CAN

NI-CAN™ Hardware and Software Manual

NATIONAL INSTRUMENTS™

October 2002 Edition
Part Number 370289E-01

Worldwide Technical Support and Product Information

`ni.com`

National Instruments Corporate Headquarters

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

Worldwide Offices

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 55 11 3262 3599,
Canada (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949,
Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, China 86 21 6555 7838,
Czech Republic 02 2423 5774, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24,
Germany 089 741 31 30, Greece 01 42 96 427, Hong Kong 2645 3186, India 91 80 4190000,
Israel 03 6393737, Italy 02 413091, Japan 03 5472 2970, Korea 02 3451 3400, Malaysia 603 9596711,
Mexico 001 800 010 0793, Netherlands 0348 433466, New Zealand 09 914 0488, Norway 32 27 73 00,
Poland 22 3390 150, Portugal 210 311 210, Russia 095 238 7139, Singapore 65 6 226 5886,
Slovenia 3 425 4200, South Africa 11 805 8197, Spain 91 640 0085, Sweden 08 587 895 00,
Switzerland 056 200 51 51, Taiwan 02 2528 7227, United Kingdom 01635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send e-mail to `techpubs@ni.com`.

Important Information

Warranty

The NI-CAN hardware is warranted against defects in materials and workmanship for a period of one year from the date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace equipment that proves to be defective during the warranty period. This warranty includes parts and labor.

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

CVI™, IMAQ™, LabVIEW™, National Instruments™, NI™, NI-CAN™, ni.com™, NI-DAQ™, NI-Motion™, and RTSI™ are trademarks of National Instruments Corporation.

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

Patents

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

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Compliance

FCC/Canada Radio Frequency Interference Compliance

Determining FCC Class

The Federal Communications Commission (FCC) has rules to protect wireless communications from interference. The FCC places digital electronics into two classes. These classes are known as Class A (for use in industrial-commercial locations only) or Class B (for use in residential or commercial locations). Depending on where it is operated, this product could be subject to restrictions in the FCC rules. (In Canada, the Department of Communications (DOC), of Industry Canada, regulates wireless interference in much the same way.)

Digital electronics emit weak signals during normal operation that can affect radio, television, or other wireless products. By examining the product you purchased, you can determine the FCC Class and therefore which of the two FCC/DOC Warnings apply in the following sections. (Some products may not be labeled at all for FCC; if so, the reader should then assume these are Class A devices.)

FCC Class A products only display a simple warning statement of one paragraph in length regarding interference and undesired operation. Most of our products are FCC Class A. The FCC rules have restrictions regarding the locations where FCC Class A products can be operated.

FCC Class B products display either a FCC ID code, starting with the letters **EXN**, or the FCC Class B compliance mark that appears as shown here on the right.

Trade Name Model Number
Tested to Comply with FCC Standards
FOR HOME OR OFFICE USE

Consult the FCC Web site at `http://www.fcc.gov` for more information.

FCC/DOC Warnings

This equipment generates and uses radio frequency energy and, if not installed and used in strict accordance with the instructions in this manual and the CE Marking Declaration of Conformity*, may cause interference to radio and television reception. Classification requirements are the same for the Federal Communications Commission (FCC) and the Canadian Department of Communications (DOC).

Changes or modifications not expressly approved by National Instruments could void the user's authority to operate the equipment under the FCC Rules.

Class A

Federal Communications Commission

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

Canadian Department of Communications

This Class A digital apparatus meets all requirements of the Canadian Interference-Causing Equipment Regulations.

Cet appareil numérique de la classe A respecte toutes les exigences du Règlement sur le matériel brouilleur du Canada.

Class B

Federal Communications Commission

This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient or relocate the receiving antenna.
- Increase the separation between the equipment and receiver.
- Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
- Consult the dealer or an experienced radio/TV technician for help.

Canadian Department of Communications

This Class B digital apparatus meets all requirements of the Canadian Interference-Causing Equipment Regulations.

Cet appareil numérique de la classe B respecte toutes les exigences du Règlement sur le matériel brouilleur du Canada.

Compliance to EU Directives

Readers in the European Union (EU) must refer to the Manufacturer's Declaration of Conformity (DoC) for information* pertaining to the CE Marking compliance scheme. The Manufacturer includes a DoC for most every hardware product except for those bought for OEMs, if also available from an original manufacturer that also markets in the EU, or where compliance is not required as for electrically benign apparatus or cables.

To obtain the DoC for this product, click **Declaration of Conformity** at `ni.com/hardref.nsf/`. This Web site lists the DoCs by product family. Select the appropriate product family, followed by your product, and a link to the DoC appears in Adobe Acrobat format. Click the Acrobat icon to download or read the DoC.

* The CE Marking Declaration of Conformity will contain important supplementary information and instructions for the user or installer.

Contents

About This Manual

Chapter 1
Introduction

Chapter 2
Installation and Configuration

Chapter 3
Developing Your Application

Chapter 4
Using the Channel API

Chapter 5
Channel API for LabVIEW

Chapter 6
Channel API for C

Chapter 7
Using the Frame API

Chapter 8
Frame API for LabVIEW

Chapter 9
Frame API for C

About This Manual

This manual is a description of the National Instruments Controller Area Network (CAN) hardware and NI-CAN software features as well as a programming reference for VIs and functions in the NI-CAN software.

The authors of this manual assume you are already familiar with your operating system.

How to Use the Manual Set

Use the *CAN Hardware and NI-CAN Software for Windows Installation Guide* in the jewel case of your program CD to install and configure your CAN hardware and the NI-CAN software. Use this manual to learn the basics of NI-CAN as well as how to develop an application.

This manual contains specific, programmer-reference information about each NI-CAN function and VI.

This manual also describes the features of the hardware. Unless otherwise noted, this manual applies to the NI-CAN hardware products, which include the following.

PCI-CAN

- PCI-CAN (high-speed; one port)
- PCI-CAN/2 (high-speed; two port)
- PCI-CAN/LS (low-speed, fault-tolerant; one port)
- PCI-CAN/LS2 (low-speed, fault-tolerant; two port)

PXI-846*x*

- PXI-8460 (low-speed, fault-tolerant; one or two port)
- PXI-8461 (high-speed; one or two port)
- PXI-8462 (dual-speed: port one high-speed, port two low-speed)

PCMCIA-CAN

- PCMCIA-CAN (high-speed; one port)
- PCMCIA-CAN/2 (high-speed; two port)
- PCMCIA-CAN/LS (low-speed, fault-tolerant; one port)

- PCMCIA-CAN/LS2 (low-speed, fault-tolerant; one port)
- PCMCIA-CAN/DS (dual-speed: one port high-speed, one port low-speed, fault-tolerant)

Conventions Used in This Manual

The following conventions appear in this manual:

» — The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.

This icon denotes a note, which alerts you to important information.

bold — Bold text denotes items that you must select or click on in the software, such as menu items and dialog box options. Bold text also denotes parameter names.

italic — Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.

`monospace` — Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.

`monospace italic` — Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

Related Documentation

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

- ANSI/ISO Standard 11898-1993, *Road Vehicles—Interchange of Digital Information—Controller Area Network (CAN) for High-Speed Communication*
- *CAN Specification Version 2.0*, 1991, Robert Bosch GmbH., Postfach 106050, D-70049 Stuttgart 1

- *CiA Draft Standard 102*, Version 2.0, CAN Physical Layer for Industrial Applications
- *CompactPCI Specification*, Revision 2.0, PCI Industrial Computers Manufacturers Group
- *DeviceNet Specification*, Version 2.0, Open DeviceNet Vendor Association
- *PXI Specification*, Revision 1.0, National Instruments Corporation
- LabVIEW Online Reference
- Measurement and Automation Explorer (MAX) Online Reference
- Microsoft Win32 Software Development Kit (SDK) Online Help

1 Introduction

This chapter provides an introduction to the Controller Area Network (CAN) and the National Instruments products for CAN.

CAN Overview

The data frame is the fundamental unit of data transfer on a CAN network. Figure 1-1 shows a simplified view of the CAN data frame.

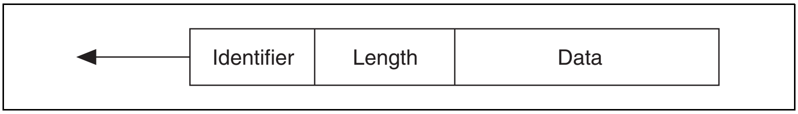


Figure 1-1. Simplified CAN Data Frame

When multiple CAN devices transmit a frame at the same time, the identifier (ID) resolves the collision. The highest priority ID continues, and the lower priority IDs retry immediately afterward. The ISO 11898 CAN standard specifies two ID formats: the standard format of 11 bits and the extended format of 29 bits.

The ID is followed by a length code that specifies the number of data bytes in the frame. The length ranges from 0 to 8 data bytes. The ID value determines the meaning of the data bytes.

In addition to the data frame, the CAN standard specifies the remote frame. The remote frame includes the ID, but no data bytes. A CAN device transmits the remote frame to request that another device transmit the associated data frame for the ID. In other words, the remote frame provides a mechanism to poll for data.

The preceding information provides a simplified description of CAN frames. The CAN frame format includes many other fields, such as for error checking and acknowledgement. For more detailed information on the ISO 11898 CAN standard, refer to Appendix F, *Summary of the CAN Standard*.

NI-CAN Hardware Overview

The National Instruments CAN hardware covered in this manual includes the PCI-CAN, PCI-CAN/2, PCI-CAN/LS (low-speed CAN), PCI-CAN/LS2, PCI-CAN/DS (dual-speed CAN), PCMCIA-CAN, PCMCIA-CAN/2, PXI-8460 (low-speed: one or two port), PXI-8461 (high-speed: one or two port) and PXI-8462 (dual-speed: port one high-speed, port two low-speed).

The PCI-CAN, PCI-CAN/LS and PCI-CAN/DS series cards are completely software configurable and compliant with the PCI Local Bus Specification. With a PCI-CAN, PCI-CAN/LS or PCI-CAN/DS series card, you can make your PC-compatible computer with PCI Local Bus slots communicate with and control CAN devices.

The PCMCIA-CAN series cards are Type II PC Cards that are completely software configurable and compliant with the PCMCIA standards for 16-bit PC Cards. With a PCMCIA-CAN series card, you can make your PC-compatible notebook with PCMCIA sockets communicate with and control CAN devices.

The PXI-8460, PXI-8461, and PXI-8462 are software configurable and compliant with the *PXI Specification* and *CompactPCI Specification*. With the PXI-846*x* cards you can make your PXI or CompactPCI chassis communicate with and control CAN devices.

The CAN hardware supports a wide variety of transfer rates up to 1 Mb/s. CAN interfacing is accomplished using the Intel 82527 CAN controller chip. The high-speed CAN physical layer fully conforms to the ISO 11898 physical layer specification for CAN and is optically isolated to 500 V. The low-speed CAN physical layer conforms to the ISO 11898 physical layer specification for CAN and is also optically isolated to 500 V.

The PCI-CAN and PXI-8461 series cards are available with two physical connector types: DB-9 D-Sub and Combicon-style pluggable screw terminals. Low-speed PCI-CAN/LS, PCI-CAN/DS, PXI-8460, and PXI-8462 boards are available with DB-9 D-Sub connectors. PCMCIA-CAN, PCMCIA-CAN/LS and PCMCIA-CAN/DS cables include both a DB-9 D-Sub and a pluggable screw terminal.

The CAN physical layer on PCI-CAN, PXI-8460 and PXI-846*x* series cards can be powered either internally (from the card) or externally (from the bus cable power). The power source for the CAN physical layer for each port is configured with a jumper.

There are four types of cables available for the PCMCIA-CAN cards:

- PCMCIA-CAN bus powered transceiver cables. The CAN physical layer is powered externally (from the bus cable power).
- PCMCIA-CAN internally powered transceiver cables. The CAN physical layer is powered internally (from the card).
- PCMCIA-CAN/LS cables. The low-speed CAN physical layer and the V-BAT pin of the low-speed transceiver are powered internally. This cable also requires that only the V–, CAN_L and CAN_H be connected to the bus.
- PCMCIA-CAN/DS cables. The high-speed port (port 1) physical layer is powered internally. The low-speed port (port 2) physical layer is identical to the PCMCIA-CAN/LS cable.

The PXI-846*x* and PCI-CAN cards use the Real-Time System Integration (RTSI) bus to solve the problem of synchronizing several functions across multiple cards to a common trigger or timing event. For PCI-CAN, the RTSI bus consists of the National Instruments RTSI bus interface and ribbon cable to route timing and trigger signals between the CAN hardware and National Instruments DAQ, IMAQ, NI-Motion, or additional CAN hardware. For the PXI-846*x*, the RTSI bus is implemented by using the National Instruments PXI trigger bus to route timing and trigger signals between the CAN hardware and National Instruments DAQ, IMAQ, NI-Motion, or additional CAN hardware. Although the PXI-846*x* series cards with RTSI bus are available in a PXI chassis, there are important issues to consider when using RTSI in a CompactPCI chassis.

Refer to Appendix E, *RTSI Bus*, for detailed information about the RTSI interface. Also refer to the *RTSI Bus Overview* and *The RTSI Solution* sections later in this chapter.

All of the CAN hardware uses the Intel 386EX embedded processor to implement time-critical features provided by the NI-CAN software. The CAN hardware communicates with the NI-CAN driver through on-card shared memory and an interrupt.

NI-CAN Software Overview

The NI-CAN software provides full-featured Application Programming Interfaces (APIs), plus tools for configuration and analysis within National Instruments Measurement & Automation Explorer (MAX). The NI-CAN APIs enable you to develop applications that are customized to your test and simulation requirements.

MAX

The NI-CAN features within MAX enable you to:

- verify the installation of your NI-CAN hardware
- configure software properties for each CAN port
- create or import configuration information for the Channel API
- interact with your CAN network using various tools

For more information, refer to Chapter 2, *Installation and Configuration*.

Frame API

As described in the *CAN Overview* section, the frame is the fundamental unit of data transfer on a CAN network. The NI-CAN Frame API provides a set of functions to write and read CAN frames.

Within the Frame API, the data bytes of each frame are not interpreted, but are transferred in their raw format. For example, you can transmit a data frame by calling a write function with the ID, length, and array of data bytes.

For more information, refer to Chapter 7, *Using the Frame API*.

Channel API

A typical CAN data frame contains multiple values encoded as raw fields. Figure 1-2 shows an example set of fields for a 6-byte data frame.

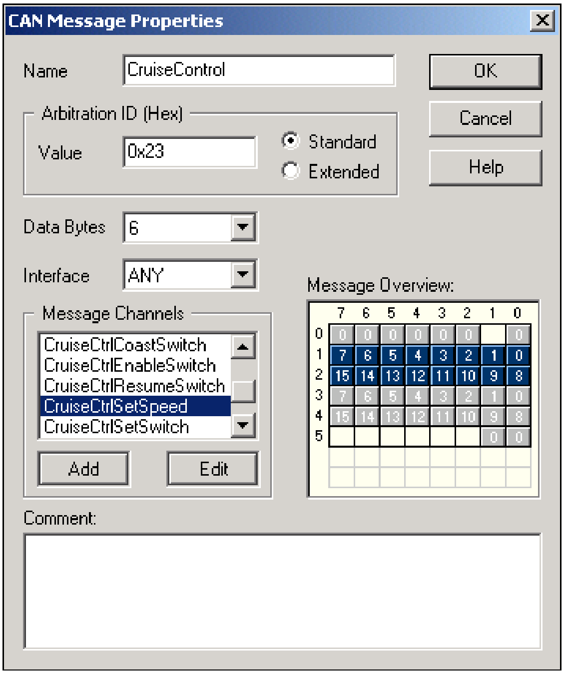


Figure 1-2. Example of CruiseControl Message

Bytes 1 to 2 contain a **CruiseCtrlSetSpeed** field that represents a vehicle speed in kilometers per hour (km/h). Most CAN devices do not transmit values as floating-point units such as 115.6 km/h. Therefore, this field consists of a 16-bit unsigned integer in which each increment represents 0.0039 km/h. For example, if the field contains the value 25000, that represents (25000 * 0.0039) = 97.5 km/h.

Bytes 3 to 4 contain another unsigned integer VehicleSpeed that represents speed in km/h. Bytes 0 and 5 contain various Boolean fields for which 1 indicates “on” and 0 indicates “off.”

When you use the NI-CAN Frame API to read CAN data frames, you must write code in your application to convert each raw field to physical units such as km/h. The NI-CAN Channel API enables you to specify this conversion information at configuration-time instead of within your application. This configuration information can be imported from Vector CANdb files, or specified directly in MAX.

For each ID you read or write on the CAN network, you specify a number of fields. For each field, you specify its location in the frame, size in bits, and a formula to convert to/from floating-point units. In other words, you specify the meaning of various fields in each CAN data frame. In NI-CAN terminology, a data frame for which the individual fields are described is called a *message*.

In other National Instruments software products such as NI-DAQ and FieldPoint, an application reads or writes a floating-point value using a *channel*, which is typically converted to/from a raw value in the measurement hardware. The NI-CAN Channel API also uses the term channel to refer to floating-point values converted to/from raw fields in messages. In CAN products of other vendors, this concept is often referred to as a signal. When a CAN message is received, NI-CAN converts the raw fields into physical units, which you then obtain using the Channel API *read function*. When you call the Channel API write function, you provide floating-point values in physical units, which NI-CAN converts into raw fields and transmits as a CAN message.

For more information, refer to Chapter 4, *Using the Channel API*.

RTSI Bus Overview

RTSI is an acronym for Real-Time System Integration. It is the National Instruments timing bus that connects CAN and DAQ boards directly. This is done via connectors on top of the PCI-CAN series boards, and the PXI trigger bus on the PXI-846*x* series boards, for precise synchronization of functions.

The RTSI Solution

A common problem with interface boards is that you cannot easily synchronize several functions across multiple boards to a common trigger or timing event. CAN boards use the RTSI bus to solve this problem.

For PCI-CAN series boards, the RTSI bus consists of connecting the National Instruments RTSI bus interface with RTSI ribbon cable to route timing and trigger signals between the CAN board and other National Instruments RTSI-equipped hardware. Refer to Appendix E, *RTSI Bus*, for detailed information about the PCI-CAN and AT-CAN series RTSI interfaces.

For the PXI-846*x* series CAN boards, the RTSI bus consists of using the National Instruments PXI trigger bus to route timing and trigger signals between the PXI-846*x* series board and other National Instruments RTSI-equipped PXI boards. Regarding the RTSI interface on your PXI-846*x* series board, there are important issues to consider when using it in a CompactPCI chassis. Refer to Appendix E, *RTSI Bus*, for detailed information about the PXI-846*x* series RTSI interface.

For information on RTSI programming, refer to the *Synchronization* section of Chapter 4, *Using the Channel API*, and the *RTSI* section in Chapter 7, *Using the Frame API*.

2

Installation and Configuration

The Measurement & Automation Explorer (MAX) provides access to all of your National Instruments products. Like other NI software products, NI-CAN uses MAX as the centralized location for all configuration and tools.

To launch MAX, select the **Measurement & Automation** shortcut on your desktop, or within your Windows **Programs** menu under **National Instruments»Measurement & Automation**.

For information on the NI-CAN software within MAX, consult the online help within MAX.

A reference is located in the MAX **Help** menu under **Help Topics» NI-CAN**.

View help for items in the MAX **Configuration** tree by using the built-in MAX help pane. If this help pane is not shown on the far right, select the **Show/Hide** button in the upper right.

View help for a dialog box by selecting the **Help** button in the window.

The following sections provide an overview of some common tasks you can perform within MAX.

Verify Installation of Your CAN Hardware

Within the **Devices & Interfaces** branch of the MAX **Configuration** tree, NI-CAN cards are listed along with other hardware in the local computer system, as shown in Figure 2-1.

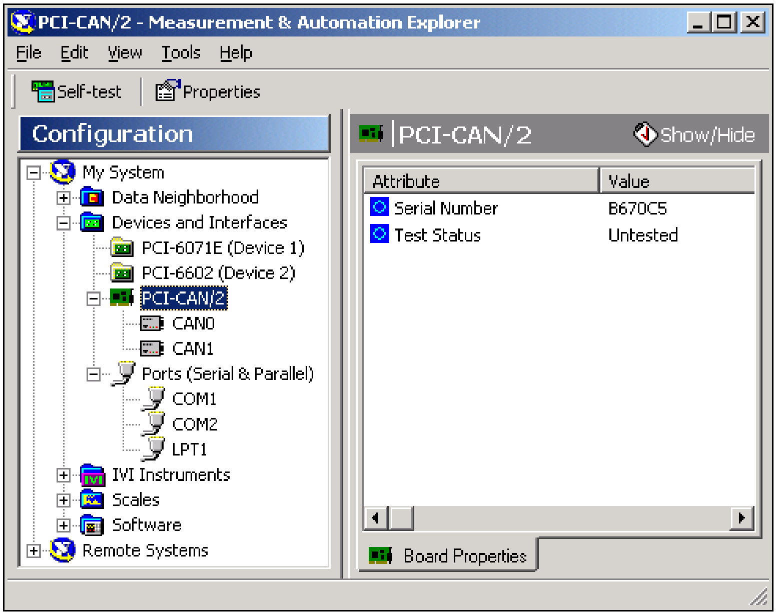


Figure 2-1. NI-CAN Cards Listed in MAX

If your NI-CAN hardware is not listed here, MAX is not configured to search for new devices on startup. In order to search for the new hardware, press the <F5> key.

To verify installation of your CAN hardware, right-click the CAN card, then select **Self-test**. If the self-test passes, the card's icon shows a checkmark. If the self-test fails, the card's icon shows an "X" mark, and the **Test Status** in the right pane describes the problem. Refer to Appendix A, *Troubleshooting and Common Questions*, for information on resolving hardware installation problems.

Configure CAN Ports

The physical ports of each CAN card are listed under the card's name. To configure software properties for each port, right-click the port and select **Properties**.

In the **Properties** dialog, you assign an interface name to the port, such as **CAN0** or **CAN1**. The interface name identifies the physical port within NI-CAN APIs.

The **Properties** dialog also contains the default baud rate for MAX tools and the Channel API.

CAN Channels

Within the **Data Neighborhood** branch of the MAX **Configuration** tree, the **CAN Channels** branch lists information for the NI-CAN Channel API, as shown in Figure 2-2.

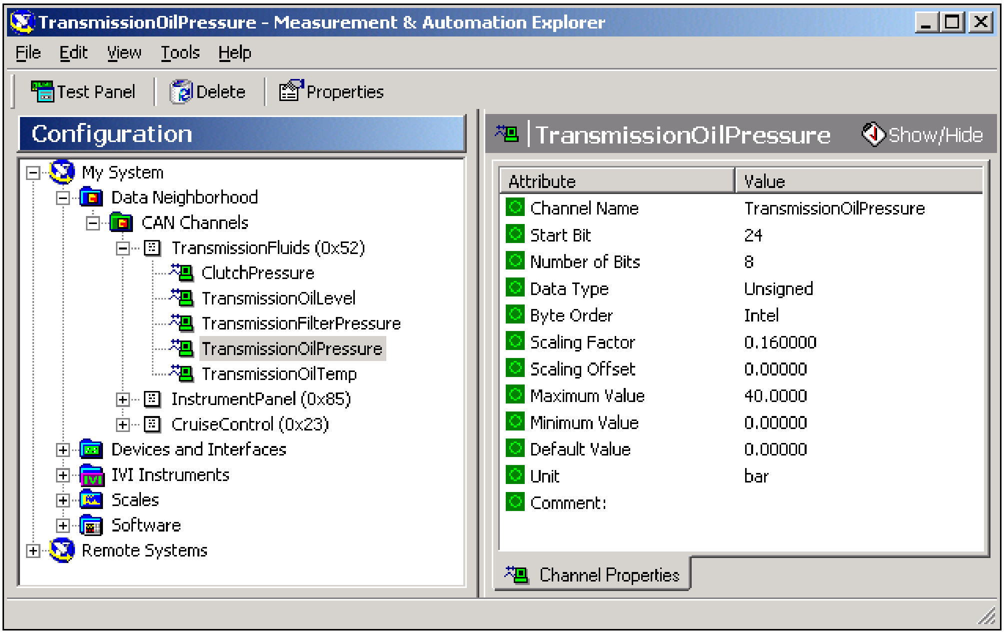


Figure 2-2. CAN Channels in MAX

The **CAN Channels** branch lists CAN messages for use with the Channel API. A set of channels is specified for each message.

For information about creating information under **CAN Channels**, refer to the *Choose Source of Channel Configuration* section of Chapter 4, *Using the Channel API*.

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 as a LabVIEW RT system, you can install a PXI CAN card and use the NI-CAN APIs to develop real-time applications. For example, you can simulate the behavior of a control algorithm within a CAN device, using data from received CAN messages to generate outgoing CAN messages with deterministic response times.

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

After you have installed your PXI CAN cards and downloaded the NI-CAN software to your LabVIEW RT system, you need to verify the installation. Within the **Tools** menu in MAX, select **NI-CAN»RT Hardware Configuration**. The **RT Hardware Configuration** tool provides features similar to **Devices & Interfaces** on your local system. Use the **RT Hardware Configuration** tool to self-test the CAN cards and assign an interface name to each physical CAN port.

To use the Channel API on your LabVIEW RT system, you must also download channel configuration information. Right-click the **CAN Channels** heading, then select **Send to RT System**. This downloads all information under **CAN Channels** to your LabVIEW RT system, so you can execute the same LabVIEW VIs on your LabVIEW RT system as your Windows system.

Tools

NI-CAN provides tools that you can launch from MAX.

- **Bus Monitor**—Displays statistics for raw CAN frames. This provides a basic tool to analyze CAN network traffic. Launch this tool by right-clicking a CAN interface (port).
- **Test Panel**—Read or write physical units for a CAN channel. This provides a simple debugging tool to experiment with CAN channels. Launch this tool by right-clicking a CAN channel.
- **NI-Spy**—Monitor function calls to the NI-CAN APIs. This tool helps in debugging programming problems in your application. Launch this tool from the MAX **Tools** menu.
- **FP1300 Configuration**—FieldPoint 1300 is the National Instruments modular I/O product for CAN. If you have installed the software for the FP1300 product, launch this tool by right-clicking a CAN interface (port).

3

Developing Your Application

This chapter explains how to develop your application using the NI-CAN APIs.

Choose Your Programming Language

The programming language you use for application development determines how to access the NI-CAN APIs.

LabVIEW

The NI-CAN software supports LabVIEW version 6.0 and later. NI-CAN support for LabVIEW RT requires version 6.0.3 or later.

NI-CAN functions and controls are available in the LabVIEW palettes. The top level of the NI-CAN function palette contains the most commonly used functions of the Channel API. Subpalettes contain the Frame API functions and advanced Channel API functions.

The reference for each NI-CAN Channel API function is in Chapter 5, *Channel API for LabVIEW*. The reference for each NI-CAN Frame API function is in Chapter 8, *Frame API for LabVIEW*. To access a function's reference from within LabVIEW, press <Ctrl-H> to open the help window, click on the NI-CAN function, and then follow the link.

The NI-CAN software includes a full set of examples for LabVIEW. These examples teach basic NI-CAN programming as well as advanced topics. The example help describes each example and includes a link you can use to open the VI.

In LabVIEW 6.0, the NI-CAN example help is in **Help»Examples»Other NI Products»Controller Area Network (CAN)**.

In LabVIEW 6.1, the NI-CAN example help is in **Help»Find Examples» Hardware Input and Output»CAN**.

LabWindows/CVI

The NI-CAN software supports LabWindows™/CVI™ version 5.5 and later.

Within LabWindows/CVI, the NI-CAN function panel is in **Libraries»NI-CAN**. Like other LabWindows/CVI function panels, the NI-CAN function panel provides help for each function and the ability to generate code.

The reference for each NI-CAN Channel API function is in Chapter 6, *Channel API for C*. The reference for each NI-CAN Frame API function is in Chapter 9, *Frame API for C*. You can access each function's reference directly from within the function panel.

The header file for both NI-CAN APIs is `nican.h`. The library for both NI-CAN APIs is `nican.lib`.

The NI-CAN software includes a full set of examples for LabWindows/CVI. The NI-CAN examples are installed in the LabWindows/CVI directory under `samples\nican`.

Each example provides a complete LabWindows/CVI project (`.prj` file). A description of each example is provided in comments at the top of the `.c` file.

Visual C++ 6

The NI-CAN software supports Microsoft Visual C/C++ version 6.

The header file and library for Visual C/C++ 6 are in the `MS Visual C` folder of the NI-CAN folder. The typical path to this folder is `\Program Files\National Instruments\NI-CAN\MS Visual C`.

To use either NI-CAN API, include the `nican.h` header file in your code, then link with the `nicanmsc.lib` library file.

For C applications (files with `.c` extension), include the header file by adding a `#include` to the beginning of your code, such as:

```
#include "nican.h"
```

For C++ applications (files with `.cpp` extension), define `_cplusplus` before including the header, such as:

```
#define _cplusplus

#include "nican.h"
```

The `_cplusplus` define enables the transition from C++ to the C language NI-CAN functions.

The reference for each NI-CAN Channel API function is in Chapter 6, *Channel API for C*. The reference for each NI-CAN Frame API function is in Chapter 9, *Frame API for C*.

You can find examples for the C language in the `MS Visual C` subfolder of the NI-CAN folder. Each example is in a separate folder. A description of each example is in comments at the top of the `.c` file.

At the command prompt, after setting MSVC environment variables (such as with MS `vcvars32.bat`), you can build each example using a command such as:

```
cl -I.. singin.c ..\nicanmsc.lib
```

Borland C/C++

The NI-CAN software supports Borland C/C++ version 5 and later.

The header file and library for Visual C/C++ 6 are in the `Borland C` folder of the NI-CAN folder. The typical path to this folder is `\Program Files\National Instruments\NI-CAN\Borland C`.

To use either NI-CAN API, include the `nican.h` header file in your code, then link with the `nicanbor.lib` library file.

For C applications (files with `.c` extension), include the header file by adding a `#include` to the beginning of your code, such as:

```
#include "nican.h"
```

For C++ applications (files with `.cpp` extension), define `_cplusplus` before including the header, such as:

```
#define _cplusplus

#include "nican.h"
```

The `_cplusplus` define enables the transition from C++ to the C language NI-CAN functions.

The reference for each NI-CAN Channel API function is in Chapter 6, *Channel API for C*. The reference for each NI-CAN Frame API function is in Chapter 9, *Frame API for C*.

You can find examples for the C language in the `Borland C` subfolder of the NI-CAN folder. Each example is in a separate folder. A description of each example is in comments at the top of the `.c` file.

Other Programming Languages

The NI-CAN software does not provide formal support for programming languages other than those described in the preceding sections. Nevertheless, you may find libraries and examples for other programming languages on the National Instruments Web site, `ni.com`.

If your programming language provides a mechanism to call a Dynamic Link Library (DLL), you can create your own code to call NI-CAN functions. All functions for the Channel API and Frame API are in `nican.dll`.

If your programming language supports the Microsoft Win32 APIs, you can load pointers to NI-CAN functions in your application. The following text demonstrates use of the Win32 functions for C/C++ environments other than Visual C/C++ 6. For more detailed information, refer to Microsoft documentation.

The following C language code fragment shows how to call Win32 `LoadLibrary` to load the NI-CAN Channel API's DLL:

```
#include <windows.h>

#include "nican.h"

HINSTANCE NicanLib = NULL;

NicanLib = LoadLibrary("nican.dll");
```

Next, your application must call the Win32 `GetProcAddress` function to obtain a pointer to each NI-CAN function that your application will use. For each NI-CAN function, you must declare a pointer variable using the function's prototype. For the prototypes of each NI-CAN function, refer to the C language chapters in this manual.

```
static nctTypeStatus (NCT_FUNC * PnctInitStart)
    (const str TaskList, i32 Interface, i32 Direction,
    f64 SampleRate, nctTypeTaskRef * TaskRef);
```

```
static nctTypeStatus (NCT_FUNC * PnctRead)
   (nctTypeTaskRef TaskRef, u32 NumberOfSamplesToRead,
   nctTypeTimestamp * StartTime, nctTypeTimestamp *
   DeltaTime, f64 * SampleArray, u32 *
   NumberOfSamplesReturned);

static nctTypeStatus (NCT_FUNC * PnctClear)
   (nctTypeTaskRef TaskRef);

PnctInitStart = (nctTypeStatus (NCT_FUNC *)
   (const str, i32, i32, f64, nctTypeTaskRef *))
   GetProcAddress(NicanLib, (LPCSTR)"nctInitStart");

PnctRead = (nctTypeStatus (NCT_FUNC *)
   (nctTypeTaskRef, u32, nctTypeTimestamp *,
   nctTypeTimestamp *, f64 *, u32 *))
   GetProcAddress(NicanLib, (LPCSTR)"nctRead");

PnctClear = (nctTypeStatus (NCT_FUNC *)
   (nctTypeTaskRef))
   GetProcAddress(NicanLib, (LPCSTR)"nctClear");
```

Your application must de-reference the pointer to call the NI-CAN function, as shown by the following code:

```
nctTypeStatus status;

nctTypeTaskRef TaskRef;

status = (*PnctInitStart)("mychannel1, mychannel2", 0,
   nctModeInput, 1000.0, &TaskRef);
```

Before exiting your application, you must unload the NI-CAN DLL as follows:

```
FreeLibrary(NicanLib);
```

Choose Which API To Use

For a given NI-CAN interface such as **CAN0**, you can use only one API at a time. Therefore, for new application development, you need to decide which API to use.

For example, if you have one application that uses the Channel API and another application that uses the Frame API, you cannot use **CAN0** with both at the same time. As an alternative, you can connect **CAN0** and **CAN1** to the same network, then use **CAN0** with one application and **CAN1** with the other, if you have a 2-port CAN card. As another alternative, you can use **CAN0** in both applications, but run each application at a different time (not simultaneously).

Because the Channel API provides access to the CAN network in easy-to-use physical units, it is recommended over the Frame API for customers who are getting started with NI-CAN.

Nevertheless, because the Frame API provides lower-level access to the CAN network, there are a few reasons why you might want to use it over the Channel API:

- You are continuing with an application developed with NI-CAN version 1.6 or earlier. The Frame API is compatible with such code.
- You need to implement a command/response protocol in which you send a command to the device, and then the device replies by sending a response. Command/response protocols typically use a fixed pair of IDs for each device, and the ID does not determine the meaning of the data bytes.
- Your devices require use of remote frames. The Channel API does not provide support for remote frames, but the Frame API has extensive features to transmit and receive remote frames. For more information, refer to the *Remote Frames* section of Chapter 7, *Using the Frame API*.
- The Frame API provides RTSI features that are lower level than the synchronization features of the Channel API. If you have advanced requirements for synchronizing CAN and DAQ cards, you may need to use the Frame API. For more information, refer to the *RTSI* section of Chapter 7, *Using the Frame API*.

4

Using the Channel API

This chapter helps you get started with the Channel API.

Choose Source of Channel Configuration

The first step in using the Channel API is to create the channel configuration for your applications. This channel configuration describes how the NI-CAN software converts raw data in messages to/from the physical units of each channel.

The NI-CAN software provides various methods to create the channel configuration. The flowchart in Figure 4-1 shows a process you can use to decide the source of your channel configuration. A description of each step in the decision process follows the flowchart.

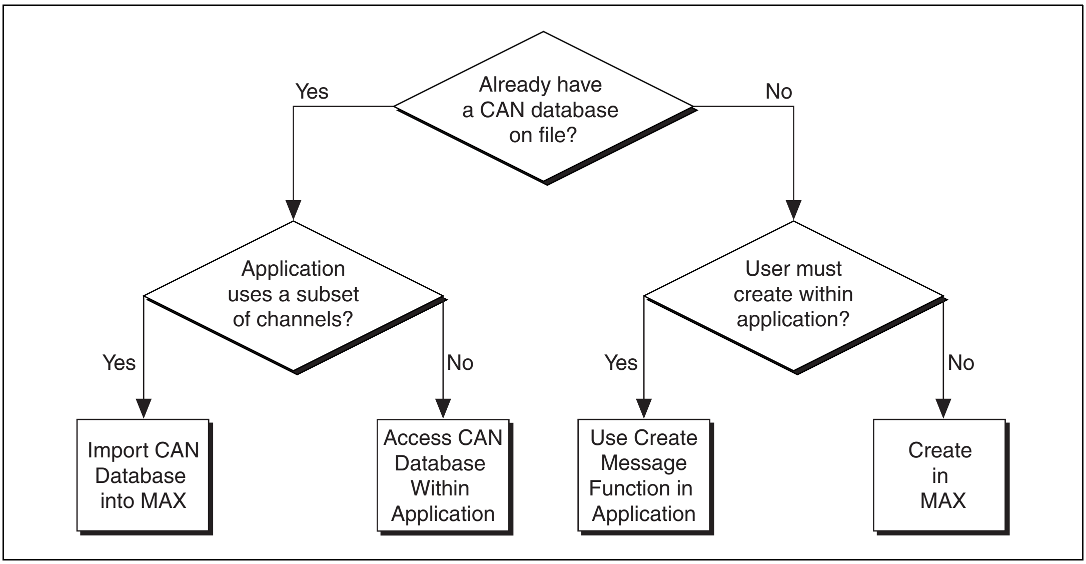


Figure 4-1. Decision Process for Choosing Source of Channel Configuration

Already Have a CAN Database File?

If you have a CAN database file, the channel configuration has already been created using a tool such as Vector's CANdb Editor. You can use each signal name in the CAN database as a channel name in the NI-CAN Channel API.

If you answer yes, refer to the *Application Uses a Subset of Channels?* section. If you answer no, refer to the *User Must Create within Application?* section.

Application Uses a Subset of Channels?

If your CAN database file contains a large number of channel descriptions (1,000 or more), does your application use only a subset of these channels (100 or less)? Importing the channels into MAX provides many benefits, but managing the transfer of large amounts of data from CAN databases can be cumbersome. For example, if the large CAN database file is updated periodically, you need to ensure that the changes are reflected in MAX after each update.

If you answer yes, refer to the *Import CAN Database into MAX* section. If you answer no, refer to the *Access CAN Database within Application* section.

Import CAN Database into MAX

The benefits of importing channels into MAX include:

- You can initialize the channel name alone within the Channel API. No path to the CAN database file is required.
- You can use the **Test Panel** in MAX to read and write the channels.

You can download the channel configuration to a LabVIEW RT system using **Send to RT System**.

To import channel configurations from a Vector CANdb file into MAX, right-click the **CAN Channels** heading, then select **Import from CANdb File**. Use shift-click to select multiple channels, and then select **Import**. If you need to select another set, you can select the channels and then **Import** again. When you are finished with the import, select **Done** to return to MAX.

Access CAN Database within Application

To access the CAN database within your application, you must initialize the channel name with the file path as a prefix. For example, if you are using a channel named `EngineRPM` in the `C:\DBC_Files\Prototype.DBC` file, you pass the following name to the Init Start function:

```
C:\DBC_Files\Prototype.DBC::EngineRPM
```

For more information, refer to the description of the Init Start function in the Channel API reference chapters.

If you are using a LabVIEW RT system, you must copy the CAN database file to the hard drive of that system, then access that file path within your application.

User Must Create within Application?

Are you developing an application that another person will use, and that person must create the channel configuration using the application itself?

If you answer yes, refer to the *Use Create Message Function in Application* section.

If you answer no, you create the channel configuration within MAX. You can save the MAX channel configuration to a file, so this method does not prevent you from deploying your application for use by others. For more information, refer to the *Create in MAX* section.

Use Create Message Function in Application

The Create Message function (**CAN Create Message** in LabVIEW and `nctCreateMessage` in other languages) takes inputs for a single message configuration, then one or more channel configurations. By using Create Message to create the channel configurations, your application is entirely self contained, not depending on MAX or a CAN database file.

The inputs to Create Message are relatively advanced for many users. Use of MAX or a CAN database helps to isolate the application end user from the specifics of CAN message encoding.

Create in MAX

To create channel configurations within MAX, right-click the **CAN Channels** heading, then select **Create Message**. Enter the message properties, then select **OK**. Right-click the message name, then select **Create Channel**. Enter the channel properties, then select **OK**. Select **Create Channel** again for each channel contained in the message.

To save channel configurations to a file, right-click the **CAN Channels** heading, then select **Save Channel Configuration**. The resulting NI-CAN database uses file extension `.ncd`. You can access the NI-CAN database using the Init Start function just like any other CAN database. By simply installing the NI-CAN database file along with your application, you can deploy your application to a variety of users.

Basic Programming Model

When you use the Channel API, the first step is to initialize a list of channels with the same direction, such as input or output. You can then read or write this list of channels as a unit. The term *task* refers to a list of channels you read or write together. A common use of the task concept is to read/write all channels of a message.

The diagram in Figure 4-2 describes the basic programming model for the NI-CAN Channel API. Within your application, you repeat this basic programming model for each task. The diagram is followed by a description of each step in the model.

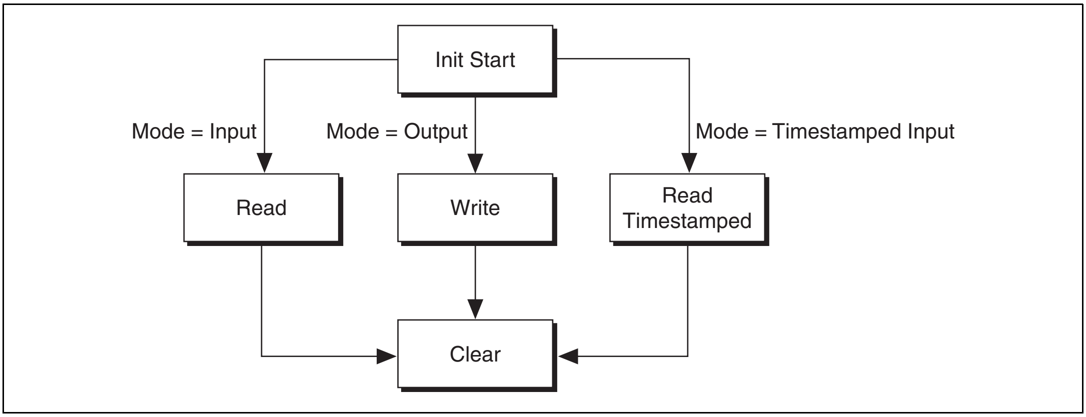


Figure 4-2. Basic Programming Model for Channel API

Init Start

The Init Start function initializes a list of channels as a single task, then starts communication for that task.

The Init Start function uses the following input parameters:

- **channel list**—Specifies the list of channels for the task, with one string for each channel.
- **interface**—Specifies the CAN interface to use for the task. The interface is an enumeration in which `0` specifies CAN0, `1` specifies CAN1, and so on. The baud rate is taken from the interface's properties in MAX.
- **mode**—Specifies the I/O mode to use for the task. This determines the direction of data transfer for the task (that is, Input or Output). It also determines the type of Read or Write function you use with the task. For more information, refer to the following sections.
- **sample rate**—Specifies the rate of sampling for input and output modes. The sample rate is specified in Hertz (samples per second). For more information, refer to the *Read* and *Write* sections.

The Init Start function simply calls the Initialize function followed by the Start function. This provides an easy way to start a list of channels.

There are a few scenarios in which you cannot use Init Start:

- **Set Property**—If you need to set properties for the task, you must call Initialize, Set Property, and Start in sequence. For example, use Set Property if you need to specify the baud rate for the interface within your application. For more information, refer to the *Set Property* section.
- **Synchronization**—If you need to synchronize multiple cards, you must call Initialize, then the appropriate functions to synchronize and start the cards. For more information, refer to the *Synchronization* section.
- **Create Message**—If you need to create channel configurations within your application, you must call Create Message and Start in sequence. For assistance is deciding whether Create Message is appropriate for your application, refer to the *Choose Source of Channel Configuration* section.

The Init Start function is **CAN Init Start** in LabVIEW and `nctInitStart` in other languages.

Read

If the mode of Init Start is Input, your application must call the Read function to obtain floating-point samples. Your application typically calls Read in a loop until done.

The Read function is **CAN Read** in LabVIEW (all types that don't end in **Time & Dbl**) and `nctRead` in other languages.

The behavior of Read depends on the initialized sample rate:

sample rate = 0

Read returns a single sample from the most recent message(s) received from the network. One sample is returned for every channel in the Init Start list.

Figure 4-3 shows an example of Read with sample rate = 0. A, B, and C represent messages for the initialized channels. If no message is received since the start of the application, the Default Value in MAX (*def*) is returned, along with a warning.

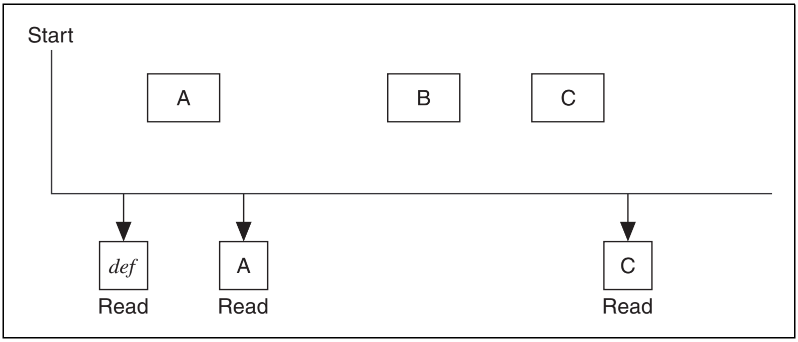


Figure 4-3. Example of Read with sample rate = 0

sample rate > 0

Read returns an array of samples for every channel in the Init Start list. Each time the clock ticks at the specified rate, a sample from the most recent message(s) is inserted into the arrays. In other words, the samples are repeated in the array at the specified rate until a new message is received. By using the same sample rate with NI-DAQ Analog Input channels, you can compare CAN and DAQ samples over time.

Figure 4-4 shows an example of Read with sample rate > 0. A, B, and C represent messages for the initialized channels. <delta–t> represents the time between samples as specified by the sample rate. *def* represents the Default Value in MAX.

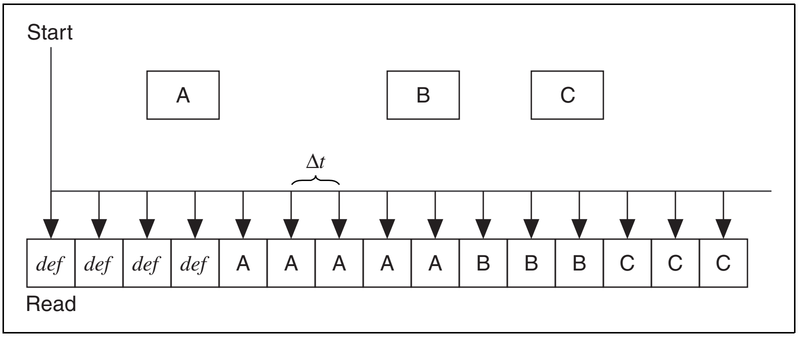


Figure 4-4. Example of Read with sample rate > 0

Read Timestamped

If the Init Start mode is Timestamped Input, your application must call the Read Timestamped function to obtain floating-point samples. Your application typically calls Read Timestamped in a loop until done.

The Read Timestamped function returns samples that correspond to messages received from network. For each message, an associated sample is returned along with a timestamp that specifies when the message arrived. An array of timestamped samples is returned for every channel in the Init Start list.

The Read Timestamped function is **CAN Read** in LabVIEW (types that end in **Time & Dbl**) and `nctReadTimestamped` in other languages.

Figure 6-5 shows an example of Read Timestamped. A, B, and C represent messages for the initialized channels. A*t*, B*t*, and C*t* represent the times when each message was received.

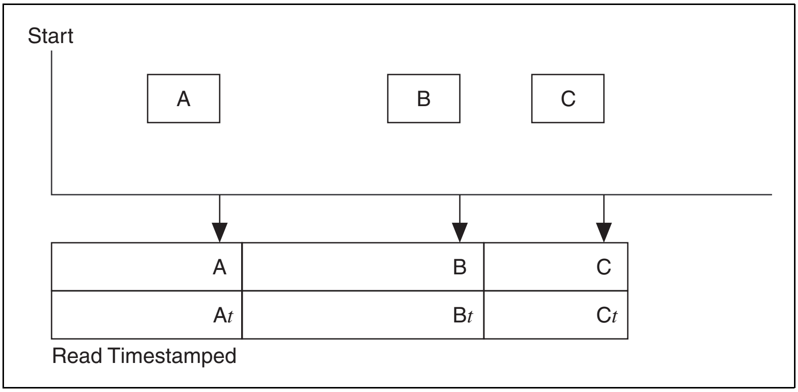


Figure 4-5. Example of Read Timestamped

Write

If the Init Start mode is Output, your application must call the Write function to output floating-point samples. Your application typically calls Write in a loop until done.

The Write function is **CAN Write** in LabVIEW and `nctWrite` in other languages.

The behavior of Write depends on the initialized sample rate:

sample rate = 0

Write transmits a message immediately on the network. The samples provided to write are used to form the message's data bytes. One sample must be specified for every channel in the Init Start list.

Figure 4-6 shows an example of Write with sample rate = 0. A, B, C and D represent messages for the initialized channels. For each Write, the associated messages are transmitted as quickly as possible.

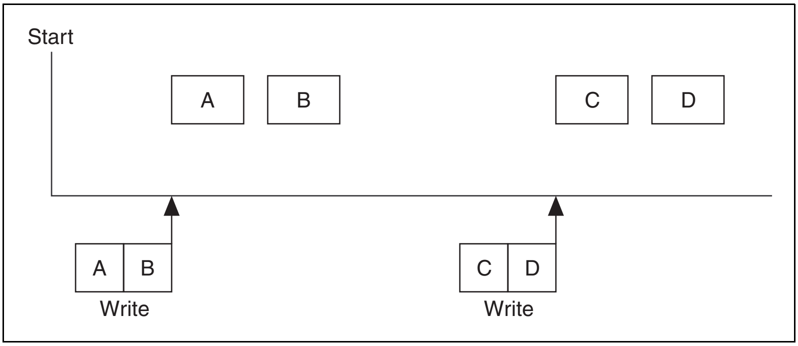


Figure 4-6. Example of Write with sample rate = 0

sample rate > 0

You provide an array of samples for every channel in the Init Start list. Each time the clock ticks at the specified rate, the next message is transmitted. Each message uses the next sample from the array(s) to form the message's data bytes. In other words, the samples from the array are transmitted periodically onto the network. By using the same sample rate with NI-DAQ Analog Output channels, you can output synchronized CAN and DAQ samples over time.

Figure 4-7 shows an example of Write with sample rate > 0. A, B, C and D represent messages for the initialized channels. <delta-t> represents the time between message transmission as specified by the sample rate.

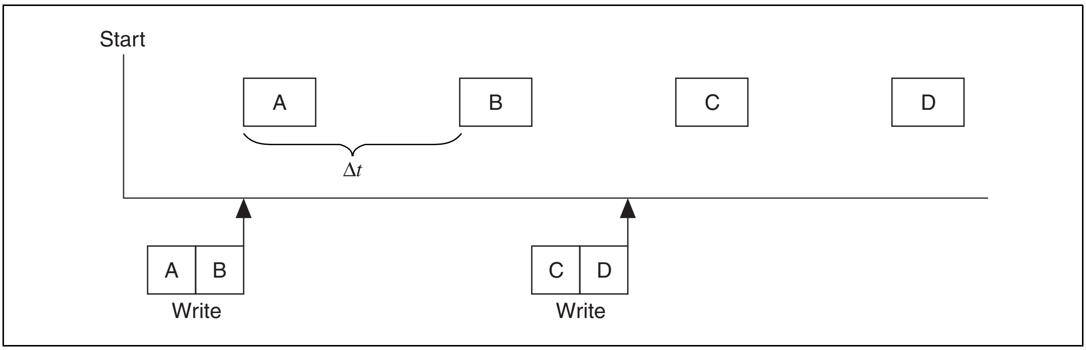


Figure 4-7. Example of Write with sample rate > 0

Clear

The Clear function stops communication for the task, then clears the configuration.

For every task that you initialize, you must call Clear prior to exiting your application.

The Clear function is **CAN Clear** in LabVIEW and `nctClear` in other languages.

Additional Programming Topics

The following sections provide information you can use to extend the basic programming model.

Get Names

If you are developing an application that another person will use, you may not want to specify a fixed channel list in your application. Ideally, you want your end-user to select the channels of interest from user interface controls, such as list boxes.

The Get Names function queries MAX or a CAN database and returns a list of all channels in that database. You can use this list to populate user-interface controls. Your end-user can then select channels from these controls, avoiding the need to type each name using the keyboard. Once the user makes his selections, your application can pass the resulting list to Init Start.

The Get Names function is **CAN Get Names** in LabVIEW and `nctGetNames` in other languages.

Synchronization

The NI-CAN Channel API uses RTSI to synchronize specific functional units on each card. For CAN cards, the functional unit is the interface (port). For DAQ cards, the functional unit is a specific measurement such as Analog Input or Analog Output. Each function routes two signals over the RTSI connection:

- **timebase**—This is a common clock shared by both cards. The shared timebase ensures that sampling does not drift. The timebase applies to all functional units on the card.

- **start trigger**—This signal is sent from one functional unit to the other functional unit when sampling starts. The shared start trigger ensures that both units start simultaneously.

Set Property

The Init Start function uses interface and channel configuration as specified in MAX or the CAN database file. If you need to change this configuration within your application, you cannot use Init Start, because most properties cannot be changed while the task is running.

For example, to set the baud rate for the interface within your application, use the following calling sequence:

- Initialize the task as stopped. The Initialize function is **CAN Initialize** in LabVIEW and `nctInitialize` in other languages.
- Use Set Property to specify the new value for the baud rate property. The Set Property function is **CAN Set Property** in LabVIEW and `nctSetProperty` in other languages.
- Start the task with the Start function. The Start function is **CAN Start** in LabVIEW and `nctStart` in other languages.

After the task is started, you may need to change properties again. To change properties within the application, use the Stop function to stop the task, Set Property to change properties, and then Start the task again.

You can also use the Get Property function to get the value of any property. The Get Property function returns values whether the task is running or not.

5

Channel API for LabVIEW

This chapter lists the LabVIEW VIs for the NI-CAN Channel API and describes the format, purpose, and parameters for each VI. The VIs in this chapter are listed alphabetically.

Unless otherwise stated, each NI-CAN VI suspends execution of the calling thread until it completes.

Section Headings

The following are section headings found in the Channel API for LabVIEW VIs.

Purpose

Each VI description includes a brief statement of the purpose of the VI.

Format

The format section describes the format of each VI.

Input and Output

The input and output parameters for each VI are listed.

Description

The description section gives details about the purpose and effect of each VI.

List of VIs

The following table is an alphabetical list of the NI-CAN VIs for the Channel API.

Table 5-1. Channel API for LabVIEW VIs

Function	Purpose
CAN Clear	Stop communication for the task and then clear the configuration.
CAN Clear with NI-DAQ	Stop and clear the CAN task and the NI-DAQ task synchronized with **CAN Sync Start with NI-DAQ.vi**.
CAN Clear Multiple with NI-DAQ	Stop and clear the list of CAN tasks and the list of NI-DAQ tasks synchronized with **CAN Sync Start Multiple with NI-DAQ.vi**.
CAN Connect Terminals	Connect terminals in the CAN hardware.
CAN Create Message	Create a message configuration and associated channel configurations within your LabVIEW application.
CAN Disconnect Terminals	Disconnect terminals in the CAN hardware.
CAN Get Names	Get an array of CAN channel names or message names from MAX or a CAN database file.
CAN Get Property	Get a property for the task, or a single channel within the task. The poly VI selection determines the property to get.
CAN Initialize	Initialize a task for the specified channel list.
CAN Init Start	Initialize a task for the specified channel list, then start communication.
CAN Read	Read samples from a CAN task initialized as input. Samples are obtained from received CAN messages. The poly VI selection determines the data type to read.
CAN Set Property	Set a property for the task, or a single channel within the task. The poly VI selection determines the property to set.
CAN Start	Start communication for the specified task.
CAN Stop	Stop communication for the specified task.

Table 5-1. Channel API for LabVIEW VIs (Continued)

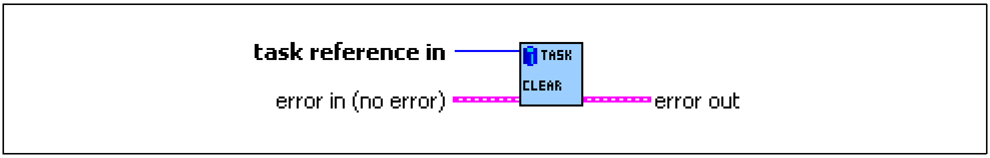
Function	Purpose
CAN Sync Start with NI-DAQ	Synchronize and start the specified CAN task and NI-DAQ task.
CAN Sync Start Multiple with NI-DAQ	Synchronize and start the specified list of multiple CAN tasks and NI-DAQ tasks. This is a more complex implementation of **CAN Sync Start with NI-DAQ.vi** that supports multiple CAN and NI-DAQ hardware products.
CAN Write	Write samples to a CAN task initialized as Output. (Refer to the **mode** parameter of **CAN Init Start.vi**.) Samples are placed into transmitted CAN messages. The poly VI selection determines the data type to write.

CAN Clear.vi

Purpose

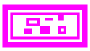
Stop communication for the task and then clear the configuration.

Format



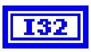
Inputs

task reference in is the task reference from the previous NI-CAN VI. The task reference is originally returned from **CAN Init Start.vi**, **CAN Initialize.vi**, or **CAN Create Message.vi**, and then wired through subsequent VIs.



Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. Unlike other VIs, this VI will execute when **status** is TRUE.



code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

The **CAN Clear** VI must always be the final NI-CAN VI called for each task. If you do not use the **CAN Clear** VI, the remaining task configurations can cause problems in execution of subsequent NI-CAN applications.

If the cleared task is the last running task for the initialized interface (refer to **CAN Init Start.vi**), the **CAN Clear** VI also stops communication on the interface's CAN controller and disconnects all terminal connections for that interface.

Unlike other VIs, this VI will execute when **status** is TRUE in **Error in**.

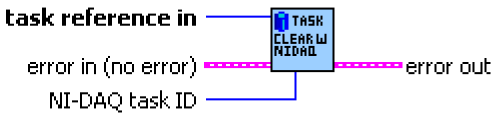
Because this VI clears the task, the task reference is not wired as an output. To change properties of a task and start again, use **CAN Stop.vi**.

CAN Clear with NI-DAQ.vi

Purpose

Stop and clear the CAN task and the NI-DAQ task synchronized with **CAN Sync Start with NI-DAQ.vi**.

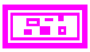
Format



Inputs

task reference in is the NI-CAN task reference you passed through the **CAN Sync Start with NI-DAQ** VI.

NI-DAQ task ID is the same NI-DAQ task ID you wired into the **CAN Sync Start with NI-DAQ** VI.



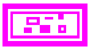
Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

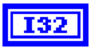
source identifies the VI where the error occurred.

Outputs



Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.



code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

Both tasks are cleared to their state prior to **CAN Sync Start with NI-DAQ**. For example, this VI clears terminal routing of the NI-DAQ device to the default state.

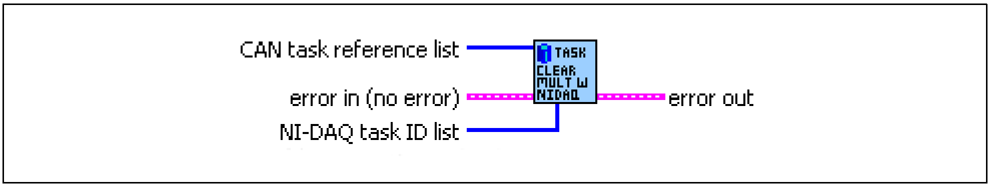
This VI is intended to serve as an example. You can use the VI as is, but the LabVIEW diagram is commented so that you can use the VI as a starting point for more complex synchronization scenarios. Before you customize the LabVIEW diagram, save a copy of the VI for your own editing.

CAN Clear Multiple with NI-DAQ.vi

Purpose

Stop and clear the list of CAN tasks and the list of NI-DAQ tasks synchronized with **CAN Sync Start Multiple with NI-DAQ.vi**.

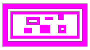
Format



Inputs

CAN task reference list is the same array of NI-CAN task references you wired into the **CAN Sync Start Multiple with NI-DAQ** VI.

NI-DAQ task ID list is the same array of NI-DAQ task IDs you wired into the **CAN Sync Start Multiple with NI-DAQ** VI.



Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

All tasks are cleared to their state prior to **CAN Sync Start Multiple with NI-DAQ**. For example, this VI clears terminal routing of all NI-DAQ devices to the default state.

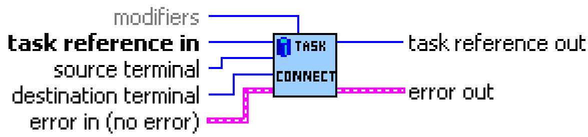
This VI is intended to serve as an example. You can use the VI as is, but the LabVIEW diagram is commented so you can use the VI as a starting point for more complex synchronization scenarios. Before you customize the LabVIEW diagram, save a copy of the VI for your own editing.

CAN Connect Terminals.vi

Purpose

Connect terminals in the CAN hardware.

Format



Inputs

task reference in is the task reference from the previous NI-CAN VI. The task reference is originally returned from **CAN Init Start.vi**, **CAN Initialize.vi**, or **CAN Create Message.vi**, and then wired through subsequent VIs.

source terminal specifies the source of the connection.

Once the connection is successfully created, behavior flows from **source terminal** to **destination terminal**.

For a list of valid source/destination pairs, refer to the Valid Combinations of Source/Destination section.

The following list describes each value of **source terminal**:

RTSI0 … RTSI6

Selects a general-purpose RTSI line as source (input) of the connection.

10 Hz Resync Event

10 Hz Resync Event selects a 10 Hz, 50 percent duty cycle clock. This slow rate is required for resynchronization of Series 1 CAN cards. On each pulse of the resync clock, the other CAN card brings its clock into sync.

By selecting **RTSI0** to **RTSI6** as the **destination terminal**, you route the 10 Hz clock to synchronize with other Series 1 CAN cards. NI-DAQ cards cannot use the 10 Hz resync clock, so this selection is limited to synchronization of two or more Series 1 CAN cards.

10 Hz Resync Event applies to the entire CAN card, including both interfaces of a 2-port CAN card. The CAN card is specified by the task interface, such as the **interface** input to **CAN Initialize.vi**.

Start Trigger Event

Start Trigger Event selects the start trigger, the event that begins sampling for tasks.

The start trigger is the same for all tasks using a given interface, such as the **interface** input to **CAN Initialize.vi**.

In the default (disconnected) state of the **Start Trigger** destination, the start trigger occurs when communication begins on the interface.

By selecting **RTSI0** to **RTSI6** as the **destination terminal**, you route the start trigger of this CAN card to the start trigger of other CAN or DAQ cards. This ensures that sampling begins at the same time on both cards. For example, you can synchronize two CAN cards by routing **Start Trigger Event** as the **source terminal** on one CAN card and then routing **Start Trigger** as the **destination terminal** on the other CAN card, with both cards using the same RTSI line for the connections.

destination terminal specifies the destination of the connection.

The following list describes each value of **destination terminal**:

RTSI0 … RTSI6

Selects a general-purpose RTSI line as destination (output) of the connection.

10 Hz Resync

10 Hz Resync instructs the CAN card to use a 10 Hz, 50 percent duty cycle clock to resynchronize its local timebase. This slow rate is required for resynchronization of CAN cards. On each pulse of the resync clock, this CAN card brings its local timebase into sync.

When synchronizing to an E-series MIO card, a typical use of this value is to use **RTSI0** to **RTSI6** as the **source terminal**, then use NI-DAQ functions to program the MIO card's Counter 0 to generate a 10 Hz 50 percent duty cycle clock on the RTSI line. For an example, refer to **CAN Sync Start with NI-DAQ.vi**.

When synchronizing to a CAN card, a typical use of this value is to use **RTSI0** to **RTSI6** as the **source terminal**, then route the other CAN card's **10 Hz Resync Event** as the **source terminal** to the same RTSI line.

10 Hz Resync applies to the entire CAN card, including both interfaces of a 2-port CAN card. The CAN card is specified by the task interface, such as the **interface** input to **CAN Initialize.vi**.

The default (disconnected) state of this destination means the CAN card does not resynchronize its local timebase.

Start Trigger

Start Trigger selects the start trigger, the event that begins sampling for tasks.

The start trigger is the same for all tasks using a given interface, such as the **interface** input to **CAN Initialize.vi**.

By selecting **RTSI0** to **RTSI6** as the **source terminal**, you route the start trigger from another CAN or DAQ card. This ensures that sampling begins at the same time on both cards. For example, you can synchronize with an E-Series DAQ MIO card by routing the MIO card's **AI start trigger** to a RTSI line and then routing the same RTSI line with **Start Trigger** as the **destination terminal** on the CAN card.

The default (disconnected) state of this destination means the start trigger occurs when communication begins on the interface. Because communication begins when the first interface task is started, this does not synchronize sampling with other NI cards.

modifiers provides optional connection information for certain source/destination pairs. The current release of NI-CAN does not use this information for any source/destination pair, so **modifiers** must be left unwired.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

task reference out is the same as **task reference in**. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

This VI connects a specific pair of source/destination terminals. One of the terminals is typically a RTSI signal, and the other terminal is an internal terminal in the CAN hardware. By connecting internal terminals to RTSI, you can synchronize the CAN card with another hardware product such as an NI-DAQ card.

The most common uses of RTSI synchronization are demonstrated by the **CAN Sync Start with NI-DAQ.vi** and **CAN Sync Start Multiple with NI-DAQ.vi** example VIs. The diagram for each of these example VIs uses **CAN Connect Terminals**, and therefore serves as a good starting point when learning this VI.

When the final task for a given interface is cleared with **CAN Clear.vi**, NI-CAN disconnects all terminal connections for that interface. Therefore, **CAN Disconnect Terminals.vi** is not required for most applications. NI-DAQ terminals remain connected after the tasks are cleared, so you must disconnect NI-DAQ terminals manually at the end of your application.

For a list of valid source/destination pairs, refer to the following section.

Valid Combinations of Source/Destination

Table 5-2 lists all valid combinations of **source terminal** and **destination terminal**.

NI-CAN hardware has the following limitations.

- PXI cards do not support **RTSI 6**.
- Signals received from a RTSI source cannot occur faster than 1 kHz. This prevents the card from receiving a 10 MHz or 20 MHz timebase, such as NI E-Series MIO hardware provides.
- Signals received from a RTSI source must be at least 100 μs in length to be detected. This prevents the card from receiving triggers in the nanoseconds range, such as the AI trigger that E-Series MIO hardware provides.

Table 5-2. Valid Combinations of Source/Destination

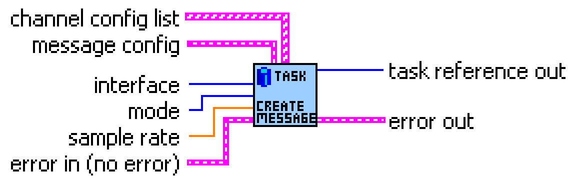
Source	Destination		
	RTSI0 to RTSI6	**10 Hz Resync**	**Start Trigger**
RTSI0 to RTSI6	—	X	X
10 Hz Resync Event	X	—	X
Start Trigger Event	X	—	—

CAN Create Message.vi

Purpose

Create a message configuration and associated channel configurations within your LabVIEW application.

Format



Inputs

interface specifies the CAN interface to use for this task.

The interface input uses a ring typedef in which value 0 selects **CAN0**, value 1 selects **CAN1**, and so on.

The **interface** input is required.

The default baud rate for the **interface** is defined within MAX, but you can change it by setting the Interface Baud Rate property with **CAN Set Property.vi**.

mode specifies the I/O mode for the task, as follows:

Input

Input channel data from received CAN messages. Use **CAN Read.vi** to obtain input samples as single point, array, or waveform.

Use this input mode to read waveforms of timed samples, such as for comparison with NI-DAQ waveforms. You can also use this input mode to read a single point from the most recent message, such as for control or simulation.

Output

Output channel data to CAN messages for transmit. Use **CAN Write.vi** to write output samples as single-point, array, or waveform.

Timestamped Input

Input channel data from received CAN messages. Use **CAN Read.vi** to obtain input samples as an array of sample/timestamp pairs (Poly VI types ending in *Timestamped Dbl*).

Use this input mode to read samples with timestamps that indicate when each message is received from the network.

sample rate specifies the timing to use for samples of the task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately.

For **mode** of **Input**, a **sample rate** of zero means that **CAN Read** returns a single point from the most recent message received, and greater than zero means that **CAN Read** returns samples timed at the specified rate.

For **mode** of **Output**, a **sample rate** of zero means that CAN messages transmit immediately when **CAN Write** is called, and greater than zero means that CAN messages are transmitted periodically at the specified rate.

For **mode** of **Timestamped Input**, **sample rate** is ignored.

message config configures properties for a new message. These properties are similar to the message properties in MAX. **Can Create Message.vi** creates a task for a single message with one or more channels.

message ID

Configures the arbitration ID of the message.

Use the **ID is Extended?** property to specify whether the ID is standard (11-bit) or extended (29-bit).

extended ID?

Configures a Boolean value that indicates whether the arbitration ID of the message is standard 11-bit format (false) or extended 29-bit format (true).

number of bytes

Configures the number of data bytes in the message. The range is 0 to 8.

channel config list configures a list of channels for the new message. The **channel config list** is an array of clusters, with one cluster for each channel. The properties of each channel entry are similar to the channel properties in MAX:

start bit

Configures the starting bit position in the message. The range is 0 (lowest bit in first byte) to 63 (highest bit in last byte).

number of bits

Configures the number of bits for the raw data in the message. The range is 0 to 64.

byte order

Configures the channel's byte order in the message.

The value of **byte order** is an enumeration:

0	**Intel**	Bytes are in little-endian order, with most-significant first.
1	**Motorola**	Bytes are in big-endian order, with least-significant first.

data type

Configures the channel's data type in the message.

The value of **Channel Data Type** is an enumeration:

0	**Signed**	Raw data in the message is a signed integer.
1	**Unsigned**	Raw data in the message is an unsigned integer.
2	**IEEE Float**	Raw data in the message is floating-point; no scaling required.

scaling factor

Configures the scaling factor used to convert raw data in the message to/from scaled floating-point units. The scaling factor is the A in the linear scaling formula $AX + B$, where X is the raw data, and B is the scaling offset.

scaling offset

Configures the scaling offset used to convert raw data in the message to/from scaled floating-point units. The scaling offset is the B in the linear scaling formula $AX + B$, where X is the raw data, and A is the scaling factor.

min value

Configures the minimum value of the channel in scaled floating-point units.

The **CAN Read** and **CAN Write** VIs do not coerce samples when converting to/from CAN messages. You can use this value with property nodes to set the range of front-panel controls and indicators.

max value

Configures the maximum value of the channel in scaled floating-point units.

The **CAN Read** and **CAN Write** VIs do not coerce samples when converting to/from CAN messages. You can use this value with property nodes to set the range of front-panel controls and indicators.

default value

Configures the default value of the channel in scaled floating-point units.

For information on how the **default value** is used, refer to **CAN Read.vi** and **CAN Write.vi**.

unit string

Configures the channel unit string. The string is no more than 64 characters in length.

You can use this value to display units (such as volts or RPM) along with the channel's samples.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

Outputs

Use **task reference out** with all subsequent VIs to reference the task. Wire this task reference to **CAN Start.vi** before you read or write samples for the message.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

To use message and channel configurations from MAX or a CAN database, use **CAN Init Start.vi** or **CAN Initialize.vi**. The **CAN Create Message** provides an alternative in which you create the message and channel configurations within your application, without use of MAX or a CAN database.

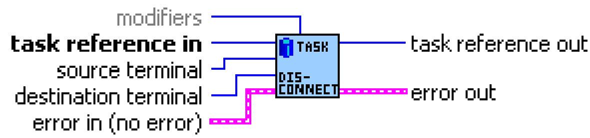
CAN Create Message returns a task reference that you wire to **CAN Start.vi** to start communication for the message and its channels.

CAN Disconnect Terminals.vi

Purpose

Disconnect terminals in the CAN hardware.

Format



Inputs

task reference in is the task reference from the previous NI-CAN VI. The task reference is originally returned from **CAN Init Start.vi**, **CAN Initialize.vi**, or **CAN Create Message.vi**, and then wired through subsequent VIs.

source terminal specifies the connection source.

For a description of values for **source terminal**, refer to **CAN Connect Terminals.vi**.

destination terminal specifies the connection destination.

For a description of values for **destination terminal**, refer to **CAN Connect Terminals.vi**.

modifiers provides optional connection information for certain source/destination pairs. The current release of NI-CAN does not use this information for any source/destination pair, so **modifiers** must be left unwired.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

task reference out is the same as **task reference in**. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

This VI disconnects a specific pair of source/destination terminals that you previously connected with **CAN Connect Terminals.vi**.

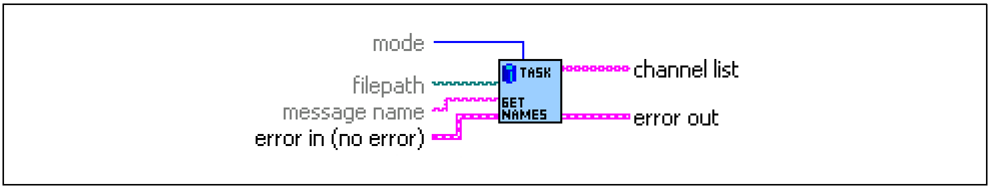
When the final task for a given interface is cleared with **CAN Clear.vi**, NI-CAN disconnects all terminal connections for that interface. Therefore, the **CAN Disconnect Terminals** VI is not required for most applications. You typically use this VI to change RTSI connections dynamically while your application is running. First, use **CAN Stop.vi** to stop all tasks for the interface, then use **CAN Disconnect Terminals** and **CAN Connect Terminals** to adjust RTSI connections, then **CAN Start.vi** to restart sampling.

CAN Get Names.vi

Purpose

Get an array of CAN channel names or message names from MAX or a CAN database file.

Format



Inputs

filepath is an optional path to a CAN database file from which to get channel names. The file must use either a `.DBC` or `.NCD` extension. Files with extension `.DBC` use the CANdb database format. Files with extension `.NCD` use the NI-CAN database format. You can generate NI-CAN database files from the Save Channels or FP 1300 selection in MAX.

The default (unwired) value of **filepath** is empty, which means to get the channel names from MAX. The MAX CAN channels are in the MAX CAN Channels listing within **Data Neighborhood**.

message name is an optional input that filters the names for a specific message. The default (unwired) value is an empty string, which means to return all names in the database. If you wire a nonempty string, the **channel list** output is limited to channels of the specified message. This input applies to **mode** of **channels** only. It is ignored for **mode** of **messages**.

mode is an optional input that specifies the type of names to return.

The value of **mode** is an enumeration:

0	**channels**	Return list of channel names. You can write this list to **CAN Init Start**. This is the default value
1	**messages**	Return list of message names.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

channel list returns the array of channel names, one string entry per channel.

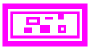
The names in **channel list** use the minimum syntax required to properly initialize the channels:

- If **filepath** is wired, **CAN Get Names** prepends the file path to the first name in **channel list**, with a double colon separating the file path and channel name.
- If a channel name is used within only one message in the database, **CAN Get Names** returns only the channel name in the array. If a channel name is used within multiple messages, **CAN Get Names** prepends the message name to that channel name, with a decimal point separating the message and channel name. This syntax ensures that the duplicate channel is associated to a single message in the database.

For more information on the syntax conventions for channel names, refer to **CAN Init Start.vi**.

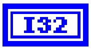
To start a task for all channels returned from **CAN Get Names**, wire **channel list** to the **CAN Init Start** VI to start a task.

You can also wire **channel list** to the property nodes of a front panel control such as a ring or list box. The user of your VI can then select names using this control, and the selected names can be wired to **CAN Init Start**.



Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.



code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

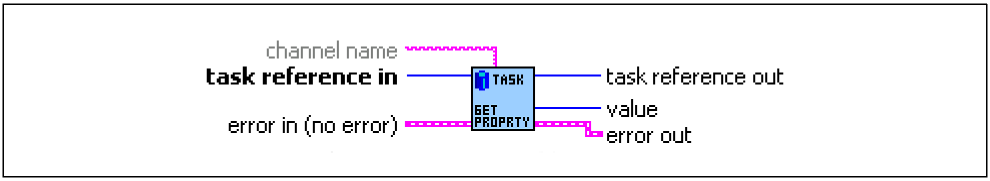
CAN Get Property.vi

Purpose

Get a property for the task, or a single channel within the task. The poly VI selection determines the property to get.

To select the property, right-click the VI, go to **Select Type** and select the property by name.

Format



Inputs

task reference in is the task reference from the previous NI-CAN VI. The task reference is originally returned from **CAN Init Start.vi**, **CAN Initialize.vi**, or **CAN Create Message.vi**, and then wired through subsequent VIs.

channel name specifies an individual channel within the task. The default (unwired) value of channel name is empty, which means the property applies to the entire task, not a specific channel.

Properties that begin with the word *Channel* or *Message* do not apply to the entire task, but an individual channel or message within the task. For these channel-specific properties, you must wire the name of a channel from channel list into the **channel name** input.

For properties that do not begin with the word *Channel* or *Message*, you must leave **channel name** unwired (empty).

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

task reference out is the same as **task reference in**. Wire the task reference to subsequent VIs for this task.

The poly output **value** returns the property value. You select the property returned in **value** by selecting the Poly VI type. The data type of **value** is also determined by the Poly VI selection. For information about the different properties provided by **CAN Get Property**, refer to the Poly VI Types section.

To select the property, right-click the VI, go to **Select Type**, and select the property by name.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Poly VI Types

Number of Channels

Returns the number of channels initialized in channel list. This is the number of array entries required when using **CAN Read** or **CAN Write**.

Timeout

Returns the **Timeout** property, which is used with some input task configurations. For more information, refer to the Timeout property in **CAN Set Property**.

U32

Number of Samples Pending

Returns the number of samples available to be read using **CAN Read**. If you set the number of samples to read input of **CAN Read** to this value, **CAN Read** returns immediately without waiting.

This property applies only to tasks initialized with **mode** of **Input** and **sample rate** greater than zero. For all other configurations, it returns an error.

U16

Behavior After Final Output

Returns the **Behavior After Final Output** property, which is used with some output task configurations. For more information, refer to the Behavior After Final Output property in **CAN Set Property**.

U16

Interface

Returns the interface initialized for the task, such as with the **CAN Init Start VI**.

U16

Mode

Returns the mode initialized for the task, such as with the **CAN Init Start** VI.

DBL

Sample Rate

Returns the sample rate initialized for the task, such as with the **CAN Init Start** VI.

U32

Message ID

Returns the arbitration ID of the channel's message.

To determine whether the ID is standard (11-bit) or extended (29-bit), get the **Message ID is Extended?** property.

The value of this property cannot be changed using **CAN Set Property**.

TF

Message ID is Extended?

Returns a Boolean value that indicates whether the arbitration ID of the channel's message is standard 11-bit format (false) or extended 29-bit format (true).

The value of this property cannot be changed using **CAN Set Property**.

Message Number of Data Bytes

Returns the number of data bytes in the channel’s message. The range is 0 to 8.

The value of this property cannot be changed using **CAN Set Property**.

Message Name

Returns the name of the channel’s message. The string is no more than 80 characters in length.

The value of this property cannot be changed using **CAN Set Property**.

Channel Start Bit

Returns the starting bit position in the message. The range is 0 (lowest bit in first byte) to 63 (highest bit in last byte).

The value of this property cannot be changed using **CAN Set Property**.

Channel Number of Bits

Returns the number of bits in the message. The range is 0 to 64.

The value of this property cannot be changed using **CAN Set Property**.

Channel Byte Order

Returns the channel’s byte order in the message.

The value of **Channel Byte Order** is an enumeration:

0	**Intel**	Bytes are in little-endian order, with most-significant first.
1	**Motorola**	Bytes are in big-endian order, with least-significant first.

The value of this property cannot be changed using **CAN Set Property**.

Channel Data Type

Returns the channel’s data type in the message.

The value of **Channel Data Type** is an enumeration:

0	**Signed**	Raw data in the message is a signed integer.
1	**Unsigned**	Raw data in the message is an unsigned integer.

2	**IEEE Float**	Raw data in the message is floating-point; no scaling required.

The value of this property cannot be changed using **CAN Set Property**.

Channel Scaling Factor

Returns the scaling factor used to convert raw bits of the message to/from scaled floating-point units. The scaling factor is the A in the linear scaling formula $AX + B$, where X is the raw data, and B is the scaling offset.

CAN messages use the raw data, and the **CAN Read** and **CAN Write** VIs provide access to samples in floating-point units.

The value of this property cannot be changed using **CAN Set Property**.

Channel Scaling Offset

Returns the scaling offset used to convert raw bits of the message to/from scaled floating-point units. The scaling offset is the B in the linear scaling formula $AX + B$, where X is the raw data, and A is the scaling factor.

CAN messages use the raw data, and the **CAN Read** and **CAN Write** VIs provide access to samples in floating-point units.

The value of this property cannot be changed using **CAN Set Property**.

Channel Minimum Value

Returns the minimum value of the channel in scaled floating-point units.

The **CAN Read** and **CAN Write** VIs do not coerce samples when converting to/from CAN messages. You can use this value with property nodes to set the range of front-panel controls and indicators.

The value of this property cannot be changed using **CAN Set Property**.

Channel Maximum Value

Returns the maximum value of the channel in scaled floating-point units.

The **CAN Read** and **CAN Write** VIs do not coerce samples when converting to/from CAN messages. You can use this value with property nodes to set the range of front-panel controls and indicators.

The value of this property cannot be changed using **CAN Set Property**.

Channel Default Value

Returns the default value of the channel in scaled floating-point units.

For information on how **Channel Default Value** is used, refer to **CAN Read.vi** and **CAN Write.vi**.

The value of this property is originally set within MAX or **Can Create Message.vi**. If the channel is initialized directly from a CAN database, the value is 0.0 by default, but it can be changed using **CAN Set Property.vi**.

Channel Unit String

Returns the unit string of the channel. The string is no more than 80 characters in length.

You can use this value to display units (such as volts or RPM) along with the channel's samples.

The value of this property cannot be changed using **CAN Set Property**.

Hardware Serial Number

Returns the hardware serial number for the NI-CAN hardware that contains Interface.

Hardware Form Factor

Returns the hardware form factor for the NI-CAN hardware that contains Interface.

The value of **Hardware Form Factor** is an enumeration:

0	**PCI**
1	**PXI**
2	**PCMCIA**
3	**AT**

Hardware Transceiver

Returns the hardware form factor for the NI-CAN hardware that contains Interface.

The value of **Hardware Transceiver** is an enumeration:

0 **HS**

1 **LS**

This property is not supported on the PCMCIA form factor.

Version Major

Returns the major version of the NI-CAN software, such as the *2* in version *2.1.5*.

Version Minor

Returns the minor version of the NI-CAN software, such as the *1* in version *.2.1.5*.

Version Update

Returns the update version of the NI-CAN software, such as the *5* in version *.2.1.5*.

Version Phase

Returns the phase of the NI-CAN software.

The value of **Version Phase** is an enumeration:

0 **Development**

1 **Alpha**

1 **Beta**

1 **Release**

Versions of NI-CAN in hardware kits or on `ni.com` will always be **Release**.

Version Build

Returns the build number of the NI-CAN software. This number applies to **Development**, **Alpha**, and **Beta** phase only, and should be ignored for **Release** phase.

Version Comment

Returns a comment string for the NI-CAN software. If you received a custom release of NI-CAN from National Instruments, this comment often describes special features of the release.

Interface Baud Rate

Returns the baud rate in use by the Interface.

Basic baud rates such as 125000 and 500000 are specified as the numeric rate.

Advanced baud rates are specified as 8000*XXYY* hex, where *YY* is the value of Bit Timing Register 0 (BTR0), and *XX* is the value of Bit Timing Register 1 (BTR1) of the CAN controller chip. For more information, refer to the Interface Properties dialog in MAX.

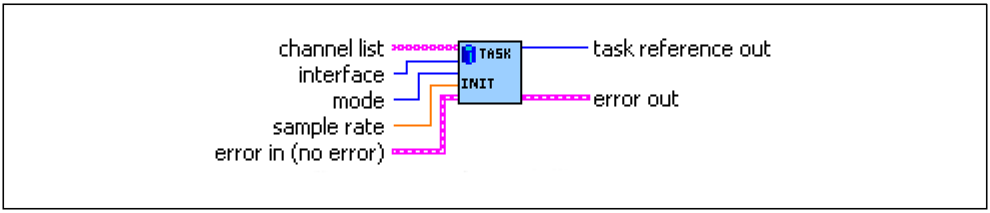
The value of this property is originally set within MAX, but it can be changed using **CAN Set Property.vi**.

CAN Initialize.vi

Purpose

Initialize a task for the specified channel list.

Format



Inputs

channel list is the array of channel names to initialize as a task. Each channel name is provided in an array entry.

For more information, refer to the channel list input of **CAN Init Start.vi**.

interface specifies the CAN interface to use for this task.

The interface input uses a ring typedef in which value 0 selects **CAN0**, value 1 selects **CAN1**, and so on.

The default (unwired) value is 65535, which means to use the default interface as defined in the MAX configuration. If the default interface in MAX is **All**, or if one or more channels in **channel list** specifies a **filepath**, the **interface** is a required input to this VI.

The Channel API and Frame API cannot use the same CAN network interface simultaneously. If the CAN network interface is already initialized in the Frame API, this function returns an error.

mode specifies the I/O mode for the task:

Input

Input channel data from received CAN messages. Use **CAN Read.vi** to obtain input samples as single-point, array, or waveform.

Use this input mode to read waveforms of timed samples, such as for comparison with NI-DAQ waveforms. You can also use this input mode to read a single point from the most recent message, such as for control or simulation.

For this mode, the channels in **channel list** can be contained in multiple messages.

Output

Output channel data to CAN messages for transmit. Use **CAN Write.vi** to write output samples as single point, array, or waveform.

For this mode, there are restrictions on using channels in **channel list** that are contained in multiple messages. Refer to **CAN Write.vi** for more information.

Timestamped Input

Input channel data from received CAN messages. Use **CAN Read.vi** to obtain input samples as an array of sample/timestamp pairs (Poly VI types ending in *Timestamped Dbl*).

Use this input mode to read samples with timestamps that indicate when each message is received from the network.

For this mode, the channels in **channel list** must be contained in a single message.

sample rate specifies the timing to use for samples of the task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately.

For **mode** of **Input**, **sample rate** of zero means that **CAN Read** returns a single point from the most recent message received, and greater than zero means that **CAN Read** returns samples timed at the specified rate.

For **mode** of **Output**, **sample rate** of zero means that CAN messages transmit immediately when **CAN Write** is called, and greater than zero means that CAN messages are transmitted periodically at the specified rate.

For **mode** of **Timestamped Input**, **sample rate** is ignored.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

Use **task reference out** with all subsequent VIs to reference the task. Wire this task reference to **CAN Start.vi** before you read or write samples for the message.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

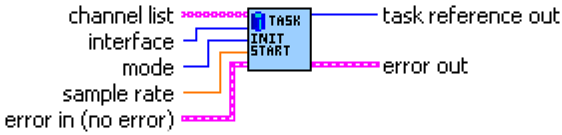
The **CAN Initialize** VI does not start communication. This enables you to use **CAN Set Property.vi** to change the task's properties, or **CAN Connect Terminals.vi** to synchronize CAN or DAQ cards. After you change properties or connections, use **CAN Start.vi** to start communication for the task.

CAN Init Start.vi

Purpose

Initialize a task for the specified channel list, then start communication.

Format



Inputs

channel list is the array of channel names to initialize and start as a task. Each channel name is provided in an array entry.

You can type in the channel list entries as string constants, or you can obtain the list from MAX or another CAN database by using **CAN Get Names.vi**.

You can initialize the same **channel list** with different **interface**, **mode**, or **sample rate**, because each task reference is unique.

The following paragraphs describe the syntax of each channel name. Brackets indicate optional fields.

[*filepath*::][*message*.]*channel*

- *filepath* is the path to a CAN database file from which to import the channel (signal) configurations. The filepath must use Windows directory syntax, and must be followed by a double-colon.

  If *filepath* is not included, the channel configuration is obtained from MAX. The MAX CAN channels are in the MAX CAN Channels listing within **Data Neighborhood**.

  Once you specify a *filepath*, it will continue to be applied to subsequent names in the channel list array until you specify a new *filepath*. After using *filepath* for a CAN database file, you can revert to using MAX by specifying an empty *filepath* (double colon only).

- *message* refers to the message in which the *channel* is contained. The message name must be followed by a decimal point.

  If the *channel* name occurs in multiple messages, you must specify the *message* name to identify the channel uniquely. Within MAX, channels with the same name in multiple messages are shown with a yellow exclamation point.

  If the *channel* name is unique across all channels, the *message* name is not required.
- *channel* refers to the channel (signal) name in MAX or the CAN database (indicated by *filepath*).

The following examples demonstrate the channel list syntax:

1. List of channels from MAX, each channel name unique across all messages.
   - myChan1
   - myChan2
   - myChan3
2. List of channels from a CANdb file, each channel name unique across all messages.
   - C:\MyCandbFiles\MyChannels.DBC::myChan1
   - myChan2
   - myChan3
3. List of channels from MAX, with one channel duplicated across two messages. MyChan2 and MyChan3 must be unique across all messages.
   - myMessage1.myChan1
   - myChan2
   - myMessage2.myChan1
   - myChan3
4. List of two channels from a CANdb file, then two channels from MAX.
   - C:\MyCandbFiles\MoreChannels.DBC::myChan1
   - myChan2

- ::myChan3
- myChan4

interface specifies the CAN interface to use for this task.

The interface input uses a ring typedef in which value 0 selects **CAN0**, value 1 selects **CAN1**, and so on.

The default (unwired) value is 65535, which means to use the default interface as defined in the MAX configuration. If the default interface in MAX is **All**, or if one or more channels in **channel list** specifies a *filepath*, the **interface** is a required input to this VI.

The Channel API and Frame API cannot use the same CAN network interface simultaneously. If the CAN network interface is already initialized in the Frame API, this function returns an error.

mode specifies the I/O mode for the task, as follows:

Input

Input channel data from received CAN messages. Use **CAN Read.vi** to obtain input samples as single-point, array, or waveform.

Use this input mode to read waveforms of timed samples, such as for comparison with NI-DAQ waveforms. You can also use this input mode to read a single point from the most recent message, such as for control or simulation.

For this mode, the channels in **channel list** can be contained in multiple messages.

Output

Output channel data to CAN messages for transmit. Use **CAN Write.vi** to write output samples as single-point, array, or waveform.

For this mode, there are restrictions on using channels in **channel list** that are contained in multiple messages. Refer to **CAN Write.vi** for more information.

Timestamped Input

Input channel data from received CAN messages. Use **CAN Read.vi** to obtain input samples as an array of sample/timestamp pairs (Poly VI types ending in *Timestamped Dbl*).

Use this input mode to read samples with timestamps that indicate when each message is received from the network.

For this mode, the channels in **channel list** must be contained in a single message.

sample rate specifies the timing to use for samples of the task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately.

For **mode** of **Input**, a **sample rate** of zero means that **CAN Read** returns a single point from the most recent message received, and greater than zero means that **CAN Read** returns samples timed at the specified rate.

For **mode** of **Output**, a **sample rate** of zero means that CAN messages transmit immediately when **CAN Write** is called, and greater than zero means that CAN messages are transmitted periodically at the specified rate.

For **mode** of **Timestamped Input**, **sample rate** is ignored.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

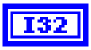
source identifies the VI where the error occurred.

Outputs

Use **task reference out** with all subsequent VIs to reference the running task. Because **CAN Init Start** starts communication, you can wire this task reference to **CAN Read.vi** or **CAN Write.vi**.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.



code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

The diagram for this VI simply calls **CAN Initialize.vi** followed by **CAN Start.vi**. This provides an easy way to start a list of channels.

The following list describes the scenarios for which **CAN Init Start.vi** cannot be used:

- If you need to set properties for the channels, use **CAN Initialize**, then **CAN Set Property.vi**, then **CAN Start.vi**. The **CAN Init Start** VI starts communication, and most channel properties cannot be changed after the task is started.
- If you need to synchronize tasks for multiple NI-CAN or NI-DAQ cards, refer to the VIs in the **CAN/DAQ Synchronization** palette, such as **CAN Sync Start with NI-DAQ.vi**.
- If you need to create channel configurations entirely within LabVIEW, without using MAX or a CAN database file, use **CAN Create Message.vi**, then **CAN Start.vi**. The **CAN Init Start** VI accepts only channel names defined in MAX or a CAN database file.

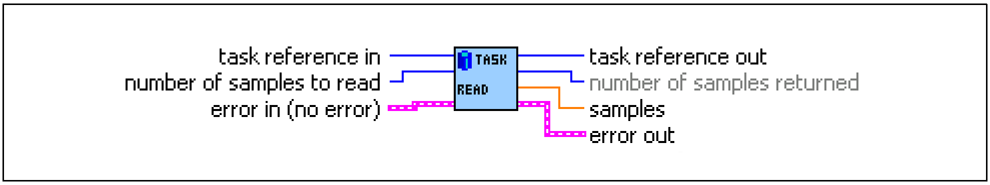
CAN Read.vi

Purpose

Read samples from a CAN task initialized as input. Samples are obtained from received CAN messages. The poly VI selection determines the data type to read.

To select the data type, right-click the VI, go to **Select Type**, and select the type by name. For an overview of CAN Read, refer to the *Read* and *Read Timestamped* sections of Chapter 4, *Using the Channel API*.

Format

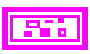


Inputs

task reference in is the task reference from the previous NI-CAN VI. The task reference is originally returned from **CAN Init Start.vi**, **CAN Initialize.vi**, or **CAN Create Message.vi**, and then wired through subsequent VIs.

The **mode** initialized for the task must be either **Input** or **Timestamped Input**.

number of samples to read specifies the number of samples to read for the task. For single-sample Poly VI types, **CAN Read** always returns one sample, so this input is ignored.



Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is

returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

task reference out is the same as **task reference in**. Wire the task reference to subsequent VIs for this task.

number of samples returned indicates the number of samples returned in the **samples** output.

The poly output **samples** returns the samples read from received CAN messages. The type of the poly output is determined by the Poly VI selection. For information on the different poly VI types provided by **CAN Read**, refer to the Poly VI Types section.

To select the data type, right-click the VI, go to **Select Type**, and select the type by name.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Poly VI Types

The name of each Poly VI type uses the following conventions:

- The first term is either **Single-Chan** or **Multi-Chan**. This indicates whether the type returns data for a single channel or multiple channels. **Multi-Chan** types return an array of analogous **Single-Chan** types, one entry for each channel initialized in **channel list** of **CAN Init Start**. **Single-Chan** types are convenient because no array indexing is required, but you are limited to reading only one CAN channel.
- The second term is either **Single-Samp** or **Multi-Samp**. This indicates whether the type returns a single sample, or an array of multiple samples. **Single-Samp** types are often used for single point control applications, such as within LabVIEW RT.

- The third term indicates the data type used for each sample. The word *Dbl* indicates double-precision (64-bit) floating point. The word *Wfm* indicates the waveform data type. The words *1D* and *2D* indicate one and two-dimensional arrays, respectively. The words **Time & Dbl** indicate a cluster of a LabVIEW timestamp and a double-precision sample.

Single-Chan Single-Samp Dbl

Returns a single sample for the first channel initialized in channel list.

If the initialized sample rate is greater than zero, this poly VI type waits for the next sample time, then returns a single sample. This enables you to execute a control loop at a specific rate.

If the initialized **sample rate** is zero, this poly VI immediately returns a single sample.

The **samples** output returns a single sample from the most recent message received. If no message has been received since you started the task, the Default Value of the channel is returned in **samples**.

You can use **error out** to determine whether a new message has been received since the previous call to **CAN Read** (or **CAN Start**). If no message has been received, the warning **code** 3FF60009 hex is returned in **error out**. If a new message has been received, the success **code** 0 is returned in **error out**.

To use this type, you must set the initialized mode to **Input** (not **Timestamped Input**).

Unless an error occurs, **number of samples returned** is one.

The Timeout property is not used with this poly VI type.

Multi-Chan Single-Samp 1D Dbl

Returns an array, one entry for each channel initialized in channel list. Each entry consists of a single sample.

The order of channel entries in **samples** is the same as the order in the original **channel list**.

If the initialized sample rate is greater than zero, this poly VI type waits for the next sample time, then returns a single sample for each channel. This enables you to execute a control loop at a specific rate.

If the initialized **sample rate** is zero, this poly VI immediately returns a single sample for each channel.

The **samples** output returns a single sample for each channel from the most recent message received. If no message has been received for a channel since you started the task, the Default Value of the channel is returned in **samples**.

You can specify channels in **channel list** that span multiple messages. A sample from the most recent message is returned for all channels.

You can use **error out** to determine whether a new message has been received since the previous call to **CAN Read** (or **CAN Start**). If no message has been received for one or more channels, the warning **code** 3FF60009 hex is returned in **error out**. If a new message has been received for all channels, the success **code** 0 is returned in **error out**.

To use this type, you must set the initialized mode to **Input** (not **Timestamped Input**).

Unless an error occurs, **number of samples returned** is one. The **samples** array is indexed by channel, and each channel's entry contains a single sample.

The Timeout property is not used with this poly VI type.

If you need to determine the number of channels in the task after initialization, get the Number of Channels property for the task reference.

Single-Chan Multi-Samp 1D Dbl

Returns an array of samples for the first channel initialized in channel list.

The initialized sample rate must be greater than zero for this poly VI, because each sample in the array indicates the value of the CAN channel at a specific point in time. In other words, the **sample rate** specifies a virtual clock that copies the most recent value from CAN messages for each sample time. The changes in sample values from message to message enable you to view the CAN channel over time, such as for comparison with other CAN or DAQ input channels.

If the initialized **sample rate** is zero, this poly VI returns an error. If your intent is simply to read the most recent sample for a task, use the **Single-Chan Single-Samp Dbl** type.

If no message has been received since you started the task, the Default Value of the channel is returned in all entries of the **samples** array.

You can use **error out** to determine whether a new message has been received since the previous call to **CAN Read** (or **CAN Start**). If no message has been received, the warning **code** 3FF60009 hex is returned in **error out**. If a new message has been received, the success **code** 0 is returned in **error out**.

To use this type, you must set the initialized mode to **Input** (not **Timestamped Input**).

Unless an error occurs, the **number of samples returned** is equal to **number of samples to read**.

The Timeout property is not used with this poly VI type.

Multi-Chan Multi-Samp 2D Dbl

Returns an array, one entry for each channel initialized in channel list. Each entry consists of an array of samples.

The order of channel entries in **samples** is the same as the order in the original **channel list**.

The initialized sample rate must be greater than zero for this poly VI, because each sample in the array indicates the value of each CAN channel at a specific point in time.

In other words, the **sample rate** specifies a virtual clock that copies the most recent value from CAN messages for each sample time. The changes in sample values from message to message enable you to view the CAN channels over time, such as for comparison with other CAN or DAQ input channels.

If the initialized **sample rate** is zero, this poly VI returns an error. If your intent is simply to read the most recent samples for a task, use the **Multi-Chan Single-Samp 1D Dbl** type.

If no message has been received for a channel since you started the task, the Default Value of the channel is returned in **samples**.

You can specify channels in **channel list** that span multiple messages. At each point in time, a sample from the most recent message is returned for all channels.

You can use **error out** to determine whether a new message has been received since the previous call to **CAN Read** (or **CAN Start**). If no message has been received for one or more channels, the warning **code** 3FF60009 hex is returned in **error out**. If a new message has been received for all channels, the success **code** 0 is returned in **error out**.

To use this type, you must set the initialized mode to **Input** (not **Timestamped Input**).

Unless an error occurs, the **number of samples returned** is equal to **number of samples to read**.

The Timeout property is not used with this poly VI type.

If you need to determine the number of channels in the task after initialization, get the Number of Channels property for the task reference.

Single-Chan Multi-Samp Wfm

Returns a single waveform for the first channel initialized in *channel list*.

The initialized sample rate must be greater than zero for this poly VI, because each sample in the array indicates the value of the CAN channel at a specific point in time. In other words, the **sample rate** specifies a virtual clock that copies the most recent value from CAN messages for each sample time. The changes in sample values from message to message enable you to view the CAN channel over time, such as for comparison with other CAN or DAQ input channels.

The waveform's start time indicates the time of the first CAN sample in the array. The waveform's delta time indicates the time between each sample in the array, as determined by the original **sample rate**.

If the initialized **sample rate** is zero, this poly VI returns an error. If your intent is to simply read the most recent sample for a task, use the **Single-Chan Single-Samp Dbl** type.

If no message has been received since you started the task, the Default Value of the channel is returned in all entries of the **samples** waveform.

You can use **error out** to determine whether a new message has been received since the previous call to **CAN Read** (or **CAN Start**). If no message has been received, the warning **code** 3FF60009 hex is returned in **error out**. If a new message has been received, the success **code** 0 is returned in **error out**.

To use this type, you must set the initialized **mode** to **Input** (not **Timestamped Input**).

Unless an error occurs, the **number of samples returned** is equal to **number of samples to read**.

The Timeout property is not used with this poly VI type.

Multi-Chan Multi-Samp 1D Wfm

Returns an array, one entry for each channel initialized in channel list. Each entry consists of a single waveform.

The order of channel entries in **samples** is the same as the order in the original **channel list**.

The initialized sample rate must be greater than zero for this poly VI, because each sample in the waveform's array indicates the value of the CAN channel at a specific point in time. In other words, the **sample rate** specifies a virtual clock that copies the most recent value from CAN messages for each sample time. The changes in sample values from message to message enable you to view the CAN channel over time, such as for comparison with other CAN or DAQ input channels.

Each waveform's start time indicates the time of the first CAN sample in the array. The waveform's delta time indicates the time between each sample in the array, as determined by the original **sample rate**.

If the initialized **sample rate** is zero, this poly VI returns an error. If your intent is simply to read the most recent samples for a task, use the **Multi-Chan Single-Samp 1D Dbl** type.

If no message has been received for a channel since you started the task, the Default Value of the channel is returned in **samples**.

You can specify channels in **channel list** that span multiple messages. At each point in time, a sample from the most recent message is returned for all channels.

You can use **error out** to determine whether a new message has been received since the previous call to **CAN Read** (or **CAN Start**). If no message has been received for one or more channels, the warning **code** 3FF60009 hex is returned in **error out**. If a new message has been received for all channels, the success **code** 0 is returned in **error out**.

To use this type, you must set the initialized mode to **Input** (not **Timestamped Input**).

Unless an error occurs, the **number of samples returned** is equal to **number of samples to read**.

The Timeout property is not used with this poly VI type.

If you need to determine the number of channels in the task after initialization, get the Number of Channels property for the task reference.

Single-Chan Multi-Samp 1D Time & Dbl

Returns an array of clusters. Each cluster corresponds to a received message for the first channel initialized in channel list. Each cluster contains the sample value, and a timestamp that indicates when the message was received.

To use this type, you must set the initialized mode to **Timestamped Input** (not **Input**).

The Timeout property determines whether this VI will wait for the **number of samples to read** messages to arrive from the network. The default value of **Timeout** is zero, but you can change it using **CAN Set Property.vi**.

If **Timeout** is greater than zero, the VI will wait for **number of samples to read** messages to arrive. If **number of samples to read** messages are not received before the **Timeout** expires, an error is returned. **Timeout** is specified as seconds.

If **Timeout** is zero, the VI will not wait for messages, but instead returns samples from the messages received since the previous call to **CAN Read**. The number of samples returned is indicated in the **number of samples returned** output, up to a maximum of **number of samples to read** messages. If no new message has been received, **number of samples returned** is 0, and **error out** indicates success.

Because the timing of values in **samples** is determined by when the message is received, the sample rate input is not used with this poly VI type.

Multi-Chan Multi-Samp 2D Time & Dbl

Returns an array, one entry for each channel initialized in channel list. Each entry consists of an array of clusters. Each cluster corresponds to a received message for the channels initialized in channel list. Each cluster contains the sample value, and a timestamp that indicates when the message was received.

The order of channel entries in **samples** is the same as the order in the original **channel list**.

To use this type, you must set the initialized mode to **Timestamped Input** (not **Input**).

You cannot specify channels in **channel list** that span multiple messages.

The Timeout property determines whether this VI waits for the **number of samples to read** messages to arrive from the network. The default value of **Timeout** is zero, but you can change it using **CAN Set Property.vi**.

If **Timeout** is greater than zero, the VI will wait for **number of samples to read** messages to arrive. If **number of samples to read** messages are not received before the **Timeout** expires, an error is returned. **Timeout** is specified as milliseconds.

If **Timeout** is zero, the VI will not wait for messages, but instead returns samples from the messages received since the previous call to **CAN Read**. The number of samples returned is indicated in the **number of samples returned** output, up to a maximum of

number of samples to read messages. If no new message has been received, **number of samples returned** is 0, and **error out** indicates success.

Because the timing of values in **samples** is determined by when the message is received, the sample rate input is not used with this poly VI type.

If you need to determine the number of channels in the task after initialization, get the Number of Channels property for the task reference.

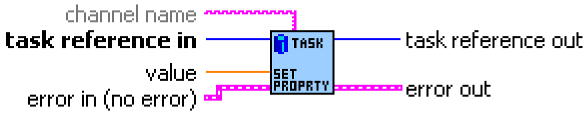
CAN Set Property.vi

Purpose

Set a property for the task, or a single channel within the task. The poly VI selection determines the property to set.

To select the property, right-click the VI, go to **Select Type** and select the property by name.

Format



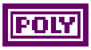
Inputs

task reference in is the task reference from the previous NI-CAN VI. The task reference is originally returned from VIs such as **CAN Initialize.vi** or **CAN Create Message.vi**, and then wired through subsequent VIs.

channel name specifies an individual channel within the task. The default (unwired) value of channel name is empty, which means that the property applies to the entire task, not a specific channel.

Properties that begin with the word *Channel* or *Message* do not apply to the entire task, but an individual channel or message within the task. For these channel-specific properties, you must wire the name of a channel from channel list into the **channel name** input.

For properties that do not begin with the word *Channel* or *Message*, you must leave **channel name** unwired (empty).



The poly input **value** specifies the property value. You select the property to set as **value** by selecting the Poly VI type. The data type of **value** is also determined by the Poly VI selection. For information on the different properties provided by **CAN Get Property**, refer to the Poly VI Types section.

To select the property, right-click the VI, go to **Select Type** and select the property by name.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

task reference out is the same as **task reference in**. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

You cannot set a property while the task is running. If you need to change a property prior to starting the task, you cannot use **CAN Init Start.vi**. First, call **CAN Initialize.vi**, followed by **CAN Set Property** and then **CAN Start.vi**. After you start the task, you can also change a property by calling **CAN Stop.vi**, followed by **CAN Set Property**, and then **CAN Start** again.

Poly VI Types

DBL

Timeout

Sets a time in milliseconds to wait for samples. The default value is zero.

Use of the **Timeout** property depends on the initialized mode of the task:

- The timeout value does *not* apply to an **Output** task.
- The timeout value does *not* apply to an **Input** task. For **Input** tasks initialized with **sample rate** greater than zero, the **number of samples to read** input to **CAN Read.vi** implicitly specifies the time to wait. For **Input** tasks initialized with **sample rate** equal to zero, the **CAN Read** VI always returns available samples immediately, without waiting.
- The timeout value *does* apply to a **Timestamped Input** task. A timeout of zero means to return available samples immediately. A timeout greater than zero means that **CAN Read.vi** will wait a maximum of **Timeout** milliseconds for **number of samples to read** samples to become available before returning.

U16

Behavior After Final Output

The **Behavior After Final Output** property applies only to tasks initialized with mode of **Output**, and sample rate greater than zero. The value specifies the behavior to perform after the final periodic sample is transmitted.

Behavior After Final Output uses the following values:

Repeat Final Sample

> Transmit messages for the final sample(s) repeatedly. The final messages are transmitted periodically as specified by **sample rate**.
>
> If there is significant delay between subsequent calls to **CAN Write.vi**, this value means that periodic messages continue between **CAN Write** calls, and messages with the final sample's data will be repeated on the network.
>
> **Repeat Final Sample** is the default value of the **Behavior After Final Output** property**.**

Cease Transmit

> Cease transmit of messages until the next call to **CAN Write**.

If there is significant delay between subsequent calls to **CAN Write**, this value means that periodic messages cease between **CAN Write** calls, and the final sample's data will not be repeated on the network.

Channel Default Value

Sets the default value of the channel in scaled floating-point units.

For information on how the **Channel Default Value** is used, refer to **CAN Read.vi** and **CAN Write.vi**.

The value of this property is originally set within MAX. If the channel is initialized directly from a CAN database, the value is 0.0 by default, but it can be changed using **CAN Set Property**.

Interface Baud Rate

Sets the baud rate in use by the Interface.

This property applies to all tasks initialized with the **Interface**.

You can specify the following basic baud rates as the numeric rate: 83333, 100000, 125000, 200000, 250000, 400000, 500000, 800000, and 1000000.

You can specify advanced baud rates as 8000*XXYY* hex, where *YY* is the value of Bit Timing Register 0 (BTR0), and *XX* is the value of Bit Timing Register 1 (BTR1) of the CAN controller. For more information, refer to the Interface Properties dialog in MAX.

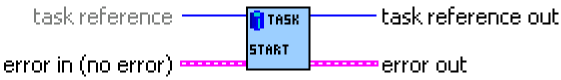
The value of this property is originally set within MAX, but it can be changed using **CAN Set Property**.

CAN Start.vi

Purpose

Start communication for the specified task.

Format



Inputs

task reference in is the task reference from the previous NI-CAN VI. The task reference is originally returned from VIs such as **CAN Initialize.vi** or **CAN Create Message.vi**, and then wired through subsequent VIs.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

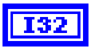
source identifies the VI where the error occurred.

Outputs

task reference out is the same as **task reference in**. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.



code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

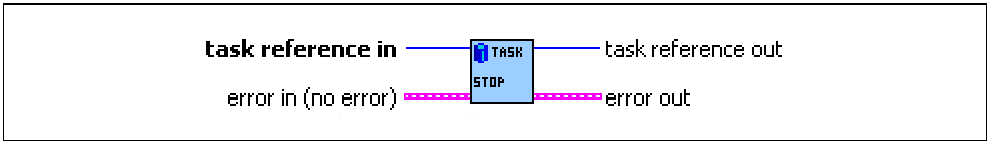
You must start communication for a task to use **CAN Read.vi** or **CAN Write.vi**. After you start communication, you can no longer change the task's configuration with **CAN Set Property.vi** or **CAN Connect Terminals.vi**.

CAN Stop.vi

Purpose

Stop communication for the specified task.

Format



Inputs

task reference in is the task reference from the previous NI-CAN VI. The task reference is originally returned from **CAN Init Start.vi**, **CAN Initialize.vi**, or **CAN Create Message.vi**, and then wired through subsequent VIs.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

task reference out is the same as **task reference in**. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

This VI stops communication so that you can change the task's configuration, such as by using **CAN Set Property.vi** or **CAN Connect Terminals.vi**. After you change the configuration, use **CAN Start.vi** to start again.

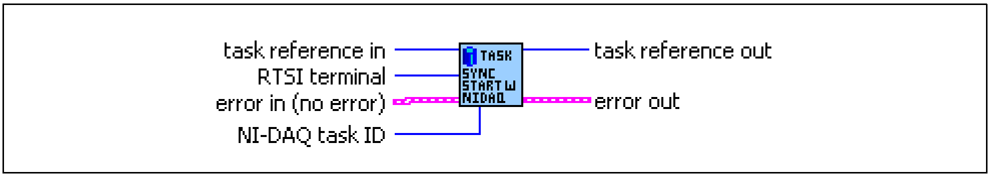
This VI does not clear the configuration for the task; therefore, do *not* use it as the last NI-CAN VI in your application. **CAN Clear.vi** must always be the last NI-CAN VI for each task.

CAN Sync Start with NI-DAQ.vi

Purpose

Synchronize and start the specified CAN task and NI-DAQ task.

Format



Inputs

task reference in is the task reference from the previous NI-CAN VI. The task reference is originally returned from VIs such as **CAN Initialize.vi** or **CAN Create Message.vi**.

NI-DAQ task ID is the task ID from an NI-DAQ configuration VI, such as **AI Config** or **AO Config**.

When this VI returns, do not call an NI-DAQ start VI for the task. The LabVIEW diagram of this VI starts the **NI-DAQ task ID** on your behalf, so you can immediately call NI-DAQ read or write VIs.

RTSI terminal specifies the RTSI terminal number to use for the shared start trigger. This input uses a ring typedef to select terminals from **RTSI0** to **RTSI6**.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

task reference out is the same as **task reference in**. Wire the task reference to subsequent NI-CAN VIs for this task.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

The CAN and NI-DAQ task execute on different NI hardware products. To use the input/output samples of these tasks together in your application, the NI hardware products must be synchronized with RTSI terminal connections. Both NI hardware products must use a common timebase to avoid clock drift, and a common start trigger to start input/output at the same time.

This VI uses NI-CAN and NI-DAQ RTSI functions to synchronize the NI hardware products to a common timebase and start trigger, and then it starts sampling on both tasks. The function used to connect RTSI terminals on the CAN card is **CAN Connect Terminals.vi**.

When you use this VI to start the tasks, you must use **CAN Clear with NI-DAQ.vi** to clear the tasks.

This VI synchronizes a single CAN hardware product to a single NI-DAQ hardware product. To synchronize multiple CAN cards and/or multiple NI-DAQ cards, refer to **CAN Sync Start Multiple with NI-DAQ.vi**.

This VI is intended to serve as an example. You can use the VI as is, but the LabVIEW diagram is commented so that you can use the VI as a starting point for more complex synchronization scenarios. Before you customize the LabVIEW diagram, save a copy of the VI for your own editing.

The diagram of this VI assumes that the NI-DAQ product is an E-series MIO device. If you are using a different NI hardware product, refer to the diagram as a starting point.

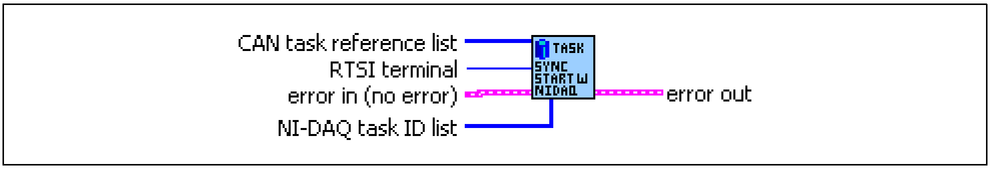
The diagram of this VI issues the start trigger immediately. To implement more complex triggering, such as using an AI trigger to start, refer to the diagram as a starting point.

CAN Sync Start Multiple with NI-DAQ.vi

Purpose

Synchronize and start the specified list of multiple CAN tasks and NI-DAQ tasks. This is a more complex implementation of **CAN Sync Start with NI-DAQ.vi** that supports multiple CAN and NI-DAQ hardware products.

Format



Inputs

CAN task reference list is an array of NI-CAN task references. Each task reference is originally returned from VIs such as **CAN Initialize.vi** or **CAN Create Message.vi**. You can build the task references into an array using the LabVIEW **Build Array** VI.

NI-DAQ task ID list is an array of NI-DAQ task IDs. Each task ID is originally returned from an NI-DAQ configuration VI, such as **AI Config** or **AO Config**.

This VI assumes that each task in **NI-DAQ task ID list** is on a different NI-DAQ card.

When this VI returns, do not call an NI-DAQ start VI for each task. The LabVIEW diagram of this VI starts each task in **NI-DAQ task ID list** on your behalf, so you can immediately call NI-DAQ read or write VIs.

RTSI terminal specifies the RTSI terminal number to use for the shared start trigger. This input uses a ring typedef to select terminals from **RTSI0** to **RTSI6**.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

The CAN and NI-DAQ tasks execute on different NI hardware products. To use the input/output samples of these tasks together in your application, the NI hardware products must be synchronized with RTSI terminal connections. Both NI hardware products must use a common timebase to avoid clock drift, and a common start trigger to start input/output at the same time.

This VI uses NI-CAN and NI-DAQ RTSI functions to synchronize the NI hardware products to a common timebase and start trigger, and then it starts sampling on all tasks. The function used to connect RTSI terminals on the CAN card is **CAN Connect Terminals.vi**.

When you use this VI to start the tasks, you must use **CAN Clear Multiple with NI-DAQ.vi** to clear the tasks.

This VI is intended to serve as an example. You can use the VI as is, but the LabVIEW diagram is commented so that you can use the VI as a starting point for more complex synchronization scenarios. Before you customize the LabVIEW diagram, save a copy of the VI for your own editing.

The diagram of this VI assumes that all NI-DAQ products are E-Series MIO devices. If you are using a different NI hardware product, refer to the diagram as a starting point.

The diagram of this VI issues the start trigger immediately. To implement more complex triggering, such as using an AI trigger to start, refer to the diagram as a starting point.

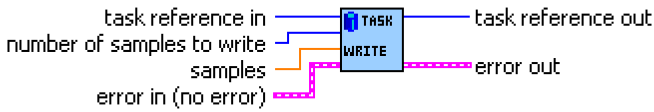
CAN Write.vi

Purpose

Write samples to a CAN task initialized as Output (refer to the **mode** parameter of **CAN Init Start.vi**). Samples are placed into transmitted CAN messages. The poly VI selection determines the data type to write.

To select the data type, right-click the VI, go to **Select Type**, and select the type by name. For an overview of CAN Write, refer to the *Write* section of Chapter 4, *Using the Channel API*.

Format



Inputs

task reference in is the task reference from the previous NI-CAN VI. The task reference is originally returned from **CAN Init Start.vi**, **CAN Initialize.vi**, or **CAN Create Message.vi**, and then wired through subsequent VIs.

The **mode** initialized for the task must be **Output**.

number of samples to write specifies the number of samples to write for the task. For single-sample Poly VI types, **CAN Write** always accepts one sample, so this input is ignored.

The poly input **samples** specifies the samples to transmit in CAN messages. The the poly input type is determined by the Poly VI selection. For information on the different poly VI types provided by **CAN Write**, refer to the Poly VI Types section.

To select the data type, right-click the VI, go to **Select Type**, and select the type by name.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Outputs

task reference out is the same as **task reference in**. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Poly VI Types

The name of each Poly VI type uses the following conventions:

- The first term is either **Single-Chan** or **Multi-Chan**. This indicates whether the type specifies data for a single channel or multiple channels. **Multi-Chan** types specify an array of analogous **Single-Chan** types, one entry for each channel initialized in **channel list** of **CAN Init Start**. **Single-Chan** types are convenient because no array indexing is required, but you are limited to writing only one CAN channel.
- The second term is either **Single-Samp** or **Multi-Samp**. This indicates whether the type specifies a single sample, or an array of multiple samples. **Single-Samp** types are often used for single-point control applications, such as within LabVIEW RT.
- The third term indicates the data type used for each sample. The word *Dbl* indicates double-precision (64-bit) floating point. The word *Wfm* indicates the waveform data type. The words *1D* and *2D* indicate one and two-dimensional arrays, respectively.

Single-Chan Single-Samp Dbl

Writes a single sample for the first channel initialized in channel list.

If the initialized sample rate is greater than zero, the task transmits a CAN message periodically at the specified rate. The first **CAN Write** transmits a message immediately, and then begins a periodic timer at the specified rate. Each subsequent message transmission is based on the timer, and uses the most recent sample provided by **CAN Write**.

If the initialized **sample rate** is zero, the message is transmitted immediately each time you call **CAN Write**.

Because all channels of a message are transmitted on the network as a unit, **CAN Write** enforces the following rules:

- You *cannot* write the same message in more than one **Output** task.
- You *can* write more than one message in a single **Output** task.
- You *can* write a subset of channels for a message in a single **Output** task. For channels that are not included in the task, the Default Value is transmitted in the CAN message. Because this Poly VI writes only one channel, the **Default Value** will always be used for any remaining channels in the associated message.

For many applications, the most straightforward technique is to assign a single **Output** task for each message you want to transmit. In each task, include all channels of that message in the **channel list**. This ensures that you can provide new samples for the entire message with each **CAN Write**.

Multi-Chan Single-Samp 1D Dbl

Writes an array, one entry for each channel initialized in channel list. Each entry consists of a single sample.

The messages transmitted by **CAN Write** are determined by the associated **channel list**. If all channels are contained in a single message, only that message is transmitted. If a few channels are contained in one message, and the remaining channels are contained in a second message, two messages are transmitted.

If the initialized sample rate is greater than zero, the task transmits associated CAN messages periodically at the specified rate. The first **CAN Write** transmits messages immediately, and then begins a periodic timer at the specified rate. Each subsequent transmission of messages is based on the timer and uses the most recent samples provided by **CAN Write**.

If the initialized **sample rate** is zero, the messages are transmitted immediately each time you call **CAN Write**.

Because all channels of a message are transmitted on the network as a unit, **CAN Write** enforces the following rules:

- You *cannot* write the same message in more than one **Output** task.
- You *can* write more than one message in a single **Output** task.
- You *can* write a subset of channels for a message in a single **Output** task. For channels that are not included in the task, the Default Value is transmitted in the CAN message.

For many applications, the most straightforward technique is to assign a single **Output** task for each message that you want to transmit. In each task, include all channels of that message in the **channel list**. This ensures that you can provide new samples for the entire message with each **CAN Write**.

Single-Chan Multi-Samp 1D Dbl

Writes an array of samples for the first channel initialized in channel list.

If the initialized sample rate is greater than zero, the task transmits a CAN message periodically at the specified rate. This Poly VI is used to transmit a sequence of messages periodically, with a unique sample value in each message. The first **CAN Write** transmits a message immediately using the first sample in the array, and then begins a periodic timer at the specified rate. Each subsequent message transmission is based on the timer, and uses the next sample in the array. After the final sample in the array has been transmitted, subsequent behavior is determined by the Behavior After Final Output property. The default **Behavior After Final Output** is to retransmit the final sample each period until **CAN Write** is called again.

If the initialized **sample rate** is zero, a message is transmitted immediately for each entry in the array, with as little delay as possible between messages. After the message for the final sample is transmitted, no further transmissions occur until **CAN Write** is called again, regardless of the **Behavior After Final Output** property.

NI-CAN uses a queue to store pending messages prior to transmission. **CAN Write** returns after the final message is written to this queue. This provides some time for you to call **CAN Write** again to provide a continual stream of samples. In LabVIEW RT, because the time between successive **CAN Write** calls is deterministic, you can ensure unique sample values in each message.

Because all channels of a message are transmitted on the network as a unit, **CAN Write** enforces the following rules:

- You *cannot* write the same message in more than one **Output** task.
- You *can* write more than one message in a single **Output** task.
- You *can* write a subset of channels for a message in a single **Output** task. For channels that are not included in the task, the Default Value is transmitted in the CAN message. Because this Poly VI writes only one channel, the **Default Value** will always be used for any remaining channels in the associated message.

For many applications, the most straightforward technique is to assign a single **Output** task for each message that you want to transmit. In each task, include all channels of that message in the **channel list**. This ensures that you can provide new samples for the entire message with each **CAN Write**.

Multi-Chan Multi-Samp 2D Dbl

Writes an array, one entry for each channel initialized in channel list. Each entry consists of an array of samples.

The messages transmitted by **CAN Write** are determined by the associated **channel list**. If all channels are contained in a single message, only that message is transmitted. If a few channels are contained in one message, and the remaining channels are contained in a second message, two messages are transmitted.

If the initialized sample rate is greater than zero, the task transmits associated CAN messages periodically at the specified rate. This Poly VI is used to transmit a sequence of messages periodically, with unique sample values in each set of messages. The first **CAN Write** transmits associated messages immediately using the first sample in each channel's array, and then begins a periodic timer at the specified rate. Each subsequent transmission of messages is based on the timer, and uses the next sample in each channel's array. After the final sample in each channel's array has been transmitted, subsequent behavior is determined by the Behavior After Final Output property. The default **Behavior After Final Output** is to retransmit the final sample each period until **CAN Write** is called again.

If the initialized **sample rate** is zero, the task transmits associated messages immediately for each entry in each channel's array, with as little delay as possible between messages. After the message for the final sample is transmitted, no further transmissions occur until **CAN Write** is called again, regardless of the **Behavior After Final Output** property.

NI-CAN uses a queue to store pending messages prior to transmission. **CAN Write** returns after the final message is written to this queue. This provides some time for you to call **CAN Write** again to provide a continual stream of samples. In LabVIEW RT, since the time between successive **CAN Write** calls is deterministic, you can ensure unique sample values in each message.

Because all channels of a message are transmitted on the network as a unit, **CAN Write** enforces the following rules:

- You *cannot* write the same message in more than one **Output** task.
- You *can* write more than one message in a single **Output** task.
- You *can* write a subset of channels for a message in a single **Output** task. For channels that are not included in the task, the Default Value is transmitted in the CAN message.

For many applications, the most straightforward technique is to assign a single **Output** task for each message that you want to transmit. In each task, include all channels of that message in the **channel list**. This ensures that you can provide new samples for the entire message with each **CAN Write**.

Single-Chan Multi-Samp Wfm

Writes a single waveform for the first channel initialized in channel list.

The start time and delta time of the waveform does not affect the beginning of message transmission. Therefore, this Poly VI type is equivalent to the Single-Chan Multi-Samp 1D Dbl Poly VI type.

Multi-Chan Multi-Samp 1D Wfm

Writes an array, one entry for each channel initialized in channel list. Each entry consists of a single waveform.

The start time and delta time of each waveform does not affect the beginning of message transmission. Therefore, this Poly VI type is equivalent to the Multi-Chan Multi-Samp 2D Dbl Poly VI type.

6 Channel API for C

This chapter lists the NI-CAN functions and describes the format, purpose and parameters.

Unless otherwise stated, each NI-CAN function suspends execution of the calling thread until it completes. The functions in this chapter are listed alphabetically.

Section Headings

The following are section headings found in the Channel API for C functions.

Purpose

Each function description includes a brief statement of the purpose of the function.

Format

The format section describes the format of each function for the C programming language.

Input and Output

The input and output parameters for each function are listed.

Description

The description section gives details about the purpose and effect of each function.

Data Types

The following data types are used with functions of the NI-CAN Channel API for C.

Table 6-1. NI-CAN Channel API for C, Data Types

Data Type	Purpose
`i8`	8-bit signed integer
`i16`	16-bit signed integer
`i32`	32-bit signed integer
`u8`	8-bit unsigned integer
`u16`	16-bit unsigned integer
`u32`	32-bit unsigned integer
`f32`	32-bit floating point number
`f64`	64-bit floating point number
`str`	ASCII string represented as an array of characters terminated by null character (`'\0'`). This type is used with output strings.
`cstr`	ASCII string represented as an array of characters terminated by null character (`'\0'`). This type is used with input strings.
`nctTypeTaskRef`	Reference to an initialized task. Refer to `nctInitStart` for more information.
`nctTypeStatus`	Status returned from NI-CAN functions. Refer to `ncStatusToString` in the Frame API for more information.
`nctTypeTimestamp`	Timestamp. Refer to `nctReadTimestamped` for more information.

List of Functions

The following table contains an alphabetical list of the NI-CAN Channel API for C functions.

Table 6-2. NI-CAN Channel API for C Functions

Function	Purpose
`nctClear`	Stop communication for the task and then clear the configuration.
`nctConnectTerminals`	Connect terminals in the CAN hardware.
`nctCreateMessage`	Create a message configuration and associated channel configurations within your application.
`nctDisconnectTerminals`	Disconnect terminals in the CAN hardware.
`nctGetNames`	Get an array of CAN channel names or message names from MAX or a CAN database file.
`nctGetProperty`	Get a property for the task, or a single channel within the task.
`nctInitialize`	Initialize a task for the specified channel list.
`nctInitStart`	Initialize a task for the specified channel list, then start communication.
`nctRead`	Read samples from a CAN task initialized with `Mode` of `nctModeInput`. Samples are obtained from received CAN messages.
`nctReadTimestamped`	Read samples from a CAN task initialized with `Mode` of `nctModeTimestampedInput`.
`nctSetProperty`	Set a property for the task, or a single channel within the task.
`nctStart`	Start communication for the specified task.
`nctStop`	Stop communication for the specified task.
`nctWrite`	Write samples to a CAN task initialized as `NctModeOutput`. Samples are placed into transmitted CAN messages.

nctClear

Purpose

Stop communication for the task and then clear the configuration.

Format

```
nctTypeStatus    nctClear(
                          nctTypeTaskRef    TaskRef);
```

Inputs

`TaskRef`	Task reference from the previous NI-CAN function. The task reference is originally returned from `nctInitStart`, `nctInitialize`, or `nctCreateMessage`.

Outputs

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

Description

The `nctClear` function must always be the final NI-CAN function called for each task. If you do not use the `nctClear` function, the remaining task configurations can cause problems in execution of subsequent NI-CAN applications.

If the cleared task is the last running task for the initialized `Interface` (refer to `nctInitStart`), the `nctClear` function also stops communication on the interface's CAN controller and disconnects all terminal connections for that interface.

Because this function clears the task, `TaskRef` cannot be used with subsequent functions. To change properties of a task and start again, use `nctStop`.

nctConnectTerminals

Purpose

Connect terminals in the CAN hardware.

Format

```
nctTypeStatus    nctConnectTerminals(
                     nctTypeTaskRef    TaskRef,
                     u32               SourceTerminal,
                     u32               DestinationTerminal,
                     u32               Modifiers);
```

Inputs

`TaskRef`

Task reference from the previous NI-CAN function. The task reference is originally returned from `nctInitStart`, `nctInitialize`, or `nctCreateMessage`.

`SourceTerminal`

Specifies the connection source.

Once the connection is successfully created, behavior flows from `SourceTerminal` to `DestinationTerminal`.

For a list of valid source/destination pairs, refer to the *Valid Combinations of Source/Destination* section.

The following list describes each value of `SourceTerminal`:

`nctSrcTermRTSI0` … `nctSrcTermRTSI6`

Selects a general-purpose RTSI line as source (input) of the connection.

`nctSrcTerm10HzResyncEvent`

`nctSrcTerm10HzResyncEvent` selects a 10 Hz, 50 percent duty cycle clock. This slow rate is required for resynchronization of CAN cards. On each pulse of the resync clock, the other CAN card brings its clock into sync.

By selecting RTSI0-6 as the `DestinationTerminal`, you route the 10 Hz clock to synchronize with other CAN cards. NI-DAQ cards cannot use the 10 Hz resync clock, so this selection is limited to synchronization of two or more CAN cards.

`nctSrcTerm10HzResyncEvent` applies to the entire CAN card, including both interfaces of a 2-port CAN card. The CAN card is specified by the task interface, such as the `Interface` input to `nctInitialize`.

`nctSrcTermStartTrigEvent`

`nctSrcTermStartTrigEvent` selects the start trigger, the event that begins sampling for tasks.

The start trigger is the same for all tasks using a given interface, such as the `Interface` input to `nctInitialize`.

In the default (disconnected) state of the `nctDestTermStartTrig` destination, the start trigger occurs when communication begins on the interface.

By selecting RTSI0-6 as the `DestinationTerminal`, you route the start trigger of this CAN card to the start trigger of other CAN or DAQ cards. This ensures that sampling begins at the same time on both cards. For example, you can synchronize two CAN cards by routing `nctSrcTermStartTrigEvent` as the `SourceTerminal` on one CAN card, and then routing `nctDestTermStartTrig` as the `DestinationTerminal` on the other CAN card, with both cards using the same RTSI line for the connections.

`DestinationTerminal` Specifies the destination of the connection.

The following list describes each value of `DestinationTerminal`:

`nctDestTermRTSI0 … nctDestTermRTSI6`

Selects a general-purpose RTSI line as destination (output) of the connection.

`nctDestTerm10HzResync`

`nctDestTerm10HzResync` instructs the CAN card to use a 10 Hz, 50 percent duty cycle clock to resynchronize its local timebase. This slow rate is required for resynchronization of CAN cards. On each pulse of the resync clock, this CAN card brings its local timebase into sync.

When synchronizing to an E-Series MIO card, a typical use of this value is to use RTSI0-6 as the `SourceTerminal`, then use NI-DAQ functions to program the MIO card's Counter 0 to generate a 10 Hz 50 percent duty cycle clock on the RTSI line.

When synchronizing to a CAN card, a typical use of this value is to use RTSI0-6 as the `SourceTerminal`, then route the other CAN card's `nctSrcTerm10HzResyncEvent` as the `SourceTerminal` to the same RTSI line.

`nctDestTerm10HzResync` applies to the entire CAN card, including both interfaces of a 2-port CAN card. The CAN card is specified by the task interface, such as the `Interface` input to `nctInitialize`.

The default (disconnected) state of this destination means the CAN card does not resynchronize its local timebase.

`nctDestTermStartTrig`

`nctDestTermStartTrig` selects the start trigger, the event that begins sampling for tasks.

The start trigger is the same for all tasks using a given interface, such as the `Interface` input to `nctInitialize`.

By selecting RTSI0-6 as the `SourceTerminal`, you route the start trigger from another CAN or DAQ card. This ensures that sampling begins at the same time on both cards. For example, you can synchronize with an E-Series DAQ MIO card by routing the MIO card's AI start trigger to a RTSI line and then routing the same RTSI line with `nctDestTermStartTrig` as the `DestinationTerminal` on the CAN card.

The default (disconnected) state of this destination means the start trigger occurs when communication begins on the interface. Because communication begins when the first task of the interface is started, this does not synchronize sampling with other NI cards.

`Modifiers`	Provides optional connection information for certain source/destination pairs. The current release of NI-CAN does not use this information for any source/destination pair, so you must pass `Modifiers` as zero.

Outputs

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

Description

This VI connects a specific pair of source/destination terminals. One of the terminals is typically a RTSI signal, and the other terminal is an internal terminal in the CAN hardware. By connecting internal terminals to RTSI, you can synchronize the CAN card with another hardware product such as an NI-DAQ card.

The most common uses of RTSI synchronization are demonstrated by the CAN/DAQ programming examples.

When the final task for a given interface is cleared with `nctClear`, NI-CAN disconnects all terminal connections for that interface. Therefore, the `nctDisconnectTerminals` function is not required for most applications. NI-DAQ terminals remain connected after the tasks are cleared, so you must disconnect NI-DAQ terminals manually at the end of your application.

For a list of valid source/destination pairs, refer to the following section.

Valid Combinations of Source/Destination

Table 6-3 lists all valid combinations of `SourceTerminal` and `DestinationTerminal`.

NI-CAN hardware have the following limitations.

- PXI cards do not support **RTSI 6**.
- Signals received from a RTSI source cannot occur faster than 1 kHz. This prevents the card from receiving a 10 MHz or 20 MHz timebase, such as provided by NI E-Series MIO hardware.

- Signals received from a RTSI source must be at least 100 μs in length to be detected. This prevents the card from receiving triggers in the nanoseconds range, such as the AI trigger provided by NI E-Series MIO hardware.

Table 6-3. Valid Combinations of Source/Destination

Source	Destination		
	RTSI0 to RTSI6	**10 Hz Resync**	**Start Trigger**
RTSI0 to RTSI6	—	X	X
10 Hz Resync Event	X	—	X
Start Trigger Event	X	—	—

nctCreateMessage

Purpose

Create a message configuration and associated channel configurations within your application.

Format

```
nctTypeStatus    nctCreateMessage(
                     nctTypeMessageConfig     MessageConfig,
                     u32                      NumberOfChannels,
                     nctTypeChannelConfig *   ChannelConfigList,
                     u32                      Interface,
                     u32                      Mode,
                     f64                      SampleRate,
                     nctTypeTaskRef *         TaskRef)
```

Inputs

`MessageConfig`

Configures properties for a new message. This function creates a task for a single message with one or more channels. You provide the properties in a C `struct`.

The properties are similar to the message properties in MAX:

`u32` `MsgArbitrationID`

Configures the arbitration ID of the message.

Use the `Extended` property to specify whether the ID is standard (11-bit) or extended (29-bit).

`u32` `Extended`

Configures a Boolean value that indicates whether the message arbitration ID is standard 11-bit format (0) or extended 29-bit format (1).

`u32` `MsgDataBytes`

Configures the number of data bytes in the message. The range is 0 to 8.

`NumberOfChannels`

Specifies the number of channel configurations you provide in `ChannelConfigList`.

`ChannelConfigList` Configures the list of channels for the new message. `ChannelConfigList` is an array of a C `struct`, with one C `struct` for each channel.

The properties of each channel are similar to the channel properties in MAX:

`u32` `StartBit`

Configures the starting bit position in the message. The range is 0 (lowest bit in first byte), to 63 (highest bit in last byte).

`u32` `NumBits`

Configures the number of bits in the message. The range is 0 to 64.

`u32` `DataType`

Configures the channel's data type in the message. Values are `nctDataSigned`, `nctDataUnsigned`, and `nctDataFloat`.

`u32` `ByteOrder`

Configures the channel's byte order in the message. Values are `nctOrderIntel` (little-endian), and `nctOrderMotorola` (big-endian).

`f64` `ScalingFactor`

Configures the scaling factor used to convert raw bits of the message to/from scaled floating-point units. The scaling factor is the A in the linear scaling formula $AX + B$, where X is the raw data, and B is the scaling offset.

`f64` `ScalingOffset`

Configures the scaling offset used to convert raw bits of the message to/from scaled floating-point units. The scaling offset is the B in the linear scaling formula $AX + B$, where X is the raw data, and A is the scaling factor.

	f64	MaxValue Configures the maximum value of the channel in scaled floating-point units. The `nctRead` and `nctWrite` functions do not coerce samples when converting to/from CAN messages. You can use this value with the user-interface functions of your development environment to set the range of front-panel controls and indicators.
	f64	MinValue Configures the minimum value of the channel in scaled floating-point units. The `nctRead` and `nctWrite` functions do not coerce samples when converting to/from CAN messages. You can use this value with the user-interface functions of your development environment to set the range of front-panel controls and indicators.
	f64	DefaultValue Configures the default value of the channel in scaled floating-point units. For information on how the `DefaultValue` is used, refer to the `nctRead` and `nctWrite` functions.
	const str	Unit Configures the unit string of the channel. The string is no more than 64 characters in length. You can use this value to display units (such as volts or RPM) along with the channel's samples.
`Interface`	Specifies the CAN interface to use for this task. The interface input uses an enumeration in which value 0 selects `CAN0`, value 1 selects `CAN1`, and so on.	

The default baud rate for the `Interface` is defined within MAX, but you can change it by setting the `nctPropIntfBaudRate` property with `nctSetProperty`.

`Mode` Specifies the I/O mode for the task:

`nctModeInput`

Input channel data from received CAN messages. Use the `nctRead` function to obtain input samples as single-point, array, or waveform.

Use this input mode to read waveforms of timed samples, such as for comparison with NI-DAQ waveforms. You can also use this input mode to read a single point from the most recent message, such as for control or simulation.

`nctModeOutput`

Output channel data to CAN messages for transmit. Use the `nctWrite` function to write output samples as single-point, array, or waveform.

`nctModeTimestampedInput`

Input channel data from received CAN messages. Use the `nctRead` function to obtain input samples as an array of sample/timestamp pairs (refer to `nctReadTimestamped`).

Use this input mode to read samples with timestamps that indicate when each message is received from the network.

`SampleRate` Specifies the timing to use for samples of the task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately.

For `Mode` of `NctModeInput`, `SampleRate` of zero means `nctRead` returns a single point from the most recent message received, and greater than zero means `nctRead` returns samples timed at the specified rate.

For `Mode` of `NctModeOutput`, `SampleRate` of zero means CAN messages transmit immediately when `nctWrite` is called, and greater than zero means CAN messages are transmitted periodically at the specified rate.

For Mode of NctModeTimestampedInput, SampleRate is ignored.

Outputs

TaskRef — Use TaskRef with all subsequent functions to reference the task. Pass this task reference to nctStart before you read or write samples for the message.

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the ncStatusToString function of the Frame API to obtain a descriptive string for the return value. The ncStatusToString and ncGetHardwareInfo functions are the only Frame API functions that can be called within a Channel API application.

Description

To use message and channel configurations from MAX or a CAN database, use the nctInitStart or nctInitialize functions. The nctCreateMessage function provides an alternative in which you create the message and channel configurations within your application, without use of MAX or a CAN database.

nctCreateMessage returns a task reference you wire to nctStart to start communication for the message and its channels.

nctDisconnectTerminals

Purpose

Disconnect terminals in the CAN hardware.

Format

```
nctTypeStatus    nctDisconnectTerminals(
                     nctTypeTaskRef     TaskRef,
                     u32                SourceTerminal,
                     u32                DestinationTerminal,
                     u32                Modifiers);
```

Inputs

`TaskRef`	Task reference from the previous NI-CAN function. The task reference is originally returned from `nctInitStart`, `nctInitialize`, or `nctCreateMessage`.
`SourceTerminal`	Specifies the source of the connection. For a description of values for `SourceTerminal`, refer to `nctConnectTerminals`.
`DestinationTerminal`	Specifies the destination of the connection. For a description of values for `DestinationTerminal`, refer to `nctConnectTerminals`.
`Modifiers`	Provides optional connection information for certain source/destination pairs. The current release of NI-CAN does not use this information for any source/destination pair, so you must pass `Modifiers` as zero.

Outputs

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

Description

This function disconnects a specific pair of source/destination terminals you previously connected with nctConnectTerminals.

When the final task for a given interface is cleared with nctClear, NI-CAN disconnects all terminal connections for that interface. Therefore, the nctDisconnectTerminals function is not required for most applications. You typically use this function to change RTSI connections dynamically while your application is running. First use nctStop to stop all tasks for the interface, then use nctDisconnectTerminals and nctConnectTerminals to adjust RTSI connections, then nctStart to restart sampling.

nctGetNames

Purpose

Get an array of CAN channel names or message names from MAX or a CAN database file.

Format

```
nctTypeStatus    nctGetNames(
                     cstr              FilePath,
                     u32               Mode,
                     cstr              MessageName,
                     u32               SizeofChannelList,
                     str               ChannelList);
```

Inputs

`FilePath`

`FilePath` is an optional path to a CAN database file from which to get channel names. The file must use either `.DBC` or `.NCD` extension. Files with extension `.DBC` use the CANdb database format. Files with extension `.NCD` use the NI-CAN database format. You can generate NI-CAN database files from the Save Channels or FP1300 Config selection in MAX.

If you pass NULL or empty-string to `FilePath`, this function gets the channel names from MAX. The MAX CAN channels are in the MAX CAN Channels listing within **Data Neighborhood**.

`Mode`

Specifies the type of names to return.

`nctGetNamesModeChannels`

Return list of channel names. You can pass the returned `ChannelList` to `nctInitStart`.

`nctGetNamesModeMessages`

Return list of message names.

`MessageName`

`MessageName` is an optional input that filters the names for a specific message. If you pass NULL or empty-string to `MessageName`, this function returns all names in the database. If you pass a non empty string, the `ChannelList` output is limited to channels of the specified message.

This input applies to `Mode` of `nctGetNamesModeChannels` only. It is ignored for `Mode` of `nctGetNamesModeMessages`.

`SizeofChannelList`

Number of bytes allocated for the `ChannelList` output.

If all of the channel names do not fit in the allocated `ChannelList`, this function returns as much as possible with an error.

Use the `nctGetNamesLength` function to determine the proper `SizeofChannelList`.

Outputs

`ChannelList`

Returns the comma-separated list of channel names.

Each name in `ChannelList` uses the minimum syntax required to properly initialize:

- If `FilePath` is wired, `nctGetNames` prepends the file path to the first name in `ChannelList`, with a double colon separating the file path and channel name.
- If a channel name is used within only one message in the database, `nctGetNames` returns only the channel name in the list. If a channel name is used within multiple messages, `nctGetNames` prepends the message name to that channel name, with a decimal point separating the message and channel name. This syntax ensures that the duplicate channel is associated to a single message in the database.

For more information on the syntax conventions for channel names, refer to `nctInitStart`.

To start a task for all channels returned from `nctGetNames`, pass `ChannelList` to the `nctInitStart` function to start a task.

You can also use `ChannelList` with a user-interface control such as a ring or list box. The user of your application can then select names using this control, and the selected names can be passed to `nctInitStart`.

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

nctGetNamesLength

Purpose

Get the required size for a specified list of channels to allocate an array for the `ChannelList` input of `nctGetNames`.

Format

```
nctTypeStatus    nctGetNamesLength(
                     cstr                FilePath,
                     u32                 Mode,
                     cstr                MessageName,
                     u32 *               SizeofChannelList);
```

Inputs

`FilePath`

`FilePath` is an optional path to a *CAN database* file from which to get channel names. The file must use either the `.DBC` or `.NCD` extension.

If you pass NULL or empty-string to `FilePath`, this function examines the channel names from MAX.

For more information on `FilePath`, refer to `nctGetNames`.

`Mode`

Specifies the type of names to examine.

`nctGetNamesModeChannels`

Examine the list of channel names.

`nctGetNamesModeMessages`

Examine the list of message names.

`MessageName`

`MessageName` is an optional input that filters the names for a specific *message*. If you pass NULL or empty-string to `MessageName`, this function returns all names in the database. If you pass a nonempty string, the `SizeofChannelList` output is limited to channels of the specified message.

This input applies to `Mode of nctGetNamesModeChannels` only. It is ignored for `Mode of nctGetNamesModeMessages`.

Outputs

`SizeofChannelList`	Number of bytes required for `nctGetNames` to return all names for the specified `FilePath`, `Mode`, and `MessageName`. After calling `nctGetNamesLength`, you can allocate an array of size `SizeofChannelList`, then pass that array to `nctGetNames` using the same input parameters. This ensures that `nctGetNames` will return all names without error.

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

nctGetProperty

Purpose

Get a property for the task, or a single channel within the task.

Format

```
nctTypeStatus    nctGetProperty(
                     nctTypeTaskRef    TaskRef,
                     cstr              ChannelName,
                     u32               PropertyId,
                     u32               SizeofValue,
                     void *            Value,
```

Inputs

`TaskRef`
Task reference from the previous NI-CAN function. The task reference is originally returned from `nctInitStart`, `nctInitialize`, or `nctCreateMessage`.

`ChannelName`
Specifies an individual channel within the task. If you pass empty-string to `ChannelName`, this means the property applies to the entire task, not a specific channel.

Properties that begin with the word *Channel* or *Message* do not apply to the entire task, but an individual channel or message within the task. For these channel-specific properties, you must pass the name of a channel from channel list into the `ChannelName` input.

For properties that do not begin with the word *Channel* or *Message*, you must pass empty-string (`" "`) into `ChannelName`. You must not pass NULL into `ChannelName`.

`PropertyId`
Selects the property to get.

For a description of each property, including its data type and `PropertyId`, refer to the Properties section.

`SizeofValue`
Number of bytes allocated for the `Value` output. This size normally depends on the data type listed in the property's description.

Outputs

`Value` Returns the property value. `PropertyId` determines the data type of the returned value.

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

Properties

`u32` `nctPropNumChannels`

Returns the number of channels initialized in channel list. This is the number of array entries required when using `nctRead` or `nctWrite`.

`f64` `nctPropTimeout`

Returns the `nctPropTimeout` property, which is used with some input task configurations. For more information, refer to the `nctPropTimeout` property in `nctSetProperty`.

`u32` `nctPropSamplesPending`

Returns the number of samples available to be read using `nctRead`. If you set the `NumberOfSamplesToRead` input of `nctRead` to this value, `nctRead` returns immediately without waiting.

This property applies only to tasks initialized with `Mode` of `NctModeInput`, and `SampleRate` greater than zero. For all other configurations, it returns an error.

`u32` `nctPropBehaviorAfterFinalOut`

Returns the `nctPropBehaviorAfterFinalOut` property, which is used with some output task configurations. For more information, refer to the `nctPropBehaviorAfterFinalOut` property in `nctSetProperty`.

`u32` `nctPropInterface`

Returns the `Interface` initialized for the task, such as with the `nctInitStart` function.

`u32` `nctPropMode`

Returns the `Mode` initialized for the task, such as with the `nctInitStart` function.

`f64` `nctPropSampleRate`

Returns the `SampleRate` initialized for the task, such as with the `nctInitStart` function.

`u32` `nctPropMsgArbitrationId`

Returns the arbitration ID of the channel's message.

To determine whether the ID is standard (11-bit) or extended (29-bit), get the `nctPropMsgIsExtended` property.

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`u32` `nctPropMsgIsExtended`

Returns a Boolean value that indicates whether the arbitration ID of the channel's message is standard 11-bit format (0) or extended 29-bit format (1).

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`u32` `nctPropMsgByteLength`

Returns the number of data bytes in the channel's message. The range is 0 to 8.

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`str` `nctPropMsgName`

Returns the name of the channel's message. The string is no more than 80 characters in length.

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`u32` `nctPropChanStartBit`

Returns the starting bit position in the message. The range is 0 (lowest bit in first byte), to 63 (highest bit in last byte).

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`u32` `nctPropChanNumBits`

Returns the number of bits in the message. The range is 0 to 64.

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`u32` `nctPropChanByteOrder`

Returns the channel's byte order in the message. Values are `nctOrderIntel` (little-endian), and `nctOrderMotorola` (big-endian).

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`u32` `nctPropChanDataType`

Returns the channel's data type in the message. Values are `nctDataSigned`, `nctDataUnsigned`, and `nctDataFloat`.

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`f64` `nctPropChanScalFactor`

Returns the scaling factor used to convert raw bits of the message to/from scaled floating-point units. The scaling factor is the A in the linear scaling formula $AX + B$, where X is the raw data, and B is the scaling offset.

CAN messages use the raw data, and the `nctRead` and `nctWrite` functions provide access to samples in floating-point units.

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`f64` `nctPropChanScalOffset`

Returns the scaling offset used to convert raw bits of the message to/from scaled floating-point units. The scaling offset is the B in the linear scaling formula $AX + B$, where X is the raw data, and A is the scaling factor.

CAN messages use the raw data, and the `nctRead` and `nctWrite` functions provide access to samples in floating-point units.

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`f64` `nctPropChanMinValue`

Returns the minimum value of the channel in scaled floating-point units.

The `nctRead` and `nctWrite` functions do not coerce samples when converting to/from CAN messages. You can use this value with the user-interface functions of your development environment to set the range of front-panel controls and indicators.

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`f64` `nctPropChanMaxValue`

Returns the maximum value of the channel in scaled floating-point units.

The `nctRead` and `nctWrite` functions do not coerce samples when converting to/from CAN messages. You can use this value with the user-interface functions of your development environment to set the range of front-panel controls and indicators.

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`f64` `nctPropChanDefaultValue`

Returns the default value of the channel in scaled floating-point units.

For information on how `nctPropChanDefaultValue` is used, refer to the `nctRead` and `nctWrite` functions.

The value of this property is originally set within MAX. If the channel is initialized directly from a CAN database, the value is 0.0 by default, but it can be changed using `nctSetProperty`.

`str` `nctPropChanUnitString`

Returns the unit string of the channel. The string is no more than 80 characters in length.

You can use this value to display units (such as volts or RPM) along with the channel's samples.

The value of this property is originally set within MAX or the CAN database and cannot be changed using `nctSetProperty`.

`u32` `nctPropHwSerialNum`

Returns the hardware serial number for the NI-CAN hardware that contains `Interface`.

`u32` `nctPropHwFormFactor`

Returns the hardware form factor for the NI-CAN hardware that contains `Interface`. Values are `nctHwFormFactorPCI`, `nctHwFormFactorPXI`, `nctHwFormFactorPCMCIA`, and `nctHwFormFactorAT`.

`u32` `nctPropHwTransceiver`

Returns the hardware form factor for the NI-CAN hardware that contains `Interface`. Values are `nctHwTransceiverHS`, and `nctHwTransceiverLS`.

This property is not supported on the PCMCIA form factor.

`u32` `nctPropVersionMajor`

Returns the major version of the NI-CAN software, such as the *2* in version *2.1.5*.

`u32` `nctPropVersionMinor`

Returns the minor version of the NI-CAN software, such as the *1* in version *2.1.5*.

`u32` `nctPropVersionUpdate`

Returns the update version of the NI-CAN software, such as the *5* in version *2.1.5*.

`u32` `nctPropVersionPhase`

Returns the phase of the NI-CAN software. Values are `nctPhaseDevelopment`, `nctPhaseAlpha`, `nctPhaseBeta`, and `nctPhaseRelease`. Versions of NI-CAN in hardware kits or on `ni.com` will always be `nctPhaseRelease`.

`u32` `nctPropVersionBuild`

Returns the build number of the NI-CAN software. This number applies to `nctPhaseDevelopment`, `nctPhaseAlpha`, and `nctPhaseBeta` phase only, and should be ignored for `nctPhaseRelease` phase.

`str` `nctPropVersionComment`

Returns a comment string for the NI-CAN software. If you received a custom release of NI-CAN from National Instruments, this comment often describes special features of the release.

`u32` `nctPropIntfBaudRate`

Returns the baud rate in use by the `Interface`.

Basic baud rates such as 125000 and 500000 are specified as the numeric rate.

Advanced baud rates are specified as 8000*XXYY* hex, where *YY* is the value of Bit Timing Register 0 (BTR0), and *XX* is the value of Bit Timing Register 1 (BTR1). For more information, refer to the **Interface Properties** dialog in MAX.

The value of this property is originally set within MAX, but it can be changed using `nctSetProperty`.

nctInitialize

Purpose

Initialize a task for the specified channel list.

Format

```
nctTypeStatus    nctInitialize(
                     cstr              ChannelList,
                     u32               Interface,
                     u32               BaudRate,
                     u32               Mode,
                     f64               SampleRate,
                     nctTypeTaskRef *  TaskRef);
```

Inputs

`ChannelList`

Comma-separated list of channel names to initialize as a task.

For more information, refer to the channel list input of `nctInitStart`.

`Interface`

Specifies the CAN interface to use for this task.

The interface input uses an enumeration in which value 0 selects `CAN0`, value 1 selects `CAN1`, and so on.

If you pass the special value `65535` to `Interface`, this function uses the default interface as defined in the MAX configuration. If the default interface in MAX is **All**, or if one or more channels in `ChannelList` specifies a *filepath*, the `Interface` is a required input to this function.

The Channel API and Frame API cannot use the same CAN network interface simultaneously. If the CAN network interface is already initialized in the Frame API, this function returns an error.

`Mode`

Specifies the I/O mode for the task:

`nctModeInput`

Input channel data from received CAN messages. Use the `nctRead` function to obtain input samples as single-point, array, or waveform.

Use this input mode to read waveforms of timed samples, such as for comparison with NI-DAQ waveforms. You can also use this input mode to read a single point from the most recent message, such as for control or simulation.

For this mode, the channels in `ChannelList` can be contained in multiple messages.

`nctModeOutput`

Output channel data to CAN messages for transmit. Use the `nctWrite` function to write output samples as single-point, array, or waveform.

For this mode, there are restrictions on using channels in `ChannelList` that are contained in multiple messages. Refer to `nctWrite` for more information.

`nctModeTimestampedInput`

Input channel data from received CAN messages. Use the `nctRead` function to obtain input samples as an array of sample/timestamp pairs (refer to `nctReadTimestamped`).

Use this input mode to read samples with timestamps that indicate when each message is received from the network.

For this mode, the channels in `ChannelList` must be contained in a single message.

`SampleRate` Specifies the timing to use for samples of the task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately.

For `Mode` of `NctModeInput`, `SampleRate` of zero means `nctRead` returns a single point from the most recent message received, and greater than zero means `nctRead` returns samples timed at the specified rate.

For `Mode` of `NctModeOutput`, `SampleRate` of zero means CAN messages transmit immediately when `nctWrite` is called, and greater than zero means CAN messages are transmitted periodically at the specified rate.

For `Mode` of `NctModeTimestampedInput`, `SampleRate` is ignored.

Outputs

TaskRef	Use TaskRef with all subsequent functions to reference the task. Pass this task reference to nctStart before you read or write samples for the message.

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the ncStatusToString function of the Frame API to obtain a descriptive string for the return value. The ncStatusToString and ncGetHardwareInfo functions are the only Frame API functions that can be called within a Channel API application.

Description

The nctInitialize function does not start communication. This enables you to use nctSetProperty to change the task's properties, or nctConnectTerminals to synchronize CAN or DAQ cards. After you change properties or connections, use nctStart to start communication for the task.

nctInitStart

Purpose

Initialize a task for the specified channel list, then start communication.

Format

```
nctTypeStatus    nctInitStart(
                     cstr                ChannelList,
                     u32                 Interface,
                     u32                 BaudRate,
                     u32                 Mode,
                     f64                 SampleRate,
                     nctTypeTaskRef *    TaskRef);
```

Inputs

`ChannelList`

Comma-separated list of channel names to initialize as a task.

You can type in the channel list as a string constant, or you can obtain the list from MAX or another CAN database by using the `nctGetNames` function.

You can initialize the same `ChannelList` with different `Interface`, `Mode`, or `SampleRate`, because each task reference is unique.

The following paragraphs describe the syntax of each channel name. Brackets indicate optional fields.

[*filepath*::][*message*.]*channel*

- *filepath* is the path to a CAN database file from which to import the channel (signal) configurations. The filepath must use Windows directory syntax, and must be followed by a double-colon.

  If *filepath* is not included, the channel configuration is obtained from MAX. The MAX CAN channels are in the MAX CAN Channels listing within **Data Neighborhood**.

  Once you specify a *filepath*, it will continue to be applied to subsequent names in the channel list until you specify a new *filepath*. After using *filepath* for a CAN database file, you can revert to using MAX by specifying an empty *filepath* (double colon only).

- *message* refers to the message in which the *channel* is contained. The message name must be followed by a decimal point.

  If the *channel* name occurs in multiple messages, you must specify the *message* name to identify the channel uniquely. Within MAX, channels with the same name in multiple messages are shown with a yellow exclamation point.

  If the *channel* name is unique across all channels, the *message* name is not required.

- *channel* refers to the channel (signal) name in MAX or the *filepath* CAN database.

The following examples demonstrate the channel list syntax:

- List of channels from MAX, each channel name unique across all messages.

  `myChan1,myChan2,myChan3`

- List of channels from a CANdb file, each channel name unique across all messages.

  `C:\MyCandb\MyChannels.DBC::myChan1`

  `myChan2,myChan3`

- List of channels from MAX, with one channel duplicated across two messages. `MyChan2` and `MyChan3` must be unique across all messages.

  `myMessage1.myChan1,myChan2,`

  `myMessage2.myChan1,myChan3`

- List of two channels from a CANdb file, then two channels from MAX.

  `C:\MyCandb\MoreChannels.DBC::myChan1,`

  `myChan2,::myChan3,myChan4`

`Interface`

Specifies the CAN interface to use for this task.

The interface input uses an enumeration in which value 0 selects `CAN0`, value 1 selects `CAN1`, and so on.

If you pass the special value `65535` to `Interface`, this function uses the default interface as defined in the MAX configuration. If the default interface in MAX is **All**, or if one or more channels in `ChannelList` specifies a *filepath*, the `Interface` is a required input to this function.

The Channel API and Frame API cannot use the same CAN network interface simultaneously. If the CAN network interface is already initialized in the Frame API, this function returns an error.

`Mode` — Specifies the I/O mode for the task:

`nctModeInput`

Input channel data from received CAN messages. Use the `nctRead` function to obtain input samples as single-point, array, or waveform.

Use this input mode to read waveforms of timed samples, such as for comparison with NI-DAQ waveforms. You can also use this input mode to read a single point from the most recent message, such as for control or simulation.

For this mode, the channels in `ChannelList` can be contained in multiple messages.

`nctModeOutput`

Output channel data to CAN messages for transmit. Use the `nctWrite` function to write output samples as single-point, array, or waveform.

For this mode, there are restrictions on using channels in `ChannelList` that are contained in multiple messages. Refer to `nctWrite` for more information.

`nctModeTimestampedInput`

Input channel data from received CAN messages. Use the `nctRead` function to obtain input samples as an array of sample/timestamp pairs (refer to `nctReadTimestamped`).

For this mode, the channels in `ChannelList` must be contained in a single message.

Use this input mode to read samples with timestamps that indicate when each message is received from the network.

`SampleRate` — Specifies the timing to use for samples of the task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately.

For Mode of NctModeInput, SampleRate of zero means nctRead returns a single point from the most recent message received, and greater than zero means nctRead returns samples timed at the specified rate.

For Mode of NctModeOutput, SampleRate of zero means CAN messages transmit immediately when nctWrite is called, and greater than zero means CAN messages are transmitted periodically at the specified rate.

For Mode of NctModeTimestampedInput, SampleRate is ignored.

Outputs

TaskRef — Use TaskRef with all subsequent functions to reference the running task. Because nctInitStart starts communication, you can pass this task reference to nctRead or nctWrite.

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the ncStatusToString function of the Frame API to obtain a descriptive string for the return value. The ncStatusToString and ncGetHardwareInfo functions are the only Frame API functions that can be called within a Channel API application.

Description

The code for this function simply calls nctInitialize followed by nctStart. This provides an easy way to start a list of channels.

The following list describes the scenarios for which nctInitStart cannot be used:

- If you need to set properties for the channels, use nctInitialize, then nctSetProperty, then nctStart. The nctInitStart function starts communication, and most channel properties cannot be changed after the task is started.
- If you need to synchronize tasks for multiple NI-CAN or NI-DAQ cards, use nctInitialize, then nctConnectTerminals to synchronize, the nctStart to start communication.
- If you need to create channel configurations entirely within your application, without using MAX or a CAN database file, use nctCreateMessage, then nctStart. The nctInitStart function accepts only channel names defined in MAX or a CAN database file.

nctRead

Purpose

Read samples from a CAN task initialized with Mode of nctModeInput. Samples are obtained from received CAN messages. For an overview of nctRead, refer to the *Read* section of Chapter 4, *Using the Channel API*.

Format

```
nctTypeStatus    nctRead(
                     nctTypeTaskRef       TaskRef,
                     u32                  NumberOfSamplesToRead,
                     nctTypeTimestamp *   StartTime,
                     nctTypeTimestamp *   DeltaTime,
                     f64 *                SampleArray,
                     u32 *                NumberOfSamplesReturned);
```

Inputs

TaskRef

Task reference from the previous NI-CAN function. The task reference is originally returned from nctInitStart, nctInitialize, or nctCreateMessage.

The Mode initialized for the task must be NctModeInput.

NumberOfSamplesToRead

Specifies the number of samples to read for the task. For single-sample input, pass 1 to this parameter.

If the initialized SampleRate is zero, you must pass NumberOfSamplesToRead no greater than 1. SampleRate of zero means nctRead returns a single sample from the most recent message(s) received.

Outputs

StartTime

Returns the time of the first CAN sample in SampleArray.

This parameter is optional. If you pass NULL for the StartTime parameter, the nctRead function proceeds normally.

If the initialized SampleRate is greater than zero, the StartTime is determined by the sample timing.

If the initialized SampleRate is zero, the StartTime is zero, because the most recent sample is returned regardless of timing.

StartTime uses the nctTypeTimestamp data type. The nctTypeTimestamp data type is a 64-bit unsigned integer compatible with the Microsoft Win32 FILETIME type. This absolute time is kept in a Coordinated Universal Time (UTC) format. UTC time is loosely defined as the current date and time of day in Greenwich England. Microsoft defines its UTC time (FILETIME) as a 64-bit counter of 100 ns intervals that have elapsed since 12:00 a.m., January 1, 1601. Because `nctTypeTimestamp` is compatible with Win32 `FILETIME`, you can pass it into the Win32 `FileTimeToLocalFileTime` function to convert it to your local time zone, and then pass the resulting local time to the Win32 `FileTimeToSystemTime` function to convert to the Win32 `SYSTEMTIME` type. `SYSTEMTIME` is a struct with fields for year, month, day, and so on. For more information on Win32 time types and functions, refer to your Microsoft Win32 documentation.

`DeltaTime`

Returns the time between each sample in `SampleArray`.

This parameter is optional. If you pass NULL for the `DeltaTime` parameter, the `nctRead` function proceeds normally.

If the initialized `SampleRate` is greater than zero, the `DeltaTime` is determined by the sample timing.

If the initialized `SampleRate` is zero, the `DeltaTime` is zero, because the most recent sample is returned regardless of timing.

`DeltaTime` uses the `nctTypeTimestamp` data type. The delta time is a relative 64-bit counter of 100 ns intervals, not an absolute UTC time. Nevertheless, you can use functions like the Win32 `FileTimeToSystemTime` function to convert to the Win32 `SYSTEMTIME` type. In addition, you can use the 32-bit `LowPart` of `DeltaTime` to obtain a simple 100 ns count, because `SampleRates` as slow as 0.4 Hz are still limited to a 32-bit 100 ns count.

`SampleArray`

Returns an array of arrays (2D array), one array for each channel initialized in the task. Each channel's array must have `NumberOfSamplesToRead` entries allocated.

For example, if you call `nctInitStart` with `ChannelList` of `mych1,mych2,mych3`, then call `nctRead` with `NumberOfSamplesToRead` of `10`, `SampleArray` must be allocated as:

```
f64 SampleArray[3][10];
```

The order of channel entries in `SampleArray` is the same as the order in the original `ChannelList`.

If you need to determine the number of channels in the task after initialization, get the `nctPropNumChannels` property for the task reference.

If no message has been received since you started the task, the default value of the channel (`nctPropChanDefaultValue`) is returned in all entries of `SampleArray`.

`NumberOfSamplesReturned` indicates the number of samples returned for each channel in `SampleArray`. The remaining entries are left unchanged (zero).

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

Description

When using `Mode` of `nctModeInput`, you can specify channels in `ChannelList` that span multiple messages.

If the initialized `SampleRate` is greater than zero, this function returns an array of samples, each of which indicates the value of the CAN channel at a specific point in time. The `nctRead` function waits for these samples to arrive in time before returning. In other words, the `SampleRate` specifies a virtual clock that copies the most recent value from CAN messages for each sample time. The changes in sample values from message to message enable you to view the CAN channel over time, such as for comparison with other CAN or DAQ input channels.

If the initialized `SampleRate` is zero, `nctRead` returns a single sample from the most recent message(s) received. To this single-point read, you must pass the `NumberOfSamplesToRead` parameter as `1`.

You can use the return value of `nctRead` to determine whether a new message has been received since the previous call to `nctRead` (or `nctStart`). If no message has been received, the warning code `CanWarnOldData` is returned. If a new message has been received, the success code 0 is returned.

If no message has been received since you started the task, the default value of the channel (`nctPropChanDefaultValue`) is returned in all entries of `SampleArray`.

The `nctPropTimeout` property is not used with `nctRead`.

nctReadTimestamped

Purpose

Read samples from a CAN task initialized with `Mode` of `nctModeTimestampedInput`. For an overview of `nctReadTimestamped`, refer to the *Read Timestamped* section of Chapter 4, *Using the Channel API*.

Format

```
nctTypeStatus    nctReadTimestamped(
                     nctTypeTaskRef       TaskRef,
                     u32                  NumberOfSamplesToRead,
                     nctTypeTimestamp *   TimestampArray,
                     f64 *                SampleArray,
                     u32 *                NumberOfSamplesReturned);
```

Inputs

`TaskRef` Task reference from the previous NI-CAN function. The task reference is originally returned from `nctInitStart`, `nctInitialize`, or `nctCreateMessage`.

The `Mode` initialized for the task must be `NctModeTimestampedInput`.

`NumberOfSamplesToRead` Specifies the number of samples to read for the task.

Outputs

`TimestampArray` Returns the time at which each corresponding sample in `SampleArray` was received in a CAN message.

The timestamps are returned as an array of arrays (2D array), one array for each channel initialized in the task. Each channel's array must have `NumberOfSamplesToRead` entries allocated.

For example, if you call `nctInitStart` with `ChannelList` of `mych1,mych2`, then call `nctReadTimestamped` with `NumberOfSamplesToRead` of `20`, both `TimestampArray` and `SampleArray` must be allocated as:

```
f64 TimestampArray[2][20];
f64 SampleArray[2][20];
```

The order of channel entries in `TimestampArray` is the same as the order in the original `ChannelList`.

If you need to determine the number of channels in the task after initialization, get the `nctPropNumChannels` property for the task reference.

Each timestamp in `TimestampArray` uses the `nctTypeTimestamp` data type. The `nctTypeTimestamp` data type is a 64-bit unsigned integer compatible with the Microsoft Win32 `FILETIME` type. This absolute time is kept in a Coordinated Universal Time (UTC) format. UTC time is loosely defined as the current date and time of day in Greenwich England Microsoft defines its UTC time (FILETIME) as a 64-bit counter of 100 ns intervals that have elapsed since 12:00 a.m., January 1, 1601. Because `nctTypeTimestamp` is compatible with Win32 `FILETIME`, you can pass it into the Win32 `FileTimeToLocalFileTime` function to convert it to your local time zone, and then pass the resulting local time to the Win32 `FileTimeToSystemTime` function to convert to the Win32 `SYSTEMTIME` type. `SYSTEMTIME` is a struct with fields for year, month, day, and so on. For more information on Win32 time types and functions, refer to your Microsoft Win32 documentation.

`SampleArray` Returns the sample value(s) for each received CAN message.

The samples are returned as an array of arrays (2D array), one array for each channel initialized in the task. Each channel's array must have `NumberOfSamplesToRead` entries allocated.

You must allocate `SampleArray` exactly as `TimestampArray`, and the order of channel entries is the same for both.

`NumberOfSamplesReturned` Indicates the number of samples returned for each channel in `SampleArray`, and the number of timestamps returned for each channel in `TimestampArray`. The remaining entries are left unchanged (zero).

Return Value

The return value indicates the function call status as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

Description

Each returned sample corresponds to a received CAN message for the channels initialized in `ChannelList`. For each sample, `nctReadTimestamped` returns the sample value and a timestamp that indicates when the message was received.

When using `Mode` of `nctModeTimestampedInput`, you *cannot* specify channels in `ChannelList` that span multiple messages.

Because the timing of samples returned by `nctReadTimestamped` is determined by when the message is received, the initialized `SampleRate` is not used.

The `nctPropTimeout` property determines whether this function waits for the `NumberOfSamplesToRead` messages to arrive from the network. The default value of `nctPropTimeout` is zero, but you can change it using the `nctSetProperty` function.

If `nctPropTimeout` is greater than zero, the function will wait for `NumberOfSamplesToRead` messages to arrive. If `NumberOfSamplesToRead` messages are not received before the `nctPropTimeout` expires, an error is returned.

If `nctPropTimeout` is zero, the function does not wait for messages, but instead returns samples from the messages received since the previous call to `nctReadTimestamped`. The number of samples returned is indicated in the `NumberOfSamplesReturned` output, up to a maximum of `NumberOfSamplesToRead` messages. If no new message has been received, `NumberOfSamplesReturned` is 0, and the return value indicates success.

nctSetProperty

Purpose

Set a property for the task, or a single channel within the task.

Format

```
nctTypeStatus    nctSetProperty(
                     nctTypeTaskRef    TaskRef,
                     cstr              ChannelName,
                     u32               PropertyId,
                     u32               SizeofValue,
                     void *            Value,
```

Inputs

`TaskRef`

Task reference from the previous NI-CAN function. The task reference is originally returned from `nctInitStart`, `nctInitialize`, or `nctCreateMessage`.

`ChannelName`

Specifies an individual channel within the task. If you pass NULL or empty-string to `ChannelName`, this means the property applies to the entire task, not a specific channel.

Properties that begin with the word *Channel* or *Message* do not apply to the entire task, but an individual channel or message within the task. For these channel-specific properties, you must pass the name of a channel from `ChannelList` into the `ChannelName` input.

For properties that do not begin with the word *Channel* or *Message*, you must pass empty-string (`" "`) into `ChannelName`. You must not pass NULL into `ChannelName`.

`PropertyId`

Selects the property to set.

For a description of each property, including its data type and `PropertyId`, refer to the Properties section.

`SizeofValue`

Number of bytes provided for the `Value` output. This size will normally depend on the data type listed in the property's description.

`Value`

Provides the property value. `PropertyId` determines the data type of the value.

Outputs

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

Description

You cannot set a property while the task is running. If you need to change a property prior to starting the task, you cannot use `nctInitStart`. First call `nctInitialize`, followed by `nctSetProperty`, and then `nctStart`. After you start the task, you can also change a property by calling `nctStop`, followed by `nctSetProperty`, and then `nctStart` again.

Properties

`f64` `nctPropTimeout`

Sets a time in milliseconds to wait for samples. The default value is zero.

Usage of the `nctPropTimeout` property depends on the initialized `Mode` of the task:

- The timeout value does *not* apply to an `NctModeOutput` task.
- The timeout value does *not* apply to an `NctModeInput` task. For `NctModeInput` tasks initialized with `SampleRate` greater than zero, the `NumberOfSamplesToRead` input to `nctRead` implicitly specifies the time to wait. For `NctModeInput` tasks initialized with `SampleRate` equal to zero, the `nctRead` function always returns available samples immediately, without waiting.
- The timeout value *does* apply to a `NctModeTimestampedInput` task. A timeout of zero means to return available samples immediately. A timeout greater than zero means `nctRead` will wait a maximum of `nctPropTimeout` milliseconds for `NumberOfSamplesToRead` samples to become available before returning.

`u32` `nctPropBehaviorAfterFinalOut`

The `nctPropBehaviorAfterFinalOut` property applies only to tasks initialized with `Mode` of `NctModeOutput`, and `SampleRate` greater than zero. The value specifies the behavior to perform after the final periodic sample is transmitted.

`nctPropBehaviorAfterFinalOut` uses the following values:

`nctOutBehavRepeatFinalSample`

Transmit messages for the final sample(s) repeatedly. The final messages are transmitted periodically as specified by `SampleRate`.

If there is significant delay between subsequent calls to `nctWrite`, this value means periodic messages continue between `nctWrite` calls, and messages with the final sample's data are repeated on the network.

`nctOutBehavRepeatFinalSample` is the default value of the `nctPropBehaviorAfterFinalOut` property.

`nctOutBehavCeaseTransmit`

Cease transmit of messages until the next call to `nctWrite`.

If there is significant delay between subsequent calls to `nctWrite`, this value means periodic messages cease between `nctWrite` calls, and the final sample's data is not repeated on the network.

`f64` `nctPropChanDefaultValue`

Sets the default value of the channel in scaled floating-point units.

For information on how the `nctPropChanDefaultValue` is used, refer to the `nctRead` and `nctWrite` functions.

The value of this property is originally set within MAX. If the channel is initialized directly from a CAN database, the value is 0.0 by default, but it can be changed using `nctSetProperty`.

`u32` `nctPropIntfBaudRate`

Sets the baud rate in use by the `Interface`.

This property applies to all tasks initialized with the `Interface`.

You can specify the following basic baud rates as the numeric rate: 83333, 100000, 125000, 200000, 250000, 400000, 500000, 800000, and 1000000.

You can specify advanced baud rates as 8000*XXYY* hex, where *YY* is the value of Bit Timing Register 0 (BTR0), and *XX* is the value of Bit Timing Register 1 (BTR1). For more information, refer to the **Interface Properties** dialog in MAX.

The value of this property is originally set within MAX, but it can be changed using `nctSetProperty`.

nctStart

Purpose

Start communication for the specified task.

Format

```
nctTypeStatus    nctStart(
                      nctTypeTaskRef    TaskRef);
```

Inputs

`TaskRef`	Task reference from the previous NI-CAN function. The task reference is originally returned from functions such as `nctInitialize`, or `nctCreateMessage`.

Outputs

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

Description

You must start communication for a task to use `nctRead` or `nctWrite`. After you start communication, you can no longer change the task's configuration with `nctSetProperty` or `nctConnectTerminals`.

nctStop

Purpose

Stop communication for the specified task.

Format

```
nctTypeStatus    nctStop(
                     nctTypeTaskRef    TaskRef);
```

Inputs

`TaskRef`	Task reference from the previous NI-CAN function. The task reference is originally returned from `nctInitStart`, `nctInitialize`, or `nctCreateMessage`.

Outputs

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

Description

This function stops communication so you can change the task's configuration, such as by using `nctSetProperty` or `nctConnectTerminals`. After you change the configuration, use `nctStart` to start again.

This function does not clear the configuration for the task; therefore, do *not* use it as the last NI-CAN function in your application. The `nctClear` function must always be used as the last NI-CAN function for each task.

nctWrite

Purpose

Write samples to a CAN task initialized as `NctModeOutput`. Samples are placed into transmitted CAN messages. For an overview of `nctWrite`, refer to the *Write* section of Chapter 4, *Using the Channel API*.

Format

```
nctTypeStatus    nctWrite(
                     nctTypeTaskRef    TaskRef,
                     u32               NumberOfSamplesToWrite,
                     f64 *             SampleArray);
```

Inputs

`TaskRef`

Task reference from the previous NI-CAN function. The task reference is originally returned from `nctInitStart`, `nctInitialize`, or `nctCreateMessage`.

The `Mode` initialized for the task must be `NctModeOutput`.

`NumberOfSamplesToWrite`

Specifies the number of samples to write for the task. For single-sample output, pass `1` to this parameter.

`SampleArray`

Provides an array of arrays (2D array), one array for each channel initialized in the task. Each channel's array must have `NumberOfSamplesToWrite` samples.

For example, if you call `nctInitStart` with `ChannelList` of `mych1,mych2,mych3`, then call `nctWrite` with `NumberOfSamplesToWrite` of `10`, `SampleArray` must be allocated as:

```
f64 SampleArray[3][10];
```

You must provide a valid sample value in each entry of the arrays.

The order of channel entries in `SampleArray` is the same as the order in the original `ChannelList`.

To determine the number of channels in the task after initialization, get the `nctPropNumChannels` property for the task reference.

Outputs

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require your attention.

Use the `ncStatusToString` function of the Frame API to obtain a descriptive string for the return value. The `ncStatusToString` and `ncGetHardwareInfo` functions are the only Frame API functions that can be called within a Channel API application.

Description

The associated `ChannelList` determines the messages transmitted by `nctWrite`. If all channels are contained in a single message, only that message is transmitted. If a few channels are contained in one message, and the remaining channels are contained in a second message, then two messages are transmitted.

If the initialized `SampleRate` is greater than zero, the task transmits associated CAN messages periodically at the specified rate. The first `nctWrite` transmits associated messages immediately using the first sample in each channel's array, and then begins a periodic timer at the specified rate. Each subsequent transmission of messages is based on the timer, and uses the next sample in each channel's array. After the final sample in each channel's array has been transmitted, subsequent behavior is determined by the `nctPropBehaviorAfterFinalOut` property. The default `nctPropBehaviorAfterFinalOut` behavior is to retransmit the final sample each period until `nctWrite` is called again.

If the initialized `SampleRate` is zero, the task transmits associated messages immediately for each entry in each channel's array, with as little delay as possible between messages. After the message for the final sample is transmitted, no further transmissions occur until `nctWrite` is called again, regardless of the `nctPropBehaviorAfterFinalOut` property.

NI-CAN uses a queue to store pending messages prior to transmission. `nctWrite` returns after the final message is written to this queue. This provides some time for you to call `nctWrite` again to provide a continual stream of samples.

Because all channels of a message are transmitted on the network as a unit, `nctWrite` enforces the following rules:

- You *cannot* write the same message in more than one `NctModeOutput` task.
- You *can* write more than one message in a single `NctModeOutput` task.

- You *can* write a subset of channels for a message in a single `NctModeOutput` task. For channels that are not included in the task, the channel default value (`nctPropChanDefaultValue`) is transmitted in the CAN message.

For many applications, the most straightforward technique is to assign a single `NctModeOutput` task for each message you want to transmit. In each task, include all channels of that message in the `ChannelList`. This ensures you can provide new samples for the entire message with each `nctWrite`.

7

Using the Frame API

This chapter provides information to help you get started with the Frame API.

Choose Which Objects To Use

An application written for the NI-CAN Frame API communicates on the network by using various objects. Which Frame API objects to use depends largely on the needs of your application. The following sections discuss the objects provided by the Frame API, and reasons why you might use each class of object.

Using CAN Network Interface Objects

The CAN Network Interface Object encapsulates a physical interface to a CAN network, usually a CAN port on an AT, PCI, PCMCIA or PXI card.

You use the CAN Network Interface Object to read and write complete CAN frames. As a CAN frame arrives from over the network, it can be placed into the read queue of the CAN Network Interface Object. You can retrieve CAN frames from this read queue using the `ncRead` or `ncReadMult` function. The read functions provide a timestamp of when the frame was received, the arbitration ID of the frame, the type of frame (data, remote, or RTSI), the data length, and the data bytes. You can also use the CAN Network Interface Object to write CAN frames using the `ncWrite` function.

Some possible uses for the CAN Network Interface Object include the following:

- You can use the read queue to log all CAN frames transferred across the network. This log is useful when you need to view CAN traffic to verify that all CAN devices are functioning properly.
- You can use the write queue to transmit a sequence of CAN frames in quick succession.

- You can read and write CAN frames for access to configuration settings within a device. Because such settings generally are not accessed during normal device operation, a dedicated CAN Object is not appropriate.
- For higher level protocols based on CAN, you can use sequences of write/read transactions to initialize communication with a device. In these protocols, specific sequences of CAN frames often need to be exchanged before you can access the data from a device. In such cases, you can use the CAN Network Interface Object to set up communication, then use CAN Objects for actual data transfer with the device.

In general, you use CAN Network Interface Objects for situations in which you need to transfer arbitrary CAN frames.

Using CAN Objects

The CAN Object encapsulates a specific CAN arbitration ID and its associated data.

Every CAN Object is always associated with a specific CAN Network Interface Object, used to identify the physical interface on which the CAN Object is located. Your application can use multiple CAN Objects in conjunction with their associated CAN Network Interface Object.

The CAN Object provides high level access to a specific arbitration ID. You can configure each CAN Object for different forms of background access. For example, you can configure a CAN Object to transmit a data frame every 100 milliseconds, or to periodically poll for data by transmitting a remote frame and receiving the data frame response. The arbitration ID, direction of data transfer, data length, and when data transfer occurs (periodic or unsolicited) are all preconfigured for the CAN Object. When you have configured and opened the CAN Object, data transfer is handled in the background using read and write queues. For example, if the CAN Object periodically polls for data, the NI-CAN driver automatically handles the periodic transmission of remote frames, and stores incoming data in the read queue of the CAN Object for later retrieval by the `ncRead` function. For CAN Objects that receive data frames, the `ncRead` function provides a timestamp of when the data frame arrived, and the data bytes of the frame. For CAN Objects that transmit data frames, the `ncWrite` function provides the outgoing data bytes.

Some possible uses for CAN Objects include the following:

- You can configure a CAN Object to periodically transmit a data frame for a specific arbitration ID. The CAN Object transmits the same data bytes repetitively until different data is provided using `ncWrite`.
- You can configure a CAN Object to watch for unsolicited data frames received for its arbitration ID, then store that data in the CAN Object's read queue. A watchdog timeout is provided to ensure that incoming data is received periodically. This configuration is useful when you want to apply a timeout to data received for a specific arbitration ID and store that data in a dedicated queue. If you do not need to apply a timeout for a given arbitration ID, it is preferable to use the CAN Network Interface Object to receive that data.
- You can configure a CAN Object to periodically poll for data by transmitting a remote frame and receiving the data frame response. This configuration is useful for communication with devices that require a remote frame to transmit their data.
- You can configure a CAN Object to transmit a data frame whenever it receives a remote frame for its arbitration ID. You can use this configuration to simulate a device which responds to remote frames.

In general, you use CAN Objects for data transfer for a specific arbitration ID, especially when that data transfer needs to occur periodically.

Programming Model

The following steps demonstrate how to use the Frame API functions in your application. The steps are shown in Figure 7-1 in flowchart form.

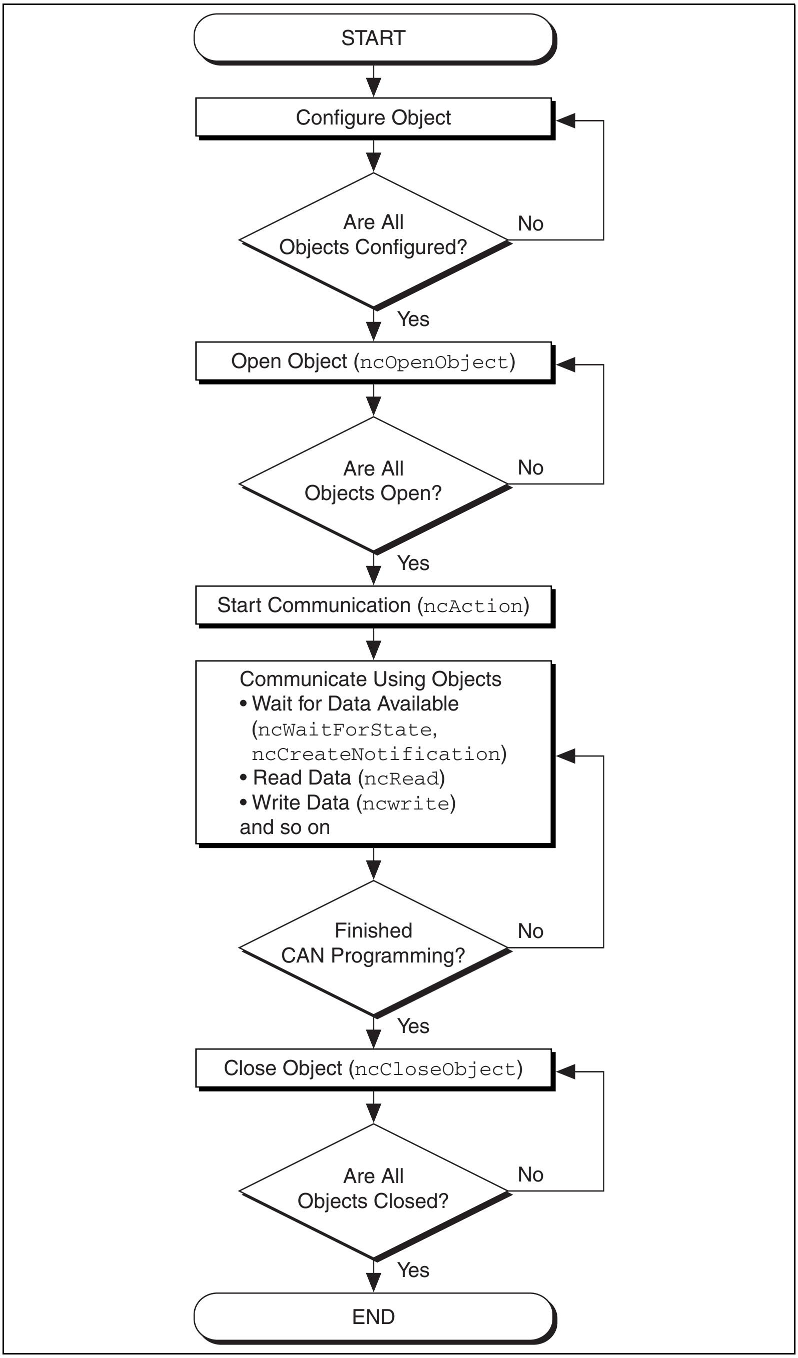


Figure 7-1. Programming Model for NI-CAN Frame API

Step 1. Configure Objects

Prior to opening the objects used in your application, you must configure the objects with their initial attribute settings. Each object is configured within your application by calling the `ncConfig` function. This function takes the name of the object to configure, along with a list of configuration attribute settings.

Step 2. Open Objects

You must call the `ncOpenObject` function to open each object you use within your application.

The `ncOpenObject` function returns a handle for use in all subsequent Frame API calls for that object. When you are using the LabVIEW function library, this handle is passed through the upper left and right terminals of each Frame API function used after the open.

Step 3. Start Communication

You must start communication on the CAN network before you can use your objects to transfer data.

If you configured your CAN Network Interface Object to start on open, that object and all of its higher level CAN Objects are started automatically by the `ncOpenObject` function, so nothing special is required for this step.

If you disabled the start-on-open attribute, when your application is ready to start communication, use the CAN Network Interface Object to call the `ncAction` function with the `Opcode` parameter set to `NC_OP_START`. This call is often useful when you want to use `ncWrite` to place outgoing data in write queues prior to starting communication. This call is also useful in high bus load situations, because it is more efficient to start communication after all objects have been opened.

If you want to reset the CAN hardware completely to clear a pending Error Passive state, you can use the CAN Network Interface Object to call the `ncAction` function with the `Opcode` parameter set to `NC_OP_RESET`. This reset must be done prior to starting communication.

Step 4. Communicate Using Objects

After you open your objects and start communication, you are ready to transfer data on the CAN network. The manner in which data is transferred depends on the configuration of the objects you are using. For this example, assume that you are communicating with a CAN device that periodically transmits a data frame. To receive this data, assume that a CAN Object is configured to watch for data frames received for its arbitration ID and store that data in its read queue.

Step 4a. Wait for Available Data

To wait for the arrival of a data frame from the device, you can call `ncWaitForState` with the `DesiredState` parameter set to `NC_ST_READ_AVAIL`. The `NC_ST_READ_AVAIL` state tells you that data for the CAN Object has been received from the network and placed into the object's read queue.

When receiving data from the device, if your only requirement is to obtain the most recent data, you are not required to wait for the `NC_ST_READ_AVAIL` state. If this is the case, you can set the read queue length of the CAN Object to zero during configuration, so that it only holds the most recent data bytes. Then you can use the `ncRead` function as needed to obtain the most recent data bytes received.

Step 4b. Read Data

Read the data bytes using `ncRead`. For CAN Objects that receive data frames, `ncRead` returns a timestamp of when the data was received, followed by the actual data bytes (the number of which you configured in step 1).

Steps 4a and 4b should be repeated for each data value you want to read from the CAN device.

Step 5. Close Objects

When you are finished accessing the CAN devices, close all objects using the `ncCloseObject` function before you exit your application.

Additional Programming Topics

The following sections outline changes to the Frame API as compared to NI-CAN 1.6.

RTSI

The Frame API provides RTSI features that are lower level than the synchronization features of the Channel API. The following list describes some of the more commonly used RTSI features in the Frame API.

- You can configure the CAN Network Interface Object to log a special RTSI frame into the read queue when a RTSI input pulses. This RTSI frame is timestamped, so you can use it to analyze the time of the RTSI pulse relative to the CAN frames on the network.
- You can configure the CAN Object to generate a RTSI output pulse when its ID is received. This allows you to trigger other products based on the reception of a specific CAN frame.
- You can configure the CAN Object to transmit a CAN frame when a RTSI input pulses. This allows you to transmit based on a functional unit in another product, such as a counter in an NI-DAQ E-series MIO product.

For more information on RTSI configuration, refer to the `ncConfig` functions in this manual.

Remote Frames

The Frame API has extensive features to transmit and receive remote frames. The following list describes some of the more commonly used remote frame features in the Frame API.

- The CAN Network Interface Object can transmit arbitrary remote frames.
- NI-CAN hardware uses the Intel 82527 CAN controller, which cannot receive arbitrary remote frames. The CAN Network Interface Object cannot receive remote frames.
- You can configure a CAN Object to transmit a remote frame and receive the corresponding data frame. The remote frame can be transmitted periodically, based on a RTSI input, or each time you call `ncWrite`.
- You can configure a CAN Object to transmit a data frame in response to reception of the corresponding remote frame.

Using Queues

To maintain an ordered history of data transfers, NI-CAN supports the use of queues, also known as FIFO (first-in-first-out) buffers. The basic behavior of such queues is common to all NI-CAN objects.

There are two basic types of NI-CAN queues: the read queue and the write queue. NI-CAN uses the read queue to store incoming network data items in the order they arrive. You access the read queue using `ncRead` to obtain the data. NI-CAN uses the write queue to transmit network frames one at a time using the network interface hardware. You access the write queue using `ncWrite` to store network data items for transmission.

State Transitions

The `NC_ST_READ_AVAIL` state transitions from false to true when NI-CAN places a new data item into an empty read queue, and remains true until you read the last data item from the queue and the queue is empty.

The `NC_ST_READ_MULT` state transitions from false to true when the number of items in a queue exceeds a threshold. The threshold is set using the `NC_ATTR_NOTIFY_MULT_LEN` attribute. The `NC_ST_READ_MULT` state and `ncReadMult` function are useful in high-traffic networks in which data items arrive quickly.

The `NC_ST_WRITE_SUCCESS` state transitions from false to true when the write queue is empty and NI-CAN has successfully transmitted the last data item onto the network. The `NC_ST_WRITE_SUCCESS` state remains true until you write another data item into the write queue. When communication starts, the `NC_ST_WRITE_SUCCESS` state is true by default.

Empty Queues

For both read and write queues, the behavior for reading an empty queue is similar. When you read an empty queue, the previous data item is returned again. For example, if you call `ncRead` when `NC_ST_READ_AVAIL` is false, the data from the previous call to `ncRead` is returned again, along with the `CanWarnOldData` warning. If no data item has yet arrived for the read queue, a default data item is returned, which consists of all zeros. You should generally wait for `NC_ST_READ_AVAIL` prior to the first call to `ncRead`.

Full Queues

For both read and write queues, the behavior for writing a full queue is similar. When you write a full queue, NI-CAN returns the `CanErrOverflowWrite` error code. For example, if you write too many data items to a write queue, the `ncWrite` function eventually returns the overflow error.

Disabling Queues

If you do not need a complete history of all data items, you can disable the read queue and/or write queue by setting its length to zero. Zero length queues are typically used only with CAN objects, not the CAN Network Interface Object. Using zero length queues generally saves memory, and often results in better performance. When a new data item arrives for a zero length queue, it overwrites the previous item without indicating an overflow. The `NC_ST_READ_AVAIL` and `NC_ST_WRITE_SUCCESS` states still behave as usual, but you can ignore them if you want only the most recent data. For example, when NI-CAN writes a new data item to the read buffer, the `NC_ST_READ_AVAIL` state becomes true until the data item is read. If you only want the most recent data, you can ignore the `NC_ST_READ_AVAIL` state, as well as the `CanWarnOldData` warning returned by `ncRead`.

Using the CAN Network Interface Object with CAN Objects

For many applications, it is desirable to use a CAN Network Interface Object in conjunction with higher level CAN Objects. For example, you can use CAN objects to transmit data or remote frames periodically, and use the CAN Network Interface Object to receive all incoming frames.

When one or more CAN Objects are open, the CAN Network Interface Object cannot receive frames which would normally be handled by the CAN Objects. The flowchart in Figure 7-2 shows the steps performed by the Frame API when a CAN frame is received.

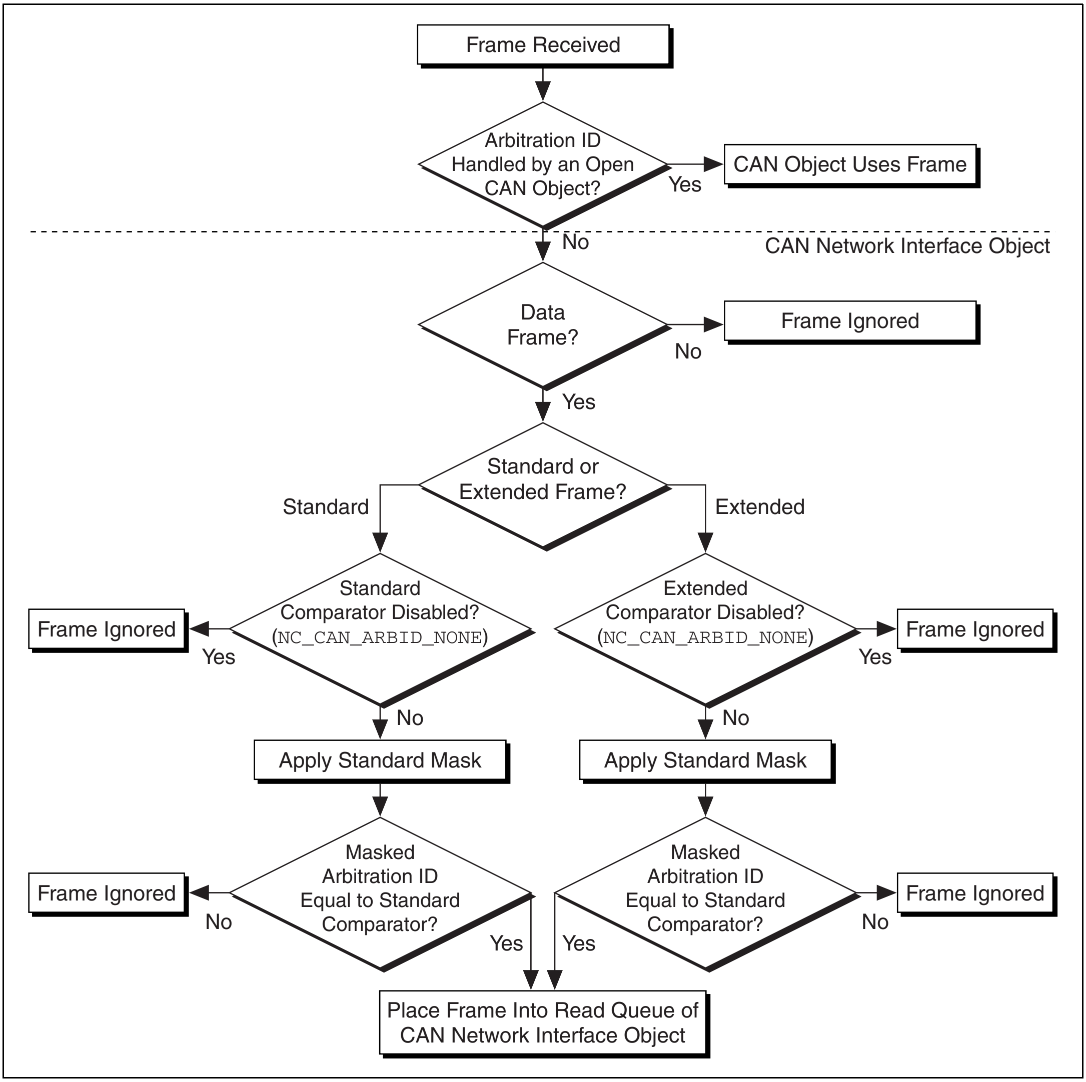


Figure 7-2. Flowchart for CAN Frame Reception

The decisions in Figure 7-2 are generally performed by the on-board CAN communications controller chip. Nevertheless, if you intend to use CAN Objects as the sole means of accessing the CAN bus, it is best to disable all frame reception in the CAN Network Interface Object by setting the comparator attributes to `NC_CAN_ARBID_NONE` (hex CFFFFFFF). By doing this, the CAN communications controller chip is best able to filter out all incoming frames except those handled by CAN Objects.

Detecting State Changes

You can detect state changes for an object using one of the following schemes:

- Call ncWaitForState to wait for one or more states to occur.
- Use ncCreateNotification in C/C++ to register a callback for one or more states.
- Use ncCreateOccurrence in LabVIEW to create an occurrence for one or more states.
- Call ncGetAttribute to get the NC_ATTR_STATE attribute.

Use the ncWaitForState function when your application must wait for a specific state before proceeding. For example, if you call ncWrite to write a frame, and your application cannot proceed until the frame is successfully transmitted, you can call ncWaitForState to wait for NC_ST_WRITE_SUCCESS.

Use the ncCreateNotification function in C/C++ when your application must handle a specific state, but can perform other processing while waiting for that state to occur. The ncCreateNotification function registers a callback function, which is invoked when the desired state occurs. For example, a callback function for NC_ST_READ_AVAIL can call ncRead and place the resulting data in a buffer. Your application can then perform any tasks desired, and process the CAN data only as needed.

Use the ncCreateOccurrence function in LabVIEW when your application must handle a specific state, but can perform other processing while waiting for that state to occur. The ncCreateOccurrence function creates a LabVIEW occurrence, which is set when the desired state occurs. Occurrences are the mechanism used in LabVIEW to provide multithreaded execution.

Use the ncGetAttribute function when you need to determine the current state of an object.

8

Frame API for LabVIEW

This chapter lists the LabVIEW VIs for the NI-CAN Frame API and describes the format, purpose, and parameters for each VI. The VIs in this chapter are listed alphabetically.

Unless otherwise stated, each NI-CAN VI suspends execution of the calling thread until it completes.

Section Headings

The following are section headings found in the Frame API for LabVIEW VIs.

Purpose

Each VI description includes a brief statement of the purpose of the VI.

Format

The format section describes the format of each VI.

Input and Output

The input and output parameters for each VI are listed.

Description

The description section gives details about the purpose and effect of each VI.

CAN Network Interface Object

The CAN Network Interface Object section gives details about using the VI with the CAN Network Interface Object.

CAN Object

The CAN Object section gives details about using the VI with the CAN Object.

List of VIs

The following table is an alphabetical list of the NI-CAN VIs for the Frame API.

Table 8-1. Frame API for LabVIEW VIs

Function	Purpose
ncAction.vi	Perform an action on an object.
ncCloseObject.vi	Close an object.
ncConfigCANNet.vi	Configure a CAN Network Interface Object before opening it.
ncConfigCANNetLS.vi	Configure a CAN Network Interface Object with logging of low-speed faults enabled.
ncConfigCANNetLS-RTSI.vi	Configure a CAN Network Interface Object with RTSI features, and with logging of low-speed faults enabled.
ncConfigCANNetRTSI.vi	Configure a CAN Network Interface Object with RTSI features.
ncConfigCANObj.vi	Configure a CAN Object before using it.
ncConfigCANObjRTSI.vi	Configure a CAN Object with RTSI features.
ncCreateOccur.vi	Create a LabVIEW occurrence for an object.
ncGetAttr.vi	Get the value of an object attribute.
ncGetHardwareInfo.vi	Get NI-CAN hardware information.
ncGetTimer.vi	Get the absolute timestamp attribute.
ncOpenObject.vi	Open an object.
ncReadNet.vi	Read single frame from a CAN Network Interface Object.
ncReadNetMult.vi	Read multiple frames from a CAN Network Interface Object.
ncReadObj.vi	Read single frame from a CAN Object.
ncReadObjMult.vi	Read multiple frames from a CAN Object.
ncReset.vi	Reset the CAN card.
ncSetAttr.vi	Set the value of an object attribute.

Table 8-1. Frame API for LabVIEW VIs (Continued)

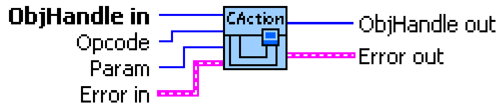
Function	Purpose
ncWait.vi	Wait for one or more states to occur in an object.
ncWriteNet.vi	Write the data value of an object.
ncWriteObj.vi	Write a single frame to a CAN Object.

ncAction.vi

Purpose

Perform an action on an object.

Format

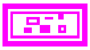


Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

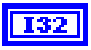
Opcode is the operation code indicating which action to perform. Refer to Tables 8-2 and 8-3.

Param is an optional parameter whose meaning is defined by **Opcode**.



Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.



code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

ncAction is a general purpose VI you can use to perform an action on the object specified by **ObjHandle in**. Its normal use is to start and stop network communication on a CAN Network Interface Object.

NI-CAN provides VIs such as **ncOpenObject** and **ncRead** for the most frequently used and/or complex actions. **ncAction** provides an easy, general purpose way to perform actions that are used less frequently or are relatively simple.

CAN Network Interface Object

NI-CAN propagates all actions on the CAN Network Interface Object up to all open CAN Objects.

Table 8-2 describes the actions supported by the CAN Network Interface Object.

Table 8-2. Actions Supported by the CAN Network Interface Object

Opcode	Param	Description
`Start CAN Network Interface`	N/A (ignored)	Transitions network interface from stopped (idle) state to started (running) state. If network interface is already started, this operation has no effect. When a network interface is started, it is communicating on the network. When you execute the Start action on a stopped CAN Network Interface Object, NI-CAN propagates it upward to all open higher-level CAN Objects. Thus, you can use it to start all higher-level network communication simultaneously.
`Stop CAN Network Interface`	N/A (ignored)	Transitions network interface from started state to stopped state. If network interface is already stopped, this operation has no effect. When a network interface is stopped, it is not communicating on the network. When you execute the Stop action on a running CAN Network Interface Object, NI-CAN propagates it upward to all open higher-level CAN Objects.
`Reset CAN Network Interface`	N/A (ignored)	Resets network interface. Stops network interface, then resets the CAN controller to clear the CAN error counters (clear error passive state). Resetting includes clearing all entries from read and write queues. The reset action is propagated up to all open higher-level CAN Objects.
`Output on RTSI line`	N/A (ignored)	Output a pulse or toggle on the RTSI line depending upon the **RTSI Behavior** attribute.

CAN Object

All actions performed on a CAN Object affect that CAN Object alone, and do not affect other CAN Objects or communication as a whole. Table 8-3 describes the actions supported by the CAN Object.

Table 8-3. Actions Supported by the CAN Object

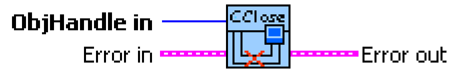
Opcode	Param	Description
`Output on RTSI line`	N/A (ignored)	Output a pulse or toggle on the RTSI line depending upon the **RTSI Behavior** attribute.

ncCloseObject.vi

Purpose

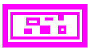
Close an object.

Format



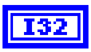
Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.



Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. Unlike other NI-CAN VIs, this VI always closes the object, regardless of the value of **status**.



code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is

returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

ncCloseObject closes an object when it no longer needs to be in use, such as when the application is about to exit. When an object is closed, NI-CAN stops all pending operations and clears RTSI configuration for the object, and your application can no longer use that specific **ObjHandle in**.

Unlike other NI-CAN VIs, this VI always closes the object, regardless of the **Status** in **Error In**.

CAN Network Interface Object

ObjHandle in refers to an open CAN Network Interface Object.

CAN Object

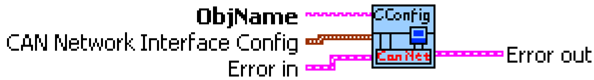
ObjHandle in refers to an open CAN Object.

ncConfigCANNet.vi

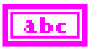
Purpose

Configure a CAN Network Interface Object before opening it.

Format

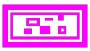


Input



ObjName is the name of the CAN Network Interface Object to configure. This name uses the syntax “CAN*x*”, where *x* is a decimal number starting at zero that indicates the CAN network interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX).

The Frame API and Channel API cannot use the same CAN network interface simultaneously. If the CAN network interface is already initialized in the Channel API, this function returns an error.



CAN Network Interface Config provides the core configuration attributes of the CAN Network Interface Object. This cluster uses the typedef **ncNetAttr.ctl**. You can wire in the cluster by first placing it on your front panel from the NI-CAN Controls palette, or you can right-click the VI input and select **Create Constant** or **Create Control**.

Start On Open indicates whether communication starts for the CAN Network Interface Object (and all applicable CAN Objects) immediately upon opening the object with **ncOpenObject**. The default is TRUE, which starts communication when **ncOpenObject** is called. If you set **Start On Open** to FALSE, you can call **ncSetAttribute** after opening the interface, then **ncAction** to start communication. The **ncSetAttribute** VI can be used to set attributes that are not contained within the **ncConfigCANNet** VI.

Baud Rate is the baud rate to use for communication. Basic baud rates are supported, including 100000, 125000, 250000, 500000, and 1000000. If you are familiar with the Bit Timing registers used in CAN controllers, you can use a special hexadecimal baud rate of 0x8000zzyy, where *yy* is the desired value for register 0 (BTR0), and *zz* is the desired value for register 1 (BTR1) of the CAN controller.

For the Frame API, the **Baud Rate** has no relationship with the baud rate property in MAX. You must always configure the **Baud Rate** with the **ncConfigCANNet** VI.

Read Queue Length is the maximum number of unread frames for the read queue of the CAN Network Interface Object. A typical value is 100. For more information, refer to **ncReadNetMult**.

Write Queue Length is the maximum number of frames for the write queue of the CAN Network Interface Object awaiting transmission. A typical value is 10. For more information, refer to **ncWriteNet**.

Standard Comparator is the CAN arbitration ID for the standard (11-bit) frame comparator. For information on how this attribute is used to filter standard frames for the Network Interface, refer to the following **Standard Mask** attribute.

If you intend to open the Network Interface, most applications can set this attribute and the **Standard Mask** to 0 in order to receive all standard frames.

If you intend to use CAN Objects as the sole means of receiving standard frames from the network, you should disable all standard frame reception in the Network Interface by setting this attribute to the special value CFFFFFFF hex. With this setting, the Network Interface is best able to filter out incoming standard frames except those handled by CAN Objects.

Standard Mask is the bit mask used in conjunction with the **Standard Comparator** attribute for filtration of incoming standard (11-bit) CAN frames. For each bit set in the mask, NI-CAN compares the corresponding bit in the **Standard Comparator** to the arbitration ID of the received frame. If the mask/comparator matches, the frame is stored in the Network Interface queue, otherwise it is discarded. Bits in the mask that are clear are treated as don't-cares. For example, hex 00000700 means to compare only the three upper bits of the 11-bit standard ID.

If you intend to open the Network Interface, most applications can set this attribute and the **Standard Comparator** to 0 in order to receive all standard frames.

If you set the **Standard Comparator** to CFFFFFFF hex, this attribute is ignored, because all standard frame reception is disabled for the Network Interface.

Extended Comparator is the CAN arbitration ID for the extended (29-bit) frame comparator. For information on how this attribute is used to filter extended frames for the Network Interface, refer to the following **Extended Mask** attribute.

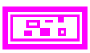
If you intend to open the Network Interface, most applications can set this attribute and the Extended Mask to 0 in order to receive all extended frames.

If you intend to use CAN Objects as the sole means of receiving extended frames from the network, you should disable all extended frame reception in the Network Interface by setting this attribute to the special value CFFFFFFF hex. With this setting, the Network Interface is best able to filter out incoming extended frames except those handled by CAN Objects.

Extended Mask is the bit mask used in conjunction with the **Extended Comparator** attribute for filtration of incoming extended (29-bit) CAN frames. For each bit set in the mask, NI-CAN compares the corresponding bit in the **Extended Comparator** to the arbitration ID of the received frame. If the mask/comparator matches, the frame is stored in the Network Interface queue, otherwise it is discarded. Bits in the mask that are clear are treated as don't-cares. For example, hex 1F000000 means to compare only the five upper bits of the 29-bit extended ID.

If you intend to open the Network Interface, most applications can set this attribute and the **Extended Comparator** to 0 in order to receive all extended frames.

If you set the **Extended Comparator** to CFFFFFFF hex, this attribute is ignored, because all extended frame reception is disabled for the Network Interface.



Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

The Network Interface provides read/write access to all IDs on the network. If you intend to use RTSI features to synchronize the Network Interface with other National Instruments cards, refer to the **ncConfigCANNetRTSI** VI.

If you need to log low-speed (LS) fault indications to the Network Interface read queue, refer to the **ncConfigCANNetLS** VI or **ncConfigCANNetLS-RTSI** VI. These VIs are not required if you simply need to communicate on a LS CAN interface.

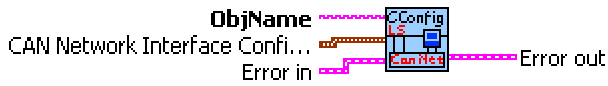
The first NI-CAN VI in your application will normally be **ncConfigCANNet**.

ncConfigCANNetLS.vi

Purpose

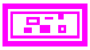
Configure a CAN Network Interface Object with logging of low-speed faults enabled.

Format



Input

ObjName is the name of the CAN Network Interface Object to configure. This name uses the syntax "CAN*x*", where *x* is a decimal number starting at zero that indicates the CAN network interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX).



CAN Network Interface Config - LS provides the core configuration attributes of the CAN Network Interface Object, plus the low-speed logging attribute. This cluster uses the typedef **ncNetAttrLS.ctl**. You can wire in the cluster by first placing it on your front panel from the NI-CAN Controls palette, or you can right-click the VI input and select **Create Constant** or **Create Control**.

Start On Open indicates whether communication starts for the CAN Network Interface Object (and all applicable CAN Objects) immediately upon opening the object with **ncOpenObject**. The default is TRUE, which starts communication when **ncOpenObject** is called. If you set **Start On Open** to FALSE, you can call **ncSetAttribute** after opening the interface, then **ncAction** to start communication. The **ncSetAttribute** VI can be used to set attributes that are not contained within the **ncConfigCANNet** VI.

Baud Rate is the baud rate to use for communication. Basic baud rates are supported, including 100000, 125000, 250000, 500000, and 1000000. If you are familiar with the Bit Timing registers used in CAN controllers, you can use a special hexadecimal baud rate of 0x8000zzyy, where *yy* is the desired value for register 0

(BTR0), and *zz* is the desired value for register 1 (BTR1) of the CAN controller.

For the Frame API, the **Baud Rate** has no relationship with the baud rate property in MAX. You must always configure the **Baud Rate** with the **ncConfigCANNet** VI.

Read Queue Length is the maximum number of unread frames for the read queue of the CAN Network Interface Object. A typical value is 100. For more information, refer to **ncReadNetMult**.

Write Queue Length is the maximum number of frames for the write queue of the CAN Network Interface Object awaiting transmission. A typical value is 10. For more information, refer to **ncWriteNet**.

Standard Comparator is the CAN arbitration ID for the standard (11-bit) frame comparator. For information on how this attribute is used to filter standard frames for the Network Interface, refer to the following **Standard Mask** attribute.

If you intend to open the Network Interface, most applications can set this attribute and the **Standard Mask** to 0 in order to receive all standard frames.

If you intend to use CAN Objects as the sole means of receiving standard frames from the network, you should disable all standard frame reception in the Network Interface by setting this attribute to the special value CFFFFFFF hex. With this setting, the Network Interface is best able to filter out incoming standard frames except those handled by CAN Objects.

Standard Mask is the bit mask used in conjunction with the **Standard Comparator** attribute for filtration of incoming standard (11-bit) CAN frames. For each bit set in the mask, NI-CAN compares the corresponding bit in the **Standard Comparator** to the arbitration ID of the received frame. If the mask/comparator matches, the frame is stored in the Network Interface queue, otherwise it is discarded. Bits in the mask that are clear are treated as don't-cares. For example, hex 00000700 means to compare only the upper three bits of the 11-bit standard ID.

If you intend to open the Network Interface, most applications can set this attribute and the **Standard Comparator** to 0 in order to receive all standard frames.

If you set the **Standard Comparator** to CFFFFFFF hex, this attribute is ignored, because all standard frame reception is disabled for the Network Interface.

Extended Comparator is the CAN arbitration ID for the extended (29-bit) frame comparator. For information on how this attribute is used to filter extended frames for the Network Interface, refer to the following **Extended Mask** attribute.

If you intend to open the Network Interface, most applications can set this attribute and the Extended Mask to 0 in order to receive all extended frames.

If you intend to use CAN Objects as the sole means of receiving extended frames from the network, you should disable all extended frame reception in the Network Interface by setting this attribute to the special value CFFFFFFF hex. With this setting, the Network Interface is best able to filter out incoming extended frames except those handled by CAN Objects.

Extended Mask is the bit mask used in conjunction with the **Extended Comparator** attribute for filtration of incoming extended (29-bit) CAN frames. For each bit set in the mask, NI-CAN compares the corresponding bit in the **Extended Comparator** to the arbitration ID of the received frame. If the mask/comparator matches, the frame is stored in the Network Interface queue, otherwise it is discarded. Bits in the mask that are clear are treated as don't-cares. For example, hex 1F000000 means to compare only the upper five bits of the 29-bit extended ID.

If you intend to open the Network Interface, most applications can set this attribute and the **Extended Comparator** to 0 in order to receive all extended frames.

If you set the **Extended Comparator** to CFFFFFFF hex, this attribute is ignored, because all extended frame reception is disabled for the Network Interface.

Log Comm Warnings specifies whether to log communication warnings (including LS faults) to the Network Interface read queue.

When set to FALSE (default), the Network Interface reports CAN communication warnings (including LS faults) in **Error out** of the read VIs. For more information, refer to **ncReadNetMult**. Using FALSE is equivalent to calling **ncConfigCANNet**.

When set to TRUE, the Network Interface reports CAN communication warnings (including LS faults) by storing a special frame in the read queue. The communication warnings are not reported in **Error out**. For more information on communication warnings and errors, refer to **ncReadNetMult**. The special communication warning frame uses the following format:

Arbitration ID:	Error/warning ID (refer to **ncReadNetMult**)
Timestamp:	Time when error/warning occurred
IsRemote:	2
DataLength:	0
Data:	N/A (ignore)

When calling **ncReadNet** or **ncReadNetMult** to read frames from the Network Interface, you typically use the IsRemote field to differentiate communications warnings from CAN frames. Refer to **ncReadNetMult** for more information.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

This VI is not required if you simply need to communicate on a LS CAN interface. Use **ncConfigCANNet** or **ncConfigCANNetRTSI** instead.

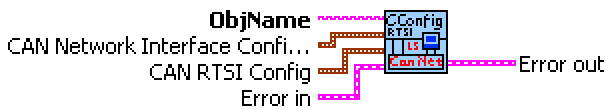
If you intend to use RTSI features to synchronize the Network Interface with other National Instruments cards, refer to the **ncConfigCANNetRTSI-LS VI**.

ncConfigCANNetLS-RTSI.vi

Purpose

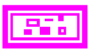
Configure a CAN Network Interface Object with RTSI features, and with logging of low-speed faults enabled.

Format

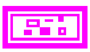


Input

ObjName is the name of the CAN Network Interface Object to configure. This name uses the syntax "CAN*x*", where *x* is a decimal number starting at zero that indicates the CAN network interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX).



CAN Network Interface Config - LS provides the core configuration attributes of the CAN Network Interface Object, plus the low-speed logging attribute. This cluster uses the typedef **ncNetAttrLS.ctl**. You can wire in the cluster by first placing it on your front panel from the NI-CAN Controls palette, or you can right-click the VI input and select **Create Constant** or **Create Control**. For more information, refer to **ncConfigCANNetLS.vi**.



CAN RTSI Config provides RTSI configuration attributes for the CAN Network Interface Object. This cluster uses the typedef **ncCANRtsiAttr.ctl**. You can wire in the cluster by first placing it on your front panel from the NI-CAN Controls palette, or you can right-click the VI input and select **Create Constant** or **Create Control**. For more information, refer to **ncConfigCANNetRTSI.vi**.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

This VI is not required if you simply need to communicate on a LS CAN interface. (Use **ncConfigCANNetRTSI** instead.)

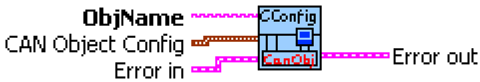
If you are not using RTSI features to synchronize the Network Interface with other National Instruments cards, refer to the **ncConfigCANNetLS** VI.

ncConfigCANNetRTSI.vi

Purpose

Configure a CAN Network Interface Object with RTSI features.

Format



Input

ObjName is the name of the CAN Network Interface Object to configure. This name uses the syntax "CAN*x*", where *x* is a decimal number starting at zero that indicates the CAN network interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX)

CAN Network Interface Config provides the core configuration attributes of the CAN Network Interface Object. This cluster uses the typedef **ncNetAttr.ctl**. You can wire in the cluster by first placing it on your front panel from the NI-CAN Controls palette, or you can right-click the VI input and select **Create Constant** or **Create Control**. For more information, refer to **ncConfigCANNet.vi**.

CAN RTSI Config provides RTSI configuration attributes for the CAN Network Interface Object. This cluster uses the typedef **ncCANRtsiAttr.ctl**. You can wire in the cluster by first placing it on your front panel from the NI-CAN Controls palette, or you can right-click the VI input and select **Create Constant** or **Create Control**.

RTSI Mode specifies the behavior of the Network Interface with respect to RTSI, including whether the RTSI signal is an input or output.

Disable RTSI

Disables RTSI behavior for the Network Interface. All other RTSI attributes are ignored. Using this mode is equivalent to calling **ncConfigCANNet**.

On RTSI Input - Transmit CAN Frame

The Network Interface will transmit a frame from its write queue when the RTSI input pulses. To begin transmission, at least one data frame must be written using **ncWriteNet**. If the write queue becomes empty due to frame transmissions, the last frame will be transmitted on each RTSI pulse until another frame is provided using **ncWriteNet**.

On RTSI Input - Timestamp RTSI event

When the RTSI input pulses, a timestamp is measured and stored in the read queue of the Network Interface. The special RTSI frame uses the following format:

Arbitration ID:	40000001 hex
Timestamp:	Time when RTSI input pulsed
IsRemote:	3
DataLength:	RTSI signal detected (**RTSI Signal**)
Data:	N/A (ignore)

When calling **ncReadNet** or **ncReadNetMult** to read frames from the Network Interface, you typically use the IsRemote field to differentiate RTSI timestamps from CAN frames. Refer to **ncReadNetMult** for more information.

Note When you configure a DAQ card to pulse the RTSI signal periodically, do not exceed 1,000 Hertz (pulse every millisecond). If the RTSI input is pulsed faster than 1kHz on a consistent basis, CAN performance will be adversely affected (for example, lost data frames).

RTSI Output on Receiving CAN Frame

The Network Interface will output the RTSI signal whenever a CAN frame is stored in the read queue.

RTSI Output on Transmitting CAN Frame

The Network Interface will output the RTSI signal whenever a CAN frame is successfully transmitted from the write queue.

RTSI Output on **ncAction** call

The Network Interface will output the RTSI signal whenever the **ncAction** VI is called with **Opcode** Output on RTSI line. This RTSI mode can be used to manually toggle/pulse a RTSI output within your application.

RTSI Signal defines the RTSI signal associated with the **RTSI Mode**. Valid values are 0 to 7, corresponding to RTSI 0 to RTSI 7 on other National Instruments cards.

Note For CAN cards with high-speed (HS) ports only, four (4) RTSI signals are available for input, and four (4) RTSI signals are available for output. Since each RTSI signal is assigned to a Network Interface or CAN Object, this means that at most four NI-CAN objects can use RTSI inputs (or outputs). For example, if you configure five (5) RTSI signals for input, NI-CAN returns an error, regardless of which **RTSI Signal** numbers were used for each.

Note For CAN cards with one or more low-speed (LS) ports, two (2) RTSI signals are available for input, and three (3) RTSI signals are available for output.

Note For PXI-CAN cards, **RTSI Signal** 6 is unavailable.

Note Many NI-DAQ cards use **RTSI Signal** 7 as the 20MHz clock, so this signal number should be avoided for other uses.

RTSI Behavior specifies whether to pulse or toggle a RTSI output. This attribute is ignored when **RTSI Mode** specifies input:

Output RTSI Pulse: Pulse the RTSI output for at least 100 microseconds.

Toggle RTSI Line: If the previous state was high, the output toggles low, then vice-versa.

RTSI Skip specifies the number of RTSI inputs to skip for **RTSI Mode** On RTSI Input - Timestamp RTSI event, and On RTSI Input - Transmit CAN Frame. It is ignored for all other **RTSI Mode** values. For example, if the RTSI input pulses every 1ms, **RTSI Skip** of 9 means that a timestamp will be stored in the read queue every 10ms.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

RTSI is a bus that interconnects National Instruments DAQ, IMAQ, NI-Motion, and CAN boards. This feature allows synchronization of DAQ, IMAQ, NI-Motion, and CAN boards by allowing exchange of timing signals. Using RTSI, a device (board) can control one or more slave devices. PCI/AT boards require a ribbon cable for the connections, but for PXI boards the connections are available on the PXI chassis backplane. Refer to the *NI-CAN User Manual* for more details on the hardware connector.

If you are not using RTSI features to synchronize the Network Interface with other National Instruments cards, refer to the **ncConfigCANNet** VI.

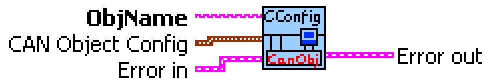
If you need to log low-speed (LS) fault indications to the Network Interface read queue, refer to the **ncConfigCANNetLS-RTSI** VI. This VI is not required if you simply need to communicate on a LS CAN interface.

ncConfigCANObj.vi

Purpose

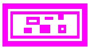
Configure a CAN Object before using it.

Format



Input

ObjName is the name of the CAN Object to configure. This name uses the syntax "CAN*x*::STD*y*" or "CAN*x*::XTD*y*". CAN*x* is the name of the CAN network interface that you used for the preceding **ncConfigCANNet** VI. STD indicates that the CAN Object uses a standard (11-bit) arbitration ID. XTD indicates that the CAN Object uses an extended (29-bit) arbitration ID. The number *y* specifies the actual arbitration ID of the CAN Object. The number *y* is decimal by default, but you can also use hexadecimal by adding "0x" to the beginning of the number. For example, "CAN0::STD25" indicates standard ID 25 decimal on CAN0, and "CAN1::XTD0x0000F652" indicates extended ID F652 hexadecimal on CAN1.



CAN Object Config provides the core configuration attributes of the CAN Object. This cluster uses the typedef **ncObjAttr.ctl**. You can wire in the cluster by first placing it on your front panel from the NI-CAN Controls palette, or you can right-click the VI input and select **Create Constant** or **Create Control**.

Period specifies the rate of periodic behavior in milliseconds.

If you wish to specify the Period in Hertz instead of milliseconds, you can use the special hexadecimal format 800000*xx*, where *xx* is the desired rate in Hertz. For example, 80000020 hex specifies 32 Hz.

The behavior depends on the **Communication Type** as follows:

Transmit Data Periodically,
Transmit Periodic Waveform,
Receive Periodic Using Remote

Period specifies the time between subsequent transmissions, and must be set greater than zero.

Receive Unsolicited,
Transmit by Response Only

Period specifies a watchdog timeout. If a frame is not received at least once every period, a timeout error is returned. Setting **Period** to zero disables the watchdog timer.

Transmit Data by Call,
Receive by Call Using Remote

Period specifies a minimum interval between subsequent transmissions. Even if **ncWriteObj** is called very frequently, frames are transmitted on the network at a rate no more than **Period**. Setting **Period** to zero disables the minimum interval timer.

Read Queue Length is the maximum number of unread frames for the read queue of the CAN Object. For more information, refer to **ncReadObj**.

If **Communication Type** is set to receive data, a typical value is 10. If **Communication Type** is set to transmit data, a typical value is 0.

Write Queue Length is the maximum number of frames for the write queue of the CAN Object awaiting transmission. For more information, refer to **ncWriteObj**.

If **Communication Type** is set to receive data, a typical value is 0. If **Communication Type** is set to transmit data, a typical value is 10.

Receive Changes Only applies only to **Communication Type** selections in which the CAN Object receives data frames (ignored for other types). For those configurations, **Receive Changes Only** specifies whether duplicated data should be placed in the read queue. When set to FALSE (default), all data frames for the CAN Object ID are placed in the read queue. When set to TRUE, data frames are placed into the read queue only if the data bytes differ from the previously received data bytes in the read queue.

This attribute has no effect on the usage of a watchdog timeout for the CAN Object. For example, if this attribute is TRUE and you also specify a watchdog timeout, NI-CAN restarts the watchdog timer every time it receives a data frame for the CAN Object's ID, regardless of whether the data differs from the previous frame.

U32 **Communication Type** specifies the behavior of the CAN Object with respect to its ID, including the direction of data transfer:

Receive Unsolicited

Receive data frames for a specific ID.

This type is useful for receiving a few IDs (1–10) into dedicated read queues. For high performance applications (more IDs, fast frame rates), the Network Interface is recommended to receive all IDs.

Period specifies a watchdog timeout, and **Receive Changes Only** specifies whether to place duplicate data frames into the read queue. **Transmit by Response** is ignored.

Receive Periodic Using Remote

Periodically transmit remote frame for a specific ID in order to receive the associated data frame. Every **Period**, the CAN Object transmits a remote frame, and then places the resulting data frame response in the read queue.

Period specifies the periodic rate, and **Receive Changes Only** specifies whether to place duplicate data frames into the read queue. **Transmit by Response** is ignored.

Receive by Call Using Remote

Transmit remote frame for a specific ID by calling **ncWriteObj**. The CAN Object places the resulting data frame response in the read queue.

Period specifies a minimum interval, and **Receive Changes Only** specifies whether to place duplicate data frames into the read queue. **Transmit by Response** is ignored.

Transmit Data Periodically

Periodically transmit data frame for a specific ID. When the CAN Object transmits the last entry from the write queue, that entry is used every period until you provide a new data frame using **ncWriteObj**. If you keep the write queue filled with unique data, this behavior allows you to ensure that each period transmits a unique data frame.

If the write queue is empty when communication starts, the first periodic transmit does not occur until you provide the first data frame with **ncWriteObj**.

This is the most commonly used CAN Object type. If you are not using remote frames, you can use multiple CAN Objects of this type, and the Network Interface for all other access (event-driven transmit and all receive).

Period specifies the periodic rate, and **Transmit by Response** specifies whether to transmit the previous period's data in response to a remote frame. **Receive Changes Only** is ignored.

Transmit by Response Only

Transmit data frame for a specific ID only in response to a received remote frame. When you call **ncWriteObj**, the data is placed in the write queue, and remains there until a remote frame is received.

Period specifies a watchdog timeout. **Transmit by Response** is assumed as TRUE regardless of the attribute setting. **Receive Changes Only** is ignored.

Transmit Data by Call

Transmit data frame when **ncWriteObj** is called. When **ncWriteObj** is called quickly, data frames are placed in the write queue for back to back transmit.

Period specifies a minimum interval, and **Transmit by Response** specifies whether to transmit the previous data frame in response to a remote frame. **Receive Changes Only** is ignored.

Transmit Periodic Waveform

Transmit a fixed sequence of data frames over and over, one data frame every **Period**.

The following steps describe typical usage of this type.

1. Configure CAN Network Interface Object with **Start On Open** FALSE, then open the Network Interface.
2. Configure the CAN Object as Transmit Periodic Waveform and a nonzero **Write Queue Length**, then open the CAN Object.
3. Call **ncWriteObj** for the CAN Object, once for every entry specified for the **Write Queue Length**.
4. Use **ncAction** to start the Network Interface (not the CAN Object). The CAN Object transmits the first frame in the write queue, then waits the specified period, then transmits the second frame, and so on. After the last frame is transmitted, the CAN Objects waits the specified period, then transmits the first frame again.

If you need to change the waveform contents at runtime, or if you need to transmit very large waveforms (more than 100 frames), we recommend using the Transmit Data Periodically type. Using that type, you can write frames to the Write Queue until full (overflow error), then wait some time for a few frames to transmit, then continue writing new frames.

Period specifies the periodic rate. **Transmit by Response** and **Receive Changes Only** are ignored.

Transmit By Response applies only to **Communication Type** of Transmit Data by Call and Transmit Data Periodically (ignored for other types). For those configurations, **Transmit By Response** specifies whether the CAN Object should automatically respond with the previously transmitted data frame when it receives a remote frame. When set to FALSE (default), the CAN Object transmits data frames only as configured, and ignores all remote frames for its ID. When set to TRUE, the CAN Object responds to incoming remote frames.

Data Length specifies the number of bytes in the data frames for this CAN Object's ID. This number is placed in the Data Length Code (DLC) of all transmitted data frames and remote frames for the CAN Object. This is also the number of data bytes returned from **ncReadObj** when the communication type indicates receive.

Examples of Different Communication Types

The following figures demonstrate how you can use the Communication Type attribute for actual network data transfer. Each figure shows two separate NI-CAN applications that are physically connected across a CAN network.

Figure 8-1 shows a CAN Object that periodically transmits data to another CAN Object. The receiving CAN Object can queue up to five data values.

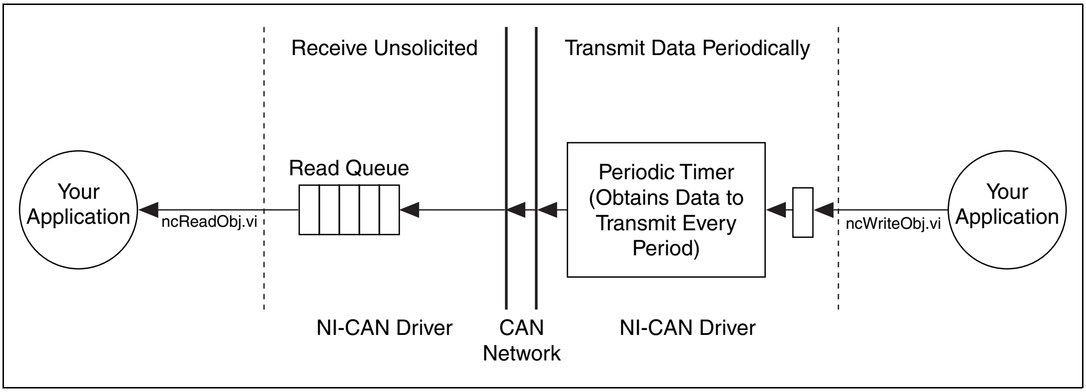


Figure 8-1. Example of Periodic Transmission

Figure 8-2 shows a CAN Object that polls data from another CAN Object. NI-CAN transmits the CAN remote frame when you call **ncWriteObj.vi**.

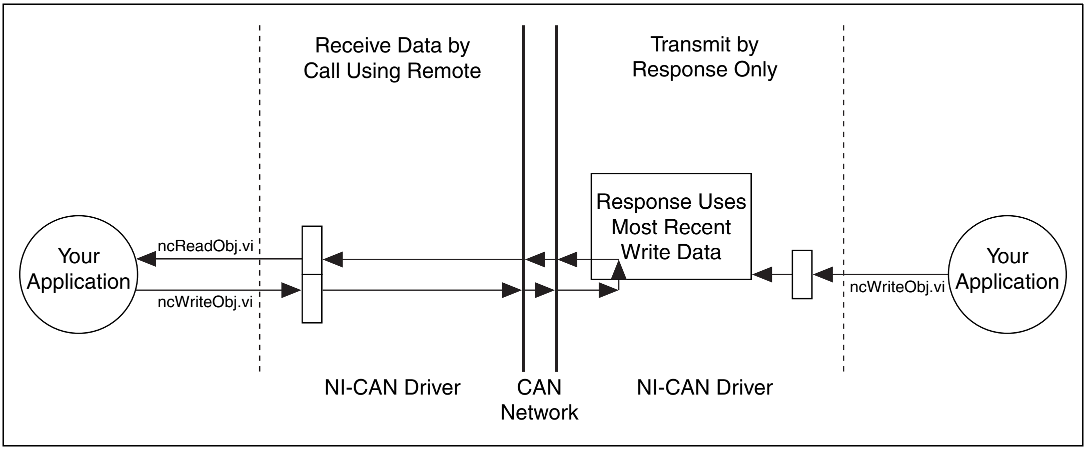


Figure 8-2. Example of Polling Remote Data Using ncWriteObj.vi

Figure 8-3 shows a CAN Object that polls data from another CAN Object. NI-CAN transmits the remote frame periodically and places only changed data into the read queue.

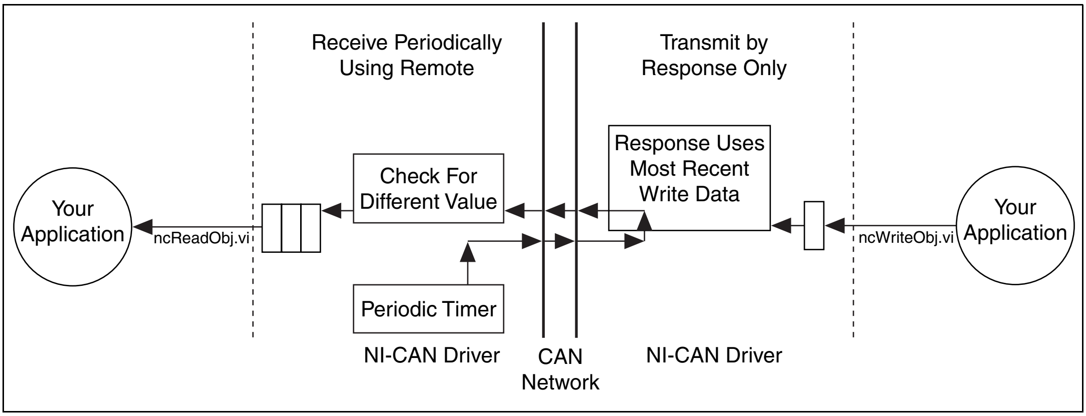


Figure 8-3. Example of Periodic Polling of Remote Data

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

The CAN Object provides read/write access to a specific ID on the network.

You normally call **ncConfigCANNet** before this VI in order to configure the Network Interface attributes, then call **ncConfigCANObj** for each CAN Object desired.

If you intend to use RTSI features to synchronize the CAN Object with other National Instruments cards, refer to the **ncConfigCANObjRTSI** VI.

When a network frame is transmitted on a CAN-based network, it always begins with the arbitration ID. This arbitration ID is primarily used for collision resolution when more than one frame is transmitted simultaneously, but often is also used as a simple mechanism to identify data. The CAN arbitration ID, along with its associated data, is referred to as a CAN Object.

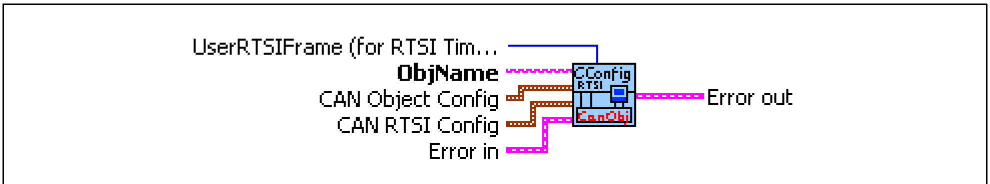
The NI-CAN implementation of CAN provides high-level access to CAN Objects on an individual basis. You can configure each CAN Object for different forms of communication (such as periodic polling, receiving unsolicited CAN data frames, and so on). After you configure a CAN Object and open it for communication, use the **ncReadObj** and **ncWriteObj** VIs to access the data of the CAN Object. The NI-CAN driver performs all other details regarding the object.

ncConfigCANObjRTSI.vi

Purpose

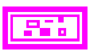
Configure a CAN Object with RTSI features.

Format

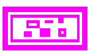


Input

ObjName is the name of the CAN Object to configure. This name uses the syntax "CAN*x*::STD*y*" or "CAN*x*::XTD*y*". CAN*x* is the name of the CAN network interface that you used for the preceding **ncConfigCANNet** VI. STD indicates that the CAN Object uses a standard (11-bit) arbitration ID. XTD indicates that the CAN Object uses an extended (29-bit) arbitration ID. The number *y* specifies the actual arbitration ID of the CAN Object. The number *y* is decimal by default, but you can also use hexadecimal by adding "0x" to the beginning of the number. For example, "CAN0::STD25" indicates standard ID 25 decimal on CAN0, and "CAN1::XTD0x0000F652" indicates extended ID F652 hexadecimal on CAN1.



CAN Object Config provides the core configuration attributes of the CAN Object. This cluster uses the typedef **ncObjAttr.ctl**. You can wire in the cluster by first placing it on your front panel from the NI-CAN Controls palette, or you can right-click the VI input and select **Create Constant** or **Create Control**. For more information, refer to **ncConfigCANObj.vi**.



CAN RTSI Config provides RTSI configuration attributes for the CAN Object. This cluster uses the typedef **ncCANRtsiAttr.ctl**. You can wire in the cluster by first placing it on your front panel from the NI-CAN Controls palette, or you can right-click the VI input and select **Create Constant** or **Create Control**.

RTSI Mode specifies the behavior of the CAN Object with respect to RTSI, including whether the RTSI signal is an input or output.

Disable RTSI

Disables RTSI behavior for the CAN Object. All other RTSI attributes are ignored. Using this mode is equivalent to calling **ncConfigCANObj**.

On RTSI Input - Transmit CAN Frame

The CAN Object will transmit a frame from its write queue when the RTSI input pulses. To begin transmission, at least one data frame must be written using **ncWriteObj**. If the write queue becomes empty due to frame transmissions, the last frame will be transmitted on each RTSI pulse until another frame is provided using **ncWriteObj**.

In order to use this **RTSI Mode**, you must configure the CAN Object's **Communication Type** to either Transmit Data by Call, Transmit Data Periodically, or Transmit Periodic Waveform. The **Period** attribute is ignored when this RTSI mode is selected.

On RTSI Input - Timestamp RTSI event

When the RTSI input pulses, a timestamp is measured and stored in the read queue of the CAN Object. The special RTSI frame uses the following format:

Timestamp:	Time when RTSI input pulsed
Data:	User-defined 4 byte data pattern (refer to **UserRTSIFrame** for details)

Note When you configure a DAQ card to pulse the RTSI signal periodically, do not exceed 1,000 Hertz (pulse every millisecond). If the RTSI input is pulsed faster than 1kHz on a consistent basis, CAN performance will be adversely affected (for example, lost data frames).

RTSI Output on Receiving CAN Frame

The CAN Object will output the RTSI signal whenever a CAN frame is stored in the read queue.

In order to use this **RTSI Mode**, you must configure the CAN Object's **Communication Type** to Receive Unsolicited.

RTSI Output on Transmitting CAN Frame

The CAN Object will output the RTSI signal whenever a CAN frame is successfully transmitted.

In order to use this **RTSI Mode**, you must configure the CAN Object's **Communication Type** to either Transmit Data by Call, Transmit Data Periodically, or Transmit Periodic Waveform.

RTSI Output on **ncAction** call

The CAN Object will output the RTSI signal whenever the **ncAction** VI is called with Opcode Output on RTSI line. This RTSI mode can be used to manually toggle/pulse a RTSI output within your application.

U32

RTSI Signal defines the RTSI signal associated with the **RTSI Mode**. Valid values are 0 to 7, corresponding to RTSI 0 to RTSI 7 on other National Instruments cards.

Note For CAN cards with high-speed (HS) ports only, four (4) RTSI signals are available for input, and four (4) RTSI signals are available for output. Since each RTSI signal is assigned to a Network Interface or CAN Object, this means that at most four NI-CAN objects can use RTSI inputs (or outputs). For example, if you configure five (5) RTSI signals for input, NI-CAN returns an error, regardless of which **RTSI Signa**l numbers were used for each.

Note For CAN cards with one or more low-speed (LS) ports, two (2) RTSI signals are available for input, and three (3) RTSI signals are available for output. The unavailable signals are used for low-speed fault detection.

Note For PXI-CAN cards, **RTSI Signal** 6 is unavailable.

Note Many NI-DAQ cards use **RTSI Signal** 7 as the 20MHz clock, so this signal number should be avoided for other uses.

U32

RTSI Behavior specifies whether to pulse or toggle a RTSI output. This attribute is ignored when **RTSI Mode** specifies input.

Output RTSI Pulse:	Pulse the RTSI output for at least 100 microseconds.
Toggle RTSI Line:	If the previous state was high, the output toggles low, then vice-versa.

RTSI Skip specifies the number of RTSI inputs to skip for **RTSI Mode** On RTSI Input - Timestamp RTSI event, and On RTSI Input - Transmit CAN Frame. It is ignored for all other **RTSI Mode** values. For example, if the RTSI input pulses every 1ms, **RTSI Skip** of 9 means that a timestamp will be stored in the read queue every 10ms.

UserRTSIFrame specifies a 4-byte pattern used to differentiate RTSI timestamps from CAN data frames. It is provided as a U32, and the high byte is stored as byte 0 from **ncReadObj**. For example, `AABBCCDD` hexadecimal is returned as `AA` in byte 0, `BB` in byte 1, and so on.

This attribute is used only for **RTSI Mode** On RTSI Input - Timestamp RTSI event. It is ignored for all other **RTSI Mode** values.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

RTSI is a bus that interconnects National Instruments DAQ, IMAQ, NI-Motion, and CAN boards. This feature allows synchronization of DAQ, IMAQ, NI-Motion, and CAN boards by allowing exchange of timing signals. Using RTSI, a device (board) can control one or more slave devices. PCI/AT boards require a ribbon cable for the connections, but for PXI boards the connections are available on the PXI chassis backplane. Refer to the *NI-CAN User Manual* for more details on the hardware connector.

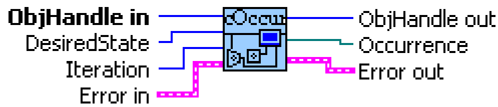
If you are not using RTSI features to synchronize the CAN Object with other National Instruments cards, refer to the **ncConfigCANObj** VI.

ncCreateOccur.vi

Purpose

Create a LabVIEW occurrence for an object.

Format



Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

DesiredState specifies a bit mask of states for which notification is desired. You can use a single state alone, or you can OR them together:

00000001 hex Read Available

At least one frame is available, which you can obtain using an appropriate read VI.

The state is set whenever a frame arrives for the object. The state is cleared when the read queue is empty.

00000002 hex Write Success

All frames provided via write VIs have been successfully transmitted onto the network. Successful transmit means that the frame won arbitration, and was acknowledged by a remote device.

The state is set when the last frame in the write queue is transmitted successfully. The state is cleared when a write VI is called.

For CAN Objects, Write Success does not always mean that transmission has stopped. For example, if a CAN Object is configured for Transmit Data Periodically, Write Success occurs when the write queue has been emptied, but periodic transmit of the last frame continues.

When communication starts the Write Success state is true by default.

00000008 hex Read Multiple

A specified number of frames are available, which you can obtain using either **ncReadNetMult** or **ncReadObjMult**. The number of frames is configured using the **ReadMult Size for Notification** attribute of **ncSetAttr**.

The state is set whenever the specified number of frames are stored in the read queue of the object. The state is cleared when you call the read VI, and less than the specified number of frames exist in the read queue.

Iteration is an optional loop iteration count. If **ncCreateOccur** is called inside a loop, the iteration count of the loop is wired to **Iteration** to ensure that the occurrence is created only once. If **Iteration** is left unwired, the occurrence is created each time **ncCreateOccur** is called, which decreases overall performance.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.

Occurrence returns the LabVIEW occurrence, for use with LabVIEW **Wait on Occurrence** VI.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

ncCreateOccur creates a notification occurrence for the object specified by **ObjHandle**. The NI-CAN driver uses the occurrence callback to communicate state changes to your application.

The **ncCreateOccur** vi is not recommended for use with LabVIEW Real-Time (RT). Due to the internal implementation of occurrences in LabVIEW, their use can have negative effects on real-time performance.

This VI is normally used when you want to allow other code to execute while waiting for NI-CAN states, especially when the other code does not call NI-CAN VIs. If such background execution is not needed, the **ncWait** VI offers better overall performance. The **ncWait** VI cannot be used at the same time as **ncCreateOccur**.

Upon successful return from **ncCreateOccur**, the occurrence is set whenever one of the states specified by **DesiredState** occurs in the object. If **DesiredState** is zero, occurrences are disabled for the object specified by **ObjHandle**.

The **Occurrence** output is normally wired into the LabVIEW **Wait on Occurrence** VI. **Wait on Occurrence** takes the **Occurrence**, and also a timeout and flag indicating whether to ignore a pending state. For more information on **Wait On Occurrence**, refer to the LabVIEW Online Reference.

When **Wait on Occurrence** completes, you should execute code to handle the **DesiredState**. For example:

- If **DesiredState** is **Read Available**, you should call **ncReadNet** or **ncReadObj** to read the available data.
- If **DesiredState** is **Read Multiple**, you should call **ncReadNetMult** or **ncReadObjMult** to read the available data.

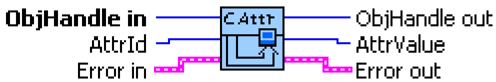
After it has been created, the **Occurrence** will be set each time a **DesiredState** goes from false to true. When you no longer want to wait on the **Occurrence** (for example, when terminating your application), call **ncCreateOccur** with **DesiredState** zero.

ncGetAttr.vi

Purpose

Get the value of an object attribute.

Format



Input

U32 **ObjHandle in** is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

U32 **AttrId** specifies the attribute to get.

Protocol

For NI-CAN, this always returns 1.

For NI-DNET, this always returns 2.

This attribute is available only from the Network Interface, not CAN Objects.

Protocol Version

For NI-CAN, this returns 02000200 hex, which corresponds to version 2.0B of the Bosch CAN specifications. For more information on the encoding of the version, refer to Software Version.

This attribute is available only from the Network Interface, not CAN Objects.

Software Version

Version of the NI-CAN software, with major, minor, update, and beta build numbers encoded in the U32 from high to low bytes. For example, 2.0.1 would be 02000100 hex, and 2.1beta5 would be 02010005 hex.

This attribute is available only from the Network Interface, not CAN Objects.

This attribute is provided for backward compatibility. **ncGetHardwareInfo** VI provides more complete version information.

Object State

Returns the object's current state bit mask. Polling with **ncGetAttr** provides an alternative method of state detection than **ncWait** or **ncCreateOccur**. For more information on the states returned from this attribute, refer to the **DesiredState** input of **ncWait**.

Read Entries Pending

Returns the number of frames available in the read queue. Polling the available frames with this attribute can be used as an alternative to the **ncWait** and **ncCreateOccur** VIs.

Write Entries Pending

Returns the number of frames pending transmission in the write queue. If your intent is to verify that all pending frames have been transmitted successfully, waiting for the Write Success state is preferable to this attribute.

ReadMult Size for Notification

Returns the number of frames used as a threshold for the Read Multiple state. For more information, refer to this attribute in **ncSetAttr**.

Serial Number

Returns the serial number of the card on which the Network Interface or CAN Object is located.

Form Factor

Returns the form factor of the card on which the Network Interface or CAN Object is located.

The returned Form Factor is an enumeration.

0	PCI
1	PXI

2	PCMCIA
3	AT

Transceiver

Returns the CAN transceiver of the port on which the Network Interface or CAN Object is located.

The returned Transceiver is an enumeration.

0	HS
1	LS

This attribute is not supported on the PCMCIA form factor.

Interface Number

Returns the interface number of the port on which the Network Interface or CAN Object is located.

This is the same number that you used in the **ObjName** string of the previous Config and Open VIs.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

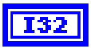
Output

ObjHandle out is the object handle for the next NI-CAN VI.

AttrValue returns the attribute value specified by **AttrId.**

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.



code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

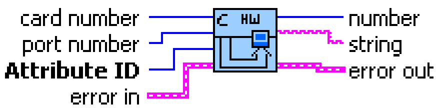
ncGetAttr gets the value of the attribute specified by **AttrId** from the object specified by **ObjHandle**. Within NI-CAN objects, you use attributes to access configuration settings, status, and other information about the object, but not data.

ncGetHardwareInfo.vi

Purpose

Get NI-CAN hardware information.

Format



Input

card number specifies the CAN card number from 1 to **Number of Cards**, where **Number of Cards** is the number of CAN cards in your system. You can determine the number of cards in your system by using this VI with **card number** = 1, **port number** = 1, and **attribute ID** = **Number of Cards**.

port number specifies the CAN port number from 1 to **Number of Ports**, where **Number of Ports** is the number of CAN ports on this CAN card. You can determine the number of ports on this CAN card by using this VI with **port number** = 1, and **attribute ID** = **Number of Ports**.

attribute ID specifies the attribute to get.

Version Major

Returns the major version of the NI-CAN software in the **number** output. Use **card number** 1 and **port number** 1 as inputs.

The major version is the ‘X’ in X.Y.Z.

Version Minor

Returns the minor version of the NI-CAN software in the **number** output. Use **card number** 1 and **port number** 1 as inputs.

The major version is the ‘Y’ in X.Y.Z.

Version Update

Returns the update version of the NI-CAN software in the **number** output. Use **card number** 1 and **port number** 1 as inputs.

The major version is the 'Z' in X.Y.Z.

Version Phase

Returns the phase of the NI-CAN software in the **number** output. Use **card number** 1 and **port number** 1 as inputs.

Phase 1 specifies Alpha, phase 2 specifies Beta, and phase 3 specifies Final release. Unless you are participating in an NI-CAN beta program, you will always see 3.

Version Build

Returns the build of the NI-CAN software in the **number** output. Use **card number** 1 and **port number** 1 as inputs.

With each software development phase, subsequent NI-CAN builds are numbered sequentially. A given Final release version always uses the same build number, so unless you are participating in an NI-CAN beta program, this build number is not relevant.

Version Comment

Returns any special comment on the NI-CAN software in the **string** output. Use **card number** 1 and **port number** 1 as inputs.

This string is normally empty for a Final release. In rare circumstances, an NI-CAN prototype or patch may be released to a specific customer. For these special releases, the version comment describes the special features of the release.

Number of Cards

Returns the number of NI-CAN cards in your system in the **number** output. Use **card number** 1 and **port number** 1 as inputs.

If you are displaying all hardware information, you get this attribute first, then iterate through all CAN cards with a For loop. Inside the card's For loop, you get all card-wide attributes including Number Of Ports, then use another For loop to get port-wide attributes.

Serial Number

Card-wide attribute that returns the serial number of the card in the **number** output. Use the desired **card number**, and **port number** 1 as inputs.

Form Factor

Card-wide attribute that returns the form factor of the card in the **number** output. Use the desired **card number**, and **port number** 1 as inputs.

The returned Form Factor is an enumeration.

0	PCI
1	PXI
2	PCMCIA
3	AT

Number of Ports

Card-wide attribute that returns the number of ports on the card in the **number** output. Use the desired **card number**, and **port number** 1 as inputs.

If you are displaying all hardware information, you get this attribute within the For loop for all cards, then iterate through all CAN ports to get port-wide attributes.

Transceiver

Port-wide attribute that returns the CAN transceiver of the port in the **number** output. Use the desired **card number** and **port number** as inputs.

The returned Transceiver is an enumeration.

0	HS
1	LS

This attribute is not supported on the PCMCIA form factor.

Interface Number

Port-wide attribute that returns the interface number of the port in the **number** output. Use the desired **card number** and **port number** as inputs.

The interface number is assigned to a physical port using the Measurement and Automation Explorer (MAX). The interface number is used as a string in the Frame API (i.e., “CAN0”). The interface number is used for the **interface** input in the Channel API.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

If the attribute is a **number**, the value is returned in this output terminal.

If the attribute is a **string**, the value is returned in this output terminal.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

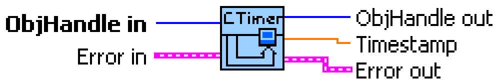
This VI provides information about available CAN cards, but does not require you to open/start sessions. First get **Number of Cards**, then loop for each card. For each card, you can get card-wide attributes (such as **Form Factor**), and you can also get the **Number of Ports**. For each port, you can get port-wide attributes such as the **Transceiver**.

ncGetTimer.vi

Purpose

Get the absolute timestamp attribute.

Format



Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.

Timestamp returns the absolute timestamp value. The value matches the absolute timestamp format used within LabVIEW itself. LabVIEW time is a DBL representing the number of seconds elapsed since 12:00 a.m., Friday, January 1, 1904, Coordinated Universal Time (UTC). You can wire this **Timestamp** to LabVIEW time functions such as **Seconds To**

Date/Time. You can also display the time in a numeric indicator of type DBL by using **Format & Precision** to select **Time & Date** format.

Note If you use **Time & Date** format, LabVIEW limits the **Seconds Precision** to 3, which shows only milliseconds. The NI-CAN timestamp provides microsecond precision. If you need to view microsecond precision, convert to milliseconds, then subtract off the non-fractional part (seconds and milliseconds), then convert to microseconds.

Timestamp = Timestamp * 1000
Microseconds = (TimeStamp - |Timestamp|) * 1000

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

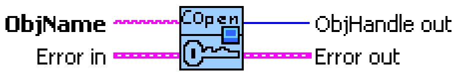
This VI can be used only with the Network Interface, and not with CAN Objects.

ncOpenObject.vi

Purpose

Open an object.

Format



Input

ObjName is the name of the object to open. You must have already wired this name into a previous config VI.

CAN Network Interface Object

This name uses the syntax "CAN*x*", where *x* is a decimal number starting at zero that indicates the CAN network interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX).

CAN Object

This name uses the syntax "CAN*x*::STD*y*" or "CAN*x*::XTD*y*". CAN*x* is the name of the CAN network interface that you used for the preceding **ncConfigCANNet** VI. STD indicates that the CAN Object uses a standard (11-bit) arbitration ID. XTD indicates that the CAN Object uses an extended (29-bit) arbitration ID. The number *y* specifies the actual arbitration ID of the CAN Object. The number *y* is decimal by default, but you can also use hexadecimal by adding "0x" to the beginning of the number. For example, "CAN0::STD25" indicates standard ID 25 decimal on CAN0, and "CAN1::XTD0x0000F652" indicates extended ID F652 hexadecimal on CAN1.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for all subsequent NI-CAN VIs for this object, including the final call to **ncCloseObject**.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

ncOpenObject.vi takes the name of an object to open and returns a handle to that object that you use with subsequent NI-CAN function calls.

The Frame API and Channel API cannot use the same CAN network interface simultaneously. If the CAN network interface is already initialized in the Channel API, this function returns an error.

Although NI-CAN can generally be used by multiple applications simultaneously, it does not allow more than one application to open the same object. For example, if one application opens `CAN0`, and another application attempts to open `CAN0`, the second **ncOpenObject.vi** returns the error `CanErrAlreadyOpen`. It is legal for one application to open `CAN0::STD14` and another application to open `CAN0::STD21`, because the two objects are considered distinct.

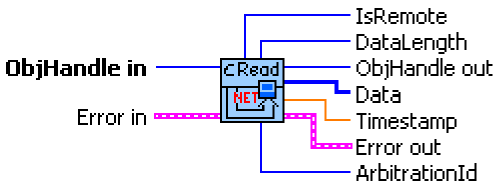
If **ncOpenObject.vi** is successful, a handle to the newly opened object is returned. You use this object handle for all subsequent function calls for the object.

ncReadNet.vi

Purpose

Read single frame from a CAN Network Interface Object.

Format



Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.

ArbitrationId returns the arbitration ID of the received frame. A standard ID (11-bit) is specified by default. An extended ID (29-bit) is specified with the bit mask 20000000 hex ORed with the ID.

The Network Interface receives frames based on the comparators and masks configured in **ncConfigCANNet**.

IsRemote indicates the type of frame:

0 Data frame

CAN data frame from network.

ArbitrationId is the ID of the received data frame. **DataLength** indicates the number of data bytes received into the **Data** array.

2 Communication warning or error

Indicates a communications problem reported by the CAN controller or the low-speed CAN transceiver. This frame type occurs only when you set the **Log Comm Warnings** attribute to TRUE (refer to **ncConfigCANNetLS** for details).

ArbitrationId indicates the type of communication problem:

8000000B hex:	Comm. error: General
4000000B hex:	Comm. warning: General
8001000B hex:	Comm. error: Stuff
4001000B hex:	Comm. warning: Stuff
8002000B hex:	Comm. error: Format
4002000B hex:	Comm. warning: Format
8003000B hex:	Comm. error: No Ack
4003000B hex:	Comm. warning: No Ack
8004000B hex:	Comm. error: Tx 1 Rx 0
4004000B hex:	Comm. warning: Tx 1 Rx 0
8005000B hex:	Comm. error: Tx 0 Rx 1
4005000B hex:	Comm. warning: Tx 0 Rx 1
8006000B hex:	Comm. error: Bad CRC
4006000B hex:	Comm. warning: Bad CRC
0000000B hex:	Comm. errors/warnings cleared
4000000C hex:	LS fault warning
0000000C hex:	LS fault cleared

DataLength and **Data** are not applicable, and should be ignored.

For more information on communication problems, refer to **Description**.

3 RTSI frame

Indicates when a RTSI input pulse occurred relative to incoming CAN frames. This frame type occurs only when you set the **RTSI Mode** attribute to On RTSI Input – Timestamp RTSI event (refer to **ncConfigCANNetRTSI** for details).

ArbitrationId is the special value 40000001 hex. **DataLength** returns the RTSI signal detected. The **Data** array is not applicable, and should be ignored.

DataLength returns the number of data bytes.

Data array returns the data bytes (8 maximum).

Timestamp returns the absolute timestamp when the frame was placed into the read queue. The value matches the absolute timestamp format used within LabVIEW itself. LabVIEW time is a DBL representing the number of seconds elapsed since 12:00 a.m., Friday, January 1, 1904, Coordinated Universal Time (UTC). You can wire this **Timestamp** to LabVIEW time functions such as **Seconds To Date/Time**. You can also display the time in a numeric indicator of type DBL by using **Format & Precision** to select **Time & Date** format.

Note If you use **Time & Date** format, LabVIEW limits the **Seconds Precision** to 3, which shows only milliseconds. The NI-CAN timestamp provides microsecond precision. If you need to view microsecond precision, convert to milliseconds, then subtract off the non-fractional part (seconds and milliseconds), then convert to microseconds.

Timestamp = Timestamp * 1000
Microseconds = (TimeStamp - |Timestamp|) * 1000

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

The **ncReadNet** VI is useful when you need to process one frame at a time, because it returns separate outputs for **ArbitrationId**, **Timestamp**, and so on. In order to read multiple frames at a time, such as for high-bandwidth networks, use the **ncReadNetMult** VI.

Since NI-CAN handles the read queue in the background, this VI does not wait for new frames to arrive. To ensure that a new frame is available before calling **ncReadNet**, first wait for the Read Available state using **ncWait**.

When you call **ncReadNet** for an empty read queue (Read Available state false), the frame from the previous call to **ncReadNet** is returned again, along with the `CanWarnOldData` warning `(status=F, code=3FF62009 hex)`.

When a frame arrives for a full read queue, NI-CAN discards the new frame, and the next call to **ncReadNet** returns the error `CanErrOverflowRead (status=T, code= BFF62028 hex)`. If you detect this overflow, switch to using **ncReadNetMult** to read in a relatively tight loop (few milliseconds each read).

Although the Network Interface allows **Read Queue Length** of zero, this is not recommended, because every new frame will always overwrite the previous frame.

You can use the Network Interface and CAN Objects simultaneously. When a CAN frame arrives from the network, NI-CAN first checks the **ArbitrationId** for an open CAN Object. If no CAN Object applies, NI-CAN checks the comparators and masks of the Network Interface. If the frame passes that filter, NI-CAN places the frame into the read queue of the Network Interface.

Error Active, Error Passive, and Bus Off States

When the CAN communication controller transfers into the error passive state, NI-CAN returns the warning `CanCommWarning (Status=F, code=3ff6200B hex)` from read VIs.

When the transmit error counter of the CAN communication controller increments above 255, the network interface transfers into the bus off state as dictated by the CAN protocol. The network interface stops communication so that you can correct the defect in the network, such as a malfunctioning cable or device. When bus off occurs, NI-CAN returns the error `CanCommError (status=T, code=BFF6200B hex)` from read VIs.

If no CAN devices are connected to the network interface port, and you attempt to transmit a frame, the `CanWarnComm` warning is returned. This warning occurs because the missing acknowledgment bit increments the transmit error counter until the network interface reaches the error passive state, but bus off state is never reached.

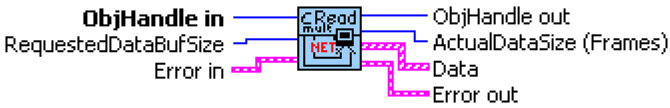
For more information about low-speed communication error handling, refer to the **Log Comm Warnings** attribute in **ncConfigCANNetLS**.

ncReadNetMult.vi

Purpose

Read multiple frames from a CAN Network Interface Object.

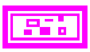
Format



Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

RequestedDataBufSize specifies the maximum number of frames desired. For most applications, this will be the same as the configured **Read Queue Length** in order to empty the read queue with each call to **ncReadNetMult**.



Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.

ActualDataSize (Frames) specifies the number of frames returned in **Data**. This number is less than or equal to **RequestedDataBufSize**.

Data returns an array of clusters. Each cluster in the array uses the typedef **CanFrameTimed.ctl**, with the following elements.

ArbitrationId returns the arbitration ID of the received frame. A standard ID (11-bit) is specified by default. An extended ID (29-bit) is specified with the bit mask 20000000 hex ORed with the ID.

The Network Interface receives frames based on the comparators and masks configured in **ncConfigCANNet**.

IsRemote indicates the type of frame.

0 Data frame

CAN data frame from network.

ArbitrationId is the ID of the received data frame. **DataLength** indicates the number of data bytes received into the **Data** array.

2 Communication warning or error

Indicates a communications problem reported by the CAN controller or the low-speed CAN transceiver. This frame type occurs only when you set the **Log Comm Warnings** attribute to TRUE (refer to **ncConfigCANNetLS** for details).

ArbitrationId indicates the type of communication problem:

8000000B hex:	Comm. error: General
4000000B hex:	Comm. warning: General
8001000B hex:	Comm. error: Stuff
4001000B hex:	Comm. warning: Stuff
8002000B hex:	Comm. error: Format
4002000B hex:	Comm. warning: Format
8003000B hex:	Comm. error: No Ack
4003000B hex:	Comm. warning: No Ack
8004000B hex:	Comm. error: Tx 1 Rx 0
4004000B hex:	Comm. warning: Tx 1 Rx 0
8005000B hex:	Comm. error: Tx 0 Rx 1
4005000B hex:	Comm. warning: Tx 0 Rx 1
8006000B hex:	Comm. error: Bad CRC

4006000B hex:	Comm. warning: Bad CRC
0000000B hex:	Comm. errors/warnings cleared
4000000C hex:	LS fault warning
0000000C hex:	LS fault cleared

DataLength and **Data** are not applicable, and should be ignored.

For more information on communication problems, refer to **Description**.

3 RTSI frame

Indicates when a RTSI input pulse occurred relative to incoming CAN frames. This frame type occurs only when you set the **RTSI Mode** attribute to On RTSI Input – Timestamp RTSI event (refer to **ncConfigCANNetRTSI** for details).

ArbitrationId is the special value 40000001 hex. **DataLength** returns the RTSI signal detected. The **Data** array is not applicable, and should be ignored.

U32 **DataLength** returns the number of data bytes.

[U8] **Data** array returns the data bytes (8 maximum).

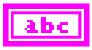
DBL **Timestamp** returns the absolute timestamp when the frame was placed into the read queue. The value matches the absolute timestamp format used within LabVIEW itself. LabVIEW time is a DBL representing the number of seconds elapsed since 12:00 a.m., Friday, January 1, 1904, Coordinated Universal Time (UTC). You can wire this **Timestamp** to LabVIEW time functions such as **Seconds To Date/Time**. You can also display the time in a numeric indicator of type DBL by using **Format & Precision** to select **Time & Date** format.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

TF **status** is TRUE if an error occurred.

I32 **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is

returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.



source identifies the VI where the error occurred.

Description

Since NI-CAN handles the read queue in the background, this VI does not wait for new frames to arrive. To ensure that new frames are available before calling **ncReadNetMult**, first wait for the Read Available state or Read Multiple state using **ncWait**.

When you call **ncReadNetMult** for an empty read queue (Read Available state false), **Error out** returns `success (status=F, code=0)`, and **ActualDataSize (Frames)** returns `0`.

When a frame arrives for a full read queue, NI-CAN discards the new frame, and the next call to **ncReadNet** returns the error `CanErrOverflowRead (status=T, code= BFF62028 hex)`. If you detect this overflow, try to read in a relatively tight loop (few milliseconds each read).

Although the Network Interface allows **Read Queue Length** of zero, this is not recommended, because every new frame will always overwrite the previous frame.

You can use the Network Interface and CAN Objects simultaneously. When a CAN frame arrives from the network, NI-CAN first checks the ArbitrationId for an open CAN Object. If no CAN Object applies, NI-CAN checks the comparators and masks of the Network Interface. If the frame passes that filter, NI-CAN places the frame into the read queue of the Network Interface

Error Active, Error Passive, and Bus Off States

When the CAN communication controller transfers into the error passive state, NI-CAN returns the warning `CanCommWarning (Status=F, code=3ff6200B hex)` from read VIs.

When the transmit error counter of the CAN communication controller increments above 255, the network interface transfers into the bus off state as dictated by the CAN protocol. The network interface stops communication so that you can correct the defect in the network, such as a malfunctioning cable or device. When bus off occurs, NI-CAN returns the error `CanCommError (status=T, code=BFF6200B hex)` from read VIs.

If no CAN devices are connected to the network interface port, and you attempt to transmit a frame, the warning `CanWarnComm` is returned. This warning occurs because the missing acknowledgment bit increments the transmit error counter until the network interface reaches the error passive state, but bus off state is never reached.

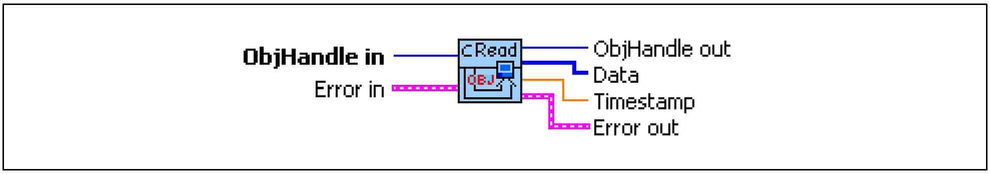
For more information about low-speed communication error handling, refer to the **Log Comm Warnings** attribute in **ncConfigCANNetLS**.

ncReadObj.vi

Purpose

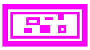
Read single frame from a CAN Object.

Format



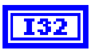
Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.



Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

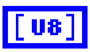


code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.



Data array returns 8 data bytes. The actual number of valid data bytes depends on the CAN Object configuration specified in **ncConfigCANObj**.

If the CAN Object **Communication Type** specifies Transmit, data frames are transmitted, not received, so the **ncReadObj** VI has no effect.

If the CAN Object **Communication Type** specifies Receive, **Data** always contains **Data Length** valid bytes, where Data Length was configured using **ncConfigCANObj**.

Timestamp returns the absolute timestamp when the frame was placed into the read queue. The value matches the absolute timestamp format used within LabVIEW itself. LabVIEW time is a DBL representing the number of seconds elapsed since 12:00 a.m., Friday, January 1, 1904, Coordinated Universal Time (UTC). You can wire this **Timestamp** to LabVIEW time functions such as **Seconds To Date/Time**. You can also display the time in a numeric indicator of type DBL by using **Format & Precision** to select **Time & Date** format.

Note If you use **Time & Date** format, LabVIEW limits the **Seconds Precision** to 3, which shows only milliseconds. The NI-CAN timestamp provides microsecond precision. If you need to view microsecond precision, convert to milliseconds, then subtract off the non-fractional part (seconds and milliseconds), then convert to microseconds.

Timestamp = Timestamp * 1000
Microseconds = (TimeStamp - |Timestamp|) * 1000

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

The **ncReadObj** VI is useful when you need to process one frame at a time. In order to read multiple frames at a time, such as for high-bandwidth networks, use the **ncReadObjMult** VI.

Since NI-CAN handles the read queue in the background, this VI does not wait for a new frame to arrive. To ensure that a new frame is available before calling **ncReadObj**, first wait for the Read Available state using **ncWait**.

When you call **ncReadObj** for an empty read queue (Read Available state false), the frame from the previous call to **ncReadObj** is returned again, along with the warning `CanWarnOldData (status=F, code=3FF62009 hex).`

When a frame arrives for a full read queue, NI-CAN discards the new frame, and the next call to **ncReadObj** returns the error `CanErrOverflowRead (status=T, code= BFF62028 hex).` If you detect this overflow, switch to using **ncReadObjMult** to read in a relatively tight loop (few milliseconds each read).

If you only need to obtain the most recent frame received for the CAN Object, you can set **Read Queue Length** to zero. When the read queue uses a zero length, only the most recent frame is stored, and overflow errors do not occur.

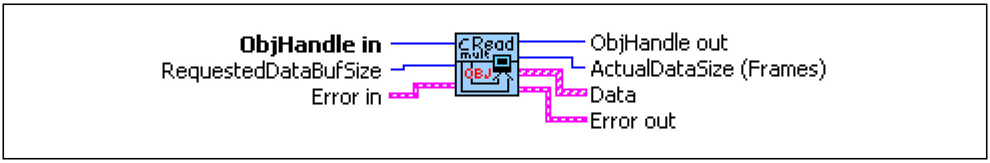
You can use the Network Interface and CAN Objects simultaneously. When a CAN frame arrives from the network, NI-CAN first checks the **ArbitrationId** for an open CAN Object. If no CAN Object applies, NI-CAN checks the comparators and masks of the Network Interface. If the frame passes that filter, NI-CAN places the frame into the read queue of the Network Interface.

ncReadObjMult.vi

Purpose

Read multiple frames from a CAN Object.

Format



Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

RequestedDataBufSize specifies the maximum number of frames desired. For most applications, this will be the same as the configured **Read Queue Length** in order to empty the read queue with each call to **ncReadObjMult**.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.

ActualDataSize (Frames) specifies the number of frames returned in **Data**. This number is less than or equal to **RequestedDataBufSize**.

Data returns an array of clusters. Each cluster in the array uses the typedef **CanDataTimed.ctl** with the following elements:

Data array returns 8 data bytes. The actual number of valid data bytes depends on the CAN Object configuration specified in **ncConfigCANObj**.

If the CAN Object **Communication Type** specifies Transmit, data frames are transmitted, not received, so **Data** always contains zero valid bytes. For this Communication Type, the **ncReadObj** VI has no effect.

If the CAN Object **Communication Type** specifies Receive, **Data** always contains **Data Length** valid bytes, where Data Length was configured using **ncConfigCANObj**.

Timestamp returns the absolute timestamp when the frame was placed into the read queue. The value matches the absolute timestamp format used within LabVIEW itself. LabVIEW time is a DBL representing the number of seconds elapsed since 12:00 a.m., Friday, January 1, 1904, Coordinated Universal Time (UTC). You can wire this **Timestamp** to LabVIEW time functions such as **Seconds To Date/Time**. You can also display the time in a numeric indicator of type DBL by using **Format & Precision** to select **Time & Date** format.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

Since NI-CAN handles the read queue in the background, this VI does not wait for new frames to arrive. To ensure that new frames are available before calling **ncReadObjMult**, first wait for the Read Available state or Read Multiple state using **ncWait**.

When you call **ncReadObjMult** for an empty read queue (Read Available state false), **Error out** returns `success (status=F, code=0)`, and **ActualDataSize (Frames)** returns `0`.

When a frame arrives for a full read queue, NI-CAN discards the new frame, and the next call to **ncReadObjMult** returns the error `CanErrOverflowRead (status=T, code=BFF62028 hex)`. If you detect this overflow, try to read in a relatively tight loop (few milliseconds each read).

If you only need to obtain the most recent frame received for the CAN Object, you can set **Read Queue Length** to zero. When the read queue uses a zero length, only the most recent frame is stored, and overflow errors do not occur.

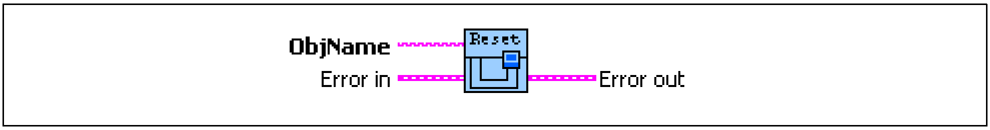
You can use the Network Interface and CAN Objects simultaneously. When a CAN frame arrives from the network, NI-CAN first checks the **ArbitrationId** for an open CAN Object. If no CAN Object applies, NI-CAN checks the comparators and masks of the Network Interface. If the frame passes that filter, NI-CAN places the frame into the read queue of the Network Interface.

ncReset.vi

Purpose

Reset the CAN card.

Format



Input

ObjName is the name of the CAN Network Interface Object to reset. This name uses the same "CAN*x*" syntax as **ncConfigCANNet**, but the reset applies to the entire CAN card.

For example, if a 2-port card contains "CAN0" and "CAN1", calling **ncReset.vi** with **ObjName** "CAN1" resets all hardware/software associated with both "CAN0" and "CAN1".

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

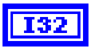
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.



code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

This VI completely resets the CAN card and ensures that all handles for that card are closed.

The **ncReset** VI resets handles for the NI-CAN Frame API only. Do not use this function to debug applications that use the NI-CAN Channel API.

If an NI-CAN application is terminated prior to closing all handles, the `CanErrNotStopped` or `CanErrAlreadyOpen` error might occur when the application is restarted. This often occurs in LabVIEW when the toolbar **Stop** button is used, or when a wiring problem with **ObjHandle** exists. By making this the first NI-CAN VI called in your application (preceding all **ncConfig.vi**), you can avoid problems related to improper termination.

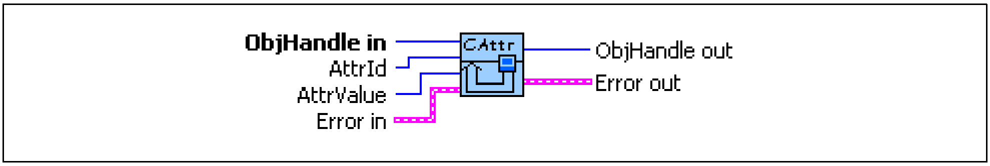
You can only use **ncReset.vi** if you plan to run a single NI-CAN application. If you run more than one NI-CAN application, each with **ncReset**, the second **ncReset** call will close all handles for the first application. You should only use **ncReset.vi** as a temporary measure. After you update your application so that it successfully closes NI-CAN handles on termination, it should no longer be used.

ncSetAttr.vi

Purpose

Set the value of an object attribute.

Format



Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

AttrId specifies the attribute to get.

ReadMult Size for Notification

Sets the number of frames used as a threshold for the Read Multiple state. For more information on the Read Multiple state, refer to **ncWait**.

The default value is one half of **Read Queue Length**.

User RTSI Frame

Sets the user RTSI frame. This attribute is normally configured using the **UserRTSIFrame** input of **ncConfigCANObjRTSI**. This attribute allows that value to be changed while running. For more information, refer to **ncConfigCANObjRTSI**.

This attribute is available only for CAN Objects, not the Network Interface.

AttrValue provides the attribute value for **AttrId**.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

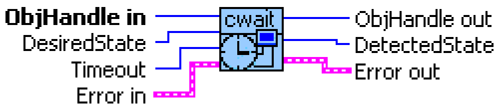
ncSetAttr.vi sets the value of the attribute specified by **AttrId** in the object specified by **ObjHandle in**.

ncWait.vi

Purpose

Wait for one or more states to occur in an object.

Format



Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

DesiredState specifies a bit mask of states for which notification is desired. You can use a single state alone, or you can OR them together:

00000001 hex Read Available

At least one frame is available, which you can obtain using an appropriate read VI.

The state is set whenever a frame arrives for the object. The state is cleared when the read queue is empty.

00000002 hex Write Success

All frames provided via write VIs have been successfully transmitted onto the network. Successful transmit means that the frame won arbitration, and was acknowledged by a remote device.

The state is set when the last frame in the write queue is transmitted successfully. The state is cleared when a write VI is called.

For CAN Objects, Write Success does not always mean that transmission has stopped. For example, if a CAN Object is configured for Transmit Data Periodically, Write Success occurs when the write queue has been emptied, but periodic transmit of the last frame continues. When communication starts, the Write Success state is true by default.

00000008 hex Read Multiple

A specified number of frames are available, which you can obtain using either **ncReadNetMult** or **ncReadObjMult**. The number of frames is configured using the **ReadMult Size for Notification** attribute of **ncSetAttr**.

The state is set whenever the specified number of frames are stored in the read queue of the object. The state is cleared when you call the read VI, and less than the specified number of frames exist in the read queue.

Timeout specifies the maximum number of milliseconds to wait for one of the states in **DesiredState**. If the **Timeout** expires before a state occurs, the error `CanErrFunctionTimeout` is returned in **Error out** `(status=T, code= BFF62001 hex)`.

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

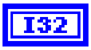
source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.



code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

Use **ncWait.vi** to wait for one or more states to occur in the object specified by `ObjHandle`.

While waiting for the desired states, **ncWait.vi** suspends execution of the current LabVIEW thread. VIs assigned to other threads can still execute. The thread of a VI can be changed in the **Priority** control in the **Execution** category of VI properties.

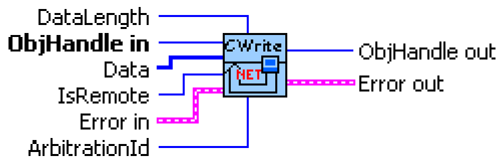
If you want to execute code in the same LabVIEW thread while waiting for NI-CAN states, refer to **ncCreateOccur.vi**. It requires more execution time than **ncWait**, but **ncCreateOccur** allows other code in the thread to execute.

ncWriteNet.vi

Purpose

Write the data value of an object.

Format



Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

ArbitrationId specifies the arbitration ID of the frame to transmit. A standard ID (11-bit) is specified by default. In order to specify an extended ID (29-bit), OR in the bit mask 20000000 hex.

IsRemote indicates the type of frame.

0 Data frame

Transmit CAN data frame.

ArbitrationId is the ID of the data frame to transmit. **DataLength** indicates the number of data bytes in the **Data** array.

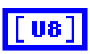
1 Remote frame

Transmit CAN remote frame.

ArbitrationId is the ID of the remote frame to transmit. **DataLength** is encoded in the remote frame Data Length Code, but the **Data** array is not used.

Data provides an array of data bytes to write.

DataLength specifies the number of data bytes.



Data array specifies the data bytes (8 maximum).

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

You use **ncWriteNet** to place a frame into the Network Interface write queue. Since NI-CAN handles the write queue in the background, this VI does not wait for the frame to be transmitted on the network.

If your goal is to transmit a set of frames as quickly as possible, simply call **ncWriteNet** once per frame, without using **ncWait** after each write. This technique makes good use of the write queue to optimize frame transmission.

Once you have written frames, if you need to wait for the final **ncWriteNet** to be transmitted successfully, use **ncWait** with the Write Success state. The Write Success state sets when all frames of the write queue have been successfully transmitted. The Write Success state clears whenever you call **ncWriteNet**.

Sporadic, recoverable errors on the network are handled automatically by the CAN protocol. As such, after **ncWriteNet** returns successfully, NI-CAN eventually transmits the frame on the network unless there is a serious network problem. Network problems such as missing or malfunctioning devices are often reported as the warning `CanWarmComm (status=F, code=3FF6200B hex).`

If the write queue is full, a call to **ncWriteNet** returns the error `CanErrOverflowWrite (status=T, code= BFF62008 hex)`. In many cases, this error is recoverable, so you should not exit your application when it occurs. For example, if you want to transmit thousands of frames in succession (i.e., downloading code), your application can check for the error `CanErrOverflowWrite`, and when detected, simply wait a few milliseconds for some of the frames to transmit, then call **ncWriteNet** again. If the second call to **ncWriteNet** returns an error, that can be treated as an unrecoverable error (no other device is ACKing the frames).

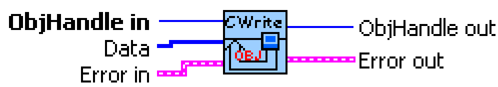
Although the Network Interface allows **Write Queue Length** of zero, this is not recommended, because every new frame will always overwrite the previous frame.

ncWriteObj.vi

Purpose

Write a single frame to a CAN Object.

Format



Input

ObjHandle in is the object handle from the previous NI-CAN VI. The handle originates from the **ncOpenObject** VI.

Data array specifies the data bytes (8 maximum).

Error in describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **Error in** cluster in **Error out**.

status is TRUE if an error occurred. If **status** is TRUE, the VI does not perform any operations.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ObjHandle out is the object handle for the next NI-CAN VI.

Error out describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

You use **ncWriteObj** to place a frame into the CAN Object write queue. Since NI-CAN handles the write queue in the background, this VI does not wait for the frame to be transmitted on the network.

Once you have written frames, if you need to wait for the final **ncWriteObj** to be transmitted successfully, use **ncWait** with the Write Success state. The Write Success state sets when all frames of the write queue have been successfully transmitted. The Write Success state clears whenever you call **ncWriteObj**.

The Write Success state does not necessarily mean that all transmission has stopped for the CAN Object. For example, when the CAN Object **Communication Type** is Transmit Data Periodically, the Write Success state sets when the final frame in the write queue is transmitted, but the previous frame will be transmitted again once the **Period** expires.

Sporadic, recoverable errors on the network are handled automatically by the CAN protocol. As such, after **ncWriteObj** returns successfully, NI-CAN eventually transmits the frame on the network unless there is a serious network problem. Network problems such as missing or malfunctioning devices are often reported as the warning `CanWarmComm (status=F, code=3FF6200B hex)`.

If the write queue is full, a call to **ncWriteObj** returns the error `CanErrOverflowWrite (status=T, code= BFF62008 hex)`. In many cases, this error is recoverable, so you should not exit your application when it occurs. For example, if you want to transmit thousands of frames in succession (i.e., large waveform transmitted periodically), your application can check for the error `CanErrOverflowWrite`, and when detected, simply wait a few milliseconds for some of the frames to transmit, then call **ncWriteObj** again. If the second call to **ncWriteObj** returns an error, that can be treated as an unrecoverable error (for example, no other device is ACKing the frames).

If you need to write a sequence of frames to the CAN Object, and ensure that each frame is transmitted, configure the **Write Queue Length** of the CAN Object to greater than zero. If you only need to transmit the most recent frame provided with **ncWriteObj,** you can set the **Write Queue Length** to zero.

If the CAN Object **Communication Type** specifies Receive behavior, the **ncWriteObj** VI can be used to transmit a remote frame. When using **ncWriteObj** to transmit a remote frame, the **Data** input can be left unwired.

9 Frame API for C

This chapter lists the NI-CAN functions and describes the format, purpose and parameters.

Unless otherwise stated, each NI-CAN function suspends execution of the calling thread until it completes. The functions in this chapter are listed alphabetically.

Section Headings

The following are section headings found in the Frame API for C functions.

Purpose

Each function description includes a brief statement of the purpose of the function.

Format

The format section describes the format of each function for the C programming language.

Input and Output

The input and output parameters for each function are listed.

Description

The description section gives details about the purpose and effect of each function.

CAN Network Interface Object

The CAN Network Interface Object section gives details about using the function with the CAN Network Interface Object.

CAN Object

The CAN Object section gives details about using the function with the CAN Object.

Data Types

The following data types are used with functions of the NI-CAN Frame API for C.

Table 9-1. NI-CAN Frame API for C, Data Types

Data Type	Purpose
`NCTYPE_INT8`	8-bit signed integer
`NCTYPE_INT16`	16-bit signed integer
`NCTYPE_INT32`	32-bit signed integer
`NCTYPE_UINT8`	8-bit unsigned integer
`NCTYPE_UINT16`	16-bit unsigned integer
`NCTYPE_UINT32`	32-bit unsigned integer
`NCTYPE_BOOL`	Boolean value. Constants `NC_TRUE (1)` and `NC_FALSE (0)` are used for comparisons.
`NCTYPE_STRING`	ASCII string represented as an array of characters terminated by null character (`'\0'`).
`NCTYPE_`*`type`*`_P`	Pointer to a variable of type *`type`*.
`NCTYPE_ANY_P`	Pointer to a variable of any type, used in cases where actual data type can vary depending on the object in use.
`NCTYPE_OBJH`	32-bit unsigned integer used to reference an open object in the Frame API.
`NCTYPE_ATTRID`	Attribute identifier. Uses constants with prefix `NC_ATTR_`.
`NCTYPE_OPCODE`	Operation code for `ncAction` function. Uses constants with prefix `NC_OP_`.
`NCTYPE_STATE`	Object states, encoded as a 32-bit mask, one bit for each state. Refer to `ncWaitForState` for more information.

Table 9-1. NI-CAN Frame API for C, Data Types (Continued)

Data Type	Purpose
NCTYPE_STATUS	Status returned from NI-CAN functions. Refer to ncStatusToString for more information.
NCTYPE_CAN_ARBID	CAN arbitration ID. The 30h bit is accessed using bitmask NC_FL_CAN_ARBID_XTD (2000000 hex). If this bit is clear, the CAN arbitration ID is standard (11-bit). If this bit is set, the CAN arbitration ID is extended (29-bit). Special constant NC_CAN_ARBID_NONE (CFFFFFFF hex) indicates no CAN arbitration ID, and is used to set the comparator attribute of the CAN Network Interface. Refer to ncConfig for more information.

List of Functions

The following table contains an alphabetical list of the NI-CAN Frame API for C functions.

Table 9-2. NI-CAN Frame API for C Functions

Function	Purpose
ncAction	Perform an action on an object.
ncCloseObject	Close an object.
ncConfig	Configure an object before using it.
ncCreateNotification	Create a notification callback for an object.
ncGetAttribute	Get the value of an object attribute.
ncGetHardwareInfo	Get NI-CAN hardware information.
ncOpenObject	Open an object.
ncRead	Read the data value of an object.
ncReadMult	Read multiple data values from the queue of an object.
ncReset	Reset CAN interface.
ncSetAttribute	Set the value of an object attribute.
ncStatusToString	Convert status code into a descriptive string.

Table 9-2. NI-CAN Frame API for C Functions (Continued)

Function	Purpose
`ncWaitForState`	Wait for one or more states to occur in an object.
`ncWrite`	Write the data value of an object.

ncAction

Purpose

Perform an action on an object.

Format

```
NCTYPE_STATUS    ncAction(
                         NCTYPE_OBJH ObjHandle,
                         NCTYPE_OPCODE Opcode,
                         NCTYPE_UINT32 Param)
```

Input

`ObjHandle`	Object handle from `ncOpenObject`.
`Opcode`	Operation code indicating which action to perform.
`Param`	Parameter whose meaning is defined by `Opcode`.

Output

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

`ncAction` is a general purpose function you can use to perform an action on the object specified by `ObjHandle`. Its normal use is to start and stop network communication on a CAN Network Interface Object.

For the most frequently used and/or complex actions, NI-CAN provides functions such as `ncOpenObject` and `ncRead`. `ncAction` provides an easy, general purpose way to perform actions that are used less frequently or are relatively simple.

CAN Network Interface Object

NI-CAN propagates all actions on the CAN Network Interface Object up to all open CAN Objects. Table 9-3 describes the actions supported by the CAN Network Interface Object.

Table 9-3. Actions Supported by the CAN Network Interface Object

Opcode	Param	Description
`NC_OP_START` (80000001 hex)	N/A (ignored)	Transitions network interface from stopped (idle) state to started (running) state. If network interface is already started, this operation has no effect. When a network interface is started, it is communicating on the network. When you execute `NC_OP_START` on a stopped CAN Network Interface Object, NI-CAN propagates it upward to all open higher-level CAN Objects. Thus, you can use it to start all higher-level network communication simultaneously.
`NC_OP_STOP` (80000002 hex)	N/A (ignored)	Transitions network interface from started state to stopped state. If network interface is already stopped, this operation has no effect. When a network interface is stopped, it is not communicating on the network. When you execute `NC_OP_STOP` on a running CAN Network Interface Object, NI-CAN propagates it upward to all open higher-level CAN Objects.
`NC_OP_RESET` (80000003 hex)	N/A (ignored)	Resets network interface. Stops network interface, then resets the CAN controller to clear the CAN error counters (clear error passive state). Resetting includes clearing all entries from read and write queues. `NC_OP_RESET` is propagated up to all open higher-level CAN Objects.
`NC_OP_RTSI_OUT` (80000004 hex)	N/A (ignored)	Output a pulse or toggle on the RTSI line depending upon the `NC_ATTR_RTSI_SIG_BEHAV`

CAN Object

All actions performed on a CAN Object affect that CAN Object alone, and do not affect other CAN Objects or communication as a whole.

Table 9-4 describes the actions supported by the CAN Object.

Table 9-4. Actions Supported by the CAN Object

Opcode	Param	Description
NC_OP_RTSI_OUT (80000004 hex)	N/A (ignored)	Output a pulse or toggle on the RTSI line depending upon the NC_ATTR_RTSI_SIG_BEHAV

ncCloseObject

Purpose

Close an object.

Format

```
NCTYPE_STATUS      ncCloseObject(NCTYPE_OBJH ObjHandle)
```

Input

`ObjHandle` Object handle.

Output

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

`ncCloseObject` closes an object when it no longer needs to be in use, such as when the application is about to exit. When an object is closed, NI-CAN stops all pending operations and clears RTSI configuration for the object, and you can no longer use the `ObjHandle` in your application.

CAN Network Interface Object

`ObjHandle` refers to an open CAN Network Interface Object.

CAN Object

`ObjHandle` refers to an open CAN Object.

ncConfig

Purpose

Configure an object before using it.

Format

C

```
NCTYPE_STATUS    ncConfig(
                        NCTYPE_STRING ObjName,
                        NCTYPE_UINT32 NumAttrs,
                        NCTYPE_ATTRID_P AttrIdList,
                        NCTYPE_UINT32_P AttrValueList)
```

Input

`ObjName`	ASCII name of the object to configure.
`NumAttrs`	Number of configuration attributes.
`AttrIdList`	List of configuration attribute identifiers.
`AttrValueList`	List of configuration attribute values.

Output

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

`ncConfig` initializes the configuration attributes of an object before opening it. The first NI-CAN function in your application will normally be `ncConfig` of the CAN Network Interface Object.

`NumAttr` indicates the number of configuration attributes in `AttrIdList` and `AttrValueList`. `AttrIdList` is an array of attribute IDs, and `AttrValueList` is an array of values. The host data type for each value in `AttrValueList` is `NCTYPE_UINT32`, which all configuration attributes can use.

The Frame API and Channel API cannot use the same CAN network interface simultaneously. If the CAN network interface is already initialized in the Channel API, this function returns an error.

The following sections describe how to use `ncConfig` with the Network Interface and CAN Object. The description for each object specifies the syntax for `ObjName`, plus a description of the commonly used attributes for `AttrIdList`.

CAN Network Interface Object

`ObjName` is the name of the CAN Network Interface Object to configure. This string uses the syntax "CAN*x*", where *x* is a decimal number starting at zero that indicates the CAN network interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX).

The following attribute IDs are commonly used for Network Interface configuration.

`NC_ATTR_START_ON_OPEN` (Start On Open)

`Start On Open` indicates whether communication starts for the CAN Network Interface Object (and all applicable CAN Objects) immediately upon opening the object with `ncOpenObject`. The default is NC_TRUE (1), which starts communication when `ncOpenObject` is called. If you set `Start On Open` to NC_FALSE (0), you can call `ncSetAttribute` after opening the interface, then `ncAction` to start communication. The `ncSetAttribute` function can be used to set attributes that are not contained within the `ncConfig` function.

`NC_ATTR_BAUD_RATE` (Baud Rate)

`Baud Rate` is the baud rate to use for communication. Common baud rates are supported, including 100000, 125000, 250000, 500000, and 1000000. If you are familiar with the Bit Timing registers used in CAN controllers, you can use a special hexadecimal baud rate of 0x8000*zzyy*, where *yy* is the desired value for register 0 (BTR0), and *zz* is the desired value for register 1 (BTR1) of the CAN controller.

`NC_ATTR_READ_Q_LEN` (Read Queue Length)

`Read Queue Length` is the maximum number of unread frames for the read queue of the CAN Network Interface Object. A typical value is 100. For more information, refer to `ncRead`.

`NC_ATTR_WRITE_Q_LEN` (Write Queue Length)

`Write Queue Length` is the maximum number of frames for the write queue of the CAN Network Interface Object awaiting transmission. A typical value is 10. For more information, refer to `ncWrite`.

`NC_ATTR_CAN_COMP_STD` (Standard Comparator)

`Standard Comparator` is the CAN arbitration ID for the standard (11-bit) frame comparator. For information on how this attribute is used to filter standard frames for the Network Interface, refer to the following NC_ATTR_CAN_MASK_STD (Standard Mask) attribute.

If you intend to open the Network Interface, most applications can set this attribute and the `Standard Mask` to 0 in order to receive all standard frames.

If you intend to use CAN Objects as the sole means of receiving standard frames from the network, you should disable all standard frame reception in the Network Interface by setting this attribute to the special value CFFFFFFF hex. With this setting, the Network Interface is best able to filter out incoming standard frames except those handled by CAN Objects.

`NC_ATTR_CAN_MASK_STD` (Standard Mask)

`Standard Mask` is the bit mask used in conjunction with the `Standard Comparator` attribute for filtration of incoming standard (11-bit) CAN frames. For each bit set in the mask, NI-CAN compares the corresponding bit in the `Standard Comparator` to the arbitration ID of the received frame. If the mask/comparator matches, the frame is stored in the Network Interface queue, otherwise it is discarded. Bits in the mask that are clear are treated as don't-cares. For example, hex 00000700 means to compare only the upper 3 bits of the 11-bit standard ID.

If you intend to open the Network Interface, most applications can set this attribute and the `Standard Comparator` to 0 in order to receive all standard frames.

If you set the `Standard Comparator` to CFFFFFFF hex, this attribute is ignored, because all standard frame reception is disabled for the Network Interface.

`NC_ATTR_CAN_COMP_XTD` (Extended Comparator)

`Extended Comparator` is the CAN arbitration ID for the extended (29-bit) frame comparator. For information on how this attribute is used to filter extended frames for the Network Interface, refer to the following NC_ATTR_CAN_MASK_XTD (Extended Mask) attribute.

If you intend to open the Network Interface, most applications can set this attribute and the `Extended Mask` to 0 in order to receive all extended frames.

If you intend to use CAN Objects as the sole means of receiving extended frames from the network, you should disable all extended frame reception in the Network Interface by setting this attribute to the special value CFFFFFFF hex. With this setting, the Network Interface is best able to filter out incoming extended frames except those handled by CAN Objects.

`NC_ATTR_CAN_MASK_XTD` (Extended Mask)

`Extended Mask` is the bit mask used in conjunction with the `Extended Comparator` attribute for filtration of incoming extended (29-bit) CAN frames. For each bit set in the mask, NI-CAN compares the corresponding bit in the `Extended Comparator` to the arbitration ID of the received frame. If the mask/comparator matches, the frame is stored in the Network Interface queue, otherwise it is discarded. Bits in the mask that are clear are treated as don't-cares. For example, hex 1F000000 means to compare only the upper 5 bits of the 29-bit extended ID.

If you intend to open the Network Interface, most applications can set this attribute and the `Extended Comparator` to 0 in order to receive all extended frames.

If you set the `Extended Comparator` to CFFFFFFF hex, this attribute is ignored, because all extended frame reception is disabled for the Network Interface.

`NC_ATTR_NOTIFY_MULT_LEN` (ReadMult Size for Notification)

Sets the number of frames used as a threshold for the Read Multiple state. For more information on the Read Multiple state, refer to `ncWaitForState`.

The default value is one half of `Read Queue Length`.

The following attribute ID is used to enable logging of low-speed (LS) faults.

`NC_ATTR_LOG_COMM_ERRS` (Log Comm Warnings)

`Log Comm Warnings` specifies whether to log communication warnings (including LS faults) to the Network Interface read queue.

When set to NC_FALSE (default), the Network Interface reports CAN communication warnings (including LS faults) in the return status of read functions. For more information, refer to `ncReadMult`.

When set to NC_TRUE, the Network Interface reports CAN communication warnings (including LS faults) by storing a special frame in the read queue. The communication warnings are not reported in the return status. For more information on communication warnings and errors, refer to `ncReadMult`. The special communication warning frame uses the following format:

Arbitration ID:	Error/warning ID (refer to `ncReadMult`)
Timestamp:	Time when error/warning occurred
IsRemote:	2
DataLength:	0
Data:	N/A (ignore)

When calling `ncRead` or `ncReadMult` to read frames from the Network Interface, you typically use the `IsRemote` field to differentiate communications warnings from CAN frames. Refer to `ncReadMult` for more information.

RTSI is a bus that interconnects National Instruments DAQ, IMAQ, NI-Motion, and CAN boards. This feature allows synchronization of DAQ, IMAQ, NI-Motion, and CAN boards by allowing exchange of timing signals. Using RTSI, a device (board) can control one or more slave devices. PCI/AT boards require a ribbon cable for the connections, but for PXI boards the connections are available on the PXI chassis backplane. Refer to the *NI-CAN User Manual* for more details on the hardware connector.

The following attribute IDs are used to enable RTSI synchronization between two or more National Instruments cards:

`NC_ATTR_RTSI_MODE` (RTSI Mode)

`RTSI Mode` specifies the behavior of the Network Interface with respect to RTSI, including whether the RTSI signal is an input or output:

`NC_RTSI_NONE`

Disables RTSI behavior for the Network Interface (default). All other RTSI attributes are ignored.

`NC_RTSI_TX_ON_IN`

The Network Interface will transmit a frame from its write queue when the RTSI input pulses. To begin transmission, at least one data frame must be written using `ncWrite`. If the write queue becomes empty due to frame transmissions, the last frame will be retransmitted on each RTSI pulse until another frame is provided using `ncWrite`.

`NC_RTSI_TIME_ON_IN`

When the RTSI input pulses, a timestamp is measured and stored in the read queue of the Network Interface. The special RTSI frame uses the following format:

Arbitration ID:	40000001 hex
Timestamp:	Time when RTSI input pulsed
IsRemote:	3 (NC_FRMTYPE_RTSI)
DataLength:	RTSI signal detected (`RTSI Signal`)
Data:	N/A (ignore)

When calling `ncRead` or `ncReadMult` to read frames from the Network Interface, use the `IsRemote` field to differentiate RTSI timestamps from CAN frames. Refer to `ncReadMult` for more information.

Note When you configure a DAQ card to pulse the RTSI signal periodically, do not exceed 1,000 Hertz (pulse every millisecond). If the RTSI input is pulsed faster than 1 kHz on a consistent basis, CAN performance will be adversely affected (i.e. lost data frames).

`NC_RTSI_OUT_ON_RX`

The Network Interface will output the RTSI signal whenever a CAN frame is stored in the read queue.

`NC_RTSI_OUT_ON_TX`

The Network Interface will output the RTSI signal whenever a CAN frame is successfully transmitted from the write queue.

`NC_RTSI_OUT_ACTION_ONLY`

The Network Interface will output the RTSI signal whenever the `ncAction` function is called with `Opcode` NC_OP_RTSI_OUT. This RTSI mode can be used to manually toggle/pulse a RTSI output within your application.

`NC_ATTR_RTSI_SIGNAL` (RTSI Signal)

`RTSI Signal` defines the RTSI signal associated with the `RTSI Mode`. Valid values are 0 to 7, corresponding to RTSI 0 to RTSI 7 on other National Instruments cards.

Note For CAN cards with high-speed (HS) ports only, four (4) RTSI signals are available for input, and four (4) RTSI signals are available for output. Since each RTSI signal is assigned to a Network Interface or CAN Object, this means that at most four NI-CAN objects can use RTSI inputs (or outputs). For example, if you configure five (5) RTSI signals for input, NI-CAN returns an error, regardless of which `RTSI Signal` numbers were used for each.

Note For CAN cards with one or more low-speed (LS) ports, two (2) RTSI signals are available for input, and three (3) RTSI signals are available for output.

Note For PXI-CAN cards, RTSI Signal 6 is unavailable.

Note Many NI-DAQ cards use RTSI Signal 7 as the 20 MHz clock, so this signal number should be avoided for other uses.

`NC_ATTR_RTSI_SIG_BEHAV` (RTSI Behavior)

`RTSI Behavior` specifies whether to pulse or toggle a RTSI output. This attribute is ignored when `RTSI Mode` specifies input:

`RTSI_SIG_PULSE`
: Pulse the RTSI output for at least 100 microseconds.

`RTSI_SIG_TOGGLE`
: If the previous state was high, the output toggles low, then vice-versa.

`NC_ATTR_RTSI_SKIP` (RTSI Skip)

`RTSI Skip` specifies the number of RTSI inputs to skip for RTSI input modes. It is ignored for RTSI output modes. For example, for `RTSI Mode` `NC_RTSI_TIME_ON_IN`, if the RTSI input pulses every 1 ms, `RTSI Skip` of 9 means that a timestamp will be stored in the read queue every 10ms.

CAN Object

`ObjName` is the name of the CAN Object to configure. This string uses the syntax "CAN*x*::STD*y*" or "CAN*x*::XTD*y*". CAN*x* is the name of the CAN network interface that you used for the preceding `ncConfig` function. STD indicates that the CAN Object uses a standard (11-bit) arbitration ID. XTD indicates that the CAN Object uses an extended (29-bit) arbitration ID. The number *y* specifies the actual arbitration ID of the CAN Object. The number *y* is decimal by default, but you can also use hexadecimal by adding "0x" to the beginning of the number. For example, "CAN0::STD25" indicates standard ID 25 decimal on CAN0, and "CAN1::XTD0x0000F652" indicates extended ID F652 hexadecimal on CAN1.

In order to configure one or more CAN objects, you must configure the CAN Network Interface Object first.

The following attribute IDs are commonly used for CAN Object configuration:

`NC_ATTR_PERIOD` (Period)

`Period` specifies the rate of periodic behavior in milliseconds.

If you wish to specify the Period in Hertz instead of milliseconds, you can use the special hexadecimal format 800000*xx*, where *xx* is the desired rate in Hertz. For example, 80000020 hex specifies 32 Hz.

The behavior depends on the `Communication Type` as follows:

`NC_CAN_COMM_TX_PERIODIC`

`NC_CAN_COMM_TX_WAVEFORM`

`NC_CAN_COMM_RX_PERIODIC`

`Period` specifies the time between subsequent transmissions, and must be set greater than zero.

`NC_CAN_COMM_RX_UNSOL`

`NC_CAN_COMM_TX_RESP_ONLY`

`Period` specifies a watchdog timeout. If a frame is not received at least once every period, a timeout error is returned. Setting `Period` to zero disables the watchdog timer.

`NC_CAN_COMM_TX_BY_CALL`

`NC_CAN_COMM_RX_BY_CALL`

`Period` specifies a minimum interval between subsequent transmissions. Even if `ncWrite` is called very frequently, frames are transmitted on the network at a rate no more than `Period`. Setting `Period` to zero disables the minimum interval timer.

`NC_ATTR_READ_Q_LEN` (Read Queue Length)

`Read Queue Length` is the maximum number of unread frames for the read queue of the CAN Object. For more information, refer to `ncRead`.

If `Communication Type` is set to receive data, a typical value is 10. If `Communication Type` is set to transmit data, a typical value is 0.

`NC_ATTR_WRITE_Q_LEN` (Write Queue Length)

`Write Queue Length` is the maximum number of frames for the write queue of the CAN Object awaiting transmission. For more information, refer to `ncWrite`.

If `Communication Type` is set to receive data, a typical value is 0. If `Communication Type` is set to transmit data, a typical value is 10.

`NC_ATTR_RX_CHANGES_ONLY` (Receive Changes Only)

`Receive Changes Only` applies only to `Communication Type` selections in which the CAN Object receives data frames (ignored for other types). For those configurations, `Receive Changes Only` specifies whether duplicated data should be placed in the read queue. When set to NC_FALSE (default), all data frames for the CAN Object ID are placed in the read queue. When set to NC_TRUE, data frames are placed into the read queue only if the data bytes differ from the previously received data bytes.

This attribute has no effect on the usage of a watchdog timeout for the CAN Object. For example, if this attribute is NC_TRUE and you also specify a watchdog timeout, NI-CAN restarts the watchdog timer every time it receives a data frame for the CAN Object ID, regardless of whether the data differs from the previous frame in the read queue.

`NC_ATTR_COMM_TYPE` (Communication Type)

`Communication Type` specifies the behavior of the CAN Object with respect to its ID, including the direction of data transfer:

`NC_CAN_COMM_RX_UNSOL` (Receive Unsolicited)
: Receive data frames for a specific ID.

  This type is useful for receiving a few IDs (1–10) into dedicated read queues. For high performance applications (more IDs, fast frame rates), the Network Interface is recommended to receive all IDs.

  `Period` specifies a watchdog timeout, and `Receive Changes Only` specifies whether to place duplicate data frames into the read queue. `Transmit by Response` is ignored.

`NC_CAN_COMM_RX_PERIODIC` (Receive Periodic Using Remote)
: Periodically transmit remote frame for a specific ID in order to receive the associated data frame. Every `Period` the CAN Object transmits a remote frame, and then places the resulting data frame response in the read queue.

  `Period` specifies the periodic rate, and `Receive Changes Only` specifies whether to place duplicate data frames into the read queue. `Transmit by Response` is ignored.

`NC_CAN_COMM_RX_BY_CALL` (Receive By Call Using Remote)
: Transmit remote frame for a specific ID by calling `ncWrite`. The CAN Object places the resulting data frame response in the read queue.

`Period` specifies a minimum interval, and `Receive Changes Only` specifies whether to place duplicate data frames into the read queue. `Transmit by Response` is ignored.

`NC_CAN_COMM_TX_PERIODIC` (Transmit Data Periodically)

Periodically transmit data frame for a specific ID. When the CAN Object transmits the last entry from the write queue, that entry is used every period until you provide a new data frame using `ncWrite`. If you keep the write queue filled with unique data, this behavior allows you to ensure that each period transmits a unique data frame.

If the write queue is empty when communication starts, the first periodic transmit does not occur until you provide the first data frame with `ncWrite`.

This is the most commonly used CAN Object type. If you are not using remote frames, you can use multiple CAN Objects of this type, and the Network Interface for all other access (event-driven transmit and all receive).

`Period` specifies the periodic rate, and `Transmit by Response` specifies whether to transmit the previous period data in response to a remote frame. `Receive Changes Only` is ignored.

`NC_CAN_COMM_TX_RESP_ONLY` (Transmit By Response Only)

Transmit data frame for a specific ID only in response to a received remote frame. When you call `ncWrite`, the data is placed in the write queue, and remains there until a remote frame is received.

`Period` specifies a watchdog timeout. `Transmit by Response` is assumed as TRUE regardless of the attribute setting. `Receive Changes Only` is ignored.

`NC_CAN_COMM_TX_BY_CALL` (Transmit Data By Call)

Transmit data frame when `ncWrite` is called. When `ncWrite` is called quickly, data frames are placed in the write queue for back to back transmit.

`Period` specifies a minimum interval, and `Transmit by Response` specifies whether to retransmit the previous data frame in response to a remote frame. `Receive Changes Only` is ignored.

`NC_CAN_COMM_TX_WAVEFORM` (Transmit Periodic Waveform)

Transmit a fixed sequence of data frames over and over, one data frame every `Period`.

The following steps describe typical usage of this type:

1. Configure CAN Network Interface Object with `Start On Open` FALSE, then open the Network Interface.
2. Configure the CAN Object as Transmit Periodic Waveform and a nonzero `Write Queue Length`, then open the CAN Object.
3. Call `ncWrite` for the CAN Object, once for every entry specified for the `Write Queue Length`.
4. Use `ncAction` to start the Network Interface (not the CAN Object). The CAN Object transmits the first frame in the write queue, then waits the specified period, then transmits the second frame, and so on. After the last frame is transmitted, the CAN Objects waits the specified period, then transmits the first frame again.

If you need to change the waveform contents at runtime, or if you need to transmit very large waveforms (more than 100 frames), we recommend using the `NC_CAN_COMM_TX_PERIODIC` type. Using that type, you can write frames to the Write Queue until full (overflow error), then wait some time for a few frames to transmit, then continue writing new frames.

`Period` specifies the periodic rate. `Transmit by Response` and `Receive Changes Only` are ignored.

`NC_ATTR_TX_RESPONSE` (Transmit By Response)

`Transmit By Response` applies only to `Communication Type` of Transmit Data by Call and Transmit Data Periodically (ignored for other types). For those configurations, `Transmit By Response` specifies whether the CAN Object should automatically respond with the previously transmitted data frame when it receives a remote frame. When set to NC_FALSE (default), the CAN Object transmits data frames only as configured, and ignores all remote frames for its ID. When set to NC_TRUE, the CAN Object responds to incoming remote frames.

`NC_ATTR_DATA_LEN` (Data Length)

`Data Length` specifies the number of bytes in the data frames for this CAN Object ID. This number is placed in the Data Length Code (DLC) of all transmitted data frames and remote frames for the CAN Object. This is also the number of data bytes returned from `ncRead` when the communication type indicates receive.

`NC_ATTR_NOTIFY_MULT_LEN` (ReadMult Size for Notification)

Sets the number of frames used as a threshold for the Read Multiple state. For more information on the Read Multiple state, refer to `ncWaitForState`.

The default value is one half of `Read Queue Length`.

RTSI is a bus that interconnects National Instruments DAQ, IMAQ, NI-Motion, and CAN boards. This feature allows synchronization of DAQ, IMAQ, NI-Motion, and CAN boards by allowing exchange of timing signals. Using RTSI, a device (board) can control one or more slave devices. PCI/AT boards require a ribbon cable for the connections, but for PXI boards the connections are available on the PXI chassis backplane. Refer to the *NI-CAN User Manual* for more details on the hardware connector.

The following attribute IDs are used to enable RTSI synchronization between two or more National Instruments cards:

`NC_ATTR_RTSI_MODE` (RTSI Mode)

`RTSI Mode` specifies the behavior of the CAN Object with respect to RTSI, including whether the RTSI signal is an input or output:

`NC_RTSI_NONE`
: Disables RTSI behavior for the CAN Object (default). All other RTSI attributes are ignored.

`NC_RTSI_TX_ON_IN`
: The CAN Object will transmit a frame from its write queue when the RTSI input pulses. To begin transmission, at least one data frame must be written using `ncWrite`. If the write queue becomes empty due to frame transmissions, the last frame will be retransmitted on each RTSI pulse until another frame is provided using `ncWrite`.

  In order to use this `RTSI Mode`, you must configure the CAN Object `Communication Type` to either Transmit Data by Call, Transmit Data Periodically, or Transmit Periodic Waveform. The `Period` attribute is ignored when this RTSI mode is selected.

`NC_RTSI_TIME_ON_IN`
: When the RTSI input pulses, a timestamp is measured and stored in the read queue of the CAN Object. The special RTSI frame uses the following format:

Timestamp:	Time when RTSI input pulsed
Data:	User-defined 4 byte data pattern (refer to `RTSI Frame` for details)

Note When you configure a DAQ card to pulse the RTSI signal periodically, do not exceed 1,000 Hertz (pulse every millisecond). If the RTSI input is pulsed faster than 1 kHz on a consistent basis, CAN performance will be adversely affected (i.e. lost data frames).

`NC_RTSI_OUT_ON_RX`
The CAN Object will output the RTSI signal whenever a CAN frame is stored in its read queue.

In order to use this `RTSI Mode`, you must configure the CAN Object `Communication Type` to Receive Unsolicited.

`NC_RTSI_OUT_ON_TX`
The CAN Object will output the RTSI signal whenever a CAN frame is successfully transmitted.

In order to use this `RTSI Mode`, you must configure the CAN Object `Communication Type` to either Transmit Data by Call, Transmit Data Periodically, or Transmit Periodic Waveform.

`NC_RTSI_OUT_ACTION_ONLY`
The CAN Object will output the RTSI signal whenever the `ncAction` function is called with `Opcode` NC_OP_RTSI_OUT. This RTSI mode can be used to manually toggle/pulse a RTSI output within your application.

`NC_ATTR_RTSI_SIGNAL` (RTSI Signal)

`RTSI Signal` defines the RTSI signal associated with the `RTSI Mode`. Valid values are 0 to 7, corresponding to RTSI 0 to RTSI 7 on other National Instruments cards.

Note For CAN cards with high-speed (HS) ports only, four (4) RTSI signals are available for input, and four (4) RTSI signals are available for output. Since each RTSI signal is assigned to a Network Interface or CAN Object, this means that at most four NI-CAN objects can use RTSI inputs (or outputs). For example, if you configure five (5) RTSI signals for input, NI-CAN returns an error, regardless of which `RTSI Signal` numbers were used for each.

Note For CAN cards with one or more low-speed (LS) ports, two (2) RTSI signals are available for input, and three (3) RTSI signals are available for output.

Note For PXI-CAN cards, RTSI Signal 6 is unavailable.

Note Many NI-DAQ cards use RTSI Signal 7 as the 20 MHz clock, so this signal number should be avoided for other uses.

`NC_ATTR_RTSI_SIG_BEHAV` (RTSI Behavior)

`RTSI Behavior` specifies whether to pulse or toggle a RTSI output. This attribute is ignored when `RTSI Mode` specifies input:

`RTSI_SIG_PULSE`
: Pulse the RTSI output for at least 100 microseconds.

`RTSI_SIG_TOGGLE`
: If the previous state was high, the output toggles low, then vice-versa.

`NC_ATTR_RTSI_SKIP` (RTSI Skip)

`RTSI Skip` specifies the number of RTSI inputs to skip for RTSI input modes. It is ignored for RTSI output modes. For example, for `RTSI Mode NC_RTSI_TIME_ON_IN`, if the RTSI input pulses every 1 ms, `RTSI Skip` of 9 means that a timestamp will be stored in the read queue every 10 ms.

`NC_ATTR_RTSI_FRAME` (RTSI Frame)

`RTSI Frame` specifies a 4-byte pattern used to differentiate RTSI timestamps from CAN data frames. It is provided as a U32, and the high byte is stored as byte 0 from `ncRead`. For example, AABBCCDD hex is returned as AA in byte 0, BB in byte 1, and so on.

This attribute is used only for `RTSI Mode NC_RTSI_TIME_ON_IN`. It is ignored for all other `RTSI Mode` values.

Examples of Different Communication Types

The following figures demonstrate how you can use the `Communication Type` attribute for actual network data transfer. Each figure shows two separate NI-CAN applications that are physically connected across a CAN network.

Figure 9-1 shows a CAN Object that periodically transmits data to another CAN Object. The receiving CAN Object can queue up to five data values.

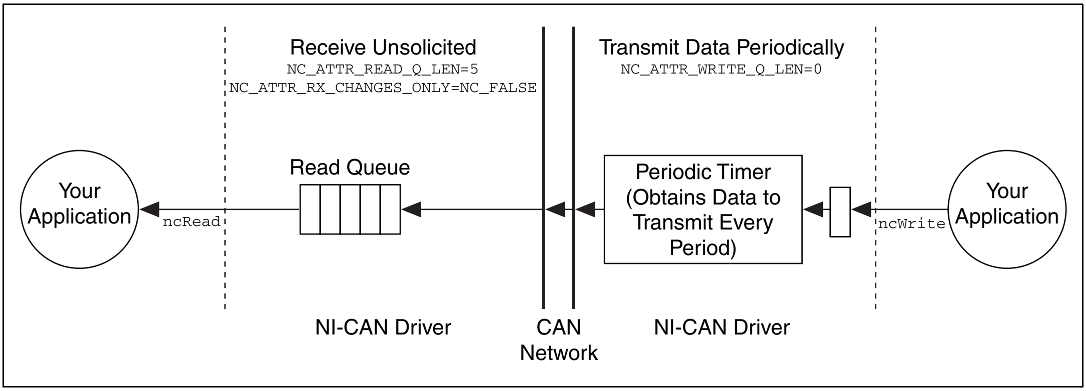


Figure 9-1. Example of Periodic Transmission

Figure 9-2 shows a CAN Object that polls data from another CAN Object. NI-CAN transmits the CAN remote frame when you call `ncWrite`.

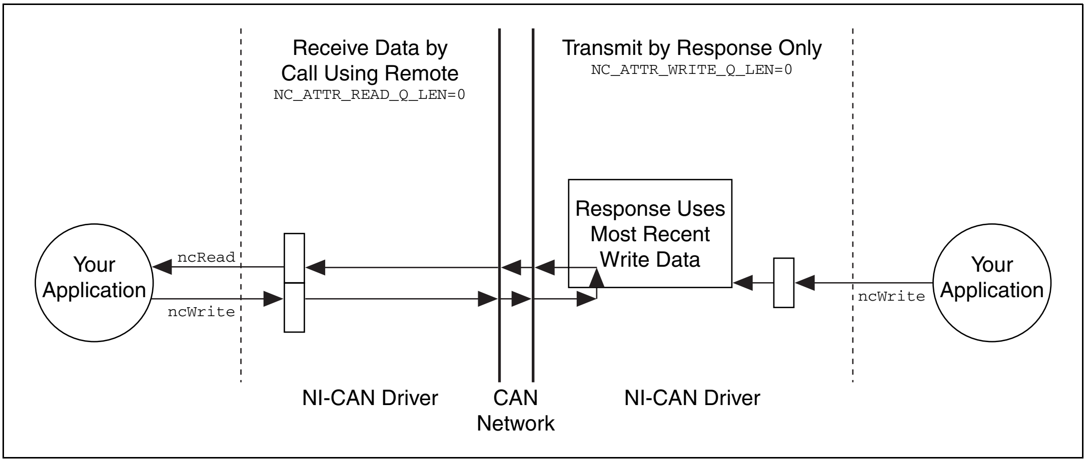


Figure 9-2. Example of Polling Remote Data Using ncWrite

Figure 9-3 shows a CAN Object that polls data from another CAN Object. NI-CAN transmits the remote frame periodically and places only changed data into the read queue.

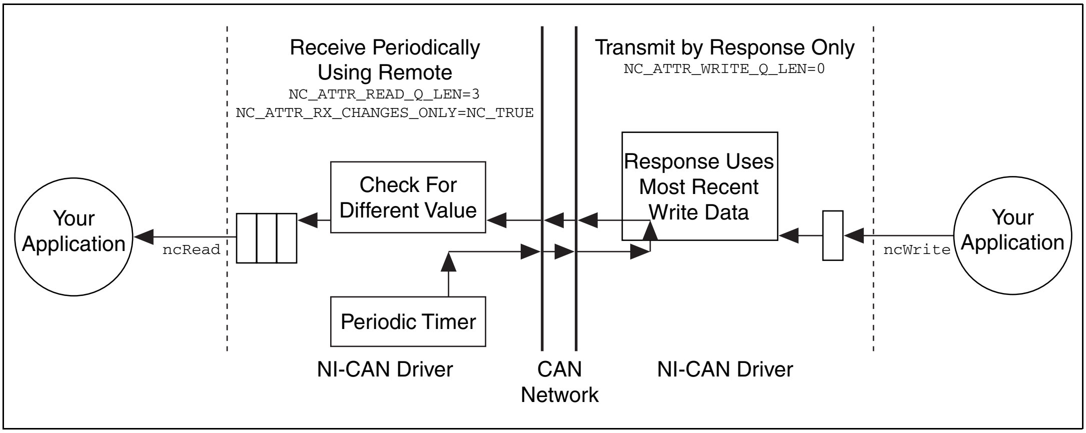


Figure 9-3. Example of Periodic Polling of Remote Data

ncCreateNotification

Purpose

Create a notification callback for an object.

Format

```
NCTYPE_STATUS    ncCreateNotification(
                          NCTYPE_OBJH ObjHandle,
                          NCTYPE_STATE DesiredState,
                          NCTYPE_UINT32 Timeout,
                          NCTYPE_ANY_P RefData,
                          NCTYPE_NOTIFY_CALLBACK
                             Callback)
```

Input

`ObjHandle`	Object handle.
`DesiredState`	States for which notification is sent.
`Timeout`	Length of time to wait in milliseconds.
`RefData`	Pointer to user-specified reference data.
`Callback`	Address of your callback function.

Output

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

`ncCreateNotification` creates a notification callback for the object specified by `ObjHandle`. The NI-CAN driver uses the notification callback to communicate state changes to your application.

This function is normally used when you want to allow other code to execute while waiting for NI-CAN states, especially when the other code does not call NI-CAN functions. If such background execution is not needed, the `ncWaitForState` function offers better overall performance. The `ncWaitForState` function cannot be used at the same time as `ncCreateNotification`.

Upon successful return from `ncCreateNotification`, the notification callback is invoked whenever one of the states specified by `DesiredState` occurs in the object.

If DesiredState is zero, notifications are disabled for the object specified by ObjHandle. DesiredState specifies a bit mask for which notification is desired. You can use a single state alone, or you can OR them together.

NC_ST_READ_AVAIL (00000001 hex)

At least one frame is available, which you can obtain using an appropriate read function.

The state is set whenever a frame arrives for the object. The state is cleared when the read queue is empty.

NC_ST_WRITE_SUCCESS (00000002 hex)

All frames provided with a write function have been successfully transmitted onto the network. Successful transmit means that the frame won arbitration, and was acknowledged by a remote device.

The state is set when the last frame in the write queue is transmitted successfully. The state is cleared when a write function is called.

For CAN Objects, Write Success does not always mean that transmission has stopped. For example, if a CAN Object is configured for Transmit Data Periodically, Write Success occurs when the write queue has been emptied, but periodic transmit of the last frame continues.

When communication starts, the NC_ST_WRITE_SUCCESS state is true by default.

NC_ST_READ_MULT (00000008 hex)

A specified number of frames are available, which you can obtain using ncReadMult. The number of frames is one half the Read Queue Length by default, but you can change it using the ReadMult Size for Notification attribute of ncSetAttribute.

The state is set whenever the specified number of frames are stored in the read queue of the object. The state is cleared when you call the read function, and less than the specified number of frames exist in the read queue.

The NI-CAN driver waits up to Timeout for one of the bits set in DesiredState to become set in the attribute NC_ATTR_STATE. You can use the special Timeout value NC_DURATION_INFINITE to wait indefinitely.

The Callback parameter provides the address of a callback function in your application. Within the Callback function, you can call any of the NI-CAN functions except ncCreateNotification and ncWaitForState.

With the RefData parameter, you provide a pointer that is sent to all notifications for the given object. This pointer normally provides reference data for use within the Callback function. For example, when you create a notification for the NC_ST_READ_AVAIL state, RefData is often the data pointer that you pass to ncRead to read available data. If the callback function does not need reference data, you can set RefData to NULL.

Callback Prototype

```
NCTYPE_STATE        _NCFUNC_ Callback (NCTYPE_OBJH ObjHandle,
                          NCTYPE_STATE State,
                          NCTYPE_STATUS Status,
                          NCTYPE_ANY_P RefData);
```

Callback Parameters

ObjHandle	Object handle.
State	Current state of object.
Status	Object status.
RefData	Pointer to your reference data.

Callback Return Value

The value you return from the callback indicates the desired states to re-enable for notification. If you no longer want to receive notifications for the callback, return a value of zero.

If you return a state from the callback, and that state is still set in the NC_ATTR_STATE attribute, the callback is invoked again immediately after it returns. For example, if you return NC_ST_READ_AVAIL when the read queue has not been emptied, the callback is invoked again.

Callback Description

In the prototype for Callback, _NCFUNC_ ensures a proper calling scheme between the NI-CAN driver and your callback.

The Callback function executes in a separate thread in your process. Therefore, it has access to any process global data, but not to thread local data. If the callback needs to access global data, you must protect that access using synchronization primitives (such as semaphores), because the callback is running in a different thread context. Alternatively, you can avoid the issue of data protection entirely if the callback simply posts a message to your application using the Win32 `PostMessage` function. For complete information on multithreading issues, refer to the Win32 Software Development Kit (SDK) online help.

In LabWindows/CVI, you cannot access User Interface library functions within the callback thread. To defer a callback for User Interface interaction, use the CVI `PostDeferredCall` function. For more information, refer to the LabWindows/CVI documentation.

The `ObjHandle` is the same object handle passed to `ncCreateNotification`. It identifies the object generating the notification, which is useful when you use the same callback function for notifications from multiple objects.

The `State` parameter holds the current state(s) of the object that generated the notification (`NC_ATTR_STATE` attribute). If the `Timeout` passed to `ncCreateNotification` expires before the desired states occur, the NI-CAN driver invokes the callback with `State` equal to zero.

The `Status` parameter holds the current status of the object. If an error occurs, `State` is zero and `Status` holds the error status. The most common notification error occurs when the `Timeout` passed to `ncCreateNotification` expires before the desired states occur (`CanErrFunctionTimeout` status code). If no error is reported, `Status` is `CanSuccess`.

The `RefData` parameter is the same pointer passed to `ncCreateNotification`, and it accesses reference data for the `Callback` function.

ncGetAttribute

Purpose

Get the value of an object attribute.

Format

```
NCTYPE_STATUS    ncGetAttribute(
                              NCTYPE_OBJH ObjHandle,
                              NCTYPE_ATTRID AttrId,
                              NCTYPE_UINT32 AttrSize,
                              NCTYPE_ANY_P AttrPtr)
```

Input

ObjHandle	Object handle.
AttrId	Identifier of the attribute to get.
AttrSize	Size of the attribute in bytes.

Output

AttrPtr	Pointer used to return attribute value.

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to ncStatusToString.

Description

ncGetAttribute gets the value of the attribute specified by AttrId from the object specified by ObjHandle. Within NI-CAN objects, you use attributes to access configuration settings, status, and other information about the object, but not data.

AttrPtr points to the variable used to receive the attribute value. Its type is undefined so that you can use the appropriate host data type for AttrId. AttrSize indicates the size of the variable that AttrPtr points to. AttrSize is typically 4, and AttrPtr references a 32-bit unsigned integer.

You can get any of the AttrId mentioned in ncConfig using ncGetAttribute. The following list describes other AttrId you can get using ncGetAttribute:

NC_ATTR_PROTOCOL (Protocol)

For NI-CAN, this always returns 1.

For NI-DNET, this always returns 2.

This attribute is available only from the Network Interface, not CAN Objects.

NC_ATTR_PROTOCOL_VERSION (Protocol Version)

For NI-CAN, this returns 02000200 hex, which corresponds to version 2.0B of the Bosch CAN specifications. For more information on the encoding of the version, refer to Software Version.

This attribute is available only from the Network Interface, not CAN Objects.

NC_ATTR_SOFTWARE_VERSION (Software Version)

Version of the NI-CAN software, with major, minor, update, and beta build numbers encoded in the U32 from high to low bytes. For example, 2.0.1 would be 02000100 hex, and 2.1beta5 would be 02010005 hex.

This attribute is available only from the Network Interface, not CAN Objects.

NC_ATTR_STATE (Object State)

Returns the current state bit mask of the object. Polling with ncGetAttr provides an alternative method of state detection than ncWaitForState or ncCreateNotification. For more information on the states returned from this attribute, refer to the DesiredState input of ncWaitForState.

NC_ATTR_READ_PENDING (Read Entries Pending)

Returns the number of frames available in the read queue. Polling the available frames with this attribute can be used as an alternative to the ncWaitForState and ncCreateNotification functions.

NC_ATTR_WRITE_PENDING (Write Entries Pending)

Returns the number of frames pending transmission in the write queue. If your intent is to verify that all pending frames have been transmitted successfully, waiting for the Write Success state is preferable to this attribute.

`NC_ATTR_NOTIFY_MULT_LEN` (ReadMult Size for Notification)

Returns the number of frames used as a threshold for the Read Multiple state. For more information, refer to this attribute in `ncSetAttribute`.

`NC_ATTR_ABS_TIME` (Absolute Timestamp)

Returns the absolute timestamp value. The timestamp format is a 64-bit unsigned integer compatible with the Win32 FILETIME type (NCTYPE_ABS_TIME). This absolute time is kept in a Coordinated Universal Time (UTC) format. UTC time is loosely defined as the current date and time of day in Greenwich England. Microsoft defines its UTC time (FILETIME) as a 64-bit counter of 100ns intervals that have elapsed since 12:00 a.m., January 1, 1601.

Since the timestamp returned by `ncRead` (and this attribute) is compatible with Win32 FILETIME, you can pass it into the Win32 FileTimeToLocalFileTime function to convert it to your local time zone, then pass the resulting local time to the Win32 FileTimeToSystemTime function to convert to the Win32 SYSTEMTIME type. SYSTEMTIME is a struct with fields for year, month, day, and so on. For more information on Win32 time types and functions, refer to your Microsoft Win32 documentation.

Since the absolute timestamp type is 64 bits (`NCTYPE_ABS_TIME`), you must use `AttrSize` of 8.

`NC_ATTR_HW_SERIAL_NUM` (Serial Number)

Returns the serial number of the card on which the Network Interface or CAN Object is located.

`NC_ATTR_HW_FORMFACTOR` (Form Factor)

Returns the form factor of the card on which the Network Interface or CAN Object is located.

The returned Form Factor is an enumeration.

`NC_HW_FORMFACTOR_PCI`	PCI
`NC_HW_FORMFACTOR_PXI`	PXI
`NC_HW_FORMFACTOR_PCMCIA`	PCMCIA
`NC_HW_FORMFACTOR_AT`	AT

`NC_ATTR_HW_TRANSCEIVER` (Transceiver)

Returns the CAN transceiver of the port on which the Network Interface or CAN Object is located.

The returned Transceiver is an enumeration.

`NC_HW_TRANSCEIVER_HS`	HS
`NC_HW_TRANSCEIVER_LS`	LS

This attribute is not supported on the PCMCIA form factor.

`NC_ATTR_INTERFACE_NUM` (Interface Number)

Returns the interface number of the port on which the Network Interface or CAN Object is located.

This is the same number that you used in the `ObjName` string of the previous `ncConfig` and `ncOpenObject` functions.

ncGetHardwareInfo

Purpose

Get NI-CAN hardware information.

Format

```
NCTYPE_STATUS _NCFUNC_ ncGetHardwareInfo(
                         NCTYPE_UINT32 CardNumber,
                         NCTYPE_UINT32 PortNumber,
                         NCTYPE_ATTRID AttrId,
                         NCTYPE_UINT32 AttrSize,
                         NCTYPE_ANY_P AttrPtr);
```

Input

`CardNumber` Specifies the CAN card number from 1 to Number of Cards, where Number of Cards is the number of CAN cards in your system. You can obtain `Number of Cards` using this function with `CardNumber` = 1, `PortNumber` = 1, and `AttrID` = `Number of Cards`.

`PortNumber` Specifies the CAN port number from 1 to `Number of Ports`, where `Number of Ports` is the number of CAN ports on this CAN card. You can obtain `Number of Ports` using this function with `PortNumber` = 1, and `AttrID` = `Number of Ports`.

`AttrID` Specifies the attribute to get:

`NC_ATTR_VERSION_MAJOR` (Version Major)

Returns the major version of the NI-CAN software. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use `CardNumber` 1 and `PortNumber` 1 as inputs.

The major version is the 'X' in X.Y.Z.

`NC_ATTR_VERSION_MINOR (Version Minor)`

Returns the minor version of the NI-CAN software. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use `CardNumber` 1 and `PortNumber` 1 as inputs.

The major version is the 'Y' in X.Y.Z.

`NC_ATTR_VERSION_UPDATE` (Version Update)

Returns the update version of the NI-CAN software. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use `CardNumber` 1 and `PortNumber` 1 as inputs.

The major version is the 'Z' in X.Y.Z.

`NC_ATTR_VERSION_PHASE` (Version Phase)

Returns the phase of the NI-CAN software. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use `CardNumber` 1 and `PortNumber` 1 as inputs.

Phase 1 specifies Alpha, phase 2 specifies Beta, and phase 3 specifies Final release. Unless you are participating in an NI-CAN beta program, you will always see 3.

`NC_ATTR_VERSION_BUILD` (Version Build)

Returns the build of the NI-CAN software. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use `CardNumber` 1 and `PortNumber` 1 as inputs.

With each software development phase, subsequent NI-CAN builds are numbered sequentially. A given Final release version always uses the same build number, so unless you are participating in an NI-CAN beta program, this build number is not relevant.

`NC_ATTR_VERSION_COMMENT` (Version Comment)

Returns any special comment on the NI-CAN software. `AttrPtr` must point to a buffer for the string, and `AttrSize` specifies the number of characters in that buffer. Use `CardNumber` 1 and `PortNumber` 1 as inputs.

This string is normally empty for a Final release. In rare circumstances, an NI-CAN prototype or patch may be released to a specific customer. For these special releases, the version comment describes the special features of the release.

`NC_ATTR_NUM_CARDS` (Number of Cards)

Returns the number of NI-CAN cards in your system. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use `CardNumber` 1 and `PortNumber` 1 as inputs.

If you are displaying all hardware information, you get this attribute first, then iterate through all CAN cards with a For loop. Inside the card For loop, you get all card-wide attributes including Number Of Ports, then use another For loop to get port-wide attributes.

`NC_ATTR_HW_SERIAL_NUM` (Serial Number)

Card-wide attribute that returns the serial number of the card. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use the desired `CardNumber`, and `PortNumber` 1 as inputs.

`NC_ATTR_HW_FORMFACTOR` (Form Factor)

Card-wide attribute that returns the form factor of the card. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use the desired `CardNumber`, and `PortNumber` 1 as inputs.

The returned Form Factor is an enumeration.

`NC_HW_FORMFACTOR_PCI`	PCI
`NC_HW_FORMFACTOR_PXI`	PXI
`NC_HW_FORMFACTOR_PCMCIA`	PCMCIA
`NC_HW_FORMFACTOR_AT`	AT

`NC_ATTR_NUM_PORTS (Number of Ports)`

Card-wide attribute that returns the number of ports on the card. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use the desired `CardNumber`, and `PortNumber` 1 as inputs.

If you are displaying all hardware information, you get this attribute within the For loop for all cards, then iterate through all CAN ports to get port-wide attributes.

`NC_ATTR_HW_TRANSCEIVER (Transceiver)`

Port-wide attribute that returns the CAN transceiver of the port. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use the desired `CardNumber` and `PortNumber` as inputs.

The returned Transceiver is an enumeration.

`NC_HW_TRANSCEIVER_HS`	HS
`NC_HW_TRANSCEIVER_LS`	LS

This attribute is not supported on the PCMCIA form factor.

`NC_ATTR_INTERFACE_NUM (Interface Number)`

Port-wide attribute that returns the interface number of the port. `AttrPtr` must point to NCTYPE_UINT32, and `AttrSize` must be 4. Use the desired `CardNumber` and `PortNumber` as inputs.

The interface number is assigned to a physical port using the Measurement and Automation Explorer (MAX). The interface number is used as a string in the Frame API (i.e. "CAN0"). The interface number is used for the `Interface` input in the Channel API.

`AttrSize` Size of the attribute in bytes.

Output

`AttrPtr` Pointer used to return attribute value.

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

This function provides information about available CAN cards, but does not require you to open/start sessions. First get `Number of Cards`, then loop for each card. For each card, you can get card-wide attributes (such as `Form Factor`), and you can also get the `Number of Ports`. For each port, you can get port-wide attributes such as the `Transceiver`.

ncOpenObject

Purpose

Open an object.

Format

```
NCTYPE_STATUS    ncOpenObject(
                          NCTYPE_STRING ObjName,
                          NCTYPE_OBJH_P ObjHandlePtr)
```

Input

`ObjName`	ASCII name of the object to open.

Output

`ObjHandlePtr`	Pointer used to return Object handle. Used with all subsequent NI-CAN function calls.

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

`ncOpenObject` takes the name of an object to open and returns a handle to that object that you use with subsequent NI-CAN function calls.

The Frame API and Channel API cannot use the same CAN network interface simultaneously. If the CAN network interface is already initialized in the Channel API, this function returns an error.

Although NI-CAN can generally be used by multiple applications simultaneously, it does not allow more than one application to open the same object. For example, if one application opens `CAN0`, and another application attempts to open `CAN0`, the second `ncOpenObject` returns the error `CanErrAlreadyOpen`. It is legal for one application to open `CAN0::STD14` and another application to open `CAN0::STD21`, because the two objects are considered distinct.

If `ncOpenObject` is successful, a handle to the newly opened object is returned. You use this object handle for all subsequent function calls for the object.

The following sections describe how to use `ncOpenObject` with the Network Interface and Can Object.

Network Interface

`ObjName` is the name of the CAN Network Interface Object to configure. This string uses the syntax "CAN*x*", where *x* is a decimal number starting at zero that indicates the CAN network interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX).

CAN Object

`ObjName` is the name of the CAN Object to configure. This string uses the syntax "CAN*x*::STD*y*" or "CAN*x*::XTD*y*". CAN*x* is the name of the CAN network interface that you used for the preceding `ncConfig` function. STD indicates that the CAN Object uses a standard (11-bit) arbitration ID. XTD indicates that the CAN Object uses an extended (29-bit) arbitration ID. The number *y* specifies the actual arbitration ID of the CAN Object. The number *y* is decimal by default, but you can also use hexadecimal by adding `0`*x* to the beginning of the number. For example, `CAN0::STD25` indicates standard ID 25 decimal on CAN0, and `CAN1::XTD0x0000F652` indicates extended ID F652 hexadecimal on CAN1.

ncRead

Purpose

Read single frame from an object.

Format

```
NCTYPE_STATUS    ncRead(
                         NCTYPE_OBJH ObjHandle,
                         NCTYPE_UINT32 DataSize,
                         NCTYPE_ANY_P DataPtr)
```

Input

`ObjHandle`	Object handle.
`DataSize`	Size of the data in bytes.

Output

`DataPtr`	Pointer used to return the frame.

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

`ncRead` reads a single frame from the object specified by `ObjHandle`.

`DataPtr` points to the variable that holds the data. Its type is undefined so that you can use the appropriate host data type. `DataSize` indicates the size of variable pointed to by `DataPtr`, and is used to verify that the size you have available is compatible with the configured read size for the object.

For information on the data type to use with `DataPtr`, refer to the following Network Interface and CAN Object descriptions.

You use `ncRead` to obtain data from the read queue of an object. Because NI-CAN handles the read queue in the background, this function does not wait for new data to arrive. To ensure that new data is available before calling `ncRead`, first wait for the `NC_ST_READ_AVAIL` state. The `NC_ST_READ_AVAIL` state transitions from false to true when NI-CAN places a new data item into an empty read queue, and remains true until you read the last data item from the queue.

The ncRead function is useful when you need to process one frame at a time. In order to read multiple frames, such as for bus analyzer applications, use the ncReadMult function.

When you call ncRead for an empty read queue (NC_ST_READ_AVAIL false), the data from the previous call to ncRead is returned to you again, along with the CanWarnOldData warning. If no data item has yet arrived for the read queue, a default data item is returned, which consists of all zeros.

When a new data item arrives for a full queue, NI-CAN discards the item, and the next call to ncRead returns the CanErrOverflowRead error. You can avoid this overflow behavior by setting the read queue length to zero. When a new data item arrives for a zero length queue, it simply overwrites the previous item without indicating an overflow. The NC_ST_READ_AVAIL state and CanWarnOldData warning still behave as usual, but you can ignore them if you only want the most recent data. You can use the NC_ATTR_READ_Q_LEN attribute to configure the read queue length.

CAN Network Interface Object

The data type that you use with ncRead of the Network Interface is NCTYPE_FRAME_STRUCT. When calling ncRead, you should pass sizeof(NCTYPE_FRAME_STRUCT) for the DataSize parameter.

Within the NCTYPE_FRAME_STRUCT structure, the FrameType field determines the meaning of all other fields. The following tables, 9-5 through 9-7, describe the fields of NCTYPE_FRAME_STRUCT for each value of FrameType.

Table 9-5. NCTYPE_FRAME_STRUCT Fields for FrameType NC_FRMTYPE_DATA (0)

Field Name	Data Type	Description
FrameType	NCTYPE_UINT8	NC_FRMTYPE_DATA (0) This value indicates a CAN data frame.
ArbitrationId	NCTYPE_CAN_ARBID	Returns the arbitration ID of the received data frame. The NCTYPE_CAN_ARBID type is an unsigned 32-bit integer that uses the bit mask NC_FL_CAN_ARBID_XTD (0x20000000) to indicate an extended ID. A standard ID (11-bit) is specified by default. The Network Interface receives data frames based on the comparators and masks configured in ncConfig.
Data	Array of 8 NCTYPE_UINT8	Returns the data bytes of the frame.

Table 9-5. NCTYPE_FRAME_STRUCT Fields for FrameType NC_FRMTYPE_DATA (0) (Continued)

Field Name	Data Type	Description
`DataLength`	`NCTYPE_UINT8`	Returns the number of data bytes received in the frame. This specifies the number of valid data bytes in `Data`.
`Timestamp`	`NCTYPE_ABS_TIME`	Returns the absolute timestamp when the data frame was received from the CAN network. The timestamp data type (NCTYPE_ABS_TIME) is a 64-bit unsigned integer compatible with the Win32 FILETIME type. This absolute time is kept in a Coordinated Universal Time (UTC) format. UTC time is loosely defined as the current date and time of day in Greenwich England. Microsoft defines its UTC time (FILETIME) as a 64-bit counter of 100 ns intervals that have elapsed since 12:00 a.m., January 1, 1601. Since `Timestamp` is compatible with Win32 FILETIME, you can pass it into the Win32 FileTimeToLocalFileTime function to convert it to your local time zone, then pass the resulting local time to the Win32 FileTimeToSystemTime function to convert to the Win32 SYSTEMTIME type. SYSTEMTIME is a struct with fields for year, month, day, and so on. For more information on Win32 time types and functions, refer to your Microsoft Win32 documentation.

Table 9-6. NCTYPE_FRAME_STRUCT Fields for FrameType NC_FRMTYPE_COMM_ERR (2)

Field Name	Data Type	Description
FrameType	NCTYPE_UINT8	NC_FRMTYPE_COMM_ERR (2) This value indicates a logged communication warning or error as reported by the CAN hardware. This frame type occurs only when you set the Log Comm Warnings attribute to TRUE. Refer to ncConfig for details.
ArbitrationId	NCTYPE_CAN_ARBID	Indicates the type of communication problem: 8000000B hex:Comm. error: General 4000000B hex:Comm. warning: General 8001000B hex:Comm. error: Stuff 4001000B hex:Comm. warning: Stuff 8002000B hex:Comm. error: Format 4002000B hex:Comm. warning: Format 8003000B hex:Comm. error: No Ack 4003000B hex:Comm. warning: No Ack 8004000B hex:Comm. error: Tx 1 Rx 0 4004000B hex:Comm. warning: Tx 1 Rx 0 8005000B hex:Comm. error: Tx 0 Rx 1 4005000B hex:Comm. warning: Tx 0 Rx 1 8006000B hex:Comm. error: Bad CRC 4006000B hex:Comm. warning: Bad CRC 0000000B hex:Comm. errors/warnings cleared 4000000C hex:LS fault warning 0000000C hex:LS fault cleared
Data	Array of 8 NCTYPE_UINT8	This field is not applicable to this frame type, and should be ignored.
DataLength	NCTYPE_UINT8	This field is not applicable to this frame type, and should be ignored.
Timestamp	NCTYPE_ABS_TIME	Returns the absolute timestamp when the communications problem occurred. For information on the timestamp data type, refer to Table 9-5.

Table 9-7. NCTYPE_FRAME_STRUCT Fields for FrameType NC_FRMTYPE_RTSI (3)

Field Name	Data Type	Description
FrameType	NCTYPE_UINT8	NC_FRMTYPE_RTSI (3) Indicates when a RTSI input pulse occurred relative to incoming CAN frames. This frame type occurs only when you set the RTSI Mode attribute to NC_RTSI_TIME_ON_IN (refer to ncConfig for details).
ArbitrationId	NCTYPE_CAN_ARBID	Returns the special value 40000001 hex.
Data	Array of 8 NCTYPE_UINT8	This field is not applicable to this frame type, and should be ignored.
DataLength	NCTYPE_UINT8	Returns the RTSI signal number detected.
Timestamp	NCTYPE_ABS_TIME	Returns the absolute timestamp when the RTSI input occurred. For information on the timestamp data type, refer to Table 9-5.

Error Active, Error Passive, and Bus Off States

When the CAN communication controller transfers into the error passive state, NI-CAN returns the warning CanCommWarning from read functions.

When the transmit error counter of the CAN communication controller increments above 255, the network interface transfers into the bus off state as dictated by the CAN protocol. The network interface stops communication so that you can correct the defect in the network, such as a malfunctioning cable or device. When bus off occurs, the NC_ST_ERROR and NC_ST_STOPPED states are set in the NC_ATTR_STATE attribute of the CAN Network Interface Object and all of its higher level CAN Objects. The background status attribute (NC_ATTR_STATUS) is set with the CanWarnComm status code.

If no CAN devices are connected to the network interface port, and you attempt to transmit a frame, the CanWarnComm status occurs. This warning occurs because the missing acknowledgment bit increments the transmit error counter until the network interface reaches the error passive state, but bus off state is never reached.

Because the error counters in the CAN controller reflect the status of the CAN network, and not necessarily your CAN application, a given CanWarnComm status code will often remain from one run of your application to the next. If you want to clear the CAN controller error counters (and the CanWarnComm warning) completely when your application starts, use ncAction of NC_OP_RESET to reset the CAN controller, then use ncAction of NC_OP_START to resume communication.

For more information about low-speed communication error handling, refer to the description of the NC_ATTR_LOG_COMM_ERRS (Log Comm Warnings) attribute ID in the ncConfig function description in this chapter.

CAN Object

The data type that you use with ncRead of the CAN Object is NCTYPE_CAN_DATA_TIMED. When calling ncRead, you should pass sizeof(NCTYPE_CAN_DATA_TIMED) for the DataSize parameter. Table 9-8 describes the fields of NCTYPE_CAN_DATA_TIMED.

Table 9-8. NCTYPE_CAN_DATA_TIMED Field Names

Field Name	Data Type	Description
Data	Array of 8 NCTYPE_UINT	Data array returns 8 data bytes. The actual number of valid data bytes depends on the CAN Object configuration specified in ncConfig. If the CAN Object Communication Type specifies Transmit, data frames are transmitted, not received, so Data always contains zero valid bytes. For this Communication Type, the ncRead function has no effect. If the CAN Object Communication Type specifies Receive, Data always contains Data Length valid bytes, where Data Length was configured using ncConfig.

Table 9-8. NCTYPE_CAN_DATA_TIMED Field Names (Continued)

Field Name	Data Type	Description
Timestamp	NCTYPE_ABS_TIME	Returns the absolute timestamp value. The timestamp data type ((NCTYPE_ABS_TIME) is a 64-bit unsigned integer compatible with the Win32 FILETIME type. This absolute time is kept in a Coordinated Universal Time (UTC) format. UTC time is loosely defined as the current date and time of day in Greenwich England. Microsoft defines its UTC time (FILETIME) as a 64-bit counter of 100 ns intervals that have elapsed since 12:00 a.m., January 1, 1601. Since Timestamp is compatible with Win32 FILETIME, you can pass it into the Win32 FileTimeToLocalFileTime function to convert it to your local time zone, then pass the resulting local time to the Win32 FileTimeToSystemTime function to convert to the Win32 SYSTEMTIME type. SYSTEMTIME is a struct with fields for year, month, day, and so on. For more information on Win32 time types and functions, refer to your Microsoft Win32 documentation.

ncReadMult

Purpose

Read multiple frames from an object.

Format

```
NCTYPE_STATUS ncReadMult(
                        NCTYPE_OBJH ObjHandle,
                        NCTYPE_UINT32  DataSize,
                        NCTYPE_ANY_P  DataPtr,
                        NCTYPE_UINT32_P  ActualDataSize);
```

Input

`ObjHandle`	Object handle.
`DataSize`	The size of the data buffer in bytes.
`DataPtr`	Points to data buffer in which the data returned.

Output

`ActualDataSize`	The number of bytes actually returned.

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

This function returns multiple frames from the read queue of the object specified by `ObjHandle`. When used with the Network Interface, `ncReadMult` is useful in analyzer applications where data frames need to be acquired at a high speed and stored for analysis in the future. For single frame and most recent data frame acquisition, you should use `ncRead`.

`DataPtr` points to an array of either `NCTYPE_CAN_STRUCT` or `NCTYPE_CAN_DATA_TIMED`. `DataSize` indicates the size of the array pointed to by `DataPtr` (in bytes). This size is specified in bytes in order to verify that the proper data type and alignment is used. When `ncReadMult` returns, the number of bytes copied into `DataPtr` is provided in `ActualDataSize`.

Because NI-CAN handles the read queue in the background, this function does not wait for new data to arrive. To ensure that new data is available before calling `ncReadMult`, first wait for the `NC_ST_READ_MULT` state. Refer to `NC_ST_READ_MULT (00000008 hex)` in the ncCreateNotification function description in this chapter for more information on this state.

Unlike the ncRead function, the ncReadMult function does not return the CanWarnOldData warning to indicate zero frames. If there is no new data, the function returns with an ActualDataSize of zero.

The description for CanErrOverflowRead and the host data types is identical to that of ncRead with the exception of CanWarnOldData, described above.

Refer to the ncRead function description for more details on the structures used with ncReadMult.

ncReset

Purpose

Reset the CAN card.

Format

```
NCTYPE_STATUS_NCFUNC_ncReset(
                                        NCTYPE_STRING ObjName,
                                        NCTYPE_UINT32 Param);
```

Input

`ObjName`	ASCII name of the interface (card) to reset
`Param`	Reserved for future use (set to 0)

Output

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

This function completely resets the CAN card and ensures that all handles for that card are closed.

`ObjName` is the name of the CAN Network Interface Object that indicates the card to reset. This name uses the same "CAN*x*" syntax as `ncConfig`, but the reset applies to the entire CAN card. For example, if a 2-port card contains "CAN0" and "CAN1", calling `ncReset` with `ObjName` "CAN1" resets all hardware/software associated with both "CAN0" and "CAN1".

If an NI-CAN application is terminated prior to closing all handles, the `CanErrNotStopped` or `CanErrAlreadyOpen` error might occur when the application is restarted. By making this the first NI-CAN function called in your application (preceding all `ncConfig`), you can avoid problems related to improper termination.

You can only use the `ncReset` function if you plan to run a single NI-CAN application. If you run more than one NI-CAN application, each with `ncReset`, the second `ncReset` call will close all handles for the first application. You should only use the `ncReset` function as a temporary measure. After you update your application so that it successfully closes NI-CAN handles on termination, it should no longer be used.

ncSetAttribute

Purpose

Set the value of an object attribute.

Format

```
NCTYPE_STATUS    ncSetAttribute(
                          NCTYPE_OBJH ObjHandle,
                          NCTYPE_ATTRID AttrId,
                          NCTYPE_UINT32 AttrSize,
                          NCTYPE_ANY_P AttrPtr)
```

Input

`ObjHandle`	Object handle.
`AttrId`	Identifier of the attribute to set.
`AttrSize`	Size of the attribute in bytes.
`AttrPtr`	New attribute value. You provide the attribute value using the pointer `AttrPtr`.

Output

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

`ncSetAttribute` sets the value of the attribute specified by `AttrId` in the object specified by `ObjHandle`.

`AttrPtr` points to the variable that holds the attribute value. Its type is undefined so that you can use the appropriate host data type for `AttrId`. `AttrSize` indicates the size of variable pointed to by `AttrPtr`. `AttrSize` is typically 4, and `AttrPtr` references a 32-bit unsigned integer.

For a listing of valid attributes for the Network Interface and CAN Object, refer to ncConfig. Unless stated otherwise, communication must be stopped prior to changing an attribute with ncSetAttribute. While the Network Interface and all CAN Objects are stopped, you can set any of the `AttrId` mentioned in `ncConfig` using `ncSetAttribute`.

ncStatusToString

Purpose

Convert status code into a descriptive string.

Format

```
void              ncStatusToString(
                                   NCTYPE_STATUS Status,
                                   NCTYPE_UINT32 SizeofString,
                                   NCTYPE_STRING String)
```

Input

`Status`	Nonzero status code returned from NI-CAN function.
`SizeofString`	Size of `String` buffer (in bytes).

Output

`String`	ASCII string that describes `Status`.

Description

When the status code returned from an NI-CAN function is nonzero, an error or warning is indicated. This function is used to obtain a description of the error/warning for debugging purposes.

If you want to avoid displaying error messages while debugging your application, you can use the `Explain.exe` utility. This console application is located in the NI-CAN installation folder, which is typically `\Program Files\National Instruments\NI-CAN`. You enter an NI-CAN status code in the command line, `Explain 0XBFF62201` for example, and the utility displays the description.

The return code is passed into the `Status` parameter. The `SizeofString` parameter indicates the number of bytes available in `String` for the description. The description will be truncated to size `SizeofString` if needed, but a size of 300 characters is large enough to hold any description. The text returned in `String` is null-terminated, so it can be used with ANSI C functions such as `printf`.

For applications written in C or C++, each NI-CAN function returns a status code as a signed 32-bit integer. Table 9-9 summarizes the NI-CAN use of this status.

Table 9-9. NI-CAN Status Codes

Status Code	Meaning
Negative	Error—Function did not perform expected behavior.
Positive	Warning—Function performed as expected, but a condition arose that may require your attention.
Zero	Success—Function completed successfully.

Your application code should check the status returned from every NI-CAN function. If an error is detected, you should close all NI-CAN handles, then exit the application. If a warning is detected, you can display a message for debugging purposes, or simply ignore the warning.

The following piece of code shows an example of handling NI-CAN status during application debugging.

```
status= ncOpenObject ("CAN0", &MyObjHandle);
PrintStat (status, "ncOpen CAN0");
```

where the function `PrintStat` has been defined at the top of the program as:

```
void PrintStat(NCTYPE_STATUS status, char *source)
{
    char statusString[300];
    if(status !=0)
    {
        ncStatusToString(status, sizeof(statusString),
                         statusString);
        printf("\n%s\nSource = %s\n", statusString,
                source);
        if (status < 0)
        {
            ncCloseObject(MyObjHandle);
            exit(1);
        }
    }
}
```

In some situations, you may want to check for specific errors in your code. For example, when `ncWaitForState` times out, you may want to continue communication, rather than exit the application. To check for specific errors, use the constants defined in `nican.h`. These constants have the same names as described in this manual. For example, to check for a function timeout, use:

```
if (status == CanErrFunctionTimeout)
```

ncWaitForState

Purpose

Wait for one or more states to occur in an object.

Format

```
NCTYPE_STATUS    ncWaitForState(
                              NCTYPE_OBJH ObjHandle,
                              NCTYPE_STATE DesiredState,
                              NCTYPE_UINT32 Timeout,
                              NCTYPE_STATE_P StatePtr)
```

Input

`ObjHandle`	Object handle.
`DesiredState`	States to wait for.
`Timeout`	Length of time to wait in milliseconds.

Output

`StatePtr`	Current state of object when desired states occur. The state is returned to you using the pointer `StatePtr`.

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

You use `ncWaitforState` to wait for one or more states to occur in the object specified by `ObjHandle`.

This function waits up to `Timeout` for one of the bits set in `DesiredState` to become set in the attribute `NC_ATTR_STATE`. You can use the special `Timeout` value `NC_DURATION_INFINITE (FFFFFFFF` hex) to wait indefinitely.

`DesiredState` specifies a bit mask of states for which the wait should return. You can use a single state alone, or you can OR them together.

`NC_ST_READ_AVAIL` (00000001 hex)

At least one frame is available, which you can obtain using an appropriate read function.

The state is set whenever a frame arrives for the object. The state is cleared when the read queue is empty.

NC_ST_WRITE_SUCCESS (00000002 hex)

All frames provided via write function have been successfully transmitted onto the network. Successful transmit means that the frame won arbitration, and was acknowledged by a remote device.

The state is set when the last frame in the write queue is transmitted successfully. The state is cleared when a write function is called.

For CAN Objects, Write Success does not always mean that transmission has stopped. For example, if a CAN Object is configured for Transmit Data Periodically, Write Success occurs when the write queue has been emptied, but periodic transmit of the last frame continues.

When communication starts, the NC_ST_WRITE_SUCCESS state is true by default.

NC_ST_READ_MULT (00000008 hex)

A specified number of frames are available, which you can obtain using ncReadMult. The number of frames is one half the Read Queue Length by default, but you can change it using the ReadMult Size for Notification attribute of ncSetAttribute.

The state is set whenever the specified number of frames are stored in the read queue of the object. The state is cleared when you call the read function, and less than the specified number of frames exist in the read queue.

When the states in DesiredState are detected, the function returns the current value of the NC_ATTR_STATE attribute. If an error occurs, the state returned is zero.

While waiting for the desired states, ncWaitForState suspends the current thread execution. Other Win32 threads in your application can still execute.

If you want to allow other code in your application to execute while waiting for NI-CAN states, refer to the ncCreateNotification function.

ncWrite

Purpose

Write a single frame to an object.

Format

```
NCTYPE_STATUS    ncWrite(
                         NCTYPE_OBJH ObjHandle,
                         NCTYPE_UINT32 DataSize,
                         NCTYPE_ANY_P DataPtr)
```

Input

`ObjHandle`	Object handle.
`DataSize`	Size of the data in bytes.
`DataPtr`	Data written to the object. You provide the data using the pointer `DataPtr`.

Output

Return Value

Status of the function call, returned as a signed 32-bit integer. Zero means the function executed successfully. Negative specifies an error, meaning the function did not perform expected behavior. Positive specifies a warning, meaning the function performed as expected, but a condition arose that might require your attention. For more information, refer to `ncStatusToString`.

Description

`ncWrite` writes a single frame to the object specified by `ObjHandle`.

`DataPtr` points to the variable from which the data is written. Its type is undefined so that you can use the appropriate host data type. `DataSize` indicates the size of variable pointed to by `DataPtr`, and is used to verify that the size you provide is compatible with the configured write size for the object.

You use `ncWrite` to place data into the write queue of an object. Because NI-CAN handles the write queue in the background, this function does not wait for data to be transmitted on the network. To make sure that the data is transmitted successfully after calling `ncWrite`, wait for the `NC_ST_WRITE_SUCCESS` state. The `NC_ST_WRITE_SUCCESS` state transitions from false to true when the write queue is empty, and NI-CAN has successfully transmitted the last data item onto the network. The `NC_ST_WRITE_SUCCESS` state remains true until you write another data item into the write queue.

When communication starts, the `NC_ST_WRITE_SUCCESS` state is true by default.

When you configure an object to transmit data onto the network periodically, it obtains data from the object write queue each period. If the write queue is empty, NI-CAN transmits the data of the previous period again. NI-CAN transmits this data repetitively until the next call to `ncWrite`.

If an object write queue is full, a call to `ncWrite` returns the `CanErrOverflowWrite` error and NI-CAN discards the data you provide. One way to avoid this overflow error is to set the write queue length to zero. When `ncWrite` is called for a zero length queue, the data item you provide with `ncWrite` simply overwrites the previous data item without indicating an overflow. A zero length write queue is often useful when an object is configured to transmit data onto the network periodically, and you simply want to transmit the most recent data value each period. It is also useful when you plan to always wait for `NC_ST_WRITE_SUCCESS` after every call to `ncWrite`. You can use the `NC_ATTR_WRITE_Q_LEN` attribute to configure the write queue length.

For information on the proper data type to use with `DataPtr`, refer to the CAN Network Interface Object and CAN Object descriptions below.

CAN Network Interface Object

The data type that you use with `ncWrite` of the Network Interface is `NCTYPE_CAN_FRAME`. When calling `ncWrite`, you should pass sizeof(`NCTYPE_CAN_FRAME`) for the `DataSize` parameter.

Within the `NCTYPE_CAN_FRAME` structure, the `FrameType` field determines the meaning of all other fields. Tables 9-10 and 9-11 describe the fields of `NCTYPE_CAN_FRAME` for each value of `FrameType`.

Table 9-10. NCTYPE_CAN_FRAME Fields for FrameType NC_FRMTYPE_DATA (0)

Field Name	Data Type	Description
`FrameType`	`NCTYPE_UINT8`	NC_FRMTYPE_DATA (0) Transmit a CAN data frame.
`ArbitrationId`	`NCTYPE_CAN_ARBID`	Specifies the arbitration ID of the frame to transmit. The NCTYPE_CAN_ARBID type is an unsigned 32-bit integer that uses the bit mask NC_FL_CAN_ARBID_XTD (`0x20000000`) to indicate an extended ID. A standard ID (11-bit) is specified by default. In order to specify an extended ID (29-bit), OR in the bit mask NC_FL_CAN_ARBID_XTD.

Table 9-10. NCTYPE_CAN_FRAME Fields for FrameType NC_FRMTYPE_DATA (0) (Continued)

Field Name	Data Type	Description
Data	Array of 8 NCTYPE_UINT8	Specifies the data bytes of the frame.
DataLength	NCTYPE_UINT8	Specifies the number of data bytes to transmit. This number of valid data bytes must be provided in Data.

Table 9-11. NCTYPE_CAN_FRAME fields for FrameType NC_FRMTYPE_REMOTE (1)

Field Name	Data Type	Description
FrameType	NCTYPE_UINT8	NC_FRMTYPE_REMOTE (1) Transmit a CAN remote frame.
ArbitrationId	NCTYPE_CAN_ARBID	Specifies the arbitration ID of the frame to transmit. The NCTYPE_CAN_ARBID type is an unsigned 32-bit integer that uses the bit mask NC_FL_CAN_ARBID_XTD (0x20000000) to indicate an extended ID. A standard ID (11-bit) is specified by default. In order to specify an extended ID (29-bit), OR in the bit mask NC_FL_CAN_ARBID_XTD.
Data	Array of 8 NCTYPE_UINT8	Remote frames do not contain data, so this array is empty.
DataLength	NCTYPE_UINT8	Specifies the Data Length Code to transmit in the remote frame.

CAN Object

The data type that you use with ncWrite of the CAN Object is NCTYPE_CAN_DATA. When calling ncWrite, you should pass sizeof(NCTYPE_CAN_DATA) for the DataSize parameter. Table 9-12 describes the fields of NCTYPE_CAN_DATA.

Table 9-12. NCTYPE_CAN_DATA Field Name

Field Name	Data Type	Description
Data	Array of 8 NCTYPE_UINT8	Data array specifies the data bytes (8 maximum). The actual number of valid data bytes depends on the CAN Object configuration specified in ncConfig. If the CAN Object's Communication Type specifies Receive, data frames are received, not transmitted, so Data always contains zero valid bytes. For this Communication Type, the ncWrite function is used solely for transmission of a remote frame. If the CAN Object's Communication Type specifies Transmit, Data must always contain Data Length valid bytes, where Data Length was configured using ncConfig.

A

Troubleshooting and Common Questions

This appendix describes how to troubleshoot problems with the NI-CAN software and answers some common questions.

Troubleshooting with the Measurement & Automation Explorer (MAX)

MAX contains configuration information for all CAN hardware installed on your system. To start MAX, double-click on the **Measurement & Automation** icon on your desktop. Your NI-CAN cards are listed in the left pane (Configuration) under **Devices and Interfaces.**

You can test your NI-CAN cards by choosing **Tools»NI-CAN»Test all Local NI-CAN Cards** from the menu, or you can right-click on an NI-CAN card and choose **Self Test**. If the Self Test fails, refer to the *Troubleshooting Self Test Failures* section of this appendix.

If there is no **National Instruments CAN Interfaces** item, and you have an NI-CAN card installed, refer to the *Missing NI-CAN Card* section of this appendix.

Missing NI-CAN Card

If you have an NI-CAN card installed, but no NI-CAN card appears in the configuration section of MAX under **Devices and Interfaces,** you need to search for hardware changes by pressing the <F5> key or choosing the **Refresh** option from the **View** menu in MAX.

If the NI-CAN card still doesn't show up, you may have a resource conflict in the Windows Device Manager. Refer to the documentation for your Windows operating system for instructions on how to resolve the problem using the Device Manager.

Troubleshooting Self Test Failures

The following topics explain common error messages generated by the NI-CAN Self Test.

Application In Use

This error occurs if you are running an application that is using the NI-CAN card. The self test aborts in order to avoid adversely affecting your application. Before running the self test, exit all applications that use NI-CAN. If you are using LabVIEW, you may need to exit LabVIEW in order to unload the NI-CAN driver.

Memory Resource Conflict

This error occurs if the memory resource assigned to a CAN card conflicts with the memory resources being used by other devices in the system. Resource conflicts typically occur when your system contains legacy boards that use resources not properly reserved with the Device Manager. If a resource conflict exists, write down the memory resource that caused the conflict and refer to the documentation for your Windows operating system for instructions on how to use the Device Manager to reserve memory resources for legacy boards. After the conflict has been resolved, run the NI-CAN Self Test again.

Interrupt Resource Conflict

This error occurs if the interrupt resource assigned to a CAN card conflicts with the interrupt resources being used by other devices in the system. Resource conflicts typically occur when your system contains legacy boards that use resources not properly reserved with the Device Manager. If a resource conflict exists, write down the interrupt resource that caused the conflict and refer to the documentation for your Windows operating system for instructions on how to use the Device Manager to reserve interrupt resources for legacy boards. After the conflict has been resolved, run the NI-CAN Self Test again.

NI-CAN Software Problem Encountered

This error occurs if the NI-CAN Self Test detects that it is unable to communicate correctly with the CAN hardware using the installed NI-CAN software. If you get this error, shut down your computer, restart it, and run the NI-CAN Self Test again.

If the error continues after restart, uninstall NI-CAN and then reinstall.

NI-CAN Hardware Problem Encountered

This error occurs if the NI-CAN Self Test detects a defect in the CAN hardware. If you get this error, write down the numeric code shown with the error, and contact National Instruments.

Common Questions

How can I determine which version of the NI-CAN software is installed on my system?

Within MAX, select **Help Topics»NI-CAN** within the Help menu. The version is displayed at the top of the help text. The NI-CAN entry provides version information.

How many CAN cards can I configure for use with my NI-CAN software?

The NI-CAN software can be configured to communicate with up to 32 NI-CAN cards on all supported operating systems.

Are interrupts required for the NI-CAN cards?

Yes, one interrupt per card is required. However, PCI and PXI CAN cards can share interrupts with other devices in the system.

How can I use non-standard baud rates?

Open MAX and right-click on the port of the baud rate you want to change. Choose **Properties** and then press the **Advanced** button.

Can I use the Channel API and the Frame API at the same time?

Yes, you can use the Channel API and the Frame API at the same time, but only on different ports. For example, you can use the Frame API on port 1 of a 2-port NI-CAN card and the Channel API on port 2 of that card.

Can high-speed NI-CAN cards and low-speed NI-CAN cards be used on the same network?

No. This is not possible due to different termination requirements of high-speed and low-speed CAN devices. Refer to Appendix B, *Cabling Requirements for High-Speed CAN*, and Appendix C, *Cabling Requirements for Low-Speed CAN*, for more information.

Does the NI-CAN card provide power to the CAN bus?

No. In order to provide power to the CAN bus, you need an external power supply.

Can I use multiple PCMCIA cards in one computer?

Yes, but make sure there are enough free resources available. Unlike PCI or PXI CAN cards, PCMCIA CAN cards cannot share resources, such as IRQs, with other devices.

I have problems with my NI PCMCIA CAN card under Windows NT. How can I resolve them?

Windows NT offers minimal support for plug and play and there are several things to consider.

Since Windows NT does not automatically assign resources to PCMCIA cards, the PCMCIA CAN cards are configured to use default values for the IRQ and the memory range. If those resources are already in use by other devices, it might be necessary to manually change those values.

To do so, right-click on the PCMCIA CAN card in MAX and choose **Properties**. Assign resource values that do not conflict with other device resources for either the Interrupt Request (IRQ) or the Memory Range.

Initially, all NI PCMCIA CAN cards will have the same resources assigned. If you have more than one PCMCIA CAN card installed, the Self Test will fail. You must change the resources of one of the cards manually.

Windows NT does not allow more than one PCMCIA card of the same type installed. Thus, you cannot use two NI PCMCIA CAN/2 cards in the same system. You can however use an NI PCMCIA CAN card and an NI PCMCIA CAN/2 card together.

Why can't I communicate with other devices on the CAN bus, even though the Self Test in MAX passed?

Check the settings for the Power Source Jumper.

The position **EXT** is required for low speed cards; high-speed cards should have it set to **INT**. Refer to Appendix B, *Cabling Requirements for High-Speed CAN*, and Appendix C, *Cabling Requirements for Low-Speed CAN*, for more information.

If the jumper settings are correct, your network may have a cabling or termination problem. Refer again to Appendix B, *Cabling Requirements for High-Speed CAN*, and Appendix C, *Cabling Requirements for Low-Speed CAN*, for more information.

Why are some components left after the NI-CAN software is uninstalled?

The uninstall program removes only items that the installation program installed. If you add anything to a directory that was created by the installation program, the uninstall program does not delete that directory, because the directory is not empty after the uninstallation. You must remove any remaining components yourself.

B

Cabling Requirements for High-Speed CAN

This section describes the cabling requirements for high-speed CAN hardware.

Cables should be constructed to meet these requirements, as well as the requirements of the other CAN or DeviceNet devices in the network.

Connector Pinouts

Depending on the type of CAN interface you are installing, the CAN hardware has DB-9 D-Sub connectors(s), or Combicon-style pluggable screw terminal connector(s), or both.

The 9-pin D-Sub follows the pinout recommended by CiA DS 102. Figure B-1 shows the pinout for this connector.

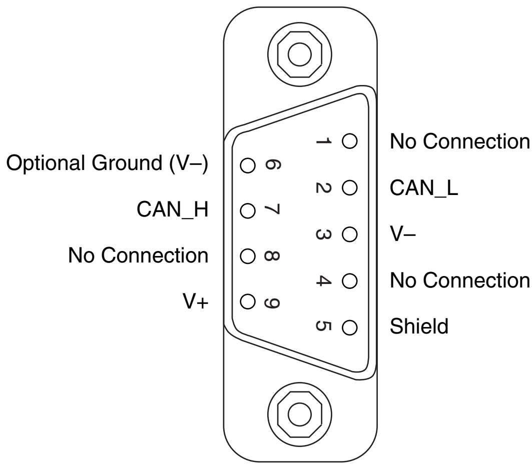
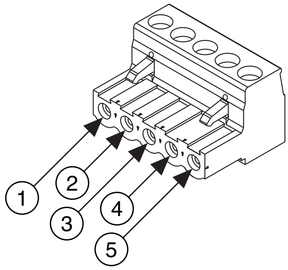


Figure B-1. Pinout for 9-Pin D-Sub Connector

The 5-pin Combicon-style pluggable screw terminal follows the pinout required by the DeviceNet specification. Figure B-2 shows the pinout for this connector.



1	V+	3	Shield	5	V–
2	CAN_H	4	CAN_L		

Figure B-2. Pinout for 5-Pin Combicon-Style Pluggable Screw Terminal

CAN_H and CAN_L are signal lines that carry the data on the CAN network. These signals should be connected using twisted-pair cable.

The V+ and V– pins are used to supply bus power to the CAN physical layer if external power is required for the CAN physical layer. If internal power for the CAN physical layer is used, the V– pin serves as the reference ground for CAN_H and CAN_L. Refer to the next section, *Power Supply Information for the High-Speed CAN Ports*, for more information.

Figure B-3 shows the end of a PCMCIA-CAN cable. The arrow points to pin 1 of the 5-pin screw terminal block. All of the signals on the 5-pin Combicon-style pluggable screw terminal are connected directly to the corresponding pins on the 9-pin D-Sub.

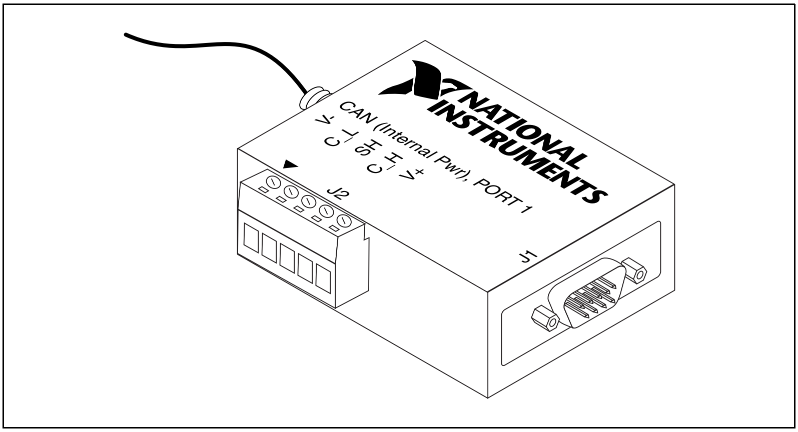
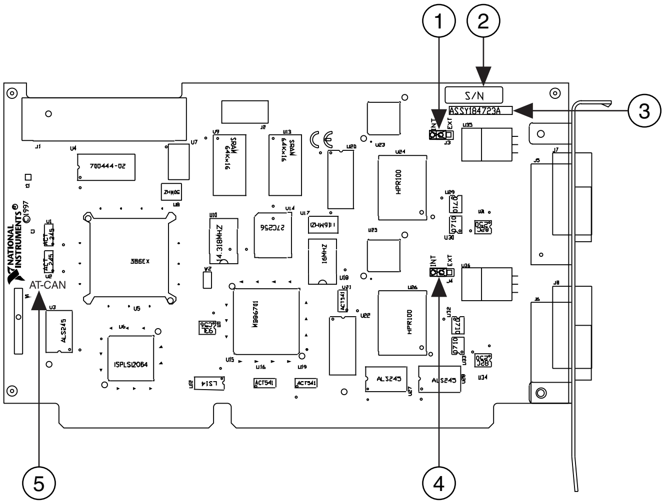


Figure B-3. PCMCIA-CAN Cable

Power Supply Information for the High-Speed CAN Ports

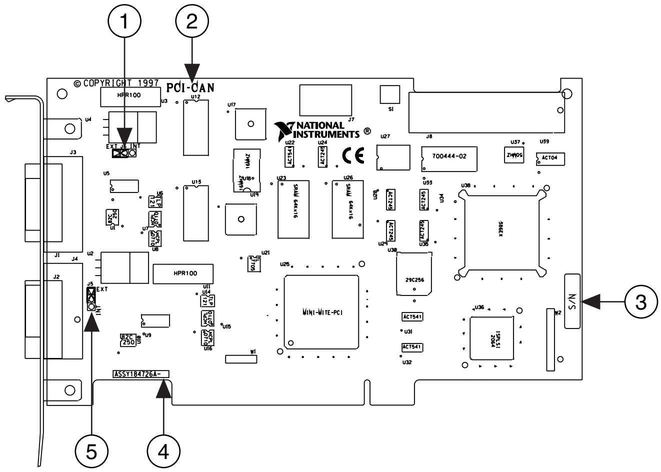
For the PCI-CAN and PXI-846*x* series cards, the power source for the CAN physical layer is configured with a jumper. The location of this jumper is shown in Figure B-4.



1	Power Supply Jumper J3	3	Assembly Number	5	Product Name
2	Serial Number	4	Power Supply Jumper J4		

Figure B-4. Parts Locator Diagram

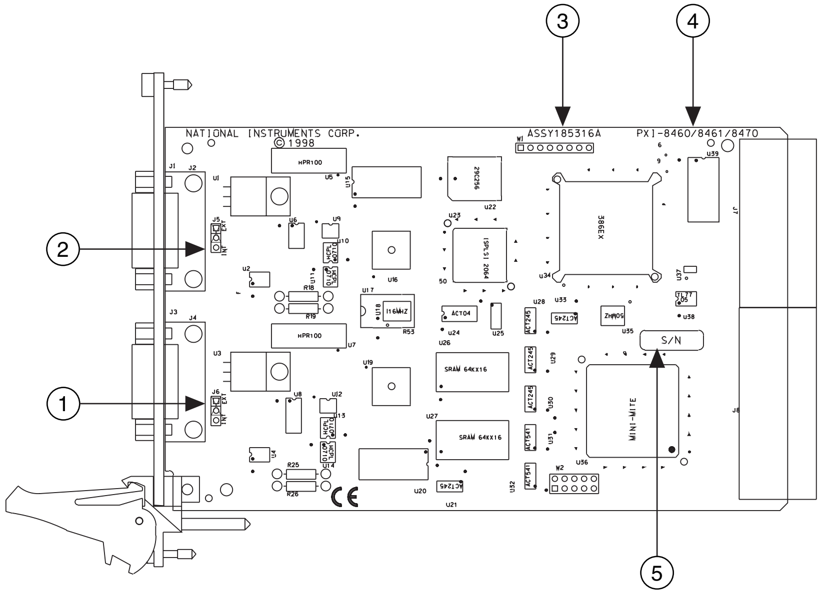
For the PCI-CAN and port one of the PCI-CAN/2 power is configured with jumper J6. For port two of the PCI-CAN/2, power is configured with jumper J5. These jumpers are shown in Figure B-5.



1	Power Supply Jumper J6	3	Serial Number	5	Power Supply Jumper J5
2	Product Name	4	Assembly Number		

Figure B-5. PCI-CAN/2 Parts Locator Diagram

For port one of the PXI-8461, power is configured with jumper J5. For port two of the PXI-8461, power is configured with jumper J6. The location of these jumper is shown in Figure B-6.



1	Power Supply Jumper J6	3	Assembly Number	5	Serial Number
2	Power Supply Jumper J5	4	Product Name		

Figure B-6. PXI-8461 Parts Locator Diagram

Connecting pins 1 and 2 of a jumper configures the CAN physical layer to be powered externally (from the bus cable power). In this configuration, the power must be supplied on the V+ and V– pins on the port connector.

Connecting pins 2 and 3 of a jumper configures the CAN physical layer to be powered internally (from the card). In this configuration, the V– signal serves as the reference ground for the isolated signals.

Figure B-7 shows how to configure your jumpers for internal or external power supplies.

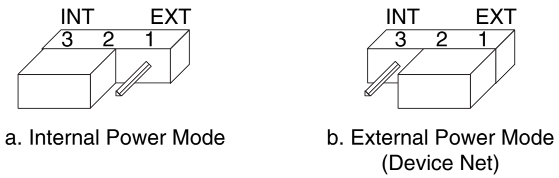


Figure B-7. Power Source Jumpers

The CAN physical layer is still isolated regardless of the power source chosen.

The PCMCIA-CAN series cards are available with two types of cable. The DeviceNet (bus powered) cable requires that the CAN physical layer be powered from the bus cable power.

The internal-powered cable supplies power to the CAN physical layer from the host computer. The V+ pin is not connected to any internal signals, but the corresponding pins on the 9-pin D-Sub and the 5 pin Combicon-style connectors are still connected. The V– pins serves as the reference ground for the isolated signals.

The CAN physical layer is isolated from the computer in both types of cable.

Bus Power Supply Requirements

If the CAN physical layer is powered from a bus power supply, the power supply should be a DC power supply with an output of 10 to 30 V. The power requirements for the CAN ports for Bus-Powered configurations are shown in Table B-1. You should take these requirements into account when determining requirements of the bus power supply for the system.

Table B-1. Power Requirements for the CAN Physical Layer for Bus-Powered Versions

Characteristic	Specification
Voltage requirement	V+ 10–30 VDC
Current requirement	40 mA typical 100 mA maximum

Cable Specifications

Cables should meet the physical medium requirements specified in ISO 11898, shown in Table B-2.

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

Table B-2. 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 allowable cable length is affected by the characteristics of the cabling and the desired bit transmission rates. Detailed cable length recommendations can be found in the ISO 11898, CiA DS 102, and DeviceNet specifications.

ISO 11898 specifies 40 m total cable length with a maximum stub length of 0.3 m for a bit rate of 1 Mb/s. 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.

Table B-3 lists the DeviceNet cable length specifications.

Table B-3. DeviceNet Cable Length Specifications

Bit Rate	Thick Cable	Thin Cable
500 kb/s	100 m	100 m
250 kb/s	200 m	100 m
100 kb/s	500 m	100 m

Number of Devices

The maximum number of devices depends on the electrical characteristics of the devices on the network. If all of the devices meet the requirements of ISO 11898, at least 30 devices may be connected to the bus. Higher numbers of devices may be connected if the electrical characteristics of the devices do not degrade signal quality below ISO 11898 signal level specifications. If all of the devices on the network meet the DeviceNet specifications, 64 devices may be connected to 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 B-8 for an example of where termination resistors should be placed in a system with more than two devices.

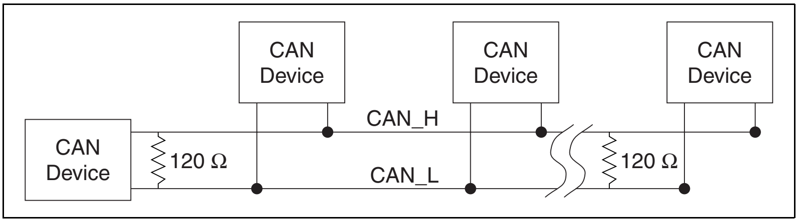


Figure B-8. Termination Resistor Placement

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; therefore, a 120 Ω resistor should be used at each end of the cable. Each termination resistor should be capable of dissipating 0.25 W of power.

Cabling Example

Figure B-9 shows an example of a cable to connect two CAN devices. For the internal power configuration, no V+ connection is required.

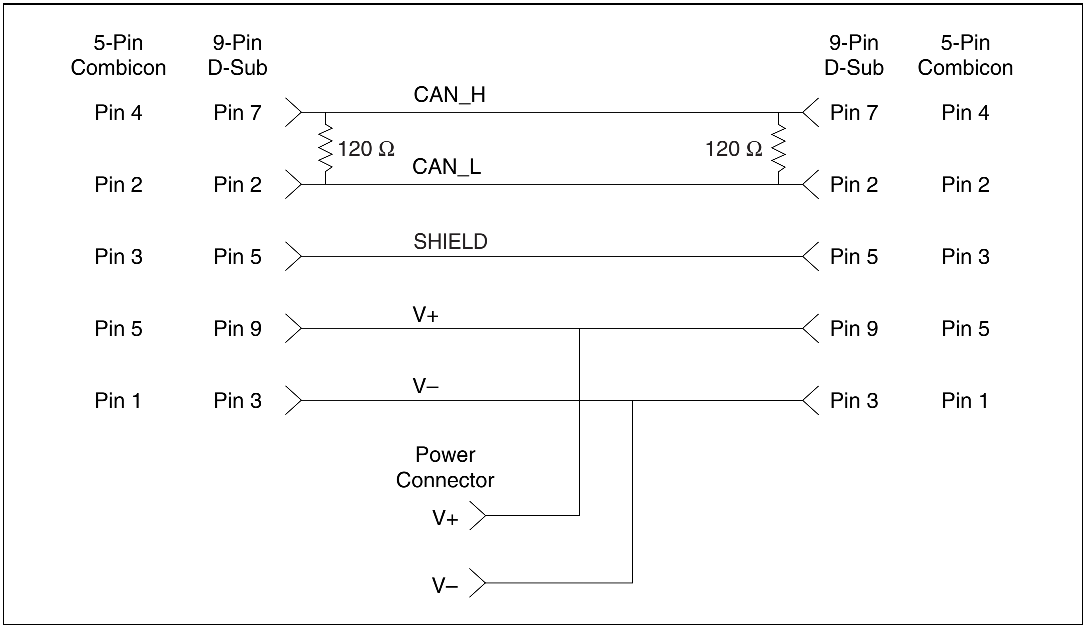


Figure B-9. Cabling Example

C

Cabling Requirements for Low-Speed CAN

This appendix describes the cabling requirements for the low-speed CAN hardware.

Cables should be constructed to meet these requirements, as well as the requirements of the other CAN devices in the network.

Connector Pinouts

The low-speed CAN hardware has DB-9 D-Sub connector(s). The 9-pin D-Sub follows the pinout recommended by CiA DS 102. Figure C-1 shows the pinout for this connector.

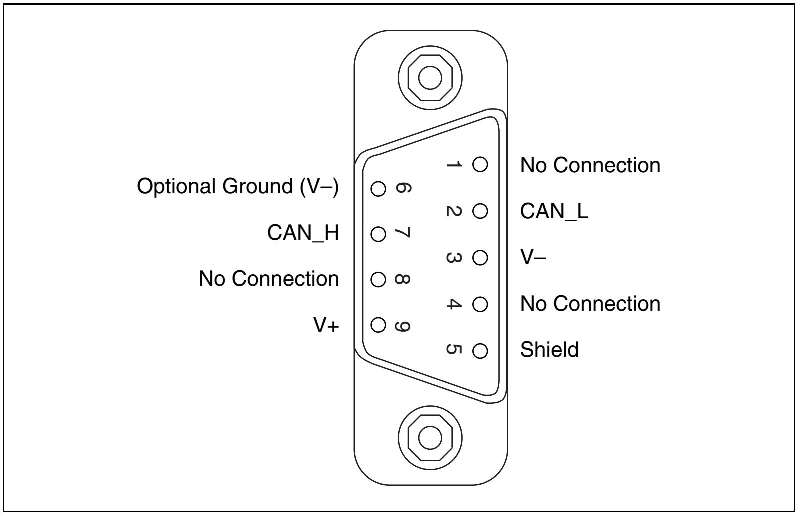


Figure C-1. Pinout for 9-Pin D-Sub Connector

CAN_H and CAN_L are signal lines that carry the data on the CAN network. These signals should be connected using twisted-pair cable.

The V+ and V– pins are used to supply bus power to the CAN physical layer if external power is required for the CAN physical layer. If internal power for the CAN physical layer is used, the V– pin serves as the reference ground for CAN_H and CAN_L. Refer to the next section, *Power Supply Information for the Low-Speed CAN Ports*, for more information.

Figure C-2 shows the end of a PCMCIA-CAN/LS cable. The arrow points to pin 1 of the 7-pin screw terminal block. All of the signals on the 7-pin pluggable screw terminal, except RTL and RTH, are connected directly to the corresponding pins on the 9-pin D-Sub.

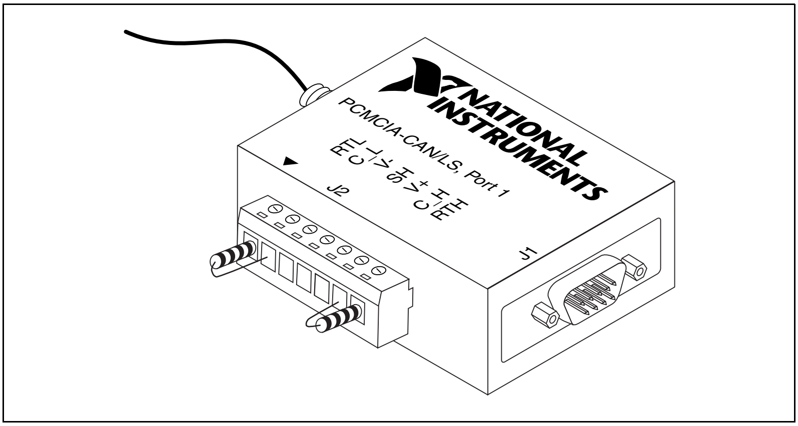
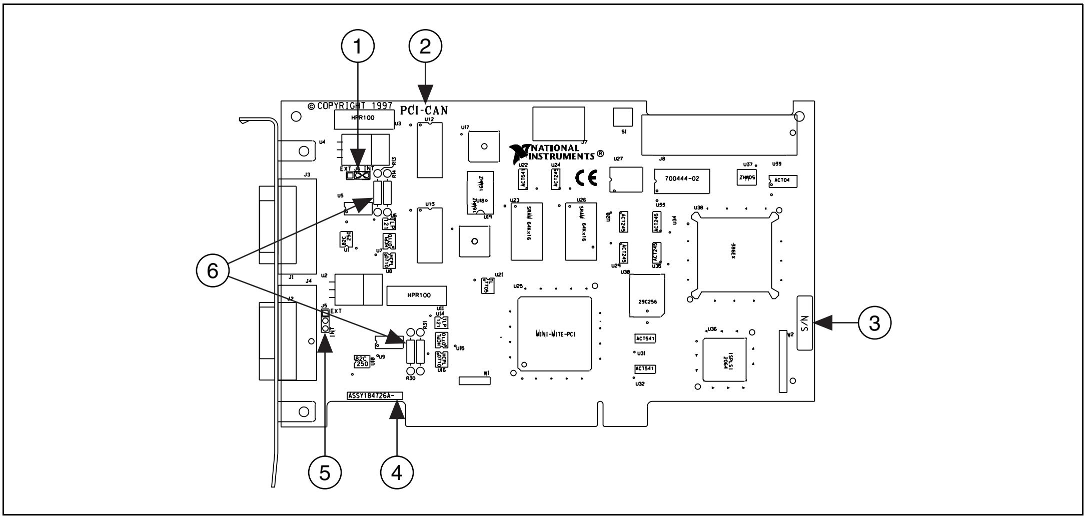


Figure C-2. PCMCIA-CAN/LS Cable

Power Supply Information for the Low-Speed CAN Ports

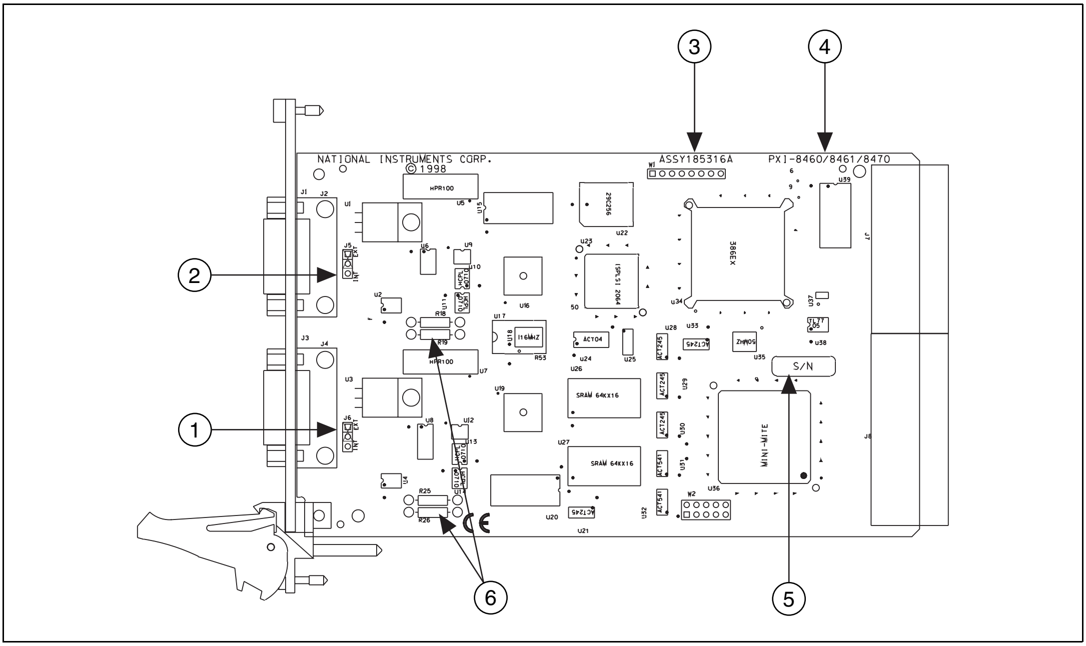
For the PCI-CAN/LS and port one of the PCI-CAN/LS2, power is configured with jumper J6. For port two of the PCI-CAN/LS2, power is configured with jumper J5. These jumpers are shown in Figure C-3.



1	Power Supply Jumper J6	3	Serial Number	5	Power Supply Jumper J5
2	Product Name	4	Assembly Number	6	Termination Resistor Sockets

Figure C-3. PCI-CAN/LS2 Parts Locator Diagram

For port one of the PXI-8460, power is configured with jumper J5. For port two of the PXI-8460, power is configured with jumper J6. These jumpers are shown in Figure C-4.



1	Power Supply Jumper J6	3	Assembly Number	5	Serial Number
2	Power Supply Jumper J5	4	Product Name	6	Termination Resistor Sockets

Figure C-4. PXI-8460 Parts Locator Diagram

Connecting pins 1 and 2 of a jumper configures the CAN physical layer to be powered externally (from the bus cable power). In this configuration, the power must be supplied on the V+ and V– pins on the port connector.

Connecting pins 2 and 3 of a jumper configures the CAN physical layer to be powered internally (from the card). In this configuration, the V– signal serves as the reference ground for the isolated signals. Even if the CAN physical layer is powered internally, the fault-tolerant CAN transceiver still requires bus power to be supplied in order for it to monitor the power supply (battery) voltage.

Figure C-5 shows how to configure your jumpers for internal or external power supplies.

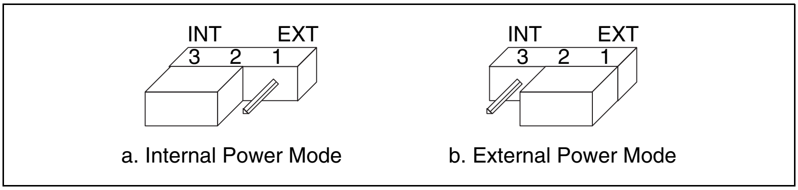


Figure C-5. Power Source Jumpers

The CAN physical layer is still isolated regardless of the power source chosen.

Bus Power Supply Requirements

If the CAN physical layer is powered from a bus power supply, the power supply should be a DC power supply with an output of 8 to 27 V. The power requirements for the CAN ports for Bus-Powered configurations are shown in Table C-1. You should take these requirements into account when determining requirements of the bus power supply for the system.

Table C-1. Power Requirements for the Low-Speed CAN Physical Layer for Bus-Powered Versions

Characteristic	Specification
Voltage requirement	V+ 8–27 VDC
Current requirement	40 mA typical 100 mA maximum

Cable Specifications

Cables should meet the physical medium requirements shown in Table C-2.

Table C-2. 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

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

Number of Devices

The maximum number of devices depends on the electrical characteristics of the devices on the network. If all of the devices meet the requirements of typical low-speed, fault-tolerant CAN, at least 20 devices may be connected to the bus. Higher numbers of devices may be connected if the electrical characteristics of the devices do not degrade signal quality below low-speed, fault-tolerant signal level specifications.

Low-Speed Termination

Every device on the low-speed CAN network requires a termination resistor for each CAN data line: R_{RTH} for CAN_H and R_{RTL} for CAN_L. Figure C-6 shows termination resistor placement in a low-speed CAN network.

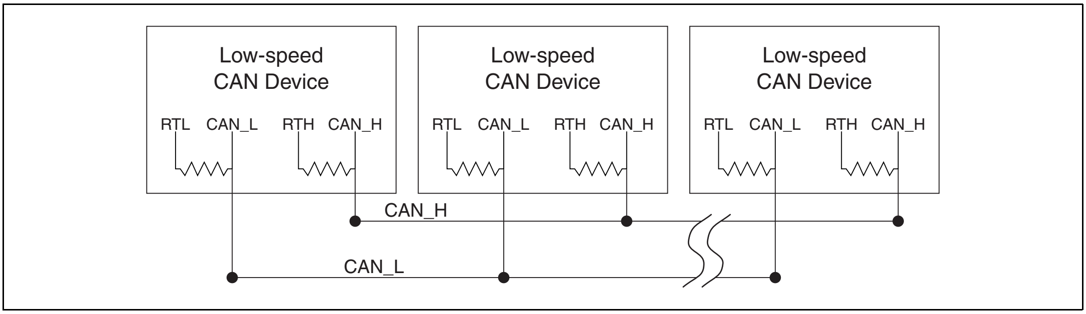


Figure C-6. Termination Resistor Placement for Low-Speed CAN

The following sections explain how to determine the correct resistor values for your low-speed CAN card, and how to replace those resistors, if necessary.

Determining the Necessary Termination Resistance for Your Board

Unlike high-speed CAN, low-speed CAN requires termination at the low-speed CAN transceiver instead of on the cable. The termination requires one resistor: RTH for CAN_H and RTL for CAN_L. This configuration allows the Philips fault-tolerant CAN transceiver to detect any of seven network faults. You can use your PCI-CAN/LS or PXI-8460 to connect to a low-speed CAN network having from two to 32 nodes as specified by Philips (including the port on the PCI-CAN/LS or PXI-8460 as a node). You can also use the PCI-CAN/LS or PXI-8460 to communicate with individual low-speed CAN devices. It is important to determine the overall termination of your existing network, or the termination of your individual device, before connecting it to a PCI-CAN/LS or PXI-8460 port. Philips recommends an overall RTH and RTL termination of 100 to 500 Ω (each) for a properly terminated low-speed network. The overall network termination may be determined as follows:

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

Philips also recommends an individual device RTH and RTL termination of 500 to 16 kΩ. The PCI-CAN/LS or PXI-8460 card ships with mounted RTH and RTL values of 510 Ω ±5% per port. The PCI-CAN/LS or PXI-8460 kit also includes a pair of 15 kΩ ±5% resistors for each port. After determining the termination of your existing network or device, you can use the following formula to indicate which value should be placed on your PCI-CAN/LS or PXI-8460 card in order to produce the proper overall RTH and RTL termination of 100 to 500 Ω upon connection of the card:

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

$*R_{\text{RTH overall}}$ should be between 100 and 500 Ω

$**R_{\text{RTH of low-speed CAN interface}}$ = 510 Ω ±5% (mounted) or 15 kΩ ±5% (in kit)

$\dagger R_{\text{RTH}} = R_{\text{RTL}}$

As the formula indicates, the 510 Ω ±5% shipped on your card will work with properly terminated networks having a total RTH and RTL termination of 125 to 500 Ω, or individual devices having an RTH and RTL termination of 500 to 16 kΩ. For communication with a network having an overall RTH and RTL termination of 100 to 125 Ω, you will need to replace the 510 Ω resistors with the 15 kΩ resistors in the kit. Refer to the next section, *Replacing the Termination Resistors on Your PCI-CAN/LS Board*.

The PCMCIA-CAN/LS cable ships with screw-terminal mounted RTH and RTL values of 510 Ω ±5% per port. The PCMCIA-CAN/LS cable also internally mounts a pair of 15.8 KΩ ±1% resistors in parallel with the external 510 Ω resistors for each port. This produces an effective RTH and RTL of 494 Ω per port for the PCMCIA-CAN/LS cable. After determining the termination of your existing network or device, you can use the formula below to indicate which configuration should be used on your PCMCIA-CAN/LS cable to produce the proper overall RTH and RTL termination of 100 to 500 Ω upon connection of the cable:

$$R_{\text{RTH overall}*,\dagger} = \frac{1}{\left(\frac{1}{R_{\text{RTH of PCMCIA-CAN/LS}**}} + \frac{1}{R_{\text{RTH of existing network or device}}}\right)}$$

*$R_{\text{RTH overall}}$ should be between 100 and 500 Ω

**$R_{\text{RTH of PCMCIA-CAN/LS}}$ = 494 Ω (510 Ω ± 5% (external) in parallel with 15.8 KΩ ± 1% (internal)), or 15.8 KΩ ± 1% (internal) only

†$R_{\text{RTH}} = R_{\text{RTL}}$

As the formula indicates, the 510 Ω ± 5% in parallel with 15.8 KΩ ± 1% shipped on your cable will work with properly terminated networks having a total RTH and RTL termination of 125 to 500 Ω, or individual devices having an RTH and RTL termination of 500 to 16 KΩ. For communication with a network having an overall RTH and RTL termination of 100 to 125 Ω, you will need to disconnect the 510 Ω resistors from the 7-pin pluggable screw terminal. This will make the RTH and RTL values of the PCMCIA-CAN/LS cable equal to the internal resistance of 15.8 KΩ ± 1%. To produce RTH and RTL values between 494 and 15.8 KΩ on the PCMCIA-CAN/LS cable, use the following formula:

$$R_{\text{External RTH of PCMCIA-CAN/LS}^{\dagger}} = \frac{1}{\left(\dfrac{1}{R_{\text{Desired RTH of PCMCIA-CAN/LS}}} - \dfrac{1}{R_{\text{Internal RTH of PCMCIA-CAN/LS***}}}\right)}$$

***$R_{\text{Internal RTH of PCMCIA-CAN/LS}}$ = 15.8 KΩ ± 1%

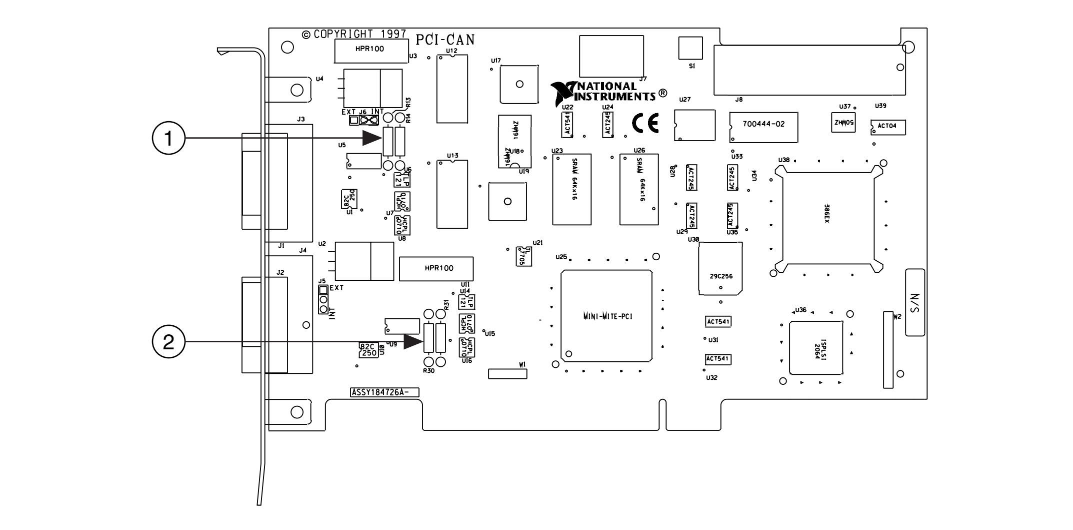
†$R_{\text{RTH}} = R_{\text{RTL}}$

For information on replacing the external RTH and RTL resistors on your PCMCIA-CAN/LS cable, refer to *Replacing the Termination Resistors on Your PCMCIA-CAN/LS Cable*.

Replacing the Termination Resistors on Your PCI-CAN/LS Board

Follow these steps to replace the termination resistors on your PCI-CAN/LS card, after you have determined the correct value in the previous section, *Determining the Necessary Termination Resistance for Your Board*.

1. Remove the termination resistors on your low-speed CAN card. Figure C-7 shows the location of the termination resistor sockets on a PCI-CAN/LS2 card.



1	Port 1 Termination Resistors	2	Port 2 Termination Resistors

Figure C-7. Location of Termination Resistors on PCI-CAN/LS2 Board

2. Cut and bend the lead wires of the resistors you want to install. Refer to Figure C-8.

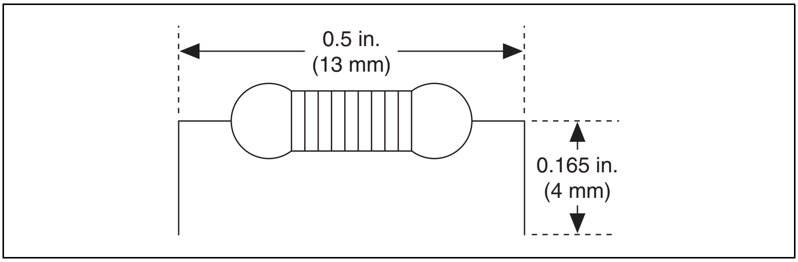


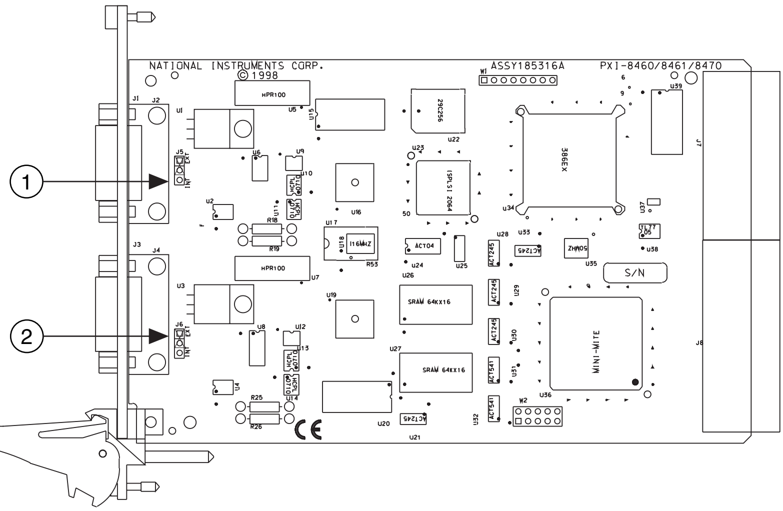
Figure C-8. Preparing Lead Wires of Replacement Resistors

3. Insert the replacement resistors into the empty sockets.
4. Refer to the *CAN Hardware and NI-CAN Software for Windows Installation Guide* in the jewel case of your program CD to complete the hardware installation.

Replacing the Termination Resistors on Your PXI-8460 Board

Follow these steps to replace the termination resistors, after you have determined the correct value in the previous section, *Determining the Necessary Termination Resistance for Your Board*.

1. Remove the termination resistors on your PXI-8460. Figure C-9 shows the location of the termination resistor sockets on a PXI-8460.



1 Port 1 Termination Resistors	2 Port 2 Termination Resistors

Figure C-9. Location of Termination Resistors on a PXI-8460

2. Cut and bend the lead wires of the resistors you want to install. Refer to Figure C-10.

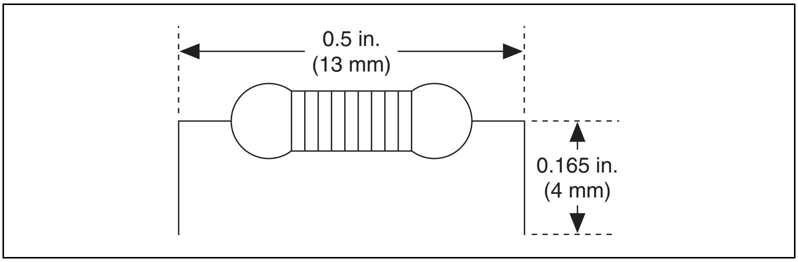


Figure C-10. Preparing Lead Wires of Replacement Resistors

3. Insert the replacement resistors into the empty sockets.
4. Refer to the *CAN Hardware and NI-CAN Software for Windows Installation Guide* in the jewel case of your program CD to complete the hardware installation.

Replacing the Termination Resistors on Your PCMCIA-CAN/LS Cable

Follow these steps to replace the termination resistors on your PCMCIA-CAN/LS cable after you have determined the correct value in the *Determining the Necessary Termination Resistance for Your Board* section.

1. Remove the two termination resistors on your PCMCIA-CAN/LS cable by loosening the pluggable terminal block mounting screws for pins 1 and 2 (RTL) and pins 6 and 7 (RTH).
2. Bend and cut the lead wires of the two resistors you want to install, as shown Figure C-11.

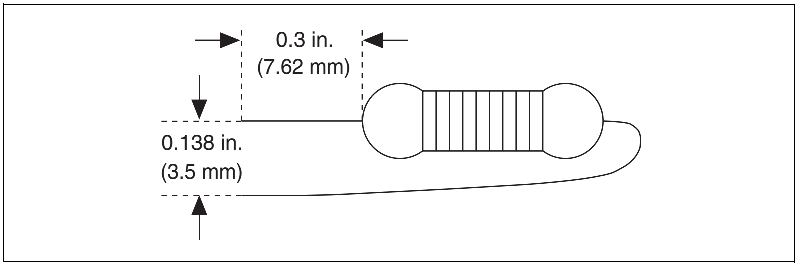


Figure C-11. Preparing Lead Wires of PCMCIA-CAN/LS Cable Replacement Resistors

3. Mount RTL by inserting the leads of one resistor into pins 1 and 2 of the pluggable terminal block and tightening the mounting screws. Mount RTH by inserting the leads of the second resistor into pins 6 and 7 of the pluggable terminal block and tightening the mounting screws.
4. Refer to the *CAN Hardware and NI-CAN Software for Windows Installation Guide* in the jewel case of your program CD to complete the hardware installation.

Cabling Example

Figure C-12 shows an example of a cable to connect two low-speed CAN devices. For the PCMCIA-CAN/LS cables, only V–, CAN_L, and CAN _H are required to be connected to the bus.

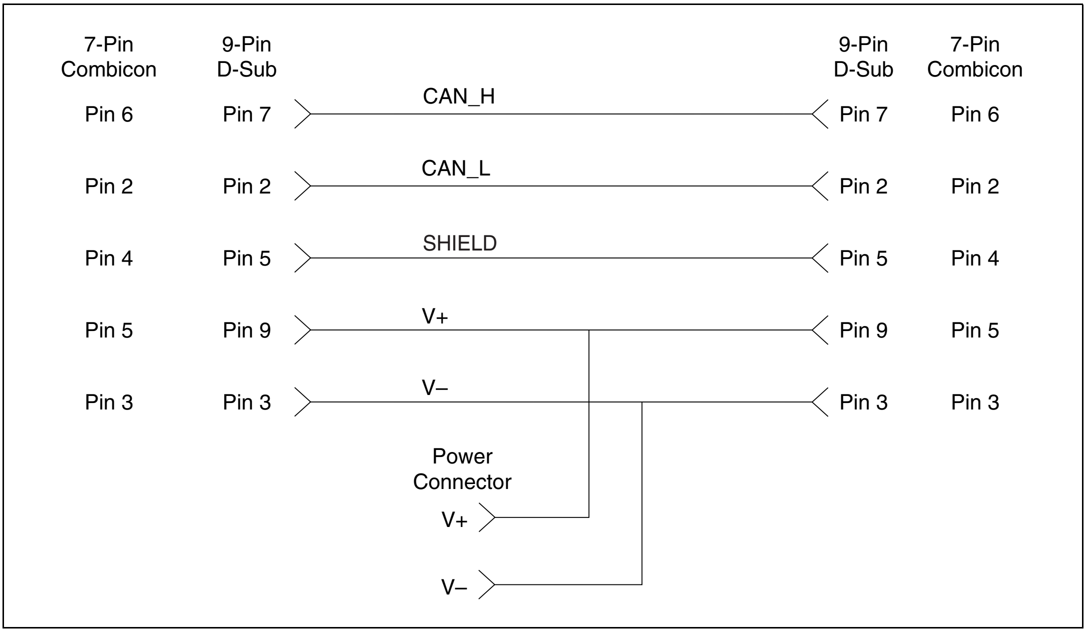


Figure C-12. Cabling Example

D

Cabling Requirements for Dual-Speed CAN

This section describes the cabling requirements for the dual-speed CAN hardware.

Port Identification

The PCI-CAN/DS card, PXI-8462 card, and PCMCIA-CAN/DS cable each provide a high-speed CAN port (port one), and a low-speed CAN port (port two). Port one of the PCI-CAN/DS is identical to port one of the PCI-CAN and PCI-CAN/2, and port two is identical to port two of the PCI-CAN/LS2.

Port one of the PXI-8462 is identical to port one of the PXI-8461 one-port and PXI-8461 two-port cards. Port two of the PXI-8462 is identical to port two of the PXI-8460 two-port card.

Port one of the PCMCIA-CAN/DS cable is identical to port one of the PCMCIA-CAN and PCMCIA-CAN/2 cables, and port two is identical to port two of the PCMCIA-CAN/LS2 cable. The PCI-CAN/DS card, PXI-8462 card and PCMCIA-CAN/DS cable allow simultaneous communication with both a high-speed and low-speed bus, each with its own specific cabling and termination requirements. For cabling requirements and port information for the high-speed CAN port, refer to Appendix B, *Cabling Requirements for Dual-Speed CAN*, in this manual. For cabling requirements and port information for the low-speed CAN port, refer to Appendix C, *Cabling Requirements for Low-Speed CAN*.

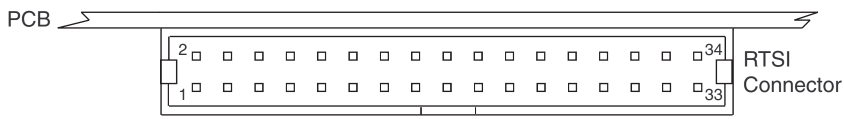
E

RTSI Bus

This appendix describes the RTSI interface on your CAN card.

RTSI and PCI

Figure E-1 shows the RTSI connector pinout for the PCI-CAN series cards.



PCI-CAN Series Trigger	PCI-CAN Series Pin Number
RTSI Trigger <0>	20
RTSI Trigger <1>	22
RTSI Trigger <2>	24
RTSI Trigger <3>	26
RTSI Trigger <4>	28
RTSI Trigger <5>	30
RTSI Trigger <6>	32
RTSI Oscillator	34
GND	19, 21, 23, 25, 27, 29, 31, 33

Figure E-1. PCI-CAN Series RTSI Connector Pinout

Using the National Instruments RTSI bus with your CAN card consists of connecting it to other RTSI-equipped cards with RTSI ribbon cable, to route timing and trigger signals between the cards. Using the RTSI bus, your CAN card can be synchronized with multiple National Instruments DAQ cards in your computer. The RTSI bus can also be used to synchronize multiple CAN cards.

The PCI-CAN and PCI-CAN/2 cards allow for the connection of four RTSI input signals and four RTSI out put signals. In order to fully support the fault reporting capabilities of the low-speed transceivers used on the PCI-CAN/LS, PCI-CAN/LS2, and PCI-CAN/DS, three RTSI lines on those cards are reserved for low-speed CAN fault reporting. This allows for the connection of three RTSI input signals and two RTSI output signals to the cards, providing them the real time synchronization benefits of RTSI without sacrificing low-speed CAN fault reporting.

RTSI, PXI and CompactPCI

Using PXI-compatible products with standard CompactPCI products is an important feature provided by the *PXI Specification*, Revision 1.0. If you use a PXI-compatible plug-in device in a standard CompactPCI chassis, you will be unable to use PXI-specific functions, but you can still use the basic plug-in device functions. For example, the RTSI bus on your PXI-846*x* series card is available in a PXI chassis, but not in a CompactPCI chassis. The CompactPCI specification permits vendors to develop sub-buses that coexist with the basic PCI interface on the CompactPCI bus. Compatible operation is not guaranteed between CompactPCI devices with different sub-buses nor between CompactPCI devices with sub-buses and PXI. The standard implementation for CompactPCI does not include these sub-buses. Your PXI-846*x* device will work in any standard CompactPCI chassis adhering to the *PICMG 2.0 R2.1 CompactPCI* core specification using the 64-bit definition for J2. PXI specific features are implemented on the J2 connector of the CompactPCI bus. Table E-1 lists the J2 pins your PXI-846*x* series card uses. Your PXI card is compatible with any CompactPCI chassis with a sub-bus that does not drive these lines. Even if the sub-bus is capable of driving these lines, the card is still compatible as long as those pins on the sub-bus are disabled by default and not ever enabled. Damage may result if these lines are driven by the sub-bus.

The PXI-8461 one-port and two-port cards allow for the connection of four RTSI input signals and four RTSI output signals. In order to fully support the fault reporting capabilities of the low-speed transceivers used on the PXI-8460 one port, PXI-8460 two port, and PXI-8462, three RTSI lines on those cards are reserved for low-speed CAN fault reporting. This allows for the connection of three RTSI input signals and two RTSI output signals to the cards, providing them the real time synchronization benefits of RTSI without sacrificing low-speed CAN fault reporting.

Table E-1. Pins Used By the PXI-846*x* Series Boards

PXI Pin Name	PXI J2 Pin Number
PXI Star	D17
PXI Trigger <0>	B16
PXI Trigger <1>	A16
PXI Trigger <2>	A17
PXI Trigger <3>	A18
PXI Trigger <4>	B18
PXI Trigger <5>	C18
PXI Trigger <7>	E16

RTSI Cables

National Instruments offers a variety of RTSI bus cables for connecting your CAN card to other CAN or DAQ hardware. For more specific information about these cables, you can refer to the National Instruments catalog, or our Web site `ni.com`.

If you searched `ni.com` and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of `ni.com/niglobal` to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

RTSI Programming

For more information on RTSI programming, refer to the *Synchronization* section of Chapter 4, *Using the Channel API*, and the *RTSI* section of Chapter 7, *Using the Frame API*. Refer to the *RTSI Bus Overview* section of Chapter 1, *Introduction*, for more information on the RTSI bus.

F

Summary of the CAN Standard

History and Use of CAN

In the past few decades, the need for improvements in automotive technology has led to increased use of electronic control systems for functions such as 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 has become 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, and CAN chips were created by major semiconductor manufacturers such as Intel, Motorola, and Philips. With these developments, many 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 many 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 industrial applications such as textile machinery, packaging machines, and production line equipment such as photoelectric sensors and motion controllers. By the mid-1990s, CAN was specified as the basis of many industrial device networking protocols, including DeviceNet, and CANopen.

With its growing popularity in automotive and industrial applications, CAN has been increasingly used in a wide variety of diverse applications. Use in systems such as agricultural equipment, nautical machinery, medical apparatus, semiconductor manufacturing equipment, and machine tools testify to the incredible 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 F-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 cease 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.

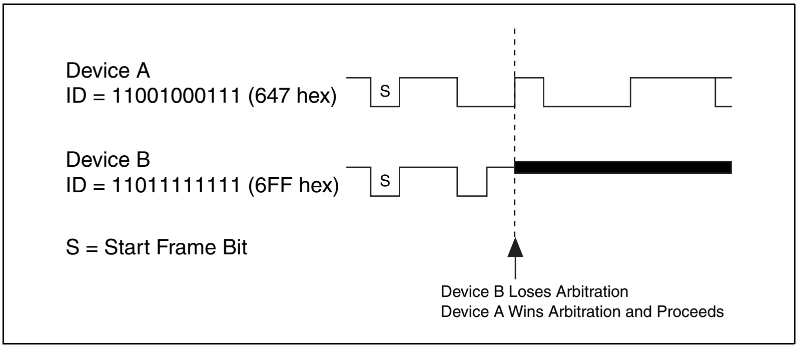


Figure F-1. Example of CAN Arbitration

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 F-2 shows the essential fields of the standard and extended frame formats, and the following sections describe each field.

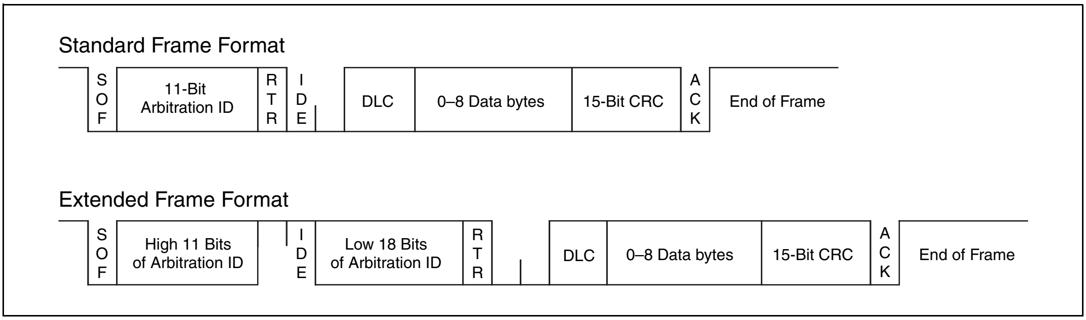


Figure F-2. Standard and Extended Frame Formats

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 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 is again possible.

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, error passive, and bus off.

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. Since 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 is that 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 your 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 is used to disable a malfunctioning CAN device which frequently transmits invalid frames, so that the device does not adversely impact 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-CAN 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 Appendix C, *Cabling Requirements for Low-Speed CAN*.

Because the low-speed transceiver switches to a fault tolerant mode on fault detection and continues to maintain communications, NI-CAN provides a special attribute, Log Comm Warnings, which when set to true enables the reporting of such warnings in the Read queue of the Network Interface rather than in the status returned from a function call. The default value of this attribute is false, which enables the reporting of low-speed transceiver warnings in the status returned from a function call.

Appendix G

Specifications

This appendix describes the physical characteristics of the CAN hardware, along with the recommended operating conditions.

PCI-CAN Series

Dimensions	10.67 by 17.46 cm (4.2 by 6.9 in.)
Power requirement	+5 VDC, 775 mA typical
I/O connector	9-pin D-Sub for each port (standard) or 5-pin Combicon-style pluggable DeviceNet screw terminal (high-speed CAN only)
Operating environment	
Ambient temperature	0 to 55 °C
Relative humidity	10 to 90%, noncondensing
Storage environment	
Ambient temperature	–20 to 70 °C
Relative humidity	5 to 90%, noncondensing

PCMCIA-CAN Series

Dimensions	8.56 by 5.40 by 0.5 cm (3.4 by 2.1 by 0.4 in.)
Power requirement	500 mA typical
I/O connector	Cable with 9-pin D-Sub and pluggable screw terminal for each port

Operating environment

Ambient temperature .... 0 to 55 °C

Relative humidity .... 10 to 90%, noncondensing

Storage environment

Ambient temperature .... –20 to 70 °C

Relative humidity .... 5 to 90%, noncondensing

PXI-CAN Series

Dimensions .... 16.0 by 10.0 cm
(6.3 by 3.9 in.)

Power requirement................................. +5 VDC, 775 mA typical

I/O connector .... 9-pin D-Sub for each port
(standard)
or
5-pin Combicon-style pluggable
DeviceNet screw terminal
(high-speed CAN only)

Operating environment

Ambient temperature .... 0 to 55 °C

Relative humidity .... 10 to 90%, noncondensing

Storage environment

Ambient temperature .... –20 to 70 °C

Relative humidity .... 5 to 95%, noncondensing

(Tested in accordance with IEC-60068-2-1, IEC-60068-2-2, IEC-60068-2-56.)

Functional Shock .... 30 g peak, half-sine, 11ms pulse

(Tested in accordance with IEC-60068-2-27. Test profile developed in accordance with MIL-T-28800E.)

Random Vibration

Operating .... 5 to 500 Hz, 0.3 grms

Nonoperating .... 5 to 500 Hz, 2.4 grms

(Tested in accordance with IEC-60068-2-64. Nonoperating test profile developed in accordance with MIL-T-28800E and MIL-STD-810E Method 514.)

High-Speed CAN Port Characteristics

Bus power .... 0 to 30 V, 40 mA typical, 100mA maximum

CAN-H, CAN-L.................................... –8 to +18V, DC or peak, CATI

Low-Speed CAN Port Characteristics

Bus Power .... 8 to 27 V, 40 mA typical, 100 mA maximum

CAN-H, CAN-L.................................... –10 to +27V, DC or peak, CATI

Safety

The NI-CAN hardware meets the requirements of the following standards for safety and electrical equipment for measurement, control, and laboratory use:

- EN 61010-1, IEC 61010-1
- UL 3111-1, UL 3121-1
- CAN/CSA C22.2 No. 1010.1

Electromagnetic Compatibility

EMC/EMI... CE, C-Tick, and FCC Part 15 (Class A) Compliant

Electrical emissions............................... EN 55011 Class A at 10 m FCC Part 15A above 1 GHz

Electrical immunity................................ Evaluated to EN 61326:1997/ A1:1998, Table 1

Note For full EMC compliance, you *must* operate this device with shielded cabling. In addition, all covers and filler panels *must* be installed. Refer to the Declaration of Conformity (DoC) for this product for any additional regulatory compliance information. To obtain the DoC for this product, click **Declaration of Conformity** at `ni.com/hardref.nsf/`. This Web site lists the DoCs by product family. Select the appropriate product family, followed by your product, and a link to the DoC appears in Adobe Acrobat format. Click the Acrobat icon to download or read the DoC.

H

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at `ni.com` for technical support and professional services:

- **Support**—Online technical support resources include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit our extensive library of technical support resources available in English, Japanese, and Spanish at `ni.com/support`. These resources are available for most products at no cost to registered users and include software drivers and updates, a KnowledgeBase, product manuals, step-by-step troubleshooting wizards, hardware schematics and conformity documentation, example code, tutorials and application notes, instrument drivers, discussion forums, a measurement glossary, and so on.
  - **Assisted Support Options**—Contact NI engineers and other measurement and automation professionals by visiting `ni.com/ask`. Our online system helps you define your question and connects you to the experts by phone, discussion forum, or email.
- **Training**—Visit `ni.com/custed` for self-paced tutorials, videos, and interactive CDs. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit `ni.com/alliance`.

If you searched `ni.com` and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of `ni.com/niglobal` to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Glossary

Prefix	Meanings	Value
n-	nano-	10^{-9}
m-	milli-	10^{-3}
k-	kilo-	10^{3}
M-	mega-	10^{6}

A

action	*See* method.
actuator	A device that uses electrical, mechanical, or other signals to change the value of an external, real-world variable. In the context of device networks, actuators are devices that receive their primary data value from over the network; examples include valves and motor starters. Also known as *final control element*.
Application Programming Interface (API)	A collection of functions used by a user application to access hardware. Within NI-CAN, you use API functions to make calls into the NI-CAN driver. NI-CAN provides two different APIs: the Frame API and Channel API.
arbitration ID	An 11- or 29-bit ID transmitted as the first field of a CAN frame. The arbitration ID determines the priority of the frame, and is normally used to identify the data transmitted in the frame.
attribute	The Frame API provides attributes to access configuration settings or other information. In the Channel API, the term *property* is used for similar settings.

B

b	Bits.
Behavior After Final Output	Property in the Channel API that specifies the behavior to perform after the final periodic output sample is transmitted. For more information, refer to **CAN Set Property.vi** for LabVIEW, or `nctSetProperty` for C.
bus off	A CAN node goes into the bus off state when its transmit error counter increments above 255. The node does not participate in network traffic, because it assumes that a defect exists that must be corrected.

C

CAN	Controller Area Network.
CAN Channels	*See* channel.
CAN controller	Communications ship used to transmit and receive frames on a CAN network. The majority of the CAN specification is implemented within the CAN controller. Examples of CAN controllers include the Intel 82527 and the Philips SJA1000.
CANdb	CAN database format defined by Vector Informatik. CANdb files use the `.dbc` file extension.
CAN database	Database file that describes channels and associated messages for a collection of CAN nodes. NI-CAN supports two CAN database formats: CANdb, and the NI-CAN database.
CAN data frame	Frame used to transmit the actual data of a CAN Object. The RTR bit is clear, and the data length indicates the number of data bytes in the frame.
CAN frame	In addition to fields used for error detection/correction, a CAN frame consists of an arbitration ID, the RTR bit, a four-bit data length, and zero to eight bytes of data.
CAN/LS	*See* Low-speed CAN.
CAN Network Interface Object	Within the NI-CAN Frame API, an object that encapsulates a CAN interface on the host computer.

CAN Object	Within the NI-CAN Frame API, an object that encapsulates a specific CAN arbitration ID along with its raw data bytes.
CAN remote frame	Frame used to request data for a CAN Object from a remote node; the RTR bit is set, and the data length indicates the amount of data desired (but no data bytes are included).
channel	Floating-point value in physical units (such as Volts, rpm, km/h, °C, and so on) that is converted to/from a raw value in measurement hardware. The NI-CAN Channel API's `Read` and `Write` functions provide access to CAN channels. When a CAN message is received, NI-CAN converts raw fields in the message into physical units, which you then obtain using the Channel API Read function. When you call a Channel API Write function, you provide floating-point values in physical units, which NI-CAN converts into raw fields and transmits as a CAN message. For an example usage of the channel concept, refer to the Channel API section in Introduction.
Channel API	NI-CAN API that you use to read and write channels.
channel list	Input parameter of the CAN Init Start function. The channel list specifies the list of channels to read or write. For more information, refer to **CAN Init Start.vi** for LabVIEW, or `nctInitStart` for C.
ChannelList	*See* channel list.
class	A set of objects that share a common structure and a common behavior.
clock drift	When two or more hardware products are used to measure a common system, you typically need to compare data from the hardware products simultaneously. Since each hardware product contains its own local oscillator to perform measurements, and all oscillators differ slightly in speed and tolerances, measurements on different hardware products can drift relative to one another. For example, if you measure the same sine wave on two different analog-input products, the measured sine waves typically drift out of phase after a few minutes. National Instruments products use RTSI to share timebases among different hardware products. Since the products share the same oscillator, clock drift is eliminated.

connection	With respect to networking, this term refers to an association between two or more nodes on a network that describes when and how data is transferred. With respect to RTSI, this term refers to a connection between two or more terminals.
controller	With respect to CAN, this term often refers to a CAN controller. With respect to real-time systems, this term refers to a device that receives input data and sends output data in order to hold one or more external, real-world variables at a certain level or condition. A thermostat is a simple example of a controller.

D

Default Value	Property in the Channel API that specifies the default value for a channel. For more information, refer to **CAN Get Property.vi** for LabVIEW, or `nctSetProperty` for C.
device	*See* node.
device network	Multi-drop digital communication network for sensors, actuators, and controllers.
DLL	Dynamic link library.
DMA	Direct memory access.

E

error active	A CAN node is in error active state when both the receive and transmit error counters are below 128.
error counters	Every CAN node keeps a count of how many receive and transmit errors have occurred. The rules for how these counters are incremented and decremented are defined by the CAN protocol specification.
error passive	A CAN node is in error passive state when one or both of its error counters increment above 127. This state is a warning that a communication problem exists, but the node is still participating in network traffic.

extended arbitration ID	A 29-bit arbitration ID. Frames that use extended IDs are often referred to as CAN 2.0 Part B (the specification that defines them).

F

FCC	Federal Communications Commission.

filepath	Complete path to a filename using Windows conventions, such as:
	`C:\Program Files\National Instruments\NI-CAN\`
	`MyDatabase.ncd`

frame	A unit of information transferred across a network from one node to another. From an OSI perspective, NI-CAN's usage of the term frame refers to a Data Link Layer unit, because individual fields are not specified.
Frame API	NI-CAN API that you use to read and write frames.

H

hex	Hexadecimal.
Hz	Hertz; cycles per second.

I

instance	An abstraction of a specific real-world thing; for example, John is an instance of the class Human. Also known as *object*.
Interface Baud Rate	Property in the Channel API that specifies the baud rate of the interface. For more information, refer to **CAN Set Property.vi** for LabVIEW, or `nctSetProperty` for C.

interface — Reference to a specific CAN port in the NI-CAN software. NI-CAN interface names are assigned within MAX, and can range from **CAN0** to **CAN63**.

In the Channel API, the interface is specified during initialization of the task. For more information, refer to **CAN Init Start.vi** for LabVIEW, or `nctInitStart` for C.

In the Frame API, the interface is specified during configuration of the CAN Network Interface Object. For more information, refer to **ncConfigCANNet.vi** for LabVIEW, or `ncConfig` for C.

Interface — *See* interface.

ISO — International Standards Organization.

K

KB — Kilobytes of memory.

L

LabVIEW — Laboratory Virtual Instrument Engineering Workbench.

local — Within NI-CAN, anything that exists on the same host (personal computer) as the NI-CAN driver.

Low-speed CAN — Fault-tolerant CAN transceiver specification as defined in ISO 11898.

M

MAX — The Measurement & Automation Explorer provides a centralized location for configuration of National Instruments hardware products. MAX also provides many useful tools for interaction with hardware.

MB — Megabytes of memory.

message — CAN data frame for which the individual fields are described. From an OSI perspective, NI-CAN usage of the term frame refers to a User Layer unit, because the Application Layer is assumed (simple peer-to-peer protocol), and the channel configurations specify User Layer meaning.

method	An action performed on an instance to affect its behavior; the externally visible code of an object. Within NI-CAN, you use NI-CAN functions to execute methods for objects. Also known as *service*, *operation*, and *action*.
minimum interval	For a given connection, the minimum amount of time between subsequent attempts to transmit frames on the connection. Some protocols use minimum intervals to guarantee a certain level of overall network performance.
mode	Input parameter of the CAN Init Start function. The mode specifies the direction of data transfer (input or output), and the type of information provided (input or timestamped input). For more information, refer to **CAN Init Start.vi** for LabVIEW, or `nctInitStart` for C.
Mode	*See* mode.
multi-drop	A physical connection in which multiple devices communicate with one another along a single cable.

N

network interface	A node's physical connection onto a network.
NI-CAN database	CAN database format defined by National Instruments. NI-CAN database files use the `.ncd` file extension.
NI-CAN driver	Device driver and/or firmware that implement all the specifics of a CAN network interface. Within NI-CAN, this software implements the CAN Network Interface Object as well as all objects above it in the object hierarchy.
node	A physical assembly, linked to a communication line (cable), capable of communicating across the network according to a protocol specification. Also known as *device*.
notification	Within NI-CAN, an operating system mechanism that the NI-CAN driver uses to communicate events to your application. You can think of a notification of as an API function, but in the opposite direction.

O

object	*See* instance.
object-oriented	A software design methodology in which classes, instances, attributes, and methods are used to hide all of the details of a software entity that do not contribute to its essential characteristics.
OSI	Open Systems Interconnection (OSI) is a collection of ISO standards for communication protocols. Most people reference OSI in the context of the layers that it specifies for all communication protocols. The Physical Layer refers to physical connectors, cabling, and signal characteristics. The Data Link Layer refers to the fundamental frame format. The Application Layer refers to connection establishment and other higher-level transactions between nodes. The User Layer is an informal term that refer to the definition of specific fields in Application Layer messages that define how an application uses the protocol.

P

peer-to-peer	Network connection in which data is transmitted from the source to its destination(s) without need for an explicit request. Although data transfer is generally unidirectional, the protocol often uses low level acknowledgments and error detection to ensure successful delivery.
periodic	Connections that transfer data on the network at a specific rate.
polled	Request/response connection in which a request for data is sent to a device, and the device sends back a response with the desired value.
poly VI	LabVIEW VI that accepts different data types for a single input or output terminal. In some cases, the data type can be selected based on the value that you wire to the poly input or output. To select a specific poly VI type, right-click the VI, go to **Select Type**, and select the desired type. For more information, refer to your LabVIEW documentation. Like many other National Instruments APIs, the NI-CAN Channel API implements Read and Write as poly VIs in order to support a variety of data types.
polymorphic VI	*See* poly VI.

port	The physical CAN connector on your NI-CAN hardware product. You assign an interface name to each port using MAX.
property	The Channel API provides properties to access configuration settings or other information. LabVIEW also uses the term property for settings of front panel controls and indicators. In the Frame API, the term attribute is used for similar settings.
property nodes	In LabVIEW, you can use property nodes to change the appearance or behavior of front panel controls and indicators. For example, you can change the label, minimum value, and maximum value of an indicator. For more information, refer to your LabVIEW documentation.
protocol	A formal set of conventions or rules for the exchange of information among nodes of a given network.

R

RAM	Random-access memory.
remote	Within NI-CAN, anything that exists in another node of the device network (not on the same host as the NI-CAN driver).
Remote Transmission Request (RTR) bit	This bit follows the arbitration ID in a frame, and indicates whether the frame is the actual data of the CAN Object (CAN data frame), or whether the frame is a request for the data (CAN remote frame).
request/response	Network connection in which a request is transmitted to one or more destination nodes, and those nodes send a response back to the requesting node. In industrial applications, the responding (slave) device is usually a sensor or actuator, and the requesting (master) device is usually a controller. Also known as *master/slave*.
resource	Hardware settings used by National Instruments CAN hardware, including an interrupt request level (IRQ) and an 8 KB physical memory range (such as D0000 to D1FFF hex).
RTSI	Real Time System Integration bus. National Instruments technology that can be used to synchronize multiple hardware products. For PCI products, this refers to the ribbon cable that is used to route signals between cards. For PXI products, the RTSI signals are provided on the backplane. For PCMCIA products, RTSI signals can be connected between a CAN card's sync cable and a DAQ card's terminal block.

S

s	Seconds.
sample	A floating-point value that represents physical units. In the NI-CAN Channel API, you Read and Write samples using channels.
sample rate	Input parameter of the CAN Init Start function. The sample rate specifies whether to transfer data in a periodic or event-driven manner. For periodic behavior, the rate specifies the number of read/write samples to perform per second. For more information, refer to **CAN Init Start.vi** for LabVIEW, or `nctInitStart` for C.
SampleRate	*See* sample rate.
sensor	A device that measures electrical, mechanical, or other signals from an external, real-world variable; in the context of device networks, sensors are devices that send their primary data value onto the network; examples include temperature sensors and presence sensors. Also known as *transmitter*.
signal	Term used by other vendors of CAN products to refer to a CAN channel. For National Instruments products, this term usually refers to a physical voltage that represents a predefined behavior. For example, RTSI connections are used to exchange signals.
standard arbitration ID	An 11-bit arbitration ID. Frames that use standard IDs are often referred to as CAN 2.0 Part A; standard IDs are by far the most commonly used.
start trigger	When two or more hardware products are used to measure a common system, you typically need to compare data from the hardware products simultaneously. Since each hardware product starts its measurement independently, measurements on different hardware products can often be skewed in time relative to one another. For example, if you measure the same sine wave on two different analog-input products, the measured sine waves start off out of phase. National Instruments products use RTSI to share start triggers among different hardware products. Since the products share the same start trigger, measurements begin at the same time.

synchronize	Connection of two or more hardware products in order to measure a common system. For National Instruments products, RTSI connections are used to synchronize. Although there are a variety of ways to synchronize National Instruments products, a common technique is to share a timebase and start trigger over RTSI in order to eliminate clock drift and startup skew.

T

task	A collection of channels that you can read or write. The task is returned as an output parameter of the CAN Init Start function, and is used for all subsequent Channel API calls such as Read or Write. For more information, refer to **CAN Init Start.vi** for LabVIEW, or `nctInitStart` for C.
terminal	A physical pin on a hardware component. RTSI signals are one type of terminal. Internal connections within hardware products are another type of terminal.
timebase	The fundamental clock used to perform measurement. National Instruments synchronization features allow the timebase of one product to be shared with another in order to eliminate clock drift.
Timeout	Property in the Channel API that specifies the behavior the timeout in seconds for Read and Write functions. For more information, refer to **CAN Set Property.vi** for LabVIEW, or `nctSetProperty` for C.

U

unsolicited	Connections that transmit data on the network sporadically based on an external event. Also known as *nonperiodic*, *sporadic*, and *event driven*.

V

VI	Virtual Instrument.

W

watchdog timeout — A timeout associated with a connection that expects to receive network data at a specific rate. If data is not received before the watchdog timeout expires, the connection is normally stopped. You can use watchdog timeouts to verify that the remote node is still operational.

waveform data type — LabVIEW data type that represents a sequential list of samples in time. The data type includes the array of samples (each a DBL), a start time that specifies when the first sample was measured, and a delay time that specifies the time between samples (sample rate) or more information, refer to your LabVIEW documentation.

The Read and Write functions of the Channel API support the LabVIEW waveform data type.

Index

Numerics

A

B

C

D

E

F

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W