 125 ns = 1000 ns = 1 µs. A 1 µs bit time is equivalent to a 1 MHz baud rate.

LIN

When the upper nibble (0xF0000000) is clear, you can set only baud rates within the LIN-specified range (2400 to 20000) for the interface.

When the upper nibble is set to 0x8 (0x80000000), no check for baud rate within LIN-specified range is performed, and the lowest 16 bits of the value may contain the custom baud rate. Any custom value higher than 65535 is masked to a 16-bit value. As with the noncustom values, the interface internally calculates the appropriate divisor values to program into its UART. Because the interface uses the Atmel ATA6620 LIN transceiver, which is guaranteed to operate within the LIN 2.0 specification limits, there are some special considerations when programming custom baud rates for LIN:

- The ATA6620 transceiver incorporates a TX dominant timeout function to prevent a faulty device it is built into from holding the LIN dominant indefinitely. If the TX line into the transceiver is held in the dominant state for too long, the transceiver switches its driver to the recessive state. This places a limit on how long the break field of a LIN header transmitted by the interface may be, and thus limits the lowest baud rate that may be set. At the point the baud rate or break length is set for the interface, it internally uses the baud rate bit time and break length settings to calculate the resulting break duration, and returns an error if that duration would be long enough to trigger the TX dominant timeout.
- At the other end of the baud range, the ATA6620 is specified to work up to 20000 baud. While the custom bit allows rates higher than that to be programmed, the transceiver behavior operating above that rate is not guaranteed.

Interface:Echo Transmit?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfEchoTx`

Description

The Interface:Echo Transmit? property determines whether Frame Input or Signal Input sessions contain frames that the interface transmits.

When this property is true, and a frame transmit is complete for an Output session, the frame is echoed to the Input session. Frame Input sessions can use the Flags field to differentiate frames received from the bus and frames the interface transmits. When using `nxReadFrame` with the raw frame format, you can parse the Flags field manually by reviewing the *Raw Frame Format* section. Signal Input sessions cannot differentiate the origin of the incoming data.

Note Echoed frames are placed into the input sessions only after the frame transmit is complete. If there are bus problems (for example, no listener) such that the frame did not transmit, the frame is not received.

Interface:Output Stream List

Data Type	Direction	Required?	Default
nxDatabaseRef_t[]	Read/Write	No	Empty Array

Property Class

XNET Session

Property ID

`nxPropSession_IntfOutStrmList`

Description

Note Only CAN and LIN interfaces currently support this property.

The Output Stream List property provides a list of frames for use with the replay feature (Interface:Output Stream Timing property set to `nxOutStrmTimng_ReplayExclusive` or `nxOutStrmTimng_ReplayInclusive`). In Replay Exclusive mode, the hardware transmits only frames that do not appear in the list. In Replay Inclusive mode, the hardware transmits only frames that appear in the list. For a LIN interface, the header of each frame written to stream output is transmitted, and the Exclusive or Inclusive mode controls the response transmission. Using these modes, you can either emulate an ECU (Replay Inclusive, where the list contains the frames the ECU transmits) or test an ECU (Replay Exclusive, where the list contains the frames the ECU transmits), or some other combination.

This property's data type is an array of database handles to frames. If you are not using a database file or prefer to specify the frames using CAN arbitration IDs or LIN unprotected IDs, you can use Interface:Output Stream List By ID instead of this property.

Interface:Output Stream List By ID

Data Type	Direction	Required?	Default
u32[]	Read/Write	No	Empty Array

Property Class

XNET Session

Property ID

`nxPropSession_IntfOutStrmListById`

Description

Note Only CAN and LIN interfaces currently support this property.

The Output Stream List By ID property provides a list of frames for use with the replay feature (Interface:Output Stream Timing property set to `nxOutStrmTimng_ReplayExclusive` or `nxOutStrmTimng_ReplayInclusive`).

This property serves the same purpose as Interface:Output Stream List, in that it provides a list of frames for replay filtering. This property provides an alternate format for you to specify the frames by their CAN arbitration ID or LIN unprotected ID. The property's data type is an array of unsigned 32-bit integer (u32). Each integer represents a CAN or LIN frame's identifier, using the same encoding as the *Raw Frame Format*.

Within each CAN frame ID value, 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. LIN frame ID values may be within the range of possible LIN IDs (0–63).

Interface:Output Stream Timing

Data Type	Direction	Required?	Default
u32	Read/Write	No	Immediate

Property Class

XNET Session

Property ID

`nxPropSession_IntfOutStrmTimng`

Description

Note Only CAN and LIN interfaces currently support this property.

The Output Stream Timing property configures how the hardware transmits frames queued using a Frame Output Stream session. The following table lists the accepted values:

Enumeration	Value
`nxOutStrmTimng_Immediate`	0
`nxOutStrmTimng_ReplayExclusive`	1
`nxOutStrmTimng_ReplayInclusive`	2

When you configure this property to be `nxOutStrmTimng_Immediate`, frames are dequeued from the queue and transmitted immediately to the bus. The hardware transmits all frames in the queue as fast as possible.

When you configure this property as `nxOutStrmTimng_ReplayExclusive` or `nxOutStrmTimng_ReplayInclusive`, the hardware is placed into a Replay mode. In this mode, the hardware evaluates the frame timestamps and attempts to maintain the original transmission times as the timestamp stored in the frame indicates. The actual transmission time is based on the relative time difference between the first dequeued frame and the time contained in the dequeued frame.

When in one of the replay modes, you can use the Interface:Output Stream List property to supply a list. In Replay Exclusive mode, the hardware transmits only frames that do not appear in the list. In Replay Inclusive mode, the hardware transmits only frames that appear in the list. Using these modes, you can either emulate an ECU (Replay Inclusive, where the list contains the frames the ECU transmits) or test an ECU (Replay Exclusive, where the list

contains the frames the ECU transmits), or some other combination. You can replay all frames by using Replay Exclusive mode without setting any list.

Special Considerations for LIN

Only LIN interface as Master supports stream output. You do not need to set the interface explicitly to Master if you want to use stream output. Just create a stream output session, and the driver automatically sets the interface to Master at interface start.

You can use immediate mode to transmit a header or full frame. You can transmit only the header for a frame by writing the frame to stream output with the desired ID and an empty data payload. You can transmit a full frame by writing the frame to stream output with the desired ID and data payload. If you write a full frame for ID *n* to stream output, and you have created a frame output session for frame with ID *n*, the stream output data takes priority (the stream output frame data is transmitted and not the frame output data). If you write a full frame to stream output, but the frame has not been defined in the database, the frame transmits with Enhanced checksum. To control the checksum type transmitted for a frame, you first must create the frame in the database and assign it to an ECU using the LIN specification you desire (the specification number determines the checksum type). You then must create a frame output object to transmit the response for the frame, and use stream output to transmit the header. Similarly, to transmit *n* corrupted checksums for a frame, you first must create a frame object in the database, create a frame output session for it, set the transmit *n* corrupted checksums property, and then use stream output to transmit the header.

Regarding event-triggered frame handling for immediate mode, if the hardware can determine that an ID is for an event-triggered frame, which means an event-triggered frame has been defined for the ID in the database, the frame is processed as if it were in an event-triggered slot in a schedule. If you write a full frame with event-triggered ID, the full frame is transmitted. If there is no collision, the next stream output frame is processed. If there is a collision, the hardware executes the collision-resolving schedule. The hardware retransmits the frame response at the corresponding slot time in the collision resolving schedule. If you write a header frame with an event-triggered ID and there is no collision, the next stream output frame is processed. If there is a collision, the hardware executes the collision-resolving schedule.

You can mix use of the hardware scheduler and stream output immediate mode. Basically, the hardware treats each stream output frame as a separate run-once schedule containing a single slot for the frame. Transmission of a stream output frame may interrupt a run-continuous schedule, but may not interrupt a run-once schedule. Transmission of stream output frames is interleaved with run-continuous schedule slot executions, depending on the application timing of writes to stream output. Stream output is prioritized to the equivalent of the lowest priority level for a run-once schedule. If you write one or more run-once schedules with higher-than-lowest priority and write frames to stream output, all the run-once schedules are executed before stream output transmits anything. If you write one or more run-once schedules with the lowest priority and write frames to stream output, the run-once schedules

execute in the order you wrote them, and are interleaved with stream output frames, depending on the application timing of writes to stream output and writes of run-once schedule changes.

In contrast to the immediate mode, neither replay mode allows for the concurrent use of the hardware scheduler, and an error is reported if you attempt to do so. Event-triggered frame handling is different for the replay modes. If the hardware can determine that an ID is for an event-triggered frame, which means an event-triggered frame has been defined for the ID in the database, the frame is transmitted as if it were being transmitted during the collision-resolving schedule for the event triggered frame. The full frame is transmitted with the Data[0] value (the underlying unconditional frame ID), copied into the header ID. If a frame cannot be found in the database, it is transmitted with Enhanced checksum. Otherwise, it is transmitted with the checksum type defined in the database.

The reply modes provide an easy means to replay headers only, full frames only, or some mix of the two. For either replay mode, the header for each frame is always transmitted and the slot delay is preserved. For replay inclusive, if you want only to replay headers, leave the Interface:Output Stream List property empty. To replay some of the responses, add their frames to Interface:Output Stream List. For frames that are not in Interface:Output Stream List, you are free to create frame output objects for them, for which you can change the checksum type or transmit corrupted checksums.

There is another consideration for the replay of diagnostic slave response frames. Because the master always transmits only the diagnostic slave response header, and a slave transmits the response if its NAD matches the one transmitted in the preceding master request frame, an array of frames for replay might include multiple slave response frames (each having the same slave response header ID) transmitted by different slaves (each having a different NAD value in the data payload). If you are using inclusive mode, you can choose not to replay any slave response frames by not including the slave response frame in Interface:Output Stream List. You can choose to replay some or all of the slave response frames by first including the slave response frame in Interface:Output Stream List, then including the NAD values for the slave responses you want to play back, in Interface:LIN:Output Stream Slave Response List By NAD. In this way, you have complete control over which slave responses are replayed (which diagnostic slaves you emulate). Replay of a diagnostic master request frame is handled like replay of any other frame; the header is always transmitted. Using the inclusive mode as an example, the response may or may not be transmitted depending on whether or not the master request frame is in Interface:Output Stream List.

Runtime Behavior

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 whose timestamp goes backwards relative to the previous timestamp, but still is 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.

When a frame with an nxFrameType_Special_Delay 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.

When a frame with an nxFrameType_Special_StartTrigger frame type is received, the hardware treats this frame as a new first frame and uses the absolute time associated with this frame as the new start time. Subsequent frames are compared to this new start time to determine the transmission time. Using a Start Trigger is especially useful when synchronizing with data acquisition products so that you can replay the first frame at the correct time relative to the start trigger for accurate synchronized replay.

Restrictions on Other Sessions

When you use Immediate mode, there are no restrictions on frames that you use in other sessions.

When you use Replay Inclusive mode, you can create output sessions that use frames that do not appear in the Interface:Output Stream List property. Attempting to create an output session that uses a frame from the Interface:Output Stream List property results in an error. Input sessions have no restrictions.

When you use Replay Exclusive mode, you cannot create any other output sessions. Attempting to create an output session returns an error. Input sessions have no restrictions.

Interface:Start Trigger Frames to Input Stream?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfStartTrigToInStrm`

Description

The `nxPropSession_IntfStartTrigToInStrm` property configures the hardware to place a start trigger frame into the Stream Input queue after it is generated. A Start Trigger frame is generated when the interface is started. The interface start process is described in *Interface Transitions*. For more information about the start trigger frame, refer to *Special Frames*.

The start trigger frame is especially useful if you plan to log and replay CAN data.

Interface:Bus Error Frames to Input Stream?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Property ID

`nxPropSession_IntfBusErrToInStrm`

Description

Note Only CAN and LIN interfaces currently support this property.

The `nxPropSession_IntfBusErrToInStrm` property configures the hardware to place a CAN or LIN bus error frame into the Stream Input queue after it is generated. A bus error frame is generated when the hardware detects a bus error. For more information about the bus error frame, refer to *Special Frames*.

Session:Application Protocol

Data Type	Direction	Required?	Default
u32	Read Only	N/A	None

Property Class

XNET Session

Short Name

`nxPropSession_ApplicationProtocol`

Description

This property returns the application protocol that the session uses.

The database used with `nxCreateSession` determines the application protocol.

The values (enumeration) for this property are:

0 nxAppProtocol_None

1 nxAppProtocol_J1939

SAE J1939:ECU

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Write Only	No	Unassigned

Property Class

XNET Session

Short Name

`nxPropSession_J1939ECU`

Description

Note This property applies to only the CAN J1939 application protocol. The database from which the ECU reference is passed in this property must be open when this property is called, because database references are valid only when the database is open.

This property assigns a database ECU to a J1939 session. Setting this property changes the node address and J1939 64-bit ECU name of the session to the values stored in the database ECU object. Changing the node address starts an address claiming procedure, as described in the SAE J1939:Node Address property.

You can assign the same ECU to multiple sessions running on the same CAN interface (for example, CAN1). All sessions with the same assigned ECU represent one J1939 node.

If multiple sessions have assigned the same ECU, setting the SAE J1939:Node Address property in one session changes the address in all sessions with the same assigned ECU running on the same CAN interface.

For more information, refer to the SAE J1939:Node Address property.

SAE J1939:ECU Busy

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Session

Short Name

`nxPropSession_J1939ECUBusy`

Description

Note This property applies to only the CAN J1939 application protocol.

Busy is a special ECU state defined in the SAE J1939 standard. A busy ECU receives subsequent RTS messages while handling a previous RTS/CTS communication.

If the ECU cannot respond immediately to an RTS request, the ECU may send CTS Hold messages. In this case, the originator receives information about the busy state and waits until the ECU leaves the busy state. (That is, the ECU no longer sends CTS Hold messages and sends the first CTS message with the requested data.)

Use the ECU Busy property to simulate this ECU behavior. If a busy XNET ECU receives a CTS message, it sends CTS Hold messages instead of CTS data messages immediately. Afterward, if clearing the busy property, the XNET ECU resumes handling the transport protocol starting with CTS data messages, as the originator expects.

SAE J1939:Hold Time Th

Data Type	Direction	Required?	Default
f64	Read/Write	No	0.5 s

Property Class

XNET Session

Short Name

`nxPropSession_J1939HoldTimeTh`

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the Hold Time Timeout value at the responder node. The value is the maximum time between a TP.CM_CTS hold message and the next TP.CM_CTS message, in seconds.

This property is related to handling the transport protocol.

SAE J1939:Maximum Repeat CTS

Data Type	Direction	Required?	Default
u32	Read/Write	No	2

Property Class

XNET Session

Short Name

`nxPropSession_J1939MaxRepeatCTS`

Description

Note This property applies to only the CAN J1939 application protocol.

This property limits the number of requests for retransmission of data packet(s) using the TP.CM_CTS message.

This property is related to handling the transport protocol.

SAE J1939:Node Address

Data Type	Direction	Required?	Default
u32	Read/Write	No	Null (254)

Property Class

XNET Session

Short Name

`nxPropSession_J1939Address`

Description

Note This property applies to only the CAN J1939 application protocol.

This property changes the node address of a J1939 session by starting an address claiming procedure. After setting this property to a valid value (≤ 253), reading the property returns the null address (254) until the address is granted. Poll the property and wait until the address gets to a valid value again before starting to write. Refer to the NI-XNET examples that demonstrate this procedure.

The node address value determines the source address in a transmitting session or a destination address in a receiving session. The source address in the extended frame identifier is overwritten with the node address of the session before transmitting.

A session with a null (254) or global address (255) receives all messages sent on the bus, but cannot transmit messages. A session with an assigned address of less than 254 receives only messages sent to this address or global messages, but not messages sent to other nodes. This session also can transmit messages.

In NI-XNET, you can assign the same J1939 node address to multiple sessions running on the same interface (for example, CAN1). Those sessions represent one J1939 node. By assigning different J1939 node addresses to multiple sessions running on the same interface, you also can create multiple nodes on the same interface.

If a J1939 ECU is assigned to multiple sessions, changing the address in one session also changes the address in all other sessions with the same assigned ECU.

For more information, refer to the SAE J1939:ECU property.

SAE J1939:NodeName

Data Type	Direction	Required?	Default
u64	Read/Write	Yes	0

Property Class

XNET Session

Short Name

`nxPropSession_J1939Name`

Description

Note This property applies to only the CAN J1939 application protocol.

This property changes the name value of a J1939 session. The name is an unsigned 64-bit integer value. Beside the SAE J1939:Node Address property, the value is specific to the ECU you want to emulate using the session. That means the session can act as if it were the real-world ECU, using the identical address and name value.

The name value is used within the address claiming procedure. If the ECU (session) wants to claim its address, it sends out an address claiming message. That message contains the ECU address and the name value of the current session's ECU. If there is another ECU within the network with an identical address but lower name value, the current session loses its address. In this case, the session cannot send out further messages, and all addressed messages using the previous address of the current session are addressed to another ECU within the network.

The most significant bit (bit 63) in the Node Name defines the ECU's arbitrary address capability (bit 63 = 1 means it is arbitrary address capable). If the node cannot use the assigned address, it automatically tries to claim another random value between 128 and 247 until it is successful.

If multiple sessions are assigned the same ECU, setting the SAE J1939.NodeName property in one session changes the address in all sessions with the same assigned ECU running on the same CAN interface.

The name value has multiple bit fields, as described in SAE J1939-81 (Network Management). A single 64-bit value represents the name value within XNET.

For more information, refer to the SAE J1939:Node Address property.

SAE J1939:Number of Packets Received

Data Type	Direction	Required?	Default
u32	Read/Write	No	255

Property Class

XNET Session

Short Name

`nxPropSession_J1939NumPacketsRecv`

Description

Note This property applies to only the CAN J1939 application protocol.

This property changes the maximum number of data packet(s) that can be received in one block at the responder node.

This property is related to handling the transport protocol.

SAE J1939:Number of Packets Response

Data Type	Direction	Required?	Default
u32	Read/Write	No	255

Property Class

XNET Session

Short Name

`nxPropSession_J1939NumPacketsResp`

Description

Note This property applies to only the CAN J1939 application protocol.

This property limits the maximum number of packets in a response. This allows the originator node to limit the number of packets in the TP.CM_CTS message. When the responder complies with this limit, it ensures the sender always can retransmit packets that the responder may not have received.

This property is related to handling the transport protocol.

SAE J1939:Response Time Tr_GD

Data Type	Direction	Required?	Default
f64	Read/Write	No	0.05 s

Property Class

XNET Session

Short Name

`nxPropSession_J1939ResponseTimeTrGD`

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the Device Response Time for global destination messages (TP.CM_BAM messages). The value is the maximum delay between sending two TP.CM_BAM messages, in seconds. The recommended range is 0.05–200 s.

This property is related to handling the transport protocol.

SAE J1939:Response Time Tr_SD

Data Type	Direction	Required?	Default
f64	Read/Write	No	0.05 s

Property Class

XNET Session

Short Name

`nxPropSession_J1939ResponseTimeTrSD`

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the Device Response Time value for specific destination messages (TP.CM_RTS/CTS messages). The value is the maximum time between receiving a message and sending the response message, in seconds. The recommended range is 0.05–0.200 s.

This property is related to handling the transport protocol.

SAE J1939:Timeout T1

Data Type	Direction	Required?	Default
f64	Read/Write	No	0.75 s

Property Class

XNET Session

Short Name

`nxPropSession_J1939TimeoutT1`

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the timeout T1 value for the responder node. The value is the maximum gap between two received TP.DT messages in seconds.

This property is related to handling the transport protocol.

SAE J1939:Timeout T2

Data Type	Direction	Required?	Default
f64	Read/Write	No	1.25 s

Property Class

XNET Session

Short Name

`nxPropSession_J1939TimeoutT2`

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the timeout T2 value at the responder node. This value is the maximum gap between sending out the TP.CM_CTS message and receiving the next TP.DT message, in seconds.

This property is related to handling the transport protocol.

SAE J1939:Timeout T3

Data Type	Direction	Required?	Default
f64	Read/Write	No	1.25 s

Property Class

XNET Session

Short Name

`nxPropSession_J1939TimeoutT3`

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the timeout T3 value at the originator node. This value is the maximum gap between sending out a TP.CM_RTS message or the last TP.DT message and receiving the TP.CM_CTS response, in seconds.

This property is related to handling the transport protocol.

SAE J1939:Timeout T4

Data Type	Direction	Required?	Default
f64	Read/Write	No	1.05 s

Property Class

XNET Session

Short Name

`nxPropSession_J1939TimeoutT4`

Description



Note This property applies to only the CAN J1939 application protocol.

This property changes the timeout T4 value at the originator node. This value is the maximum gap between the TP.CM_CTS hold message and the next TP.CM_CTS message, in seconds.

This property is related to handling the transport protocol.

Frame Properties

This section includes the frame-specific properties in the session property node.

CAN Frame Properties

This category includes CAN-specific frame properties.

Frame:CAN:Start Time Offset

Data Type	Direction	Required?	Default
double	Write Only	No	–1

Property Class

XNET Session

Property ID

`nxPropSessionSub_CANStartTimeOff`

Description

Use this property to configure the amount of time that must elapse between the session being started and the time that the first frame is transmitted across the bus. This is different than the cyclic rate, which determines the time between subsequent frame transmissions.

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

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

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

Note This property affects the active frame object in the session. Review the `nxSetSubProperty` function to learn more about setting a property on an active frame.

Frame:CAN:Transmit Time

Data Type	Direction	Required?	Default
double	Write Only	No	From Database

Property Class

XNET Session

Property ID

`nxPropSessionSub_CANTxTime`

Description

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

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



Note This property affects the active frame object in the session. Review the `nxSetSubProperty` function to learn more about setting a property on an active frame.



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

Frame:LIN:Transmit N Corrupted Checksums

Data Type	Direction	Required?	Default
u32	Write Only	No	0

Property Class

XNET Session

Property ID

`nxPropSessionSub_LINTxNCorruptedChksums`

Description

When set to a nonzero value, this property causes the next *N* number of checksums to be corrupted. The checksum is corrupted by negating the value calculated per the database; (`EnhancedValue * -1`) or (`ClassicValue * -1`). This property is valid only for output sessions. If the frame is transmitted in an unconditional or sporadic schedule slot, *N* is always decremented for each frame transmission. If the frame is transmitted in an event-triggered slot and a collision occurs, *N* is not decremented. In that case, *N* is decremented only when the collision resolving schedule is executed and the frame is successfully transmitted. If the frame is the only one to transmit in the event-triggered slot (no collision), *N* is decremented at event-triggered slot time.

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



Note This property affects the active frame object in the session. Review the `nxSetSubProperty` property to learn more about setting a property on an active frame.

Frame:Skip N Cyclic Frames

Data Type	Direction	Required?	Default
u32	Write Only	No	0

Property Class

XNET Session

Property ID

`nxPropSessionSub_SkipNCyclicFrames`

Description

Note Only CAN interfaces currently support this property.

When set to a nonzero value, this property causes the next *N* cyclic frames to be skipped. When the frame's transmission time arrives and the skip count is nonzero, a frame value is dequeued (if this is not a single-point session), and the skip count is decremented, but the frame actually is not transmitted across the bus. When the skip count decrements to zero, subsequent cyclic transmissions resume. This property is valid only for output sessions and frames with cyclic timing (that is, not event-based frames).

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

Note This property affects the active frame object in the session. Review the `nxSetSubProperty` property to learn more about setting a property on an active frame.

Auto Start?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	True

Property Class

XNET Session

Property ID

`nxPropSession_AutoStart`

Description

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

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

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

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

ClusterName

Data Type	Direction	Required?	Default
cstr	Read Only	N/A	N/A

Property Class

XNET Session

Property ID

`nxPropSession_ClusterName`

Description

This property returns the cluster (network) used with `nxCreateSession`.

DatabaseName

Data Type	Direction	Required?	Default
cstr	Read Only	N/A	N/A

Property Class

XNET Session

Property ID

`nxPropSession_DatabaseName`

Description

This property returns the database used with `nxCreateSession`.

List

Data Type	Direction	Required?	Default
cstr	Read Only	N/A	N/A

Property Class

XNET Session

Property ID

`nxPropSession_List`

Description

This property returns a comma separated list of frames or signals in the session.

For a Frame Input or Frame Output session, this property returns a list of frames. For a Signal Input/Output session, it returns the list of signals.

Mode

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET Session

Property ID

`nxPropSession_Mode`

Description

This property returns the session mode (ring). You provided this mode when you created the session. For more information, refer to *Session Modes*.

Number in List

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET Session

Property ID

`nxPropSession_NumInList`

Description

This property returns the number of frames or signals in the session's list. This is a quick way to get the size of the List property.

Number of Values Pending

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET Session

Property ID

`nxPropSession_NumPend`

Description

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

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

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

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

The largest possible frames sizes are:

- CAN FD: 64 byte payload.
- FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay:Payload Length Maximum property provides this value.

Number of Values Unused

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET Session

Property ID

`nxPropSession_NumUnused`

Description

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

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

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

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

The largest possible frames sizes are:

- CAN FD: 64 byte payload.
- FlexRay: The higher value of the frame size in the static segment and the maximum frame size in the dynamic segment. The XNET Cluster FlexRay:Payload Length Maximum property provides this value.

Payload Length Maximum

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET Session

Property ID

`nxPropSession_PayldLenMax`

Description

This property returns the maximum payload length of all frames in this session, expressed as bytes (0–254).

This property does not apply to Signal sessions (only Frame sessions).

For CAN Stream (Input and Output), this property depends on the XNET Cluster CAN:I/O Mode property. If the I/O mode is CAN, this property is 8 bytes. If the I/O mode is nxCANioMode_CAN_FD or nxCANioMode_CAN_FD_BRS, this property is 64 bytes.

For LIN Stream (Input and Output), this property always is 8 bytes. For FlexRay Stream (Input and Output), this property is the same as the XNET Cluster FlexRay:Payload Length Maximum property value. For Queued and Single-Point (Input and Output), this is the maximum payload of all frames specified in the List property.

Protocol

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET Session

Property ID

`nxPropSession_Protocol`

Description

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

The values (enumeration) for this property are:

0 CAN

1 FlexRay

2 LIN

Queue Size

Data Type	Direction	Required?	Default
u32	Read/Write	No	Refer to Description

Property Class

XNET Session

Property ID

`nxPropSession_QueueSize`

Description

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

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

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

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

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

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

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

For Signal I/O Waveform sessions, you can use signals from more than one frame. Within the implementation, each frame uses a dedicated queue. The Queue Size property does not

represent the memory in these queues, but rather the amount of time stored. The default queue allocations store Application Time worth of resampled signal values. If you read the default Queue Size property for a Signal I/O Waveform session, it returns Application Time multiplied by the time Resample Rate. If you write the Queue Size property for a Signal I/O Waveform session, that value is translated from a number of samples to a time, and that time is used to allocate memory for each queue.

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

Default Value

You calculate the default queue size based on the following assumptions:

- **Application Time**: The time between calls to the appropriate `nxRead`/`nxWrite` function in your application.
- **Frame Time**: The time between frames on the network for this session.

The following pseudo code describes the default queue size formula:

```
if (session is Signal I/O Waveform)
    Queue_Size = (Application_Time * Resample_Rate);
else
    Queue_Size = (Application_Time / Frame_Time);
if (Queue_Size < 64)
    Queue_Size = 64;
if (session mode is Frame I/O)
    Queue_Size = Queue_Size * Frame_Size;
```

For Signal I/O Waveform sessions, the initial formula calculates the number of resampled values that occur within the Application Time. This is done by multiplying Application Time by the XNET Session Resample Rate property.

For all other

, the initial formula divides Application Time by Frame Time.

The minimum for this formula is 64. This minimum ensures that you can read or write at least 64 elements. If you need to read or write more elements for a slow frame, you can set the Queue Size property to a larger number than the default. If you set a large Queue Size, this may limit the maximum number of frames you can use in all sessions.

For Frame I/O sessions, this formula result is multiplied by each frame value size to obtain a queue size in bytes.

For Signal I/O sessions, this formula result is used directly for the queue size property to provide the number of signal values for the appropriate `nxRead` or `nxWrite` function. Within

the Signal I/O session, the memory allocated for the queue incorporates frame sizes, because the signal values are mapped to/from frame values internally.

Application Time

The target in which your application runs determines the Application Time:

- **Windows**: 400 ms (0.4 s)
- **Real-Time (RT)**: 100 ms (0.1 s)

This works under the assumption that for Windows, more memory is available for input queues, and you have limited control over the application timing. RT targets typically have less available memory, but your application has better control over application timing.

Frame Time

Frame Time is calculated differently for Frame I/O Stream sessions compared to other modes. For Frame I/O Stream, you access all frames in the network (cluster), so the Frame Time is related to the average bus load on your network. For other modes, you access specific frames only, so the Frame Time is obtained from database properties for those frames.

The Frame Time used for the default varies by session mode and protocol, as described below.

CAN, Frame I/O Stream

Frame Time is 100 µs (0.0001 s).

This time assumes a baud rate of 1 Mbps, with frames back to back (100 percent busload).

For CAN sessions created for a standard CAN bus, the Frame Size is 24 bytes. For CAN sessions created for a CAN FD Bus (the cluster I/O mode is CAN FD or CAN FD+BRS), the frame size can vary up to 64 bytes. However, the default queue size is based on the 24-byte frame time. When connecting to a CAN FD bus, you may need to adjust this size as necessary.

When you create an application to stress test NI-XNET performance, it is possible to generate CAN frames faster than 100 µs. For this application, you must set the queue size to larger than the default.

FlexRay, Frame I/O Stream

Frame Time is 20 µs (0.00002 s).

This time assumes a baud rate of 10 Mbps, with a cycle containing static slots only (no minislots or NIT), and frames on channel A only.

Small frames at a fast rate require a larger queue size than large frames at a slow rate. Therefore, this default assumes static slots with 4 bytes, for a Frame Size of 24 bytes.

When you create an application to stress test NI-XNET performance, it is possible to generate FlexRay frames faster than 20 µs. For this application, you must set the queue size to larger than the default.

LIN, Frame I/O Stream

Frame Time is 2 ms (0.002 s).

This time assumes a baud rate of 20 kbps, with 1 byte frames back to back (100 percent busload).

For all LIN sessions, Frame Size is 24 bytes.

CAN, Other Modes

For Frame I/O Queued, Signal I/O XY, and Signal I/O Waveform, the Frame Time is different for each frame in the session (or frame within which signals are contained).

For CAN frames, Frame Time is the frame property CAN Transmit Time, which specifies the time between successive frames (in floating-point seconds).

If the frame's CAN Transmit Time is 0, this implies the possibility of back-to-back frames on the network. Nevertheless, this back-to-back traffic typically occurs in bursts, and the average rate over a long period of time is relatively slow. To keep the default queue size to a reasonable value, when CAN Transmit Time is 0, the formula uses a Frame Time of 50 ms (0.05 s).

For CAN sessions using a standard CAN cluster, the frame size is 24 bytes. For CAN sessions using a CAN FD cluster, the frame size may differ for each frame in the session. Each frame size is obtained from its XNET Frame Payload Length property in the database.

FlexRay, Other Modes

For Frame I/O Queued, Signal I/O XY, and Signal I/O Waveform, the Frame Time is different for each frame in the session (or frame within which signals are contained).

For FlexRay frames, Frame Time is the time between successive frames (in floating-point seconds), calculated from cluster and frame properties. For example, if a cluster Cycle (cycle duration) is 10000 µs, and the frame Base Cycle is 0 and Cycle Repetition is 1, the frame's Transmit Time is 0.01 (10 ms).

For these session modes, the Frame Size is different for each frame in the session. Each Frame Size is obtained from its XNET Frame Payload Length property in the database.

LIN, Other Modes

For LIN frames, Frame Time is a property of the schedule running in the LIN master node. It is assumed that the Frame Time for a single frame always is larger than 8 ms, so that the default queue size is set to 64 frames throughout.

For all LIN sessions, Frame Size is 24 bytes.

Examples

The following table lists example session configurations and the resulting default queue sizes.

Session Configuration	Default Queue Size	Formula
Frame Input Stream, CAN, Windows	96000	(0.4 / 0.0001) = 4000; 4000 × 24 bytes
Frame Output Stream, CAN, Windows	96000	(0.4 / 0.0001) = 4000; 4000 × 24 bytes; output is always same as input
Frame Input Stream, FlexRay, Windows	480000	(0.4 / 0.00002) = 20000; 20000 × 24 bytes
Frame Input Stream, CAN, RT	24000	(0.1 / 0.0001) = 1000; 1000 × 24 bytes
Frame Input Stream, FlexRay, RT	120000	(0.1 / 0.00002) = 5000; 5000 × 24 bytes
Frame Input Queued, CAN, Transmit Time 0.0, Windows	1536*	(0.4 / 0.05) = 8; Transmit Time 0 uses Frame Time 50 ms; use minimum of 64 frames (64 × 24)
Frame Input Queued, CAN, Transmit Time 0.0005, Windows	19200*	(0.4 / 0.0005) = 800; 800 × 24 bytes
Frame Input Queued, CAN, Transmit Time 1.0 (1 s), Windows	1536*	(0.4 / 1.0) = 0.4; use minimum of 64 frames (64 × 24)
Frame Input Queued, FlexRay, every 2 ms cycle, payload length 4, Windows	4800	(0.4 / 0.002) = 200; 200 × 24 bytes
Frame Input Queued, FlexRay, every 2 ms cycle, payload length 16, RT	2048	(0.1 / 0.002) = 50, use minimum of 64; payload length 16 requires 32 bytes; 64 × 32 bytes

Session Configuration	Default Queue Size	Formula
Signal Input XY, two CAN frames, Transmit Time 0.0 and 0.0005, Windows	64* and 800* (read as 800)	(0.4 / 0.05) = 8, use minimum of 64; (0.4 / 0.0005) = 800; expressed as signal values
Signal Output XY, two CAN frames, Transmit Time 0.0 and 0.0005, Windows	64* and 800* (read as 64)	(0.4 / 0.05) = 8, use minimum of 64; (0.4 / 0.0005) = 800; expressed as signal values
Signal Output Waveform, two CAN frames, 1 ms and 400 ms, resample rate 1000 Hz, Windows	400*	Memory allocation is 400 and 64 frames to provide 0.4 sec of storage, queue size represents number of samples, or (0.4 × 1000.0)
Signal Output Waveform, two CAN frames, 1 ms and 400 ms, resample rate 1000 Hz, Windows	400*	Memory allocation is 400 and 64 frames to provide 0.4 sec of storage, queue size represents number of samples, or (0.4 × 1000.0)

* For a CAN FD cluster, the default queue size is based on the frame's database payload length, which may be larger than 24 bytes (up to 64 bytes).

Resample Rate

Data Type	Direction	Required?	Default
f64	Read/Write	No	1000.0 (Sample Every Millisecond)

Property Class

XNET Session

Property ID

`nxPropSession_ResampRate`

Description

Rate used to resample frame data to/from signal data in waveforms.

This property applies only when the session mode is Signal Input Waveform or Signal Output Waveform. This property is ignored for all other modes.

The data type is 64-bit floating point (DBL). The units are in Hertz (samples per second).

XNET Signal Properties

This section includes the XNET Signal properties.

Byte Order

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET Signal

Property ID

`nxPropSig_ByteOrdr`

Description

Signal byte order in the frame payload.

This property defines how signal bytes are ordered in the frame payload when the frame is loaded in memory.

- **Little Endian**: Higher significant signal bits are placed on higher byte addresses. In NI-CAN, this was called Intel Byte Order.

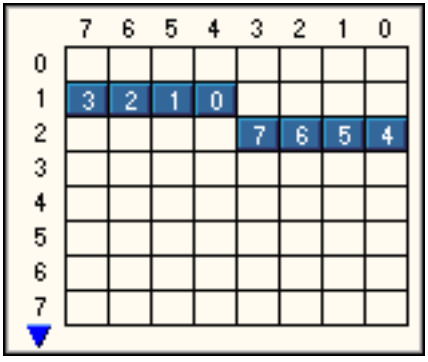


Figure 5-2. Little Endian Signal with Start Bit 12

- Big Endian: Higher significant signal bits are placed on lower byte addresses. In NI-CAN, this was called Motorola Byte Order.

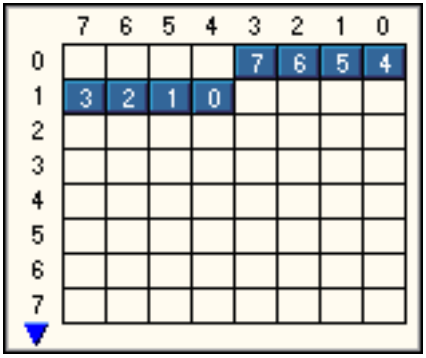


Figure 5-3. Big Endian Signal with Start Bit 12

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Comment

Data Type	Direction	Required?	Default
char *	Read/Write	No	Empty String

Property Class

XNET Signal

Property ID

`nxPropSig_Comment`

Description

Comment describing the signal object.

A comment is a string containing up to 65535 characters.

Configuration Status

Data Type	Direction	Required?	Default
i32	Read Only	No	N/A

Property Class

XNET Signal

Property ID

`nxPropSig_ConfigStatus`

Description

The signal object configuration status.

Configuration Status returns an NI-XNET error code. You can pass the value to the `nxStatusToString` error code input to convert the value to a text description of the configuration problem.

By default, incorrectly configured signals in the database are not returned from the XNET Frame Signals property because they cannot be used in the bus communication. You can change this behavior by setting the XNET Database ShowInvalidFromOpen? property to true. When a signal configuration status becomes invalid after the database is opened, the signal still is returned from the Signals property even if the ShowInvalidFromOpen? property is false.

Examples of invalid signal configuration:

- The signal is specified using bits outside the frame payload.
- The signal overlaps another signal in the frame. For example, two multiplexed signals with the same multiplexer value are using the same bit in the frame payload.
- The signal with integer data type (signed or unsigned) is specified with more than 52 bits. This is not allowed due to internal limitation of the double data type that NI-XNET uses for signal values.
- The frame containing the signal is invalid (for example, a CAN frame is defined with more than 8 payload bytes).

Data Type

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET Signal

Property ID

`nxPropSig_DataType`

Description

The signal data type.

This property determines how the bits of a signal in a frame must be interpreted to build a value.

- **Signed**: Signed integer with positive and negative values.
- **Unsigned**: Unsigned integer with no negative values.
- **IEEE Float**: Float value with 7 or 15 significant decimal digits (32 bit or 64 bit).

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.
- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Default Value

Data Type	Direction	Required?	Default
Double	Read/Write	No	0.0 (If Not in Database)

Property Class

XNET Signal

Property ID

`nxPropSig_Default`

Description

The signal default value, specified as scaled floating-point units.

The data type is 64-bit floating point (DBL).

The initial value of this property comes from the database. If the database does not provide a value, this property uses a default value of 0.0.

For all three signal output sessions, this property is used when a frame transmits prior to a call to `nxWrite`. The XNET Frame Default Payload property is used as the initial payload, then the default value of each signal is mapped into that payload using this property, and the result is used for the frame transmit.

For all three signal input sessions, this property is returned for each signal when `nxRead` is called prior to receiving the first frame.

For more information about when this property is used, refer to the discussion of `nxRead`/`nxWrite` for each session mode.

Frame

Data Type	Direction	Required?	Default
nxPropSig_FrameRef	Read Only	N/A	Parent Frame

Property Class

XNET Signal

Property ID

`nxPropSig_FrameRef`

Description

Reference to the signal parent frame.

This property returns the refnum to the signal parent frame. The parent frame is defined when the signal object is created. You cannot change it afterwards.

Maximum Value

Data Type	Direction	Required?	Default
Double	Read/Write	No	1000.0

Property Class

XNET Signal

Property ID

`nxPropSig_Max`

Description

The scaled signal value maximum.

`nxRead` and `nxWrite` do not limit the signal value to a maximum value. Use this database property to set the maximum value.

Minimum Value

Data Type	Direction	Required?	Default
Double	Read/Write	No	0.0

Property Class

XNET Signal

Property ID

`nxPropSig_Min`

Description

The scaled signal value minimum.

`nxRead` and `nxWrite` do not limit the signal value to a minimum value. Use this database property to set the minimum value.

Mux:Data Multiplexer?

Data Type	Direction	Required?	Default
Boolean	Read/Write	No	False

Property Class

XNET Signal

Property ID

`nxPropSig_MuxIsDataMux`

Description

This property defines the signal that is a multiplexer signal. A frame containing a multiplexer value is called a multiplexed frame.

A multiplexer defines an area within the frame to contain different information (dynamic signals) depending on the multiplexer signal value. Dynamic signals with a different multiplexer value (defined in a different subframe) can share bits in the frame payload. The multiplexer signal value determines which dynamic signals are transmitted in the given frame.

To define dynamic signals in the frame transmitted with a given multiplexer value, you first must create a subframe in this frame and set the multiplexer value in the subframe. Then you must create dynamic signals using `nxdbCreateObject` to create child signals of this subframe.

Multiplexer signals may not overlap other static or dynamic signals in the frame.

Dynamic signals may overlap other dynamic signals when they have a different multiplexer value.

A frame may contain only one multiplexer signal.

The multiplexer signal is not scaled. Scaling factor and offset do not apply.

In NI-CAN, the multiplexer signal was called mode channel.

Mux:Dynamic?

Data Type	Direction	Required?	Default
Boolean	Read Only	No	False

Property Class

XNET Signal

Property ID

`nxPropSig_MuxIsDynamic`

Description

Use this property to determine if a signal is static or dynamic. Dynamic signals are transmitted in the frame when the multiplexer signal in the frame has a given value specified in the subframe. Use the Mux:Multiplexer Value property to determine with which multiplexer value the dynamic signal is transmitted.

This property is read only. To create a dynamic signal, create the signal object as a child of a subframe instead of a frame. The dynamic signal cannot be changed to a static signal afterwards.

In NI-CAN, dynamic signals were called mode-dependent signals.

Mux:Multiplexer Value

Data Type	Direction	Required?	Default
u32	Read Only	N/A	N/A

Property Class

XNET Signal

Property ID

`nxPropSig_MuxValue`

Description

The multiplexer value applies to dynamic signals only (the XNET Signal Mux:Dynamic? property returns true). This property defines which multiplexer value is transmitted in the multiplexer signal when this dynamic signal is transmitted in the frame.

The multiplexer value is determined in the subframe. All dynamic signals that are children of the same subframe object use the same multiplexer value.

Dynamic signals with the same multiplexer value may not overlap each other, the multiplexer signal, or static signals.

Mux:Subframe

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	Parent Subframe

Property Class

XNET Signal

Property ID

`nxPropSig_MuxSubfrmRef`

Description

Reference to the subframe parent.

This property is valid only for dynamic signals that have a subframe parent. For static signals or the multiplexer signal, this property returns 0 and an error indication.

Name (Short)

Data Type	Direction	Required?	Default
char *	Read/Write	Yes	Defined in nxdbCreateObject

Property Class

XNET Signal

Property ID

nxPropSig_Name

Description

String identifying a signal object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A signal name must be unique for all signals in a frame.

This short name does not include qualifiers to ensure that it is unique, such as the database, cluster, and frame name. It is for display purposes.

You can write this property to change the signal's short name.

Name Unique to Cluster

Data Type	Direction	Required?	Default
cstr	Read Only	N/A	N/A

Property Class

XNET Signal

Property ID

`nxPropSig_NameUniqueToCluster`

Description

This property returns a signal name unique to the cluster that contains the signal. If the single name is not unique within the cluster, the name is <*frame-name*>.<*signal-name*>.

You can pass the name to the `nxdbFindObject` function to retrieve the reference to the object, while the single name is not guaranteed success in `nxdbFindObject` because it may be not unique in the cluster.

Number of Bits

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET Signal

Property ID

`nxPropSig_NumBits`

Description

The number of bits the signal uses in the frame payload.

IEEE Float numbers are limited to 32 bit or 64 bit.

Integer (signed and unsigned) numbers are limited to 1–52 bits. NI-XNET converts all integers to doubles (64-bit IEEE Float). Integer numbers with more than 52 bits (the size of the mantissa in a 64-bit IEEE Float) cannot be converted exactly to double, and vice versa; therefore, NI-XNET does not support this.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

PDU

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET Signal

Property ID

`nxPropSig_PDURef`

Description

Reference to the signal's parent PDU.

This property returns the reference to the signal's parent PDU. The parent PDU is defined when the signal object is created. You cannot change it afterwards.

Scaling Factor

Data Type	Direction	Required?	Default
Double	Read/Write	No	1.0

Property Class

XNET Signal

Property ID

`nxPropSig_ScaleFac`

Description

Factor *a* for linear scaling *ax*+*b*.

Linear scaling is applied to all signals with the IEEE Float data type, unsigned and signed. For identical scaling 1.0*x*+0.0, NI-XNET optimized scaling routines do not perform the multiplication and addition.

Scaling Offset

Data Type	Direction	Required?	Default
Double	Read/Write	No	0.0

Property Class

XNET Signal

Property ID

`nxPropSig_ScaleOff`

Description

Offset *b* for linear scaling *ax*+*b*.

Linear scaling is applied to all signals with the IEEE Float data type, unsigned and signed. For identical scaling 1.0*x*+0.0, NI-XNET optimized scaling routines do not perform the multiplication and addition.

Start Bit

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET Signal

Property ID

`nxPropSig_StartBit`

Description

The least significant signal bit position in the frame payload.

This property determines the signal starting point in the frame. For the integer data type (signed and unsigned), it means the binary signal representation least significant bit position. For IEEE Float signals, it means the mantissa least significant bit.

The NI-XNET Database Editor shows a graphical overview of the frame. It enumerates the frame bytes on the left and the byte bits on top. The bit number in the frame is calculated as byte number × 8 + bit number. The maximum bit number in a CAN or LIN frame is 63 (7 × 8 + 7); the maximum bit number in a FlexRay frame is 2031 (253 × 8 + 7).

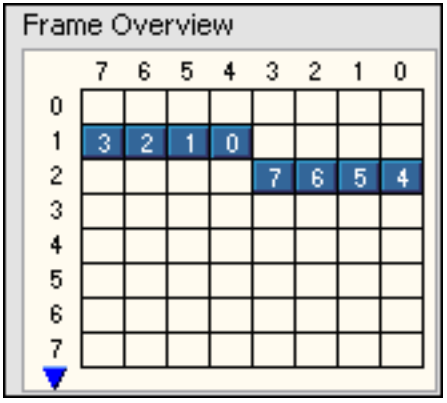


Figure 5-4. Frame Overview in the NI-XNET Database Editor with a Signal Starting in Bit 12

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this signal, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Unit

Data Type	Direction	Required?	Default
char *	Read/Write	No	Empty String

Property Class

XNET Signal

Property ID

`nxPropSig_Unit`

Description

This property describes the signal value unit. NI-XNET does not use the unit internally for calculations. You can use the string to display the signal value along with the unit.

XNET Subframe Properties

This section includes the XNET Subframe properties.

Dynamic Signals

Data Type	Direction	Required?	Default
nxDatabaseRef_t *	Read Only	N/A	N/A

Property Class

XNET Subframe

Property ID

`nxPropSubfrm_DynSigRefs`

Description

Dynamic signals in the subframe.

This property returns an array of refnums to dynamic signals in the subframe. Those signals are transmitted when the multiplexer signal in the frame has the multiplexer value defined in the subframe.

Dynamic signals are created with `nxdbCreateObject` by specifying a subframe as the parent.

Frame

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET Subframe

Property ID

`nxPropSubfrm_FrmRef`

Description

Returns the refnum to the parent frame. The parent frame is defined when the subframe is created, and you cannot change it afterwards.

Multiplexer Value

Data Type	Direction	Required?	Default
u32	Read/Write	Yes	N/A

Property Class

XNET Subframe

Property ID

`nxPropSubfrm_MuxValue`

Description

Multiplexer value for this subframe.

This property specifies the multiplexer signal value used when the dynamic signals in this subframe are transmitted in the frame. Only one subframe is transmitted at a time in the frame.

There also is a multiplexer value for a signal object as a read-only property. It reflects the value set on the parent subframe object.

This property is required. If the property does not contain a valid value, and you create an XNET session that uses this subframe, the session returns an error. To ensure that the property contains a valid value, you can do one of the following:

- Use a database file (or alias) to create the session.

  The file formats require a valid value in the text for this property.

- Set a value using the `nxdbSetProperty` function.

  This is needed when you create your own in-memory database (*:memory:*) rather than use a file. The property does not contain a default in this case, so you must set a valid value prior to creating a session.

For more information about using database files and in-memory databases, refer to *Databases*.

Name (Short)

Data Type	Direction	Required?	Default
char *	Read/Write	Yes	Defined in nxdbCreateObject

Property Class

XNET Subframe

Property ID

nxPropSubfrm_Name

Description

String identifying a subframe object.

Lowercase letters, uppercase letters, numbers, and the underscore (_) are valid characters for the short name. The space ( ), period (.), and other special characters are not supported within the name. The short name must begin with a letter (uppercase or lowercase) or underscore, and not a number. The short name is limited to 128 characters.

A subframe name must be unique for all subframes in a frame.

This short name does not include qualifiers to ensure that it is unique, such as the database, cluster, and frame name. It is for display purposes.

You can write this property to change the subframe's short name.

Name Unique to Cluster

Data Type	Direction	Required?	Default
cstr	Read Only	N/A	N/A

Property Class

XNET Subframe

Property ID

`nxPropSubfrm_NameUniqueToCluster`

Description

This property returns a subframe name unique to the cluster that contains the subframe. If the single name is not unique within the cluster, the name is <*frame-name*>.<*subframe-name*>.

You can pass the name to the `nxdbFindObject` function to retrieve the reference to the object, while the single name is not guaranteed success in `nxdbFindObject` because it may be not unique in the cluster.

PDU

Data Type	Direction	Required?	Default
nxDatabaseRef_t	Read Only	N/A	N/A

Property Class

XNET Subframe

Property ID

`nxPropSubfrm_PDURef`

Description

Reference to the subframe's parent PDU.

This property returns the reference to the subframe's parent PDU. The parent PDU is defined when the subframe object is created. You cannot change it afterwards.

XNET System Properties

Description

The XNET System properties provide information about all NI-XNET hardware in your system, including all devices and interfaces.

You retrieve a system handle with `nxSystemOpen` and release it with `nxSystemClose`. Pass the system handle to all system property calls.

Devices

Data Type	Direction	Required?	Default
u32[]	Read Only	No	N/A

Property Class

XNET System

Property ID

`nxPropSys_DevRefs`

Description

Returns an array of handles to physical XNET devices in the system. Each physical XNET board is a hardware product such as a PCI/PXI board.

You can pass the XNET Device handle to `nxGetProperty` and `nxGetPropertySize` to access properties of the device.

Interfaces (All)

Data Type	Direction	Required?	Default
u32[]	Read Only	No	N/A

Property Class

XNET System

Property ID

`nxPropSys_IntfRefs`

Description

Returns an array of handles to all available interfaces on the system.

Interfaces (CAN)

Data Type	Direction	Required?	Default
u32[]	Read Only	No	N/A

Property Class

XNET System

Property ID

`nxPropSys_IntfRefsCAN`

Description

Returns an array of handles to all available interfaces on the system that support the CAN Protocol.

Interfaces (FlexRay)

Data Type	Direction	Required?	Default
u32[]	Read Only	No	N/A

Property Class

XNET System

Property ID

`nxPropSys_IntfRefsFlexRay`

Description

Returns an array of handles to all available interfaces on the system that support the FlexRay protocol.

Interfaces (LIN)

Data Type	Direction	Required?	Default
u32[]	Read Only	No	N/A

Property Class

XNET System

Property ID

`nxPropSys_IntfRefsLIN`

Description

Returns an array of handles to all available interfaces on the system that support the LIN Protocol.

Version:Build

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET System

Property ID

`nxPropSys_VerBuild`

Description

Returns the driver version [Build] as a u32.

Remarks

The driver version is specified in the following format:
[Major].[Minor].[Update][Phase][Build].

For example, 1.2.3f4 returns:

- [Major] = 1
- [Minor] = 2
- [Update] = 3
- [Phase] = Final/Release
- [Build] = 4

A larger version number implies a newer XNET driver version.

Use this property for:

- Determining driver functionality or release date.
- Determining upgrade availability.

Version:Major

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET System

Property ID

`nxPropSys_VerMajor`

Description

Returns the driver version [Major] as a u32.

Remarks

The driver version is specified in the following format:
[Major].[Minor].[Update][Phase][Build].

For example, 1.2.3f4 returns:

- [Major] = 1
- [Minor] = 2
- [Update] = 3
- [Phase] = Final/Release
- [Build] = 4

A larger version number implies a newer XNET driver version.

Use this property for:

- Determining driver functionality or release date.
- Determining upgrade availability.

Version:Minor

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET System

Property ID

`nxPropSys_VerMinor`

Description

Returns the driver version [Minor] as a u32.

Remarks

The driver version is specified in the following format:
[Major].[Minor].[Update][Phase][Build].

For example, 1.2.3f4 returns:

- [Major] = 1
- [Minor] = 2
- [Update] = 3
- [Phase] = Final/Release
- [Build] = 4

A larger version number implies a newer XNET driver version.

Use this property for:

- Determining driver functionality or release date.
- Determining upgrade availability.

Version:Phase

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET System

Property ID

`nxPropSys_VerPhase`

Description

Returns the driver version [Phase] as a u32.

Enumeration	Value
nxPhase_Development	0
nxPhase_Alpha	1
nxPhase_Beta	2
nxPhase_Release	3

Note The driver's official version always has a phase of Release.

Remarks

The driver version is specified in the following format:
[Major].[Minor].[Update][Phase][Build].

For example, 1.2.3f4 returns:

- [Major] = 1
- [Minor] = 2
- [Update] = 3
- [Phase] = Final/Release
- [Build] = 4

A larger version number implies a newer XNET driver version.

Use this property for:

- Determining driver functionality or release date.
- Determining upgrade availability.

Version:Update

Data Type	Direction	Required?	Default
u32	Read Only	No	N/A

Property Class

XNET System

Property ID

`nxPropSys_VerUpdate`

Description

Returns the driver version [Update] as a u32.

Remarks

The driver version is specified in the following format:
[Major].[Minor].[Update][Phase][Build].

For example, 1.2.3f4 returns:

- [Major] = 1
- [Minor] = 2
- [Update] = 3
- [Phase] = Final/Release
- [Build] = 4

A larger version number implies a newer XNET driver version.

Use this property for:

- Determining driver functionality or release date.
- Determining upgrade availability.

Additional Topics

This section includes additional CAN, FlexRay, and LIN-related information.

Overall

Cyclic and Event Timing

For all embedded network protocols (for example, CAN, LIN, and FlexRay), 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 `nxWrite...` function. The call to the `nxWrite...` 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 `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* section, 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 `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 `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 `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.

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 simply 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.

Raw Frame Format

This section describes the raw data format for frames. `nxReadFrame` and `nxWriteFrame` use this format.

The raw frame format is for examples that demonstrate access to log files. 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.

Refer to the NI-XNET logfile examples for functions that convert raw frame data for CAN, FlexRay, or LIN frames.

The raw frame format consists of one or more frames encoded in a sequence of bytes. 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).

Table 5-3. Base Unit Elements

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 12:00 a.m. January 1 1601 Coordinated Universal Time (UTC). 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 big-endian computing platforms (for example, CompactRIO with a PowerPC), Byte Offset 0 is the most significant byte. For more information, refer to the NI-XNET examples for logfile 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). All unused bits are 0.

Table 5-3. Base Unit Elements (Continued)

Element	Byte Offset	Description
Type	12	The frame type. This element specifies the fundamental frame type. The Identifier, Flag, and Info element interpretation is different for each type. The upper 4 bits of this element specify the protocol: The valid values in decimal are: 0 CAN 2 FlexRay 4 LIN 14 Special The lower 4 bits of this element contain the specific type. The following Type values may occur for CAN: **CAN Data (0)** The CAN data frame contains payload data. This is the most commonly used frame type for CAN. **CAN Remote (1)** 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. **Delay (224)** 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*.

Table 5-3. Base Unit Elements (Continued)

Element	Byte Offset	Description
		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. (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*. The following Type values may occur for FlexRay: **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. 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.

Table 5-3. Base Unit Elements (Continued)

Element	Byte Offset	Description
		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.
		FlexRay Symbol (34) — FlexRay symbol frame. The frame contains a symbol received on the FlexRay bus. For this frame type, the 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. (Refer to *Start Interface* for more information.) For information about this frame, including the other frame fields, refer to *Special Frames*.
		The following Type values may occur for LIN:
		LIN Data (64) — The LIN data frame contains payload data. This currently is the only frame type for LIN.

Table 5-3. Base Unit Elements (Continued)

Element	Byte Offset	Description	
		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. (Refer to *Start Interface* for more information.) For information about this frame, including the other frame fields, refer to *Special Frames*.
		LIN Bus Error (65)	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*.

Table 5-3. Base Unit Elements (Continued)

Element	Byte Offset	Description
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 `nxRead` function 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 • Bit 2 specifies the frame Preamble bit • Bit 4 specifies if the frame transfers on Channel A • Bit 5 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, if bit 0 of the Flags element is clear, the Info element is 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.

Table 5-3. Base Unit Elements (Continued)

Element	Byte Offset	Description
PayloadLength	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. 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 Payload, bytes 8–11 are in the first byte 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 logfile, but PayloadLength determines the number of valid bytes.

Payload Unit

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

Table 5-4. Payload Unit Elements

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

Special Frames

The NI-XNET driver offers a few special frames not directly used in bus communication.

Delay Frame

A Delay frame is used during replay. When a frame with a Delay frame type is in the stream output queue while the Interface:Output Stream Timing property is set to a replay mode, the hardware delays for the requested time. The fields of the Delay frame are as follows:

Element	Description
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, absolute time of 6:00:00.250PM 12/31/1903) will delay 250 ms.
Identifier	0 (Ignored)
Type	`nxFrameType_Special_Delay`
Flags	0 (Ignored)
Info	0 (Ignored)
Payload Length	0
Payload	N/A

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 `nxConnectTerminals` to connect the external connection to the internal LogTrigger

terminal. A Log Trigger frame is applicable to CAN, LIN, and FlexRay. The fields of the Log Trigger frame are as follows:

Element	Description
Timestamp	Time when the trigger occurred.
Identifier	0
Type	`nxFrameType_Special_LogTrigger`
Flags	0
Info	0
Payload Length	0
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. (Refer to *Start Interface* for more information.) 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, LIN, and FlexRay. The fields of the Start Trigger frame are as follows:

Element	Description
Timestamp	Time when the interface started.
Identifier	0
Type	`nxFrameType_Special_StartTrigger`
Flags	0
Info	0
Payload Length	0
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:

CAN Frame

Element	Description
Timestamp	Time when the bus error was detected.
Identifier	0
Type	`nxFrameType_Special_CANBusError`
Flags	0
Info	0
Payload Length	5 (may increase in the future)
Payload	Byte 0: CAN Comm State 0 = Error Active 1 = Error Passive 2 = Bus Off Byte 1: TX Error Counter Byte 2: RX Error Counter Byte 3: Detected Bus Error 0 = None (never returned) 1 = Stuff 2 = Form 3 = Ack 4 = Bit 1 5 = Bit 0 6 = CRC Byte 4: Transceiver Error? 0 = no error 1 = error

LIN Frame

Element	Description
Timestamp	Time when the bus error was detected.
Identifier	0
Type	`nxFrameType_Special_LINBusError`
Flags	0
Info	0
Payload Length	5 (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 6 = CRC Byte 2: Identifier on bus Byte 3: Received byte on bus Byte 4: Expected byte on bus

Required Properties

When you create a new object in a database, the object 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 section lists all required properties. Properties with a protocol prefix (for example, *FlexRay:*) in the property name apply only a session that uses the specified protocol.

The Cluster object class requires the following properties:

- Baud Rate[1]
- 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

[1] 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 Baud Rate property or XNET Session Interface:Baud Rate property. For CAN and LIN with other session modes, the XNET Cluster Baud Rate property is required.

- 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 property:

- 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

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 can be accessed using the API. In other words, the session controls receive or transmit 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 the appropriate `nxRead` or `nxWrite` function, you can perform a more specific transition using `nxStart` and `nxStop`. 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*.

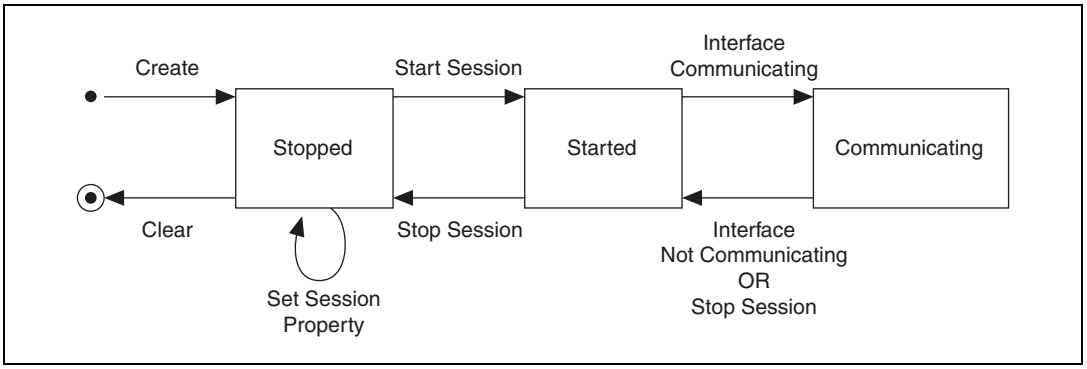


Figure 5-5. Session State Model

Interface State Model

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

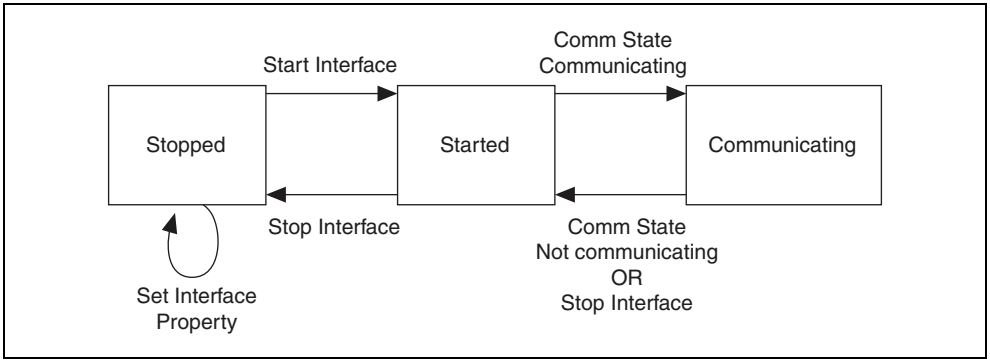


Figure 5-6. Interface State Model

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 property in the Session Property Node 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 appropriate `nxRead` or `nxWrite` function or `nxStart` using defaults, 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 `nxStart` 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 (`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 `nxRead` or `nxWrite` function.

Session Transitions

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 property in the XNET Session Property 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 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, then you call nxStop, you still can call the appropriate nxRead function to read the frame values from the queues. To flush the queues of a session, 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.

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 Property Node Interface category. 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*.

Interface Transitions

Set Interface Property

While the interface is Stopped, you can change interface-specific properties. These properties are in the Session Property Node Interface category. 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 the 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:

- **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.
- `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 `nxConnectTerminals` with a destination of Interface Start Trigger, or by setting 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 (the appropriate `nxRead` or `nxWrite` function or `nxStart`).
- **Connected**: Start Interface occurs when the connected source terminal transistions 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, `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 `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 `nxStop`. In other words, using `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 `nxReadState`.

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 `nxReadState`.

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 `nxReadState`.

Comm State Not Communicating

This transition occurs when the interface no longer is 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 `nxReadState`.

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 `nxReadState`.

For LIN, this occurs when communication enters the Idle state. For more information about the specific LIN interface communication states, refer to `nxReadState`.

CAN

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.

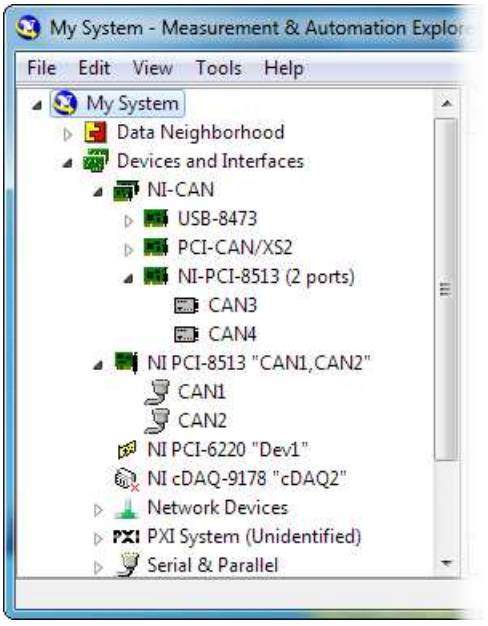
If you are starting a new application, you typically use NI-XNET (not NI-CAN).

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 Hardware and Software 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.

Table 5-5. NI-CAN and 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 format.
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.

Table 5-5. NI-CAN and NI-XNET Terms (Continued) (Continued)

NI-CAN Term	NI-XNET Term	Comment
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 section 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

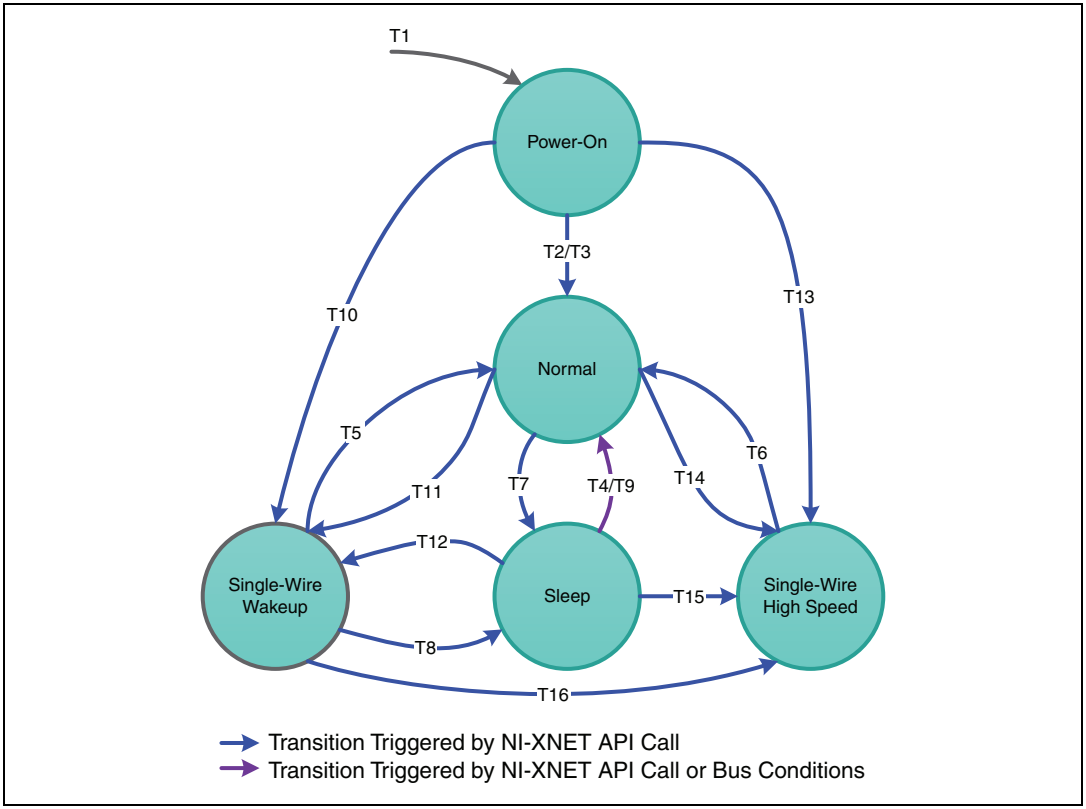
The behavior is the same as Cyclic Remote. When you write data using the appropriate `nxWrite` function, the CAN data frame transmits onto the network when the associated CAN remote frame is received (same identifier). Unlike Cyclic Data, the remote ECU sends the associated CAN remote frame in an event-driven 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 transceriver by setting 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
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

T#	Condition	From	To
10	Interface:CAN:Transceiver State with value SW Wakeup	Power-on	SW Wakeup
11	Interface:CAN:Transceiver State with value SW Wakeup	Normal	SW Wakeup
12	Interface:CAN:Transceiver State with value SW Wakeup	Sleep	SW Wakeup
13	Interface:CAN:Transceiver State with value SW HighSpeed	Power-on	SW High Speed
14	Interface:CAN:Transceiver State with value SW HighSpeed	Normal	SW High Speed
15	Interface:CAN:Transceiver State with value SW HighSpeed	Sleep	SW High Speed
16	Interface:CAN:Transceiver State with value SW HighSpeed	SW Wakeup	SW High Speed

FlexRay

FlexRay Timing Type and Session Mode

For each XNET Frame FlexRay:Timing Type property value, this section 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.

Protocol Data Units (PDUs) in NI-XNET

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.

Relationship Between Frames, Signals, and PDUs

Frames and PDUs

The frame element contains an arbitrary number of nonoverlapping PDUs. A frame can have multiple PDUs, and the same PDU can exist in different frames. Figure 5-7 shows the one-to-*n* (one PDU in *n* number of frames) and *n*-to-one (*n* number of PDUs in one frame) relationships.

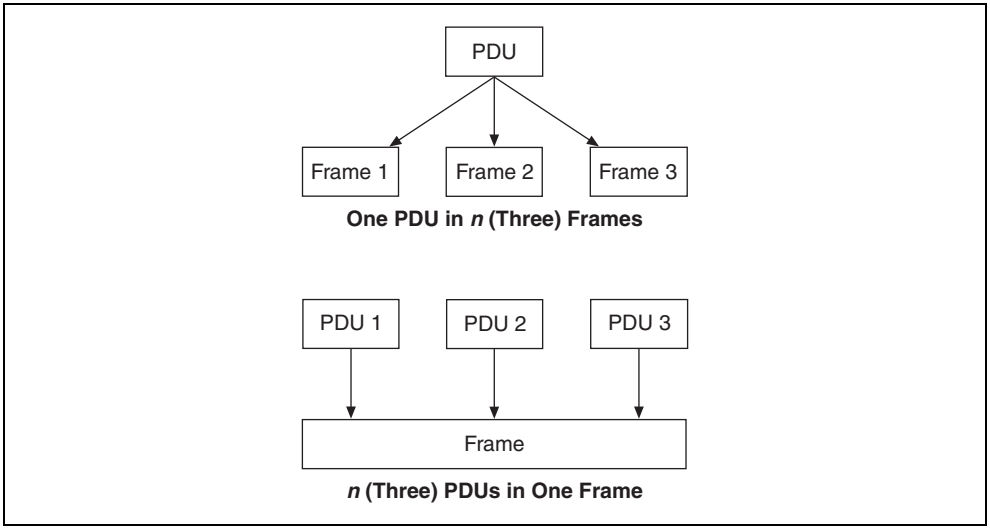


Figure 5-7. Relationships Between PDUs and Frames

Signals and PDUs

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

Figure 5-8 represents the relationship between frames, PDUs, and signals.

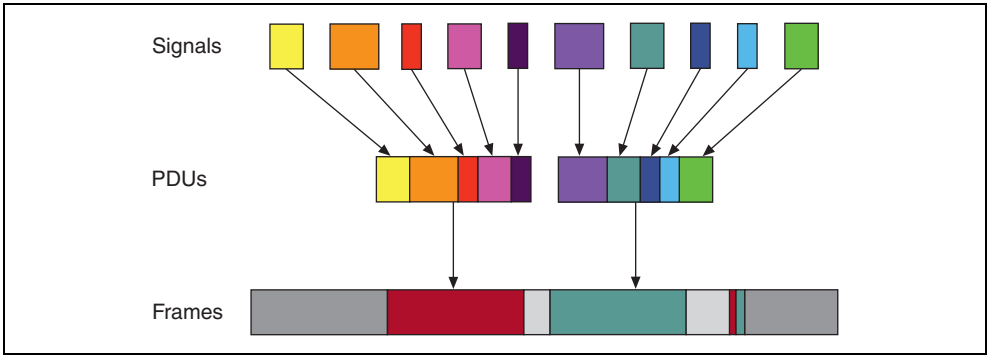


Figure 5-8. Relationships Between Frames, PDUs, and Signals

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.

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 Figure 5-9.)

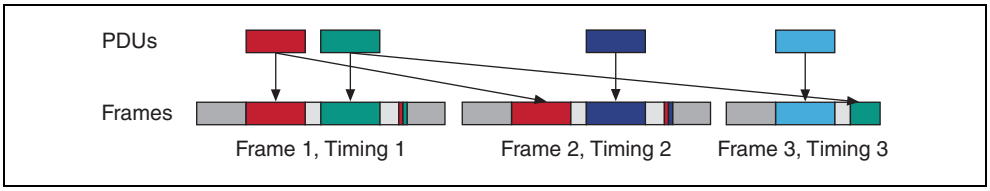


Figure 5-9. PDU Timing and Frame Timing

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 the PDU(s) to the special Create Session modes for PDU. (Refer to `nxCreateSession` for more information.) 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.

FlexRay Startup/Wakeup

Use the FlexRay Startup mechanism to take an idle interface and properly integrate into a FlexRay cluster.

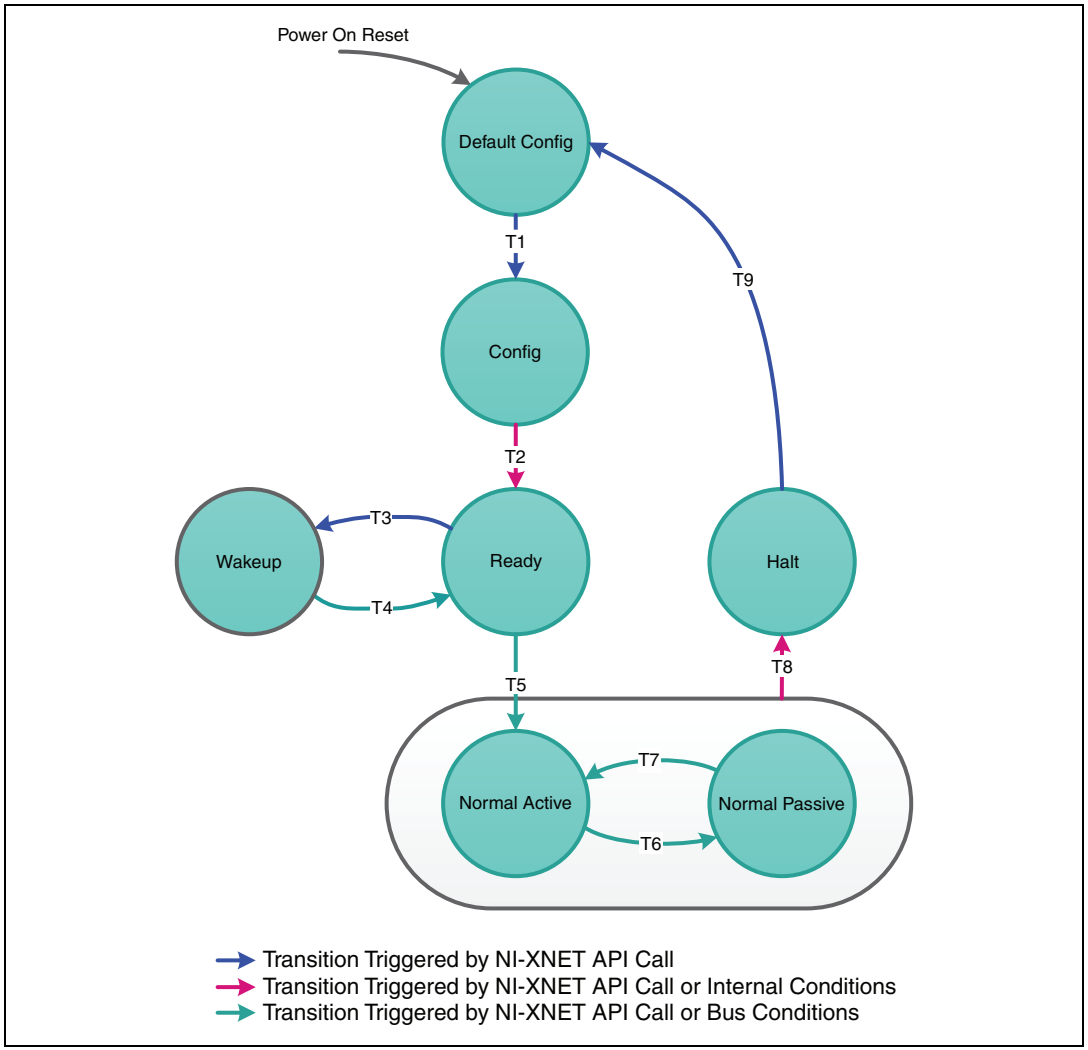
If your cluster does not support the wakeup mechanism, this process is straightforward. After creating your FlexRay session, call `nxStart`, which causes the interface to transition from **Default Config** to **Ready**, where it attempts to integrate with the FlexRay cluster. If your node is a coldstart 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:AutoAsleepWhenStopped 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 (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.

LIN

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`). 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.

6

Troubleshooting and Common Questions

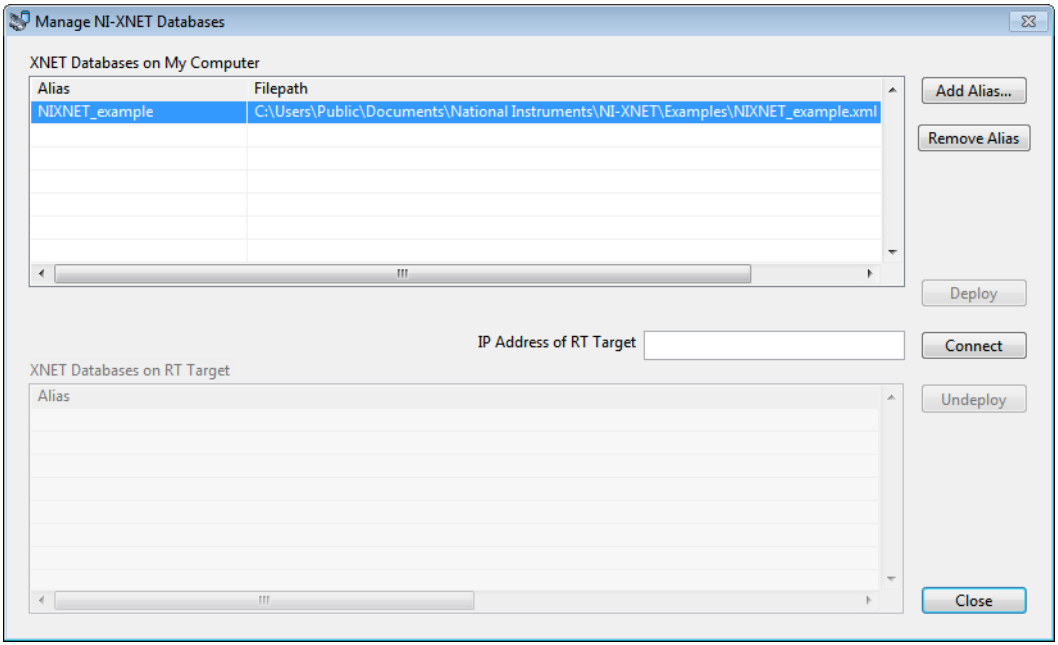
This chapter includes NI-XNET troubleshooting tips and common questions

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 following window appears, and you can see where your database file is on the disk.



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 add 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.

Notes 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.

On LabVIEW RT

Open the **Manage NI-XNET Databases** dialog (see above) and connect to your LabVIEW RT target by entering the IP address and clicking **Connect**. Select the NIXNET_example database and click **Undeploy**.

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.

In LabVIEW

Use a property node (shown below) for the session.



In 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. Examples of how to do this are in the example finder at **Hardware Input and Output»CAN»NI-XNET» Advanced» CAN Change Objects 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.

A

Summary of the CAN Standard

This appendix summarizes the CAN standard.

History and Use of CAN

In the past few decades, advances in automotive technology have led to increased use of electronic control systems for engine timing, anti-lock brake systems, and distributorless ignition. With conventional wiring, data is exchanged in these systems using dedicated signal lines. As the complexity and number of devices has increased, using dedicated signal lines becomes increasingly difficult and expensive.

To overcome the limitations of conventional automotive wiring, Bosch developed the Controller Area Network (CAN) in the mid-1980s. Using CAN, devices (controllers, sensors, and actuators) are connected on a common serial bus. This network of devices can be thought of as a scaled-down, real-time, low-cost version of networks used to connect personal computers. Any device on a CAN network can communicate with any other device using a common pair of wires.

As CAN implementations increased in the automotive industry, CAN was standardized internationally as ISO 11898. CAN chips were created by major semiconductor manufacturers such as Intel, Motorola, and Philips. With these developments, manufacturers of industrial automation equipment began to consider CAN for use in industrial applications. Comparison of the requirements for automotive and industrial device networks showed numerous similarities, including the transition away from dedicated signal lines, low cost, resistance to harsh environments, and high real-time capabilities.

Because of these similarities, CAN became widely used in photoelectric sensors and motion controllers for textile machinery, packaging machines, and production line equipment. By the mid-1990s, CAN was specified as the basis of many industrial device networking protocols, including DeviceNet, and CANopen.

On April 17, 2012, Bosch released an updated CAN specification, *CAN with Flexible Data-Rate*. This specification improves CAN performance by making two key additions to the CAN standard: increasing the maximum payload size from 8 to 64 bytes and maximum baud rate from 1 to 2 Mb/s or more. Remote frames always are transmitted in the CAN 2.0 standard format.

With its growing popularity in automotive and industrial applications, CAN has been increasingly used in a wide variety of diverse applications. Use in agricultural equipment, nautical machinery, medical apparatus, semiconductor manufacturing equipment, and machine tools testify to the versatility of CAN.

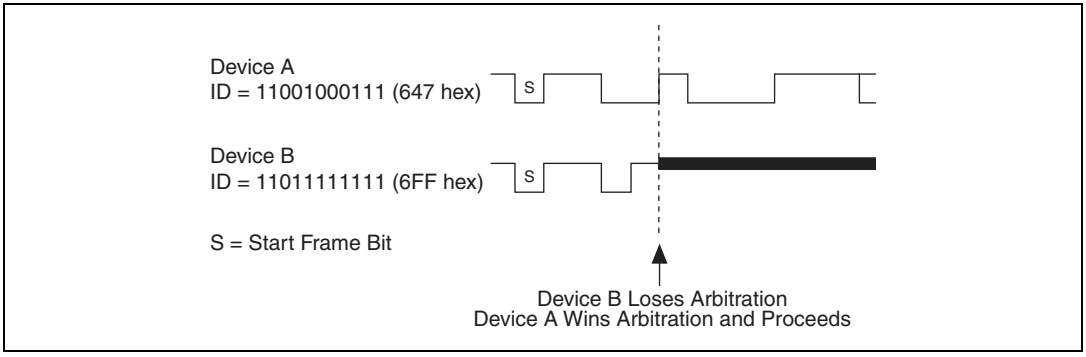
CAN Identifiers and Message Priority

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.

Figure A-1 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.



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

Figure A-1. Example of CAN Arbitration

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. Figure A-2 shows the essential fields of the standard and extended frame formats, and the following sections describe each field.

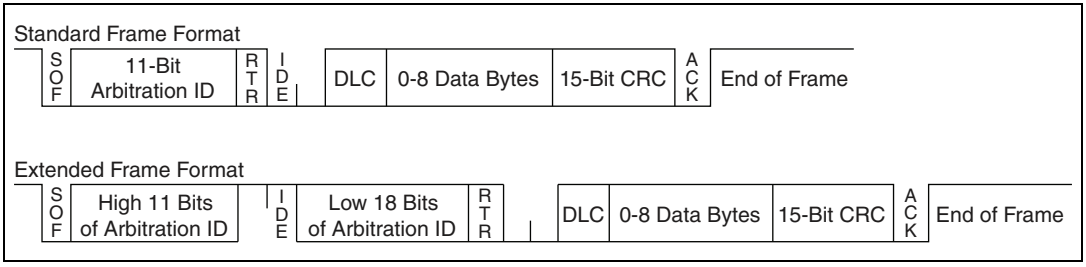


Figure A-2. Standard and Extended Frame Formats

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

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. Figure A-3 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.

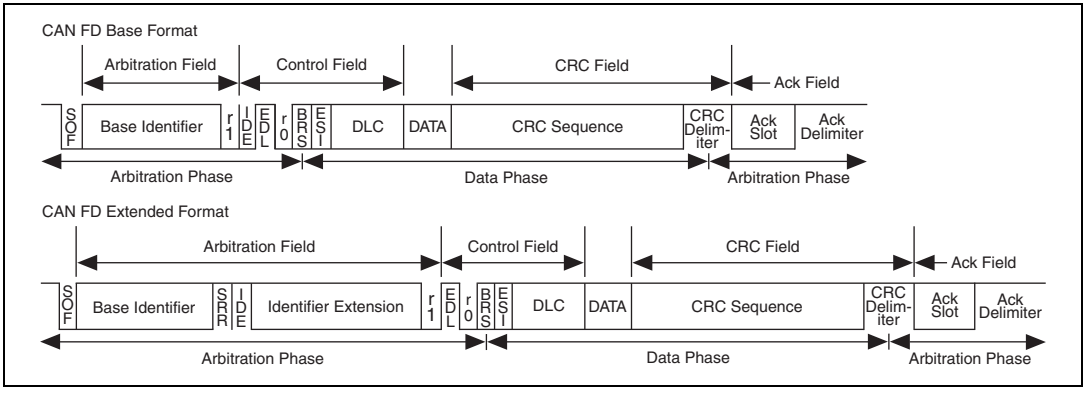


Figure A-3. CAN FD Standard and Extended Frame Formats

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 nonarbitration 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.

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 sections:

- 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.

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.

CRC Error

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

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.

Acknowledgment Error

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

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 on 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 State, Error Passive State, and 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.

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).

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.

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 125 K 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 the *Cabling Requirements for Low-Speed/Fault-Tolerant CAN* section of Chapter 3, *NI-XNET Hardware Overview*.

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 CAN Physical Layer* in Chapter 3, *NI-XNET Hardware Overview*.

B

Summary of the FlexRay Standard

This appendix summarizes the FlexRay standard.

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.

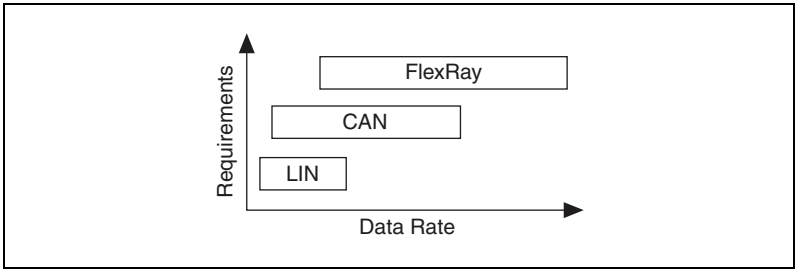


Figure B-1. Requirements Comparison

As shown in Figure B-1, 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 Network

FlexRay Bus Benefits

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

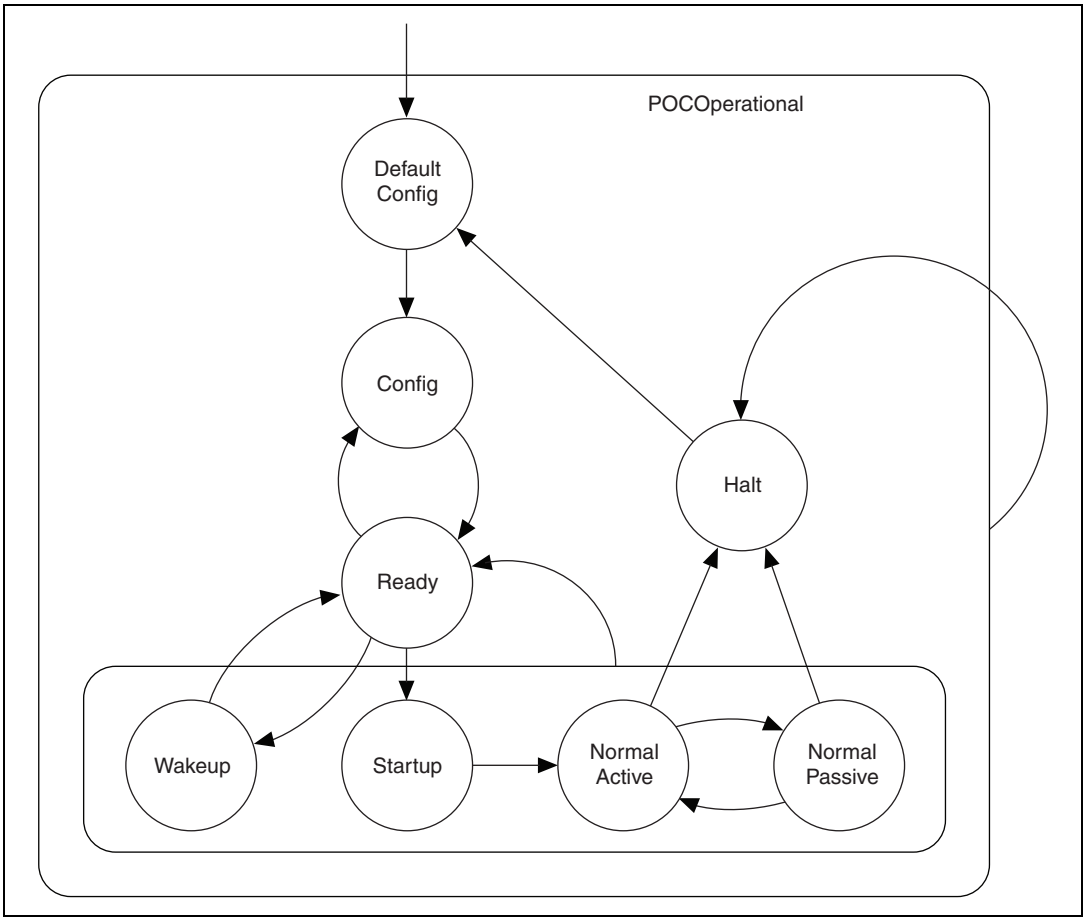
- Provides up to 10 Mbits/s data rate on each channel, or a gross data rate up to 20 Mbits/s.
- Significantly increases Frame Length (compared to CAN—8 bytes per frame).
- Makes synchronous and asynchronous data transfer possible.
- Guarantees frame latency and jitter during synchronous transfer (real-time 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 nonredundant 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 clusterwide 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.

Figure B-2 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.

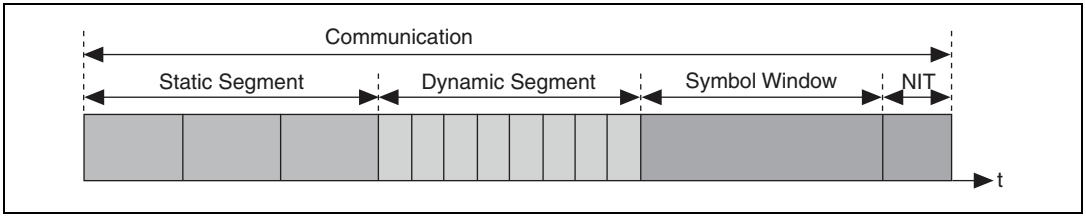


Figure B-2. Communication Cycle

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.

Figure B-3 shows the startup state machine as the *FlexRay Protocol Specification* v. 2.1 defines it.

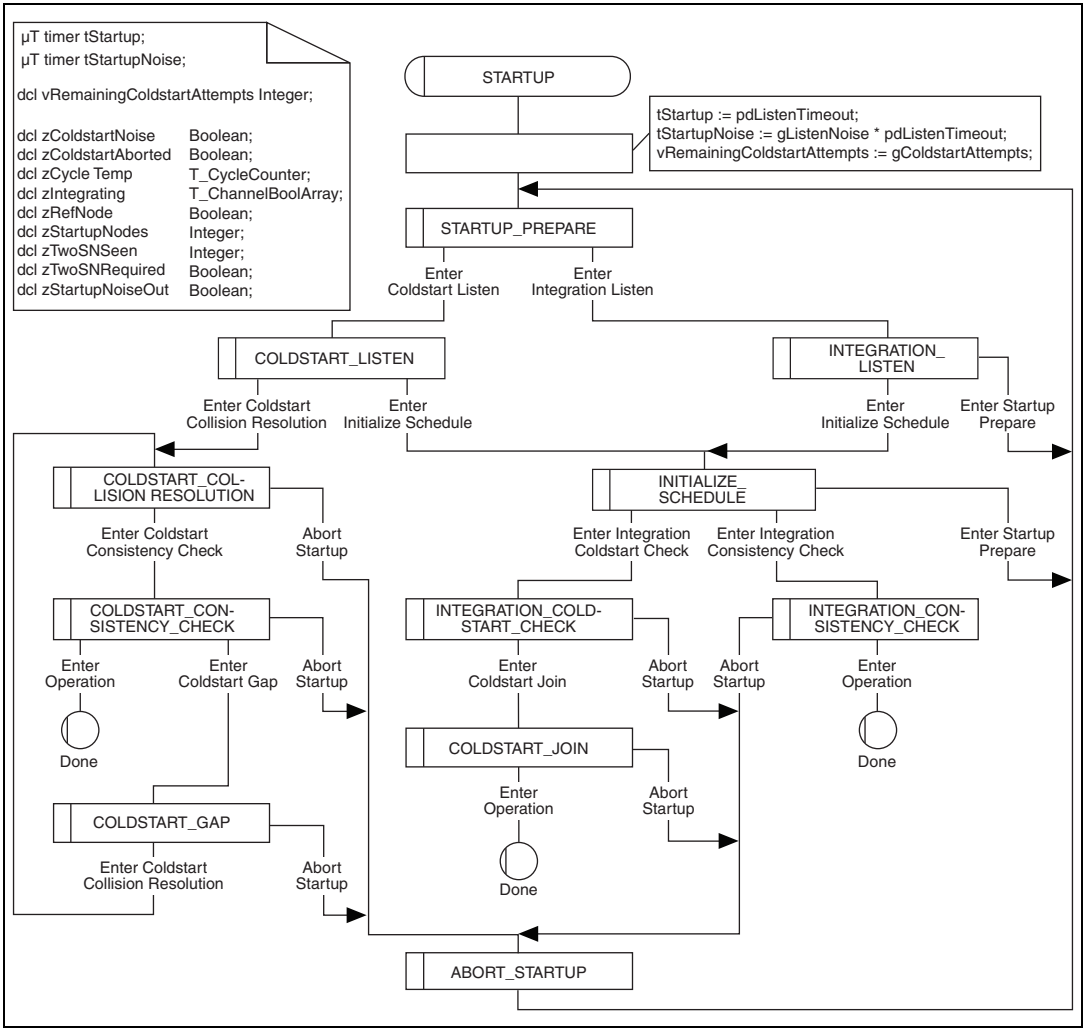


Figure B-3. Startup State Machine

Figure B-4 shows the state transitions for a leading coldstart node (Node A), following coldstart node (Node B), and non-coldstart node (Node C).

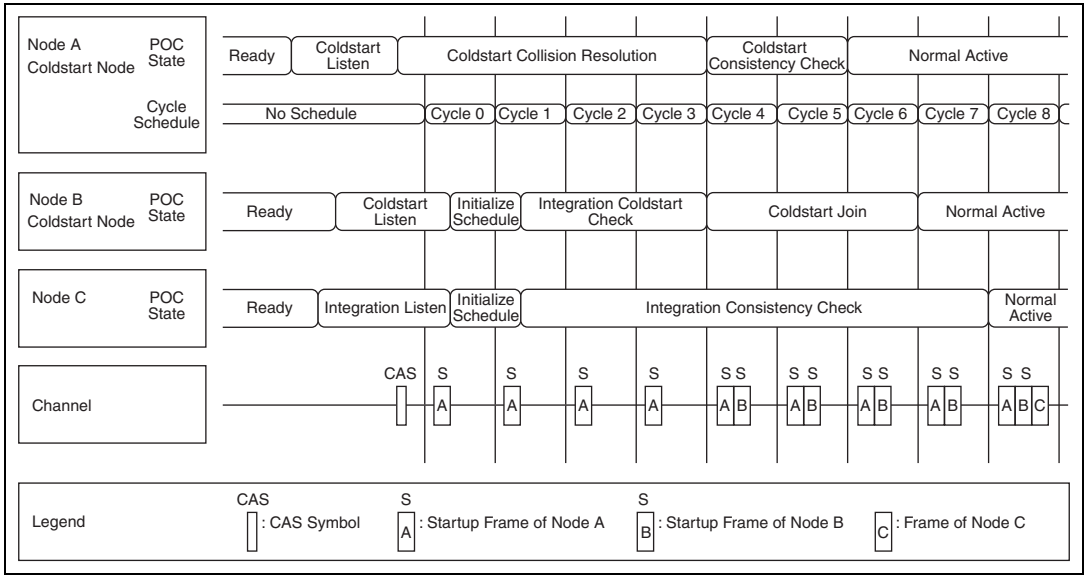


Figure B-4. State Transitions

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.

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

Figure B-5 shows the FlexRay frame format. The FlexRay frame has three segments: header, payload, and trailer.

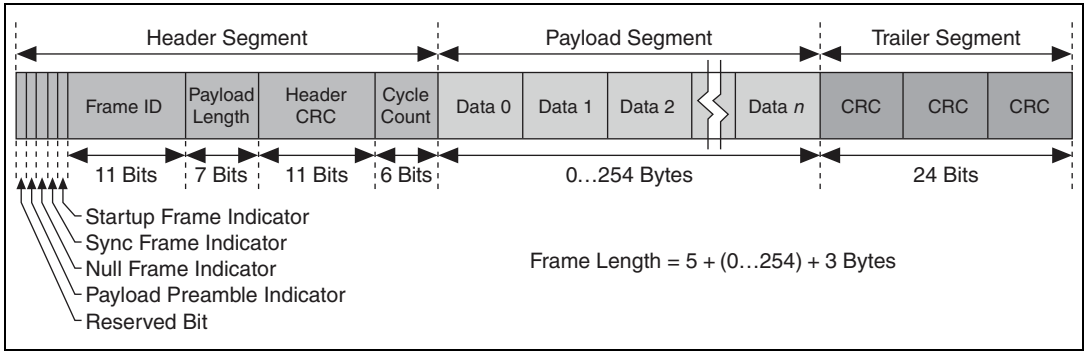


Figure B-5. FlexRay Frame Format

- **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.

C

Summary of the LIN Standard

This appendix summarizes the LIN standard.

History and Use of LIN

Local Interconnect Network (LIN) was developed to create a standard for low-cost, low-end multiplexed communication in automotive networks. Whereas CAN addressed the need for high-bandwidth, advanced error-handling networks, the hardware and software costs of CAN implementation became prohibitive for lower performance devices like power window and seat controllers. LIN provides cost-efficient communication in applications where the bandwidth and versatility of CAN are not required. LIN can be implemented relatively inexpensively using the standard serial UART embedded into most modern low-cost 8-bit microcontrollers.

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 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 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.

Figure C-1 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

Figure C-1. Parity Calculation Method

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.

Figure C-2 shows how a master task header and slave task response combine to create a LIN full frame.

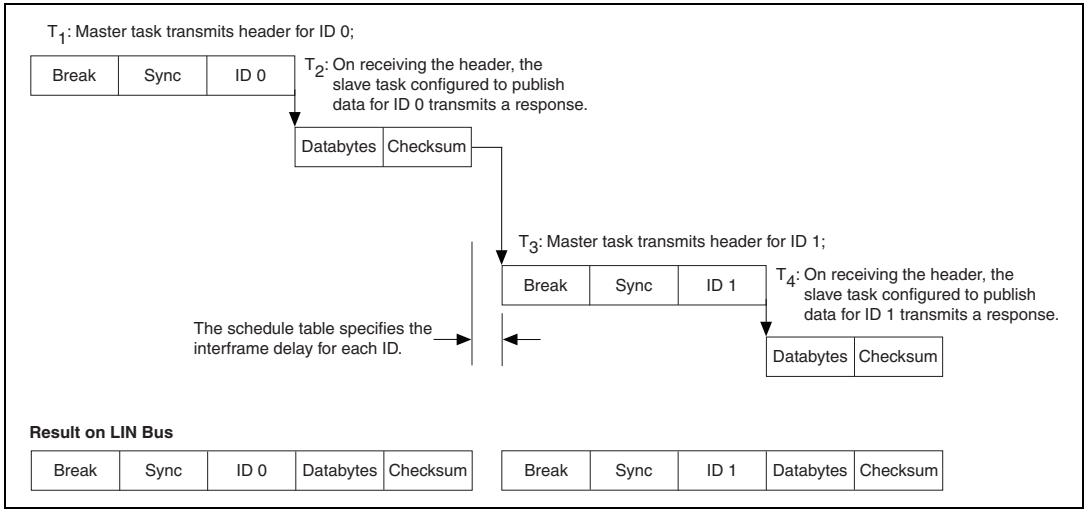


Figure C-2. Creation of LIN Full Frames

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} = Time it takes to transmit 1 bit (1/Baud_Rate)

N_{Data} = Number of data bytes in response

$T_{Header\_Nominal} = 34 * T_{Bit}$

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

$T_{Frame\_Nominal} = T_{Header\_Nominal} + T_{Response\_Nominal}$

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.

$T_{Header\_Maximum} = 1.4 * T_{Header\_Nominal}$

$T_{Response\_Maximum} = 1.4 * T_{Response\_Nominal}$

$T_{Frame\_Maximum} = T_{Header\_Maximum} + T_{Response\_Maximum}$

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. Afterwards, 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.

Additional LIN Information

For further LIN specification details, refer to the LIN consortium Web site at `www.lin-subbus.org`.

D

Specifications

This appendix lists specifications for PXI-XNET, PCI-XNET, and C Series NI XNET hardware.

PXI-XNET

This section lists specifications for PXI-XNET hardware.

Physical Layers

CAN Physical Layers

High-Speed CAN

Transceiver ..... NXP TJA1041

Max baud rate ..... 1 Mbps

Min baud rate ..... 40 kbps

CAN_H, CAN_L bus lines ..... –27 to +40 VDC

Low-Speed/Fault-Tolerant CAN

Transceiver[1] ..... NXP TJA1054A or TJA 1055T

Max baud rate ..... 125 kbps

Min baud rate ..... 40 kbps, 10 kbps min for all error modes

Single Wire CAN

Transceiver[2] ..... NXP AU5790 or ON Semiconductor NCV7356

[1] Refer to *Low-Speed/Fault-Tolerant Physical Layer* in Chapter 3, *NI-XNET Hardware Overview*, to determine the transceiver used.

[2] Refer to *Single Wire CAN Physical Layer* in Chapter 3, *NI-XNET Hardware Overview*, to determine the transceiver used.

Max baud rate .....83.3 kbps

Min baud rate.....33.3 kbps

Bus Power Required .....+8 to +18 V

External CAN Transceiver

Digital I/O Characteristics

Parameter	Min	Max
Output0, Output1		
V_{OH} (I_{OH} = –8 mA)	3.8 V	—
V_{OL} (I_{OL} = 8 mA)	—	0.44 V
NERR		
V_{IH}	2.0 V	—
V_{IL}	—	0.8 V

FlexRay Physical Layer

Transceiver .....NXP TJA1080 × 2

Max baud rate .....10 Mbps

Min baud rate.....1 Mbps

LIN Physical Layer

Transceiver[1] .....ATMEL ATA6620 or ATA6625

Max baud rate .....20 kbps

Min baud rate.....2.4 kbps

Bus Power Required .....+8 to +18 V

RTSI/Front Panel Sync Connectors

Trigger lines.....7 input/output

Clock lines .....1 input/output

[1] Refer to *LIN Physical Layer* in Chapter 3, *NI-XNET Hardware Overview*, to determine the transceiver used.

Front Panel Sync Connectors ..... 2 input/output (XS/FlexRay only)

I/O compatibility ..... TTL

Power-on state ..... Input (High-Z)

Response ..... Rising Edge Triggers

Physical Dimensions

Dimensions ..... 10.00 cm × 16.00 cm (3.9 in. × 6.3 in.)

I/O connector ..... 9-pin male D-SUB for each port

Sync connector ..... SMB jack × 2 (XS/FlexRay only)

Power Requirements

CAN

+5 VDC (±5%) ..... 640 mA typical

+3.3 VDC (±5%) ..... 940 mA typical

FlexRay

+5 VDC (±5%) ..... 210 mA typical

+3.3 VDC (±5%) ..... 940 mA typical

LIN

+3.3 VDC (±5%) ..... 940 mA typical

Shock

Operating ..... 30 g peak, half-sine, 11 ms pulse (Tested in accordance with IEC-60068-2-27. Test profile developed in accordance with MIL-PRF-28800F.)

Random Vibration

Operating .....5 to 500 Hz, 0.3 grms

Non-operating .....5 to 500 Hz, 2.4 grms
(Tested in accordance with IEC-60068-2-64. Non-operating test profile exceeds the requirements of MIL-PRF-28800F, Class 3.)

Safety

Isolation Voltages

Port-to-port ground

Continuous.....60 VDC, Measurement Category I

Port-to-earth ground

Continuous.....60 VDC, Measurement Category I

This isolation is intended to prevent ground loops.

Safety Standards

This product meets the requirements of the following standards of safety for electrical equipment for measurement, control, and laboratory use:

- IEC 61010-1, EN 61010-1
- UL 61010-1, CSA 61010-1

Note For UL and other safety certifications, refer to the product label or the *Online Product Certification* section.

Environmental

Operating temperature .....0 to 55 °C

Storage temperature .....–20 to 70 °C
(Tested in accordance with IEC-60068-2-1 and IEC-60068-2-2.)

Operating humidity .....10 to 90% RH, noncondensing

Storage humidity ..... 5 to 95% RH, noncondensing (Tested in accordance with IEC-60068-2-56.)

Maximum altitude ..... 2000 m

Pollution Degree (IEC 60664) 2

Indoor use only.

PCI-XNET

This section lists specifications for PCI-XNET hardware.

Physical Layers

CAN Physical Layers

High-Speed CAN

Transceiver..... NXP TJA1041

Max baud rate..... 1 Mbps

Min baud rate ..... 40 kbps

CAN_H, CAN_L bus lines ..... –27 to +40 VDC

Low-Speed/Fault-Tolerant CAN

Transceiver[1] ..... NXP TJA1054A or TJA 1055T

Max baud rate..... 125 kbps

Min baud rate ..... 40 kbps, 10 kbps min for all error modes

Single Wire CAN

Transceiver[2] ..... NXP AU5790 or ON Semiconductor NCV7356

Max baud rate..... 83.3 kbps

[1] Refer to *Low-Speed/Fault-Tolerant Physical Layer* in Chapter 3, *NI-XNET Hardware Overview*, to determine the transceiver used.

[2] Refer to *Single Wire CAN Physical Layer* in Chapter 3, *NI-XNET Hardware Overview*, to determine the transceiver used.

Min baud rate.....33.3 kbps

Bus Power Required .....+8 to +18 V

External CAN Transceiver

Digital I/O Characteristics

Parameter	Min	Max
Output0, Output1		
V_{OH} (I_{OH} = –8 mA)	3.8 V	—
V_{OL} (I_{OL} = 8 mA)	—	0.44 V
NERR		
V_{IH}	2.0 V	—
V_{IL}	—	0.8 V

FlexRay Physical Layer

Transceiver .....NXP TJA1080 × 2

Max baud rate .....10 Mbps

Min baud rate.....1 Mbps

LIN Physical Layer

Transceiver[1] .....ATMEL ATA6620 or ATA6625

Max baud rate .....20 kbps

Min baud rate.....2.4 kbps

Bus Power Required .....+8 to +18 V

RTSI/Front Panel Sync Connectors

Trigger lines.....7 input/output

Clock lines .....1 input/output

Front Panel Sync Connectors .....2 input/output
(XS/FlexRay only)

[1] Refer to *LIN Physical Layer* in Chapter 3, *NI-XNET Hardware Overview*, to determine the transceiver used.

I/O compatibility ..... TTL

Power-on state ..... Input (High-Z)

Response ..... Rising edge triggers

Physical Dimensions

Dimensions ..... 10.67 cm × 16.76 cm (4.2 in. × 6.6 in.)

I/O connector ..... 9-pin male D-SUB for each port

Sync connector ..... SMB jack × 2 (XS/FlexRay only)

Power Requirements

CAN

+5 VDC (±5%) ..... 640 mA typical

+3.3 VDC (±5%) ..... 940 mA typical

FlexRay

+5 VDC (±5%) ..... 210 mA typical

+3.3 VDC (±5%) ..... 940 mA typical

LIN

+3.3 VDC (±5%) ..... 940 mA typical

Safety

Isolation Voltages

Port-to-port ground

Continuous ..... 60 VDC, Measurement Category I

Port-to-earth ground

Continuous ..... 60 VDC, Measurement Category I

This isolation is intended to prevent ground loops.

Safety Standards

This product meets the requirements of the following standards of safety for electrical equipment for measurement, control, and laboratory use:

- IEC 61010-1, EN 61010-1
- UL 61010-1, CSA 61010-1

Note For UL and other safety certifications, refer to the product label or the *Online Product Certification* section.

Environmental

Operating temperature .....0 to 55 °C

Storage temperature .....–20 to 70 °C (Tested in accordance with IEC-60068-2-1 and IEC-60068-2-2.)

Operating humidity .....10 to 90% RH, noncondensing

Storage humidity.....5 to 95% RH, noncondensing (Tested in accordance with IEC-60068-2-56.)

Maximum altitude.....2000 m

Pollution Degree (IEC 60664) 2

Indoor use only.

C Series XNET

For C Series hardware specifications, refer to your C Series hardware operating instructions.

NI-XNET Transceiver Cables

For NI-XNET Transceiver Cable hardware specifications, refer to your NI-XNET Transceiver Cable hardware operating instructions.

Electromagnetic Compatibility

This product meets the requirements of the following EMC standards for electrical equipment for measurement, control, and laboratory use:

- EN 61326 (IEC 61326): Class A emissions; Basic immunity
- EN 55011 (CISPR 11): Group 1, Class A emissions
- AS/NZS CISPR 11: Group 1, Class A emissions
- FCC 47 CFR Part 15B: Class A emissions
- ICES-001: Class A emissions

Note For the standards applied to assess the EMC of this product, refer to the *Online Product Certification* section.

Note For EMC compliance, operate this product according to the documentation.

Caution When operating this product, use shielded cables and accessories.

CE Compliance CE

This product meets the essential requirements of applicable European Directives as follows:

- 2006/95/EC; Low-Voltage Directive (safety)
- 2004/108/EC; Electromagnetic Compatibility Directive (EMC)

Online Product Certification

Refer to the product Declaration of Conformity (DoC) for additional regulatory compliance information. To obtain product certifications and the DoC for this product, visit `ni.com/certification`, search by model number or product line, and click the appropriate link in the Certification column.

Environmental Management

NI is committed to designing and manufacturing products in an environmentally responsible manner. NI recognizes that eliminating certain hazardous substances from our products is beneficial to the environment and to NI customers.

For additional environmental information, refer to the *NI and the Environment* Web page at ni.com/environment. This page contains the environmental regulations and directives with which NI complies, as well as other environmental information not included in this document.

Waste Electrical and Electronic Equipment (WEEE)



EU Customers At the end of the product life cycle, all products *must* be sent to a WEEE recycling center. For more information about WEEE recycling centers, National Instruments WEEE initiatives, and compliance with WEEE Directive 2002/96/EC on Waste and Electronic Equipment, visit ni.com/environment/weee.

电子信息产品污染控制管理办法 （中国 RoHS）



中国客户 National Instruments 符合中国电子信息产品中限制使用某些有害物质指令 (RoHS)。
关于 National Instruments 中国 RoHS 合规性信息，请登录 ni.com/environment/rohs_china。
(For information about China RoHS compliance, go to ni.com/environment/rohs_china.)

E

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 **XNET Read.vi** and **XNET Write.vi**.

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.vi** 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 **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, **XNET Create Session.vi** is ideal.

F

Bus Monitor

Overview

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 MAX or the NI-XNET Windows **Start** menu category. 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.

G

Database Editor

The NI-XNET Database Editor is a small standalone tool for creating and maintaining embedded network databases. You can use the editor to:

- Configure the basic network
- Define frames and signals exchanged on the network
- Assign frames to ECUs that send and receive them

To launch the Database Editor, go to **Start»All Programs»National Instruments»NI-XNET»Database Editor**.

Why Databases?

Databases are the means of choice for managing your embedded networks. Although it is possible (and supported) in principle to run a network without a database, using a database is highly recommended to have a consistent set of network parameters for all nodes in the network. This is especially true for FlexRay, where you need 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.

Database Formats

For NI-XNET, NI adopted the ASAM FIBEX standard as a database storage format. FIBEX (FIeld Bus EXchange) is a vendor-independent exchange format for embedded network data. It is an XML-based text format. The NI-XNET Database Editor can read and write this format.

In addition, the NI-XNET Database Editor can import the NI-CAN database format (`.ncd`), vector CANdb format (`.dbc`), and LIN description file format (`.ldf`) and convert them to FIBEX.

Clusters

The basic entity of a database is a cluster. A cluster is the description of a single network (for example, a CAN or FlexRay bus).

For CAN, the cluster contains only the baud rate. For FlexRay, there are about 30 global network parameters to set for a cluster. The NI-XNET Database Editor includes an Easy view, where you can set the six most important parameters; the other parameters are then chosen automatically to obtain a functioning network. If you start with FlexRay, this is probably the method of choice. However, if you have an existing database, you can use the Expert view to set individual parameters.

Usually, a database contains only one cluster. For example, the NI-CAN database and Vector CANdb formats support only one cluster. However, FIBEX supports multiple clusters per database; for example, you might describe all of a car's networks in a single database.

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.

PDUs

A Protocol Data Unit (PDU) is a data unit defined in a cluster and exchanged within a frame. Like a frame, a PDU contains an arbitrary number of signals. You can map one or more PDUs to a frame by defining a start bit and update bit in the frame properties window. You can map one PDU to multiple frames.

For CAN and LIN, NI-XNET supports only a one-to-one relationship between frames and PDUs, 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** editor option to hide PDUs in the editor. If the file contains frames with advanced PDU

configuration (using a one-to-*n* or *n*-to-one relationship or update bits), you cannot deactivate **Use PDUs** in the editor.

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

Signals

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**: the signal start position within the frame
- **Number of bits**: the signal length within the frame
- **Data type**: the data type (signed, unsigned, or float)
- **Byte order**: little or big endian
- **Scaling factor and offset**: for converting physical data to binary representation

ECUs

ECUs appear in the NI-XNET Database Editor only as transmitters and receivers of frames within clusters. They are not separate entities. That is, the same ECU might appear in different database clusters, but in the exported FIBEX file, they appear as different ECU entities.

In the LabVIEW Project Provider, you can sort frames by ECUs.

H

NI Services

National Instruments provides global services and support as part of our commitment to your success. Take advantage of product services in addition to training and certification programs that meet your needs during each phase of the application life cycle; from planning and development through deployment and ongoing maintenance.

To get started, register your product at ni.com/myproducts.

As a registered NI product user, you are entitled to the following benefits:

- Access to applicable product services.
- Easier product management with an online account.
- Receive critical part notifications, software updates, and service expirations.

Log in to your National Instruments ni.com User Profile to get personalized access to your services.

Services and Resources

- **Maintenance and Hardware Services—**NI helps you identify your systems' accuracy and reliability requirements and provides warranty, sparing, and calibration services to help you maintain accuracy and minimize downtime over the life of your system. Visit ni.com/services for more information.
  - **Warranty and Repair—**All NI hardware features a one-year standard warranty that is extendable up to five years. NI offers repair services performed in a timely manner by highly trained factory technicians using only original parts at a National Instruments service center.
  - **Calibration—**Through regular calibration, you can quantify and improve the measurement performance of an instrument. NI provides state-of-the-art calibration services. If your product supports calibration, you can obtain the calibration certificate for your product at ni.com/calibration.

- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.
- **Training and Certification—**The NI training and certification program is the most effective way to increase application development proficiency and productivity. Visit ni.com/training for more information.
  - The Skills Guide assists you in identifying the proficiency requirements of your current application and gives you options for obtaining those skills consistent with your time and budget constraints and personal learning preferences. Visit ni.com/skills-guide to see these custom paths.
  - NI offers courses in several languages and formats including instructor-led classes at facilities worldwide, courses on-site at your facility, and online courses to serve your individual needs.
- **Technical Support—**Support at ni.com/support includes the following resources:
- **Self-Help Technical Resources—**Visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
- **Software Support Service Membership—**The Standard Service Program (SSP) is a renewable one-year subscription included with almost every NI software product, including NI Developer Suite. This program entitles members to direct access to NI Applications Engineers through phone and email for one-to-one technical support, as well as exclusive access to online training modules at ni.com/self-paced-training. NI also offers flexible extended contract options that guarantee your SSP benefits are available without interruption for as long as you need them. Visit ni.com/ssp for more information.
- **Declaration of Conformity (DoC)—**A DoC is our claim of compliance with the Council of the European Communities using the manufacturer's declaration of conformity. This system affords the user protection for electromagnetic compatibility (EMC) and product safety. You can obtain the DoC for your product by visiting ni.com/certification.

For information about other technical support options in your area, visit ni.com/services, or contact your local office at ni.com/contact.

You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office websites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Index

A

acknowledgement error, A-8
Acknowledgment Bit (ACK), A-5
advanced database example using property nodes (figure), 4-11
Advanced subpalette, 4-499
alias, definition of
  in C, 5-5
  in LabVIEW, 4-8
API
  for C, 5-1
  for LabVIEW, 4-1
  reference
    NI-XNET API for C, 5-47
    NI-XNET API for LabVIEW, 4-61
Application Protocol
  in C, 5-197, 5-246
  in LabVIEW, 4-319, 4-351
Arbitration ID, A-4
Auto Start?
  in C, 5-372
  in LabVIEW, 4-185

B

basic programming model, NI-XNET API, in LabVIEW, 4-3
Baud Rate
  in C, 5-147
  in LabVIEW, 4-320
bit error, A-7
bit rate switch bit (BRS), A-6
Break field in LIN, C-2
BRS (bit rate switch bit), A-6
built application, creating using a LabVIEW project, 4-560
Bus Monitor, F-1
bus off state, A-9
bus power requirements
  CAN hardware
    High-Speed physical layer, 3-4
    Low-Speed/Fault-Tolerant physical layer, 3-8
    Single Wire physical layer, 3-12
  FlexRay hardware, 3-1
  LIN hardware, 3-15
Byte Order
  in C, 5-387
  in LabVIEW, 4-394

C

C Series modules firmware update, 2-5
cable connecting two CAN devices (figure), 3-7
cable lengths
  CAN hardware
    High-Speed, 3-5
    Single Wire physical layer, 3-12
  FlexRay hardware, 3-2
  LIN hardware, 3-15
cable termination, CAN hardware
  High-Speed, 3-5
  High-Speed termination resistor placement (figure), 3-6
  Low-Speed/Fault-Tolerant, 3-9
  Low-Speed/Fault-Tolerant termination resistor placement (figure), 3-9
cabling example, High-Speed CAN hardware, 3-7
cabling requirements
  CAN hardware
    High-Speed, 3-5
    Low-Speed/Fault-Tolerant, 3-8
    Single Wire physical layer, 3-12

FlexRay hardware, 3-1
LIN hardware, 3-15
CAN
DB9 pinouts
C Series NI-XNET (table), 3-14
PXI and PCI NI-XNET
external CAN transceiver (table), 3-13
interfaces (table), 3-13
error detection and confinement, A-7
external transceiver, 3-12
FD frames, A-5
Frame properties
in C, 5-368
in LabVIEW, 4-180
frames, A-3
hardware, 3-3
High-Speed physical layer
bus power requirements, 3-4
cable example, 3-7
cable lengths, 3-5
cable termination, 3-5
cabling requirements, 3-5
number of devices, 3-5
transceiver, 3-4
Low-Speed/Fault-Tolerant physical layer
bus power requirements, 3-8
cabling requirements, 3-8
determining necessary termination resistance for a board, 3-10
number of devices, 3-9
termination, 3-9
transceiver, 3-7
Single Wire physical layer
bus power requirements, 3-12
cable lengths, 3-12
cabling requirements, 3-12
number of devices, 3-12
termination (bus loading), 3-12
transceiver, 3-11
history and use, A-1
identifiers and message priority, A-2
interface configuration, 2-7
Interface properties
in C, 5-273
in LabVIEW, 4-83
Low-Speed, A-10
NI-XNET transceiver cables, 3-3
Single Wire, A-11
summary of CAN standard, A-1
using with NI-XNET API for LabVIEW
configuring frame I/O stream sessions, 4-49
understanding CAN frame timing, 4-48
XS software selectable physical layer, 3-3
CAN FD standard and extended frame formats (figure), A-5
CAN frames
Acknowledgment Bit (ACK), A-5
Arbitration ID, A-4
CAN FD standard and extended frame formats (figure), A-5
Cyclic Redundancy Check (CRC), A-4
Data Bytes, A-4
Data Length Code (DLC), A-4
End of Frame, A-5
Identifier Extension (IDE), A-4
Remote Transmit Request (RTR), A-4
standard and extended frame formats (figure), A-3
Start of Frame (SOF), A-4
CAN timing type and session mode
in C, 5-443
cyclic data, 5-443
cyclic remote, 5-445
event data, 5-445
event remote, 5-446

in LabVIEW
cyclic data, 4-595
cyclic remote, 4-596
event data, 4-596
event remote, 4-597
CAN:Extended Identifier?
in C, 5-215
in LabVIEW, 4-347
CAN:FD Baud Rate
in C, 5-148
in LabVIEW, 4-321
CAN:I/O Mode
in C, 5-149
in LabVIEW, 4-322
CAN:Timing Type
in C, 5-216
in LabVIEW, 4-348
CAN:Transmit Time
in C, 5-218
in LabVIEW, 4-350
CAN.Termination Capability
in C, 5-247
in LabVIEW, 4-528
CAN.Transceiver Capability
in C, 5-248
in LabVIEW, 4-529
CE compliance specifications, D-9
changing LIN schedule, 4-51
Checksum field in LIN, C-4
clock synchronization in FlexRay, B-9
Cluster
XNET ECU
in C, 5-204
in LabVIEW, 4-335
XNET Frame
in C, 5-219
in LabVIEW, 4-352
XNET LIN Schedule
in C, 5-253
in LabVIEW, 4-467
XNET PDU
in C, 5-266
in LabVIEW, 4-379
XNET Session, 4-186
ClusterName, 5-373
Clusters
in C, 5-198
in LabVIEW, 4-279
clusters, in databases, G-2
coldstart nodes in FlexRay, B-6
Collision Resolving Schedule
XNET LIN Schedule Entry
in C, 5-258
in LabVIEW, 4-475
Comment
XNET Cluster
in C, 5-150
in LabVIEW, 4-323
XNET ECU
in C, 5-204
in LabVIEW, 4-339
XNET Frame
in C, 5-219
in LabVIEW, 4-352
XNET LIN Schedule
in C, 5-253
in LabVIEW, 4-468
XNET PDU
in C, 5-266
in LabVIEW, 4-379
XNET Signal
in C, 5-389
in LabVIEW, 4-396
common questions, 6-1
Communication Cycle in FlexRay, B-5
CompactRIO, getting started with, 2-8
configuration, 2-1
in LabVIEW Real-Time (RT), 2-7
NI-XNET interfaces, 2-7
NI-XNET tools, 2-12

Configuration Status
XNET Cluster
in C, 5-150
in LabVIEW, 4-323
XNET ECU
in C, 5-205
in LabVIEW, 4-339
XNET Frame
in C, 5-220
in LabVIEW, 4-353
XNET LIN Schedule
in C, 5-254
in LabVIEW, 4-469
XNET PDU
in C, 5-267
in LabVIEW, 4-380
XNET Signal
in C, 5-390
in LabVIEW, 4-397
configuring frame I/O stream sessions, 4-49
Controls palette, 4-558
Conversion Mode
in C, 5-38
in LabVIEW, 4-45
CRC (cyclic redundancy check sequence), A-6
CRC error, A-8
creating a built application using a LabVIEW project
databases, 4-560
NI-XNET sessions, 4-560
creating a built real-time application in LabVIEW Real-Time (RT), 4-55
creating cluster and frame for CAN (figure), 4-13
cyclic data
CAN
in C, 5-443
in LabVIEW, 4-595
FlexRay
in C, 5-449
in LabVIEW, 4-601
Cyclic Redundancy Check (CRC), A-4
cyclic redundancy check sequence (CRC), A-6
cyclic remote
in C, 5-445
in LabVIEW, 4-596
cyclic timing
CAN
in C, 5-418
in LabVIEW, 4-561
FlexRay
in C, 5-419
in LabVIEW, 4-562
in C, 5-418
in LabVIEW, 4-561
LIN
in C, 5-419
in LabVIEW, 4-562

D

Data Bytes, A-4
Data Length Code (DLC), A-4
Data Payload field in LIN, C-3
data security in FlexRay, B-3
Data Type
in C, 5-391
in LabVIEW, 4-398
Database
in C, 5-151
in LabVIEW, 4-324
database classes, 4-614
database controls, 4-558
Database Editor
clusters, G-2
database formats, G-1
ECUs, G-3
frames, G-2
PDUs, G-2
reasons to use databases, G-1
signals, G-3

database programming
NI-XNET API for C
creating a new file using NI-XNET Database Editor, 5-7
creating in memory, 5-7
editing and selecting
editing in file, 5-7
editing in memory, 5-7
selecting from file, 5-7
using a file, 5-7
using existing file, 5-6
using existing file as is, 5-6
NI-XNET API for LabVIEW
creating a new file using NI-XNET Database Editor, 4-12
creating in memory, 4-12
creating cluster and frame for CAN (figure), 4-13
editing and selecting
editing in file, 4-12
editing in memory, 4-11
multiple databases simultaneously, 4-13
selecting from file
advanced database example using property nodes (figure), 4-11
using I/O names, 4-10
using LabVIEW project, 4-10
using a file, 4-12
using existing file, 4-9
using existing file as is, 4-9
Database subpalette, 4-278
DatabaseName, 5-373
databases
definition
in C, 5-4
in LabVIEW, 4-7
deploying in LabVIEW Real-Time (RT), 4-54
formats, G-1
reasons to use, G-1
using
with NI-XNET API for C, 5-4
with NI-XNET API for LabVIEW, 4-7
Default Payload
in C, 5-221
in LabVIEW, 4-354
Default Value
in C, 5-392
in LabVIEW, 4-399
Delay, XNET LIN Schedule Entry
in C, 5-258
in LabVIEW, 4-476
deploying databases in LabVIEW Real-Time (RT), 4-54
determining necessary termination resistance for Low-Speed/Fault-Tolerant CAN hardware, 3-10
Device
in C, 5-249
in LabVIEW, 4-530
Devices
in C, 5-410
in LabVIEW, 4-515
documentation
related documentation, *xxxiii*
Dynamic Signals
in C, 5-405
in LabVIEW, 4-388

E

ECUs
in C, 5-151
in databases, G-3
in LabVIEW, 4-324
EDL (extended data length bit), A-6
electromagnetic compatibility specifications, D-9
End of Frame, A-5

Entries, XNET LIN Schedule
in C, 5-255
in LabVIEW, 4-470
environmental management
specifications, D-9
error active state, A-9
error detection and confinement, CAN, A-7
acknowledgement error, A-8
bit error, A-7
bus off state, A-9
CRC error, A-8
error active state, A-9
error passive state, A-9
form error, A-8
stuff error, A-7
error handling
in FlexRay, B-3
in LabVIEW, 4-562
error passive state, A-9
error state indicator bit (ESI), A-6
ESI (error state indicator bit), A-6
event data
CAN
in C, 5-445
in LabVIEW, 4-596
FlexRay
in C, 5-451
in LabVIEW, 4-602
Event Identifier, XNET LIN Schedule Entry
in C, 5-259
in LabVIEW, 4-476
event remote
in C, 5-446
in LabVIEW, 4-597
event timing
CAN
in C, 5-418
in LabVIEW, 4-561
FlexRay
in C, 5-419
in LabVIEW, 4-562
in C, 5-418
in LabVIEW, 4-561
LIN
in C, 5-419
in LabVIEW, 4-562
examples
NI-XNET API for LabVIEW, 4-1
extended data length bit (EDL), A-6
external CAN transceiver, 3-12

F

fault handling in LabVIEW, 4-563
FD frames, CAN, A-5
FIBEX database format, definition, G-1
File Management subpalette, 4-459
firmware update, XNET C series modules, 2-5
FlexRay
bus benefits, B-3
cable characteristics (table), 3-2
clock synchronization, B-9
coldstart nodes, B-6
Communication Cycle, B-5
data security, B-3
DB9 pinouts (table), 3-2
error handling, B-3
frame format, B-10
hardware, 3-1
bus power requirements, 3-1
cable lengths, 3-2
cabling requirements, 3-1
number of devices, 3-2
physical layer, 3-1
termination, 3-2
transceiver, 3-1
interface configuration, 2-7
Interface properties
in C, 5-291
in LabVIEW, 4-102
network, B-3
overview, B-1

Protocol Data Units, 4-51
protocol operation control, B-4
startup, B-6
path of following coldstart node, B-9
path of leading coldstart node, B-8
path of non-coldstart node, B-9
startup state machine (figure), B-7
state transitions (figure), B-8
summary of FlexRay standard, B-1
timing source in LabVIEW Real-Time (RT), 4-55
timing type and session mode
in C
cyclic data, 5-449
event data, 5-451
in LabVIEW
cyclic data, 4-601
event data, 4-602
using with NI-XNET API for LabVIEW
starting communication, 4-50
understanding FlexRay frame timing, 4-51
FlexRay startup/wakeup
in C, 5-455
in LabVIEW, 4-606
FlexRay:Action Point Offset
in C, 5-152
in LabVIEW, 4-282
FlexRay:Base Cycle
in C, 5-223
in LabVIEW, 4-356
FlexRay:CAS Rx Low Max
in C, 5-153
in LabVIEW, 4-283
FlexRay:Channel Assignment
in C, 5-225
in LabVIEW, 4-358
FlexRay:Channels
in C, 5-154
in LabVIEW, 4-284
FlexRay:Cluster Drift Damping
in C, 5-155
in LabVIEW, 4-285
FlexRay:Cold Start Attempts
in C, 5-156
in LabVIEW, 4-286
FlexRay:Coldstart?
in C, 5-206
in LabVIEW, 4-335
FlexRay:Connected Channels
in C, 5-206
in LabVIEW, 4-336
FlexRay:Cycle
in C, 5-157
in LabVIEW, 4-287
FlexRay:Cycle Repetition
in C, 5-226
in LabVIEW, 4-359
FlexRay:Dynamic Segment Start
in C, 5-158
in LabVIEW, 4-288
FlexRay:Dynamic Slot Idle Phase
in C, 5-159
in LabVIEW, 4-289
FlexRay:In Cycle Repetitions:Channel Assignments
in C, 5-228
in LabVIEW, 4-365
FlexRay:In Cycle Repetitions:Enabled?
in C, 5-229
in LabVIEW, 4-366
FlexRay:In Cycle Repetitions:Identifiers
in C, 5-230
in LabVIEW, 4-367
FlexRay:Latest Guaranteed Dynamic Slot
in C, 5-160
in LabVIEW, 4-290
FlexRay:Latest Usable Dynamic Slot
in C, 5-161
in LabVIEW, 4-291

FlexRay:Listen Noise
in C, 5-162
in LabVIEW, 4-292
FlexRay:Macro Per Cycle
in C, 5-163
in LabVIEW, 4-293
FlexRay:Macrotick
in C, 5-164
in LabVIEW, 4-294
FlexRay:Max Without Clock Correction Fatal
in C, 5-165
in LabVIEW, 4-295
FlexRay:Max Without Clock Correction Passive
in C, 5-166
in LabVIEW, 4-296
FlexRay:Minislot
in C, 5-167
in LabVIEW, 4-298
FlexRay:Minislot Action Point Offset
in C, 5-168
in LabVIEW, 4-297
FlexRay:Network Management Vector Length
in C, 5-169
in LabVIEW, 4-299
FlexRay:NIT
in C, 5-170
in LabVIEW, 4-301
FlexRay:NIT Start
in C, 5-171
in LabVIEW, 4-300
FlexRay:Number of Minislots
in C, 5-172
in LabVIEW, 4-302
FlexRay:Number of Static Slots
in C, 5-173
in LabVIEW, 4-303
FlexRay:Offset Correction Start
in C, 5-174
in LabVIEW, 4-304
FlexRay:Payload Length Dynamic Maximum
in C, 5-175
in LabVIEW, 4-305
FlexRay:Payload Length Maximum
in C, 5-176
in LabVIEW, 4-306
FlexRay:Payload Length Static
in C, 5-177
in LabVIEW, 4-307
FlexRay:Payload Preamble?
in C, 5-231
in LabVIEW, 4-361
FlexRay:Startup Frame
in C, 5-207
in LabVIEW, 4-336
FlexRay:Startup?
in C, 5-232
in LabVIEW, 4-362
FlexRay:Static Slot
in C, 5-178
in LabVIEW, 4-308
FlexRay:Symbol Window
in C, 5-179
in LabVIEW, 4-310
FlexRay:Symbol Window Start
in C, 5-180
in LabVIEW, 4-309
FlexRay:Sync Node Max
in C, 5-181
in LabVIEW, 4-311
FlexRay:Sync?
in C, 5-233
in LabVIEW, 4-363
FlexRay:Timing Type
in C, 5-234
in LabVIEW, 4-364
FlexRay:TSS Transmitter
in C, 5-182
in LabVIEW, 4-312

FlexRay:Use Wakeup
in C, 5-183
in LabVIEW, 4-313
FlexRay:Wakeup Channels
in C, 5-207
in LabVIEW, 4-337
FlexRay:Wakeup Pattern
in C, 5-208
in LabVIEW, 4-337
FlexRay:Wakeup Symbol Rx Idle
in C, 5-184
in LabVIEW, 4-314
FlexRay:Wakeup Symbol Rx Low
in C, 5-185
in LabVIEW, 4-315
FlexRay:Wakeup Symbol Rx Window
in C, 5-186
in LabVIEW, 4-316
FlexRay:Wakeup Symbol Tx Idle
in C, 5-187
in LabVIEW, 4-317
FlexRay:Wakeup Symbol Tx Low
in C, 5-188
in LabVIEW, 4-318
form error, A-8
Form Factor
in C, 5-200
in LabVIEW, 4-523
Frame
in C, 5-393, 5-406
in LabVIEW, 4-388, 4-401
frame format in FlexRay, B-10
Frame Input Queued mode
in C, 5-10
example, 5-10
in LabVIEW, 4-15
example, 4-16
Frame Input Single-Point mode
in C, 5-12
example, 5-12
in LabVIEW, 4-18
example, 4-18
Frame Input Stream mode
in C, 5-13
example, 5-14
in LabVIEW, 4-19
example, 4-20
Frame Output Queued mode
in C, 5-16
examples, 5-16
in LabVIEW, 4-22
examples, 4-22
Frame Output Single-Point mode
in C, 5-18
example, 5-19
in LabVIEW, 4-24
example, 4-25
Frame Output Stream mode
in C, 5-21
example, 5-21
in LabVIEW, 4-27
example, 4-28
Frame properties
CAN
in C, 5-368
in LabVIEW, 4-180
in C, 5-368
in LabVIEW, 4-180
frame timing in LabVIEW
CAN, 4-48
FlexRay, 4-51
LIN, 4-52
Frame:Active
in LabVIEW, 4-182
Frame:CAN:Start Time Offset
in C, 5-368
in LabVIEW, 4-180

Frame:CAN:Transmit Time
in C, 5-369
in LabVIEW, 4-181
Frame:LIN:Transmit N Corrupted Checksums
in C, 5-370
in LabVIEW, 4-183
Frame:Skip N Cyclic Frames
in C, 5-371
in LabVIEW, 4-184
Frames
XNET Cluster
in C, 5-189
in LabVIEW, 4-325
XNET LIN Schedule Entry
in C, 5-260
in LabVIEW, 4-477
XNET PDU
in C, 5-268
in LabVIEW, 4-381
Frames Received
in C, 5-208
in LabVIEW, 4-340
Frames Transmitted
in C, 5-209
in LabVIEW, 4-340
frames, CAN, A-3
in databases, G-2

G

getting started
with Compact RIO, 2-8
with NI-XNET API
for C, 5-1
for LabVIEW, 4-1

H

hardware overview, 3-1
high-priority loops in LabVIEW Real-Time (RT), 4-53
High-Speed physical layer, CAN, 3-4
history and use of CAN, A-1

I

I/O name classes
database classes, 4-614
session, 4-614
system classes, 4-615
I/O name, viewing available interfaces in, 4-6
ID field in LIN, C-3
Identifier
in C, 5-235
in LabVIEW, 4-368
Identifier Extension (IDE), A-4
increasing communication demands, B-2
installation, 2-1
verifying NI-XNET hardware installation, 2-4
XNET C Series modules firmware update, 2-5
Interface properties
CAN
in C, 5-273
in LabVIEW, 4-83
FlexRay
in C, 5-291
in LabVIEW, 4-102
in C, 5-273
in LabVIEW, 4-83
LIN
in C, 5-330
in LabVIEW, 4-141
source terminal
in C, 5-341
in LabVIEW, 4-152

interface state model
in C, 5-435
in LabVIEW, 4-581
interface states
in C, 5-438
in LabVIEW, 4-584
interface transitions
in C, 5-439
in LabVIEW, 4-585
Interface:Baud Rate
in C, 5-342
in LabVIEW, 4-153
Interface:Bus Error Frames to Input Stream?
in C, 5-353
in LabVIEW, 4-164
Interface:CAN:External Transceiver Config
in C, 5-273
in LabVIEW, 4-84
Interface:CAN:FD Baud Rate
in C, 5-276
in LabVIEW, 4-87
Interface:CAN:I/O Mode
in C, 5-278
in LabVIEW, 4-89
Interface:CAN:Listen Only?
in C, 5-279
in LabVIEW, 4-90
Interface:CAN:Pending Transmit Order
in C, 5-280
in LabVIEW, 4-91
Interface:CAN:Single Shot Transmit?
in C, 5-282
in LabVIEW, 4-93
Interface:CAN:Termination
in C, 5-283
in LabVIEW, 4-94
Interface:CAN:Transceiver State
in C, 5-285
in LabVIEW, 4-96
Interface:CAN:Transceiver Type
in C, 5-288
in LabVIEW, 4-99
Interface:CAN:Transmit I/O Mode
in C, 5-290
in LabVIEW, 4-101
Interface:Echo Transmit?
in C, 5-345
in LabVIEW, 4-156
Interface:FlexRay:Accepted Startup Range
in C, 5-291
in LabVIEW, 4-102
Interface:FlexRay:Allow Halt Due To Clock?
in C, 5-292
in LabVIEW, 4-103
Interface:FlexRay:Allow Passive to Active
in C, 5-293
in LabVIEW, 4-104
Interface:FlexRay:Auto Asleep When Stopped
in LabVIEW, 4-105
Interface:FlexRay:AutoAsleepWhenStopped
in C, 5-294
Interface:FlexRay:Cluster Drift Damping
in C, 5-295
in LabVIEW, 4-106
Interface:FlexRay:Coldstart?
in C, 5-296
in LabVIEW, 4-107
Interface:FlexRay:Connected Channels
in C, 5-297
in LabVIEW, 4-108
Interface:FlexRay:Decoding Correction
in C, 5-298
in LabVIEW, 4-109
Interface:FlexRay:Delay Compensation Ch A
in C, 5-299
in LabVIEW, 4-110
Interface:FlexRay:Delay Compensation Ch B
in C, 5-300
in LabVIEW, 4-111

Interface:FlexRay:Key Slot Identifier
in C, 5-301
in LabVIEW, 4-112
Interface:FlexRay:Latest Tx
in C, 5-303
in LabVIEW, 4-114
Interface:FlexRay:Listen Timeout
in C, 5-304
in LabVIEW, 4-115
Interface:FlexRay:Macro Initial Offset Ch A
in C, 5-305
in LabVIEW, 4-116
Interface:FlexRay:Macro Initial Offset Ch B
in C, 5-306
in LabVIEW, 4-117
Interface:FlexRay:Max Drift
in C, 5-307
in LabVIEW, 4-118
Interface:FlexRay:Micro Initial Offset Ch A
in C, 5-308
in LabVIEW, 4-119
Interface:FlexRay:Micro Initial Offset Ch B
in C, 5-309
in LabVIEW, 4-120
Interface:FlexRay:Microtick
in C, 5-310
in LabVIEW, 4-121
Interface:FlexRay:Null Frames To Input Stream?
in C, 5-311
in LabVIEW, 4-122
Interface:FlexRay:Offset Correction
in C, 5-312
in LabVIEW, 4-123
Interface:FlexRay:Offset Correction Out
in C, 5-313
in LabVIEW, 4-124
Interface:FlexRay:Rate Correction
in C, 5-314
in LabVIEW, 4-125
Interface:FlexRay:Rate Correction Out
in C, 5-315
in LabVIEW, 4-126
Interface:FlexRay:Samples Per Microtick
in C, 5-316
in LabVIEW, 4-127
Interface:FlexRay:Single Slot Enabled?
in C, 5-317
in LabVIEW, 4-128
Interface:FlexRay:Sleep
in C, 5-318
in LabVIEW, 4-129
Interface:FlexRay:Statistics Enabled?
in C, 5-320
in LabVIEW, 4-131
Interface:FlexRay:Symbol Frames To Input Stream?
in C, 5-321
in LabVIEW, 4-132
Interface:FlexRay:Sync Frame Status
in C, 5-322
in LabVIEW, 4-137
Interface:FlexRay:Sync Frames Channel A Even
in C, 5-323
in LabVIEW, 4-133
Interface:FlexRay:Sync Frames Channel A Odd
in C, 5-324
in LabVIEW, 4-134
Interface:FlexRay:Sync Frames Channel B Even
in C, 5-325
in LabVIEW, 4-135
Interface:FlexRay:Sync Frames Channel B Odd
in C, 5-326
in LabVIEW, 4-136
Interface:FlexRay:Termination
in C, 5-327
in LabVIEW, 4-138

Interface:FlexRay:Wakeup Channel
in C, 5-328
in LabVIEW, 4-139
Interface:FlexRay:Wakeup Pattern
in C, 5-329
in LabVIEW, 4-140
Interface:I/O Name, 4-157
Interface:LIN:Break Length
in C, 5-330
in LabVIEW, 4-141
Interface:LIN:DiagP2min
in C, 5-331
in LabVIEW, 4-142
Interface:LIN:DiagSTmin
in C, 5-332
in LabVIEW, 4-143
Interface:LIN:Master?
in C, 5-333
in LabVIEW, 4-144
Interface:LIN:Output Stream Slave Response List By NAD
in C, 5-334
in LabVIEW, 4-145
Interface:LIN:Schedule Names
in C, 5-335
Interface:LIN:Schedules
in LabVIEW, 4-146
Interface:LIN:Sleep
in C, 5-336
in LabVIEW, 4-147
Interface:LIN:Start Allowed without Bus Power?
in C, 5-339
in LabVIEW, 4-150
Interface:LIN:Termination
in C, 5-340
in LabVIEW, 4-151
Interface:Output Stream List
in C, 5-346
in LabVIEW, 4-158
Interface:Output Stream List By ID
in C, 5-347
in LabVIEW, 4-159
Interface:Output Stream Timing
in C, 5-348
in LabVIEW, 4-160
Interface:Source Terminal:Start Trigger
in C, 5-341
in LabVIEW, 4-152
Interface:Start Trigger Frames to Input Stream?
in C, 5-352
in LabVIEW, 4-164
Interfaces
in C, 5-201
in LabVIEW, 4-524
interfaces
definition
in C, 5-3
in LabVIEW, 4-4
in NI-XNET API
for C, 5-3
for LabVIEW, 4-4
viewing
in C, 5-4
in LabVIEW, 4-5
Interfaces (All)
in C, 5-411
in LabVIEW, 4-516
Interfaces (CAN)
in C, 5-411
in LabVIEW, 4-516
Interfaces (FlexRay)
in C, 5-412
in LabVIEW, 4-515
Interfaces (LIN)
in C, 5-412
in LabVIEW, 4-517
introduction, 1-1
in-vehicle application requirements comparison (figure), B-2

ISO 11898 specifications for characteristics of a CAN_H and CAN_L pair of wires (table), 3-5
isolation, 3-17

J

J1939
address claiming procedure, 4-59, 5-44
basics, 4-57, 5-42
compatibility issue, 4-56, 5-41
functionality not supported, 4-60, 5-46
mixing J1939 and CAN messages, 4-59, 5-45
NI-XNET sessions, 4-60, 5-46
node address in NI-XNET, 4-58, 5-43
sessions, 4-55, 5-41
signal ranges, 4-60, 5-46
transmitting frames, 4-59, 5-45
without granted node address, 4-59, 5-45
transport protocol, 4-60, 5-45

L

LabVIEW project, 4-1
viewing available interfaces in, 4-6
LabVIEW project provider, E-1
LabVIEW Real-Time (RT)
configuration in, 2-7
using with NI-XNET API for LabVIEW
creating a built real-time application, 4-55
deploying databases, 4-54
FlexRay timing source, 4-55
high-priority loops, 4-53
memory use for databases, 4-54
XNET I/O names, 4-54
LabWindows/CVI, 5-1
examples, 5-1
LEDs, 3-18
LIN
additional information, C-7
advanced frame types, C-6
Break field, C-2
bus timing, C-5
C Series DB9 pinouts (table), 3-17
Checksum field, C-4
Data Payload field, C-3
diagnostics, 4-52
error detection and confinement, C-5
frame format, C-2
frame timing and session mode
in C, 5-457
in LabVIEW, 4-609
full frames, creation (figure), C-4
hardware, 3-14
bus power requirements, 3-15
cable lengths, 3-15
cabling requirements, 3-15
number of devices, 3-16
physical layer, 3-14
termination, 3-16
transceiver, 3-15
history and use, C-1
ID field, C-3
interface configuration, 2-7
Interface properties
in C, 5-330
in LabVIEW, 4-141
PXI and PCI DB9 pinouts (table), 3-16
schedule, changing, 4-51
sleep and wakeup, C-6
Steam Output Mode, using with, 4-52
summary of LIN standard, C-1
Sync field, C-3
topology and behavior, C-1
using with NI-XNET API for LabVIEW, 4-51
LIN diagnostics, 4-52

Stream Output Mode, using with LIN, 4-52
understanding LIN frame timing, 4-52
LIN Master
XNET ECU
in C, 5-209
LIN Version
XNET ECU
in C, 5-210
LIN:Checksum
in C, 5-237
in LabVIEW, 4-370
LIN:Configured NAD
in C, 5-211
in LabVIEW, 4-342
LIN:Function ID
in C, 5-212
in LabVIEW, 4-343
LIN:Initial NAD
in C, 5-210
in LabVIEW, 4-342
LIN:Master?
in LabVIEW, 4-341
LIN:P2min
in C, 5-212
in LabVIEW, 4-344
LIN:Protocol Version
in LabVIEW, 4-341
LIN:Schedules
in LabVIEW, 4-326
LIN:STmin
in C, 5-213
in LabVIEW, 4-344
LIN:Supplier ID
in C, 5-211
in LabVIEW, 4-343
LIN:Tick
in LabVIEW, 4-327
List, 5-374
List of Frames, 4-188
List of Signals, 4-189
Low-Speed CAN, A-10
Low-Speed/Fault-Tolerant physical layer, 3-7

M

Maximum Value
in C, 5-393
in LabVIEW, 4-401
Measurement & Automation Explorer
configuring NI-XNET hardware and software in, 2-3
viewing interfaces
in C, 5-4
in LabVIEW, 4-5
memory use for databases in LabVIEW Real-Time (RT), 4-54
Minimum Value
in C, 5-394
in LabVIEW, 4-402
Mode
in C, 5-374
in LabVIEW, 4-190
multiplexed signals
in C, 5-419
creating, 5-420
reading, 5-420
support for, 5-420
writing, 5-420
in LabVIEW, 4-565
creating, 4-566
reading, 4-566
support for, 4-566
writing, 4-566
Multiplexer Value
in C, 5-407
in LabVIEW, 4-389
Mux:Data Multiplexer Signal
XNET Frame
in C, 5-238
in LabVIEW, 4-371

XNET PDU
in C, 5-268
in LabVIEW, 4-381
Mux:Data Multiplexer?
in C, 5-395
in LabVIEW, 4-403
Mux:Dynamic?
in C, 5-396
in LabVIEW, 4-400
Mux:Is Data Multiplexed?
XNET Frame
in C, 5-238
in LabVIEW, 4-371
XNET PDU
in C, 5-269
in LabVIEW, 4-382
Mux:Multiplexer Value
XNET Frame
in LabVIEW, 4-402
XNET Signal
in C, 5-397
Mux:Static Signals
XNET Frame
in C, 5-239
in LabVIEW, 4-372
XNET PDU
in C, 5-269
in LabVIEW, 4-382
Mux:Subframe
in C, 5-397
in LabVIEW, 4-411
Mux:Subframes
XNET Frame
in C, 5-239
in LabVIEW, 4-372
XNET PDU
in C, 5-270
in LabVIEW, 4-383

N

Name
XNET Interface
in C, 5-249
in LabVIEW, 4-530
XNET LIN Schedule
in C, 5-255
XNET LIN Schedule Entry
in C, 5-261
Name (Short)
XNET Cluster
in C, 5-190
in LabVIEW, 4-328
XNET ECU
in C, 5-214
in LabVIEW, 4-345
XNET Frame
in C, 5-240
in LabVIEW, 4-373
XNET LIN Schedule
in LabVIEW, 4-471
XNET LIN Schedule Entry
in LabVIEW, 4-478
XNET PDU
in C, 5-270
in LabVIEW, 4-384
XNET Signal
in C, 5-398
in LabVIEW, 4-404
XNET Subframe
in C, 5-408
in LabVIEW, 4-390
Name Unique to Cluster
XNET LIN Schedule Entry
in C, 5-262
XNET Signal
in C, 5-399
XNET Subframe
in C, 5-409

NI-CAN
in C
CAN timing type and session mode, 5-443
CAN transceiver state machine, 5-447
compatibility, 5-441
NI-XNET CAN products in MAX, 5-441
transition, 5-442
in LabVIEW
CAN timing type and session mode, 4-594
CAN transceiver state machine, 4-598
compatibility, 4-592
NI-XNET CAN products in MAX, 4-592
transition, 4-593
terms (table), 4-594, 5-442
NI-CAN and NI-XNET terms comparison (table), 4-594, 5-442
NI-XNET
Bus Monitor, F-1
Database Editor
clusters, G-2
database formats, G-1
ECUs, G-3
frames, G-2
PDUs, G-2
reasons to use databases, G-1
signals, G-3
hardware overview, 3-1
programming PDUs with
in C, 5-454
in LabVIEW, 4-605
System Configuration API, 2-13
terms (table), 4-594, 5-442
tools, 2-12
NI-XNET API
for C
API reference
functions, 5-47
properties, 5-147
CAN timing type and session mode, 5-443
CAN transceiver state machine, 5-447
cyclic and event timing
CAN, 5-418
FlexRay, 5-419
LIN, 5-419
databases, 5-4
getting started, 5-1
interface states, 5-438
interface transitions, 5-439
interfaces, 5-3
LIN frame timing and session mode, 5-457
multiplexed signals, 5-419
creating, 5-420
reading, 5-420
support for, 5-420
writing, 5-420
NI-CAN
compatibility, 5-441
NI-XNET CAN products in MAX, 5-441
transition, 5-442
raw frame format
base unit, 5-421
payload unit, 5-429
required properties, 5-432
session, 5-8
session states, 5-436
session transitions, 5-437
special frames, 5-429
Bus Error frame, 5-430
Delay frame, 5-429
Log Trigger frame, 5-429

Start Trigger frame, 5-430
state models
interface state model, 5-435
session state model, 5-435
for LabVIEW, 4-1, 4-50, 4-51
API reference, 4-61
basic programming model, 4-3
CAN timing type and session mode, 4-594
CAN transceiver state machine, 4-598
creating a built application, 4-560
cyclic and event timing
CAN, 4-561
FlexRay, 4-562
LIN, 4-562
databases, 4-7
error handling, 4-562
fault handling, 4-563
getting started, 4-1
I/O name classes, 4-614
interface states, 4-584
interface transitions, 4-585
interfaces, 4-4
J1939 sessions, 4-55, 5-41
LIN frame timing and session mode, 4-609
multiplexed signals, 4-565
creating, 4-566
reading, 4-566
support for, 4-566
writing, 4-566
NI-CAN
compatibility, 4-592
NI-XNET CAN products in MAX, 4-592
transition, 4-593
raw frame format
base unit, 4-567
payload unit, 4-572
required properties, 4-577
session, 4-13
session states, 4-581
session transitions, 4-582
special frames, 4-572
Bus Error frame, 4-576
Delay frame, 4-572
Lot Trigger frame, 4-572
Start Trigger frame, 4-574
state models
interface state model, 4-581
session state model, 4-580
Technical Data Management Streaming (TDMS)
using CAN, 4-48
using LabVIEW Real-Time (RT), 4-53
XNET Cluster I/O name, 4-615
XNET Database I/O name, 4-618
XNET Device I/O name, 4-621
XNET ECU I/O name, 4-621
XNET Frame I/O name, 4-624
XNET I/O names, 4-613
XNET Interface I/O name, 4-627
XNET LIN Schedule Entry I/O name, 4-637
XNET LIN Schedule I/O name, 4-635
XNET PDU I/O name, 4-638
XNET Session I/O name, 4-628
XNET Signal I/O name, 4-630
XNET Subframe I/O name, 4-633
XNET Terminal I/O name, 4-634
NI-XNET transceiver cables, 3-3
Node Configuration:Free Format:Data Bytes
XNET LIN Schedule Entry
in C, 5-263
in LabVIEW, 4-479
Notify subpalette, 4-484

Number
in C, 5-250
in LabVIEW, 4-531
Number in List
in C, 5-375
in LabVIEW, 4-190
Number of Bits
in C, 5-400
in LabVIEW, 4-406
number of CAN hardware devices
High-Speed, 3-5
Low-Speed/Fault-Tolerant, 3-9
Single Wire physical layer, 3-12
number of devices
FlexRay hardware, 3-2
number of LIN hardware devices, 3-16
Number of Ports
in C, 5-201
in LabVIEW, 4-525
Number of Values Pending
in C, 5-375
in LabVIEW, 4-191
Number of Values Unused
in C, 5-377
in LabVIEW, 4-192
nxBlink, 5-47
nxClear, 5-49
nxConnectTerminals, 5-50
nxConvertFramesToSignalsSinglePoint, 5-57
nxConvertSignalsToFramesSinglePoint, 5-59
nxCreateSession, 5-61
nxCreateSessionByRef, 5-66
nxdbAddAlias, 5-68
nxdbCloseDatabase, 5-70
nxdbCreateObject, 5-71
nxdbDeleteObject, 5-73
nxdbDeploy, 5-74
nxdbFindObject, 5-76
nxdbGetDatabaseList, 5-78
nxdbGetDatabaseListSizes, 5-80
nxdbGetDBCAttribute, 5-82
nxdbGetDBCAttributeSize, 5-84
nxdbGetProperty, 5-85
nxdbGetPropertySize, 5-86
nxdbMerge, 5-87
nxdbOpenDatabase, 5-90
nxdbRemoveAlias, 5-91
nxdbSaveDatabase, 5-92
nxdbSetProperty, 5-94
nxdbUndeploy, 5-95
nxDisconnectTerminals, 5-96
nxFlush, 5-98
nxGetProperty, 5-99
nxGetPropertySize, 5-101
nxGetSubProperty, 5-102
nxGetSubPropertySize, 5-103
nxReadFrame, 5-104
nxReadSignalSinglePoint, 5-107
nxReadSignalWaveform, 5-109
nxReadSignalXY, 5-111
nxReadState, 5-113
nxSetProperty, 5-125
nxSetSubProperty, 5-126
nxStart, 5-127
nxStatusToString, 5-129
nxStop, 5-130
nxSystemClose, 5-132
nxSystemOpen, 5-133
nxWait, 5-134
nxWriteFrame, 5-136
nxWriteSignalSinglePoint, 5-139
nxWriteSignalWaveform, 5-140
nxWriteSignalXY, 5-142
nxWriteState, 5-144

O

online product certification specifications, D-9

P

palettes, NI-XNET API for LabVIEW, 4-2
parity calculation method (figure), C-3
Payload Length
    XNET Frame
        in C, 5-241
        in LabVIEW, 4-375
    XNET PDU
        in C, 5-271
        in LabVIEW, 4-385
Payload Length Maximum
    in C, 5-378
    in LabVIEW, 4-193
PCI-XNET specifications
    environmental, D-8
    isolation voltages, D-7
    physical dimensions, D-7
    physical layers
        external CAN transceiver, D-6
        FlexRay, D-6
        High-Speed CAN, D-5
        LIN, D-6
        Low-Speed/Fault-Tolerant CAN, D-5
        Single Wire CAN, D-5
    power requirements
        CAN, D-7
        FlexRay, D-7
        LIN, D-7
    RTSI/front panel sync connectors, D-6
    safety, D-7
PDU
    XNET Signal
        in C, 5-401
        in LabVIEW, 4-407
    XNET Subframe
        in C, 5-409
        in LabVIEW, 4-392
PDU References
    in C, 5-242
PDU Start Bits
    in C, 5-243
PDU Update Bits
    in C, 5-244
PDU_Mapping, 4-376
PDUs
    in C, 5-191
    in LabVIEW, 4-330
PDUs Required?
    in C, 5-192
    in LabVIEW, 4-331
PDUs, CAN
    in databases, G-2
physical layer, FlexRay hardware, 3-1
pinouts, 3-13
    C Series LIN DB9 (table), 3-17
    C Series NI 9861/9862, 3-14
    C Series NI 9866, 3-16
    CAN DB9
        C Series NI-XNET (table), 3-14
        PXI and PCI NI-XNET
            external CAN transceiver (table), 3-13
            interfaces (table), 3-13
    FlexRay DB9 (table), 3-2
    PCI-8511/8512/8513, 3-13
    PCI-8516, 3-16
    PCI-8517, 3-2
    PXI and PCI LIN DB9 (table), 3-16
    PXI-8511/8512/8513, 3-13
    PXI-8516, 3-16
    PXI-8517, 3-2
Port Number
    in C, 5-251
    in LabVIEW, 4-532
Priority, XNET LIN Schedule
    in C, 5-256
    in LabVIEW, 4-472
Product Name
    in C, 5-202
    in LabVIEW, 4-525

Product Number
in C, 5-202
in LabVIEW, 4-526
Protocol
XNET Cluster
in C, 5-194
in LabVIEW, 4-333
XNET Interface
in C, 5-252
in LabVIEW, 4-533
XNET Session
in C, 5-378
in LabVIEW, 4-194
Protocol Data Units (PDUs), 4-51
in C, 5-452
introduction, 5-452
programming with NI-XNET, 5-454
properties, 5-453
timing compared to frame timing, 5-454
in LabVIEW, 4-603
introduction, 4-603
programming with NI-XNET, 4-605
properties, 4-605
timing compared to frame timing, 4-605
protocol operation control in FlexRay, B-4
PXI-XNET specifications
environmental, D-4
isolation voltages, D-4
physical dimensions, D-3
physical layers
external CAN transceiver, D-2
FlexRay, D-2
High-Speed CAN, D-1
LIN, D-2
Low-Speed/Fault-Tolerant CAN, D-1
Single Wire CAN, D-1
power requirements
CAN, D-3
FlexRay, D-3
LIN, D-3
random vibration, D-4
RTSI/front panel sync connectors, D-2
safety, D-4
shock, D-3

Q

Queue Size
in C, 5-380
in LabVIEW, 4-195

R

raw frame format
in C
base unit, 5-421
payload unit, 5-429
in LabVIEW
base unit, 4-567
payload unit, 4-572
related documentation, *xxxiii*
Remote Transmit Request (RTR), A-4
required properties
in C, 5-432
in LabVIEW, 4-577
Resample Rate
in C, 5-386
in LabVIEW, 4-201
Run Mode, XNET LIN Schedule
in C, 5-257
in LabVIEW, 4-473

S

SAE J1939:ECU
in C, 5-354
in LabVIEW, 4-166

SAE J1939:ECU Busy
in C, 5-355
in LabVIEW, 4-167
SAE J1939:Hold Time Th
in C, 5-356
in LabVIEW, 4-168
SAE J1939:Maximum Repeat CTS
in C, 5-357
in LabVIEW, 4-169
SAE J1939:Node Address
in C, 5-358
in LabVIEW, 4-170
SAE J1939:NodeName
in C, 5-359
in LabVIEW, 4-171
SAE J1939:Number of Packets Received
in C, 5-360
in LabVIEW, 4-172
SAE J1939:Number of Packets Response
in C, 5-361
in LabVIEW, 4-173
SAE J1939:Response Time Tr_GD
in C, 5-362
in LabVIEW, 4-174
SAE J1939:Response Time Tr_SD
in C, 5-363
in LabVIEW, 4-175
SAE J1939:Timeout T1
in C, 5-364
in LabVIEW, 4-176
SAE J1939:Timeout T2
in C, 5-365
in LabVIEW, 4-177
SAE J1939:Timeout T3
in C, 5-366
in LabVIEW, 4-178
SAE J1939:Timeout T4
in C, 5-367
in LabVIEW, 4-179
safety
information, 2-1
Scaling Factor
in C, 5-401
in LabVIEW, 4-408
Scaling Offset
in C, 5-402
in LabVIEW, 4-408
Schedule, XNET LIN Schedule Entry
in C, 5-264
in LabVIEW, 4-480
Schedules, XNET Cluster
in C, 5-194
Serial Number
in C, 5-203
in LabVIEW, 4-526
session
creating in LabVIEW
using LabVIEW project, 4-48
using XNET Create Session.vi, 4-48
definition
in C, 5-8
in LabVIEW, 4-13
modes
in C, 5-9
in LabVIEW, 4-14
session state model
in C, 5-435
in LabVIEW, 4-580
session states
in C, 5-436
in LabVIEW, 4-581
session transitions
in C, 5-437
in LabVIEW, 4-582
Session:Application Protocol
in C, 5-353
in LabVIEW, 4-165

ShowInvalidFromOpen?
in C, 5-199
in LabVIEW, 4-280
Signal Input Single-Point mode
in C, 5-24
example, 5-24
in LabVIEW, 4-29
example, 4-30
Signal Input Waveform mode
in C, 5-26
example, 5-26
in LabVIEW, 4-32
example, 4-33
Signal Input XY mode
in C, 5-28
example, 5-28
in LabVIEW, 4-35
example, 4-35
Signal Output Single-Point mode
in C, 5-30
example, 5-30
in LabVIEW, 4-37
example, 4-37
Signal Output Waveform mode
in C, 5-31
example, 5-32
in LabVIEW, 4-38
example, 4-39
Signal Output XY mode
in C, 5-34
examples, 5-34
in LabVIEW, 4-41
examples, 4-41
Signals
XNET Cluster
in C, 5-195
in LabVIEW, 4-333
XNET Frame
in C, 5-245
in LabVIEW, 4-377
XNET PDU
in C, 5-272
in LabVIEW, 4-386
signals, in databases, G-3
Single Wire CAN, A-11
physical layer, 3-11
Slot Number
in C, 5-203
in LabVIEW, 4-527
source terminal
Interface properties
in C, 5-341
in LabVIEW, 4-152
special frames
in C, 5-429
Bus Error frame, 5-430
Delay frame, 5-429
Log Trigger frame, 5-429
Start Trigger frame, 5-430
in LabVIEW, 4-572
Bus Error frame, 4-576
Delay frame, 4-572
Lot Trigger frame, 4-572
Start Trigger frame, 4-574
specifications
C Series XNET, D-8
CE compliance, D-9
electromagnetic compatibility, D-9
environmental management, D-9
for characteristics of a CAN_H and CAN_L Pair of wires (table), 3-9
NI-XNET Transceiver Cables, D-8
online product certification, D-9
PCI-XNET, D-5
environmental, D-8
isolation voltages, D-7
physical dimensions, D-7
physical layers
external CAN transceiver, D-6
FlexRay, D-6
High-Speed CAN, D-5

LIN, D-6
Low-Speed/Fault-Tolerant CAN, D-5
Single Wire CAN, D-5
power requirements
CAN, D-7
FlexRay, D-7
LIN, D-7
RTSI/front panel sync connectors, D-6
safety, D-7
PXI-XNET, D-1
environmental, D-4
isolation voltages, D-4
physical dimensions, D-3
physical layers
external CAN transceiver, D-2
FlexRay, D-2
High-Speed CAN, D-1
LIN, D-2
Low-Speed/Fault-Tolerant CAN, D-1
Single Wire CAN, D-1
power requirements
CAN, D-3
FlexRay, D-3
LIN, D-3
random vibration, D-4
RTSI/front panel sync connectors, D-2
safety, D-4
shock, D-3
standard and extended frame formats (figure), A-3
Start Bit
in C, 5-403
in LabVIEW, 4-409
Start of Frame (SOF), A-4
starting FlexRay communication in LabVIEW, 4-50
startup in FlexRay, B-6
path of following coldstart node, B-9
path of leading coldstart node, B-8
path of non-coldstart node, B-9
startup state machine in FlexRay (figure), B-7
state models
in C
interface state model, 5-435
session state model, 5-435
in LabVIEW
interface state model, 4-581
session state model, 4-580
state transitions in FlexRay (figure), B-8
stuff error, A-7
Sync field in LIN, C-3
synchronization, 3-20
C Series and NI-XNET Transceiver Cables, 3-20
PXI NI-XNET and PCI NI-XNET, 3-20
system classes, 4-615
System Configuration API, 2-13
system controls, 4-559
system node, viewing available interfaces in, 4-6

T

TDMS
See Technical Data Management Streaming (TDMS)
Technical Data Management Streaming (TDMS)
channel data, 4-588
channel name, 4-587
channel properties, 4-588
group name, 4-587
termination
CAN hardware
Low-Speed/Fault-Tolerant, 3-9
Single Wire physical layer, 3-12
FlexRay hardware, 3-2
LIN hardware, 3-16

termination resistor placement, CAN
High Speed (figure), 3-6
Low-Speed/Fault-Tolerant (figure), 3-9
Tick, XNET Cluster
in C, 5-196
transceiver
CAN hardware
High-Speed physical layer, 3-4
Low-Speed/Fault-Tolerant physical layer, 3-7
Single Wire physical layer, 3-11
FlexRay hardware, 3-1
LIN hardware, 3-15
troubleshooting, 6-1
Type, XNET LIN Schedule Entry
in C, 5-265
in LabVIEW, 4-481

U

Unit
in C, 5-405
in LabVIEW, 4-411
using CAN, 4-48
using FlexRay, 4-50
using LIN, 4-51

V

verifying NI-XNET hardware installation, 2-4
Version:Build
in C, 5-413
in LabVIEW, 4-518
Version:Major
in C, 5-414
in LabVIEW, 4-519
Version:Minor
in C, 5-415
in LabVIEW, 4-520
Version:Phase
in C, 5-416
in LabVIEW, 4-521
Version:Update
in C, 5-417
in LabVIEW, 4-522
Visual C++, 5-2
Visual C++ 6
examples, 5-3

X

XNET Blink.vi, 4-534
XNET C Series modules firmware update, 2-5
XNET Clear.vi, 4-504
XNET Cluster constant, 4-334
XNET Cluster control, 4-558
XNET Cluster I/O name
string use, 4-617
user interface, 4-616
XNET Cluster properties, 5-147
XNET Cluster property node, 4-281
XNET Connect Terminals.vi, 4-506
XNET Convert (Frame CAN to Signal).vi, 4-539
XNET Convert (Frame FlexRay to Signal).vi, 4-542
XNET Convert (Frame LIN to Signal).vi, 4-545
XNET Convert (Frame Raw to Signal).vi, 4-547
XNET Convert (Signal to Frame CAN).vi, 4-549
XNET Convert (Signal to Frame FlexRay).vi, 4-551
XNET Convert (Signal to Frame LIN).vi, 4-554
XNET Convert (Signal to Frame Raw).vi, 4-556
XNET Convert.vi, 4-538
XNET Create Session (Conversion).vi, 4-63

XNET Create Session (Frame Input Queued).vi, 4-64
XNET Create Session (Frame Input Single-Point).vi, 4-65
XNET Create Session (Frame Input Stream).vi, 4-66
XNET Create Session (Frame Output Queued).vi, 4-69
XNET Create Session (Frame Output Single-Point).vi, 4-70
XNET Create Session (Frame Output Stream).vi, 4-71
XNET Create Session (Generic).vi, 4-74
XNET Create Session (PDU Input Queued).vi, 4-68
XNET Create Session (PDU Input Single Point).vi, 4-68
XNET Create Session (PDU Output Queued).vi, 4-73
XNET Create Session (PDU Output Single-Point).vi, 4-73
XNET Create Session (Signal Input Single-Point).vi, 4-76
XNET Create Session (Signal Input Waveform).vi, 4-77
XNET Create Session (Signal Input XY).vi, 4-78
XNET Create Session (Signal Output Single-Point).vi, 4-79
XNET Create Session (Signal Output Waveform).vi, 4-80
XNET Create Session (Signal Output XY).vi, 4-81
XNET Create Session.vi, 4-62
  using to create a session, 4-48
XNET Create Timing Source (FlexRay Cycle).vi, 4-490
XNET Create Timing Source.vi, 4-490
XNET Database Add Alias.vi, 4-459
XNET Database Close (Cluster).vi, 4-414
XNET Database Close (Database).vi, 4-415
XNET Database Close (ECU).vi, 4-416
XNET Database Close (Frame).vi, 4-417
XNET Database Close (LIN Schedule Entry).vi, 4-422
XNET Database Close (LIN Schedule).vi, 4-421
XNET Database Close (PDU).vi, 4-418
XNET Database Close (Signal).vi, 4-419
XNET Database Close (Subframe).vi, 4-420
XNET Database Close.vi, 4-413
XNET Database constant, 4-281
XNET Database control, 4-558
XNET Database Create (Cluster).vi, 4-424
XNET Database Create (Dynamic Signal).vi, 4-426
XNET Database Create (ECU).vi, 4-428
XNET Database Create (Frame).vi, 4-429
XNET Database Create (LIN Schedule Entry).vi, 4-435
XNET Database Create (LIN Schedule).vi, 4-434
XNET Database Create (PDU).vi, 4-430
XNET Database Create (Signal).vi, 4-431
XNET Database Create (Subframe).vi, 4-432
XNET Database Create Object.vi, 4-423
XNET Database Delete (Cluster).vi, 4-438
XNET Database Delete (ECU).vi, 4-439
XNET Database Delete (Frame).vi, 4-440
XNET Database Delete (LIN Schedule Entry).vi, 4-445
XNET Database Delete (LIN Schedule).vi, 4-444
XNET Database Delete (PDU).vi, 4-441
XNET Database Delete (Signal).vi, 4-442
XNET Database Delete (Subframe).vi, 4-443
XNET Database Delete Object.vi, 4-437
XNET Database Deploy.vi, 4-464
XNET Database Export.vi, 4-458
XNET Database Get DBC Attribute.vi, 4-482
XNET Database Get List.vi, 4-462

XNET Database I/O name, 4-618
refnum use, 4-620
string use, 4-619
user interface, 4-618
XNET Database Merge (Cluster).vi, 4-455
XNET Database Merge (ECU).vi, 4-451
XNET Database Merge (Frame).vi, 4-447
XNET Database Merge (LIN Schedule).vi, 4-453
XNET Database Merge (PDU).vi, 4-449
XNET Database Merge.vi, 4-446
XNET Database Open.vi, 4-412
XNET Database properties, 5-198
XNET Database property node, 4-278
XNET Database Remove Alias.vi, 4-461
XNET Database Save.vi, 4-457
XNET Database Undeploy.vi, 4-466
XNET Device I/O name, 4-621
refnum use, 4-621
string use, 4-621
user interface, 4-621
XNET Device properties, 5-200
XNET Device property node, 4-523
XNET Disconnect Terminals.vi, 4-513
XNET ECU constant, 4-347
XNET ECU control, 4-558
XNET ECU I/O name, 4-621
refnum use, 4-624
string use, 4-623
user interface, 4-621
XNET ECU properties, 5-204
XNET ECU property node, 4-334
XNET Flush.vi, 4-505
XNET Frame constant, 4-378
XNET Frame control, 4-559
XNET Frame I/O name, 4-624
refnum use, 4-626
string use, 4-625
user interface, 4-624
XNET Frame properties, 5-215
XNET Frame property node, 4-347
XNET I/O names, 4-613
I/O name classes, 4-614
in LabVIEW Real-Time (RT), 4-54
XNET Interface constant, 4-534
XNET Interface control, 4-559
XNET Interface I/O name, 4-627
refnum use, 4-628
string use, 4-628
user interface, 4-627
XNET Interface properties, 5-247
XNET Interface property node, 4-527
XNET LIN Schedule control, 4-559
XNET LIN Schedule Entry control, 4-559
XNET LIN Schedule Entry I/O name, 4-637
refnum use, 4-638
string use, 4-638
user interface, 4-638
XNET LIN Schedule Entry properties, 5-258
XNET LIN Schedule Entry property node, 4-474
XNET LIN Schedule I/O name, 4-635
refnum use, 4-637
string use, 4-636
user interface, 4-635
XNET LIN Schedule properties, 5-253
XNET LIN Schedule property node, 4-467
XNET PDU constant, 4-386
XNET PDU I/O name, 4-638
refnum use, 4-641
string use, 4-640
user interface, 4-638
XNET PDU properties, 5-266
XNET PDU property node, 4-378
XNET Read (Frame CAN).vi, 4-204
XNET Read (Frame FlexRay).vi, 4-208
XNET Read (Frame LIN).vi, 4-213
XNET Read (Frame Raw).vi, 4-218
XNET Read (Signal Single-Point).vi, 4-221
XNET Read (Signal Waveform).vi, 4-222
XNET Read (Signal XY).vi, 4-224
XNET Read (State CAN Comm).vi, 4-227

XNET Read (State FlexRay Comm).vi, 4-231
XNET Read (State FlexRay Cycle Macrotick).vi, 4-240
XNET Read (State FlexRay Statistics).vi, 4-242
XNET Read (State LIN Comm).vi, 4-235
XNET Read (State Session Info).vi, 4-248
XNET Read (State Time Comm).vi, 4-244
XNET Read (State Time Current).vi, 4-245
XNET Read (State Time Start).vi, 4-246
XNET Read.vi, 4-202
XNET Session constant, 4-61
XNET Session control, 4-558
XNET Session I/O name, 4-628
    refnum use, 4-630
    string use, 4-629
    user interface, 4-629
XNET Session properties, 5-273
XNET Session property node, 4-82
XNET Signal constant, 4-412
XNET Signal control, 4-559
XNET Signal I/O name, 4-630
    refnum use, 4-633
    string use, 4-632
    user interface, 4-631
XNET Signal properties, 5-387
XNET Signal property node, 4-393
XNET Start.vi, 4-499
XNET Stop.vi, 4-502
XNET String to IO Name.vi, 4-537
XNET Subframe I/O name, 4-633
    refnum use, 4-634
    string use, 4-634
    user interface, 4-633
XNET Subframe properties, 5-405
XNET Subframe property node, 4-387
XNET System Close.vi, 4-536
XNET System properties, 5-410
XNET System property node, 4-514
XNET Terminal constant, 4-514
XNET Terminal control, 4-559
XNET Terminal I/O name, 4-634
    refnum use, 4-635
    string use, 4-634
    user interface, 4-634
XNET Wait (CAN Remote Wakeup).vi, 4-488
XNET Wait (Interface Communicating).vi, 4-486
XNET Wait (LIN Remote Wakeup).vi, 4-489
XNET Wait (Transmit Complete).vi, 4-485
XNET Wait.vi, 4-484
XNET Write (Frame CAN).vi, 4-256
XNET Write (Frame FlexRay).vi, 4-260
XNET Write (Frame LIN).vi, 4-264
XNET Write (Frame Raw).vi, 4-268
XNET Write (Signal Single-Point).vi, 4-251
XNET Write (Signal Waveform).vi, 4-252
XNET Write (Signal XY).vi, 4-254
XNET Write (State FlexRay Symbol).vi, 4-271
XNET Write (State LIN Diagnostic Schedule Change).vi, 4-275
XNET Write (State LIN Schedule Change).vi, 4-272
XNET Write.vi, 4-249
XS software selectable physical layer, 3-3