FOUNDATION™ Fieldbus

NI-FBUS Hardware and Software User Manual

January 2014
371994H-01

NATIONAL INSTRUMENTS™

Worldwide Technical Support and Product Information

ni.com

Worldwide Offices

Visit 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.

National Instruments Corporate Headquarters

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

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

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, he 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

End-User License Agreements and Third-Party Legal Notices

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

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

Trademarks

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

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

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

TETRIX by Pitsco is a trademark of Pitsco, Inc.

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

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

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

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

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

Xilinx is the registered trademark of Xilinx, Inc.

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

FireWire® is the registered trademark of Apple Inc.

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

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

Patents

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

Export Compliance Information

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

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(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

Electromagnetic Compatibility Information

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

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

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

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

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

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

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

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

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

Contents

About This Manual

This manual contains information on how to configure and use National Instruments Fieldbus hardware and software.

Related Documentation

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

- *Fieldbus Standard for Use in Industrial Control Systems, Part 2, ISA-S50.02.1992*
- *NI-FBUS Installation Guide*
- *Wiring and Installation 31.25 kbit/s, Voltage Mode, Wire Medium Application Guide*, Fieldbus Foundation

1 Introduction

This chapter provides an introduction to the FOUNDATION™ Fieldbus (FF) and the National Instruments hardware and software products for FF.

FF Overview

FOUNDATION Fieldbus is an all-digital, two-way, multi-drop communication system that brings the control algorithms into instrumentation. FOUNDATION Fieldbus is a Local Area Network (LAN) for FOUNDATION Fieldbus devices including process control sensors, actuators, and control devices. FOUNDATION Fieldbus supports digital encoding of data and many types of messages. Unlike many traditional system which requires a set of wires for each device, multiple FOUNDATION Fieldbus devices can be connected to the same set of wires.

FOUNDATION Fieldbus has two communication protocols: H1 and HSE. The first, H1, transmits at 31.25 Kb/s and is used to connect the field devices. The second protocol, High Speed Ethernet (HSE), uses 10 or 100 Mbps Ethernet as the physical later and provides a high-speed backbone for the network.

Please refer to *FOUNDATION™ Fieldbus Overview* document for more information about FOUNDATION Fieldbus technology.

NI-FBUS Hardware Products

PCI, PCMCIA, and USB

National Instruments provides interface devices for the PCI bus (PCI-FBUS), PCMCIA (PCMCIA-FBUS), and USB (USB-8486). Each National Instruments device connects FOUNDATION Fieldbus devices to standard desktop, industrial, and notebook personal computers. PCMCIA-FBUS is available in 1- and 2-port configurations. PCI-FBUS is available in a 2-port configuration. USB-8486 is available in a 1-port configuration.

The PCI-FBUS/USB-8486 uses a standard DB-9 male D-SUB connector to attach to the Fieldbus network. The PCMCIA-FBUS connects to the fieldbus by using a cable that provides two connectors to attach to the fieldbus network DB-9 male D-SUB connector and Combicon-style pluggable screw terminals.

NI-FBUS Software Products

Communications Manager

The NI-FBUS Communications Manager implements a high-level Application Program Interface (API) that lets you communicate with the National Instruments FOUNDATION Fieldbus communication stack and hardware. It provides a high-level API advanced users can use to interface with the National Instruments FOUNDATION Fieldbus communication stack and hardware.

Configurator

Most NI-FBUS users use the NI-FBUS Configurator. In addition to providing the functionality of the NI-FBUS Communications Manager in a graphical format, it includes additional functionality to allow you to configure a Fieldbus network. It can automatically generate the schedule for the network and configure field devices and hosts to transmit and receive alarms and trends.

Monitor

The NI-FBUS Monitor helps you monitor and debug Fieldbus data traffic. It symbolically decodes data packets from the Fieldbus, monitors the live list, and performs statistical analysis of packets. You can use the NI-FBUS Monitor to diagnose the communication of H1 network or debug the development of device.

You can use FOUNDATION Fieldbus products with National Instruments HMI software packages, including Lookout and LabVIEW DSC. And you can also use third-party HMI software through NI-FBUS OPC Server.

2

Connector and Cabling

This chapter provides hardware connector and interface cabling information for PCI-FBUS, PCMCIA-FBUS, and USB-8486. Install the software and hardware before cabling the hardware. Refer to the *NI-FBUS Installation Guide* available in PDF-format in the *NI-FBUS Software* media or in printed-format shipped with the media.

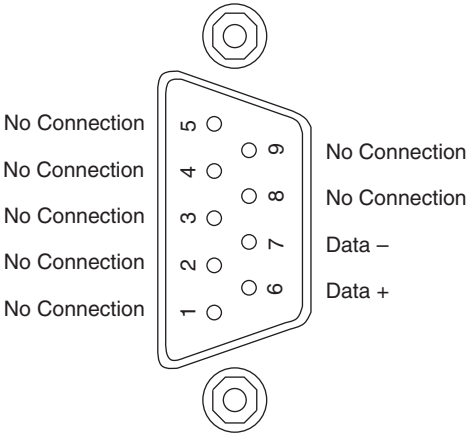
PCI-FBUS/2

This section contains information about the pinout of the PCI-FBUS connectors.

Fieldbus Cable Connector Pinout

To make a Fieldbus cable, ensure that pins 6 and 7 are used for the Fieldbus signals as shown in Figure 2-1. The cable must also follow the technical specifications listed in the document *Fieldbus Standard for Use in Industrial Control Systems, Part 2, ISA-S50.02.1992*.

Figure 2-1. Fieldbus Connector Pinout for the PCI-FBUS



PCMCIA-FBUS

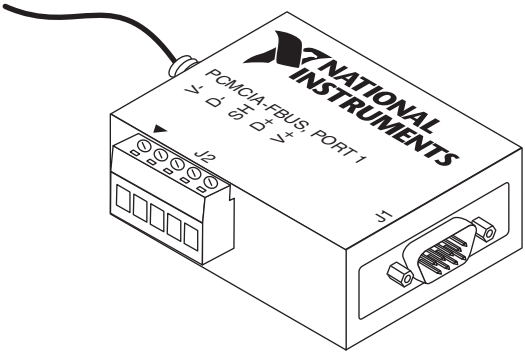
This section contains information about the pinout of the PCMCIA-FBUS connectors.

Pinout Information

A PCMCIA-FBUS cable has been included in your kit. The following figures show the pinout of the PCMCIA-FBUS connectors so you can make your own cable if you need a longer cable than the one provided in your kit.

Figure 2-2 shows the PCMCIA-FBUS cable. An arrow on the cable points to pin 1 of the screw terminal block.

Figure 2-2. PCMCIA-FBUS Cable



The PCMCIA-FBUS/2 cable has two Fieldbus connectors that are similar to the one shown in Figure 2-2. The connector labeled **PCMCIA-FBUS, PORT 1** is the connector for Fieldbus port 1 and the connector labeled **PCMCIA-FBUS, PORT 2** is the connector for Fieldbus port 2. Refer to Figure 2-3 and Figure 2-4 for the pinouts of both connectors.

Figure 2-3 shows J1, the Fieldbus connector pinout.

Figure 2-3. Fieldbus Connector Pinout

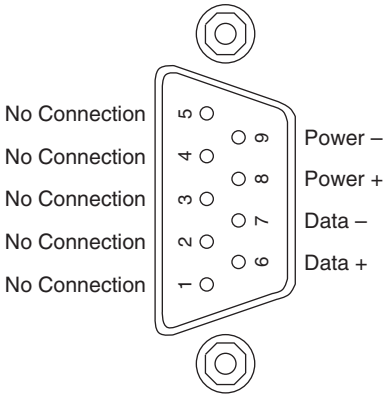
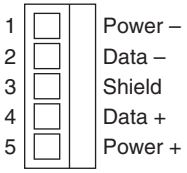


Figure 2-4 shows J2, the screw terminal block pinout.

Figure 2-4. Screw Terminal Block Pinout



The pinout of the PCMCIA-FBUS uses pins 6 and 7 of the J1 connector for the Fieldbus signals as specified in the *Fieldbus Standard for Use in Industrial Control Systems, Part 2, ISA-S50.02.1992*. Pins 2 and 4 of the J2 screw terminal block provide an alternate connection to the Fieldbus. However, the screw terminal block is not an independent link.

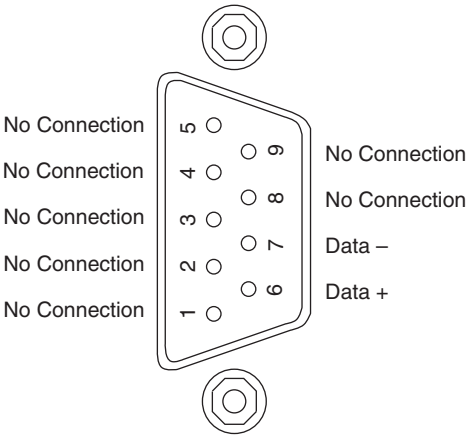
All of the signals on the screw terminal block provide a direct connection to the 9-pin D-SUB. National Instruments provides the Power+ and Power- connections as passive connections from the D-SUB to the screw terminal. The PCMCIA-FBUS itself does not supply power to, or draw power from, these pins.

USB-8486

The USB-8486 hardware has a 9-pin male D-SUB (DB-9) connector for the H1 port.

Figure 2-5 shows the male DB-9 connector pinout.

Figure 2-5. Male DB-9 Connector Pinout for the USB-8486



The pinout of the USB-8486 uses pins 6 and 7 of the connector for the Fieldbus signals as specified in the *Fieldbus Standard for Use in Industrial Control Systems, Part 2, ISA-S50.02.1992*.

9-Pin D-SUB (DB-9) Cable Information

A 2-meter cable has been included in your kit which converts the 9-pin D-SUB connector to three wire pigtails.

Figure 2-6. DB-9 Cable for the USB-8486

Figure 2-7 shows the pinout of the 9-pin D-SUB female connector so you can make your own cable if you need a longer cable than the one provided in your kit.

Figure 2-7. Pinout for 9-Pin D-SUB Female Connector of the DB-9 Cable

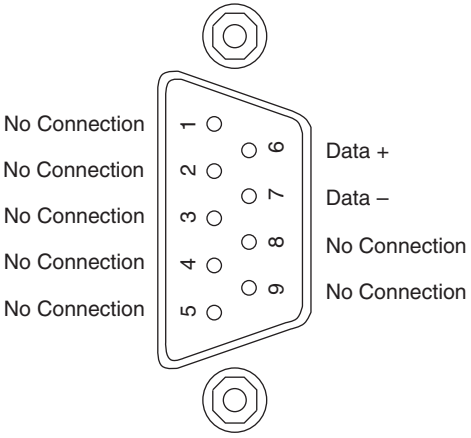


Table 2-1 provides the cable pigtail "pinout."

Table 2-1. Information for Cable Pigtails

Signal	Color	Size
Data +	Red	22 AWG
Data -	Black	22 AWG
Shield	Green	22 AWG

All of the signals on the three wire pigtails provide a direct corresponding connection to the 9-pin D-SUB.

3

NI-FBUS CM Software

This chapter provides information on the NI-FBUS Communications Manager (CM) software. It assumes that you are already familiar with your Microsoft operating system.

NI-FBUS Communications Manager Overview

The NI-FBUS Communications Manager implements a high-level Application Program Interface (API) that facilitates communication with the National Instruments FOUNDATION Fieldbus communication stack and hardware. The main purpose of the NI-FBUS Communications Manager is to make the details of the Fieldbus communication protocols transparent by providing an API that supports *TAG.PARAMETER* access. You need a general knowledge of the Fieldbus architecture (outlined in the *FOUNDATION™ Fieldbus Overview* document) to understand and use the NI-FBUS Communications Manager.

The NI-FBUS Communications Manager handles communication between the communication stack and the user application. It also handles the details of communicating with the Fieldbus Messaging Specification (FMS) and lower layers of the communications stack. The NI-FBUS Communications Manager hides the low-level details of Virtual Communication Relationships (VCRs), connection management, addresses, and Object Dictionary indices, and offers name access to physical devices, Virtual Field Devices (VFDs), function blocks, transducer blocks, and parameters.

The NI-FBUS Communications Manager API is independent of the National Instruments Fieldbus hardware and the operating system. With the NI-FBUS Communications Manager, you can insert multiple National Instruments Fieldbus interfaces into the same PC and use them through the NI-FBUS Communications Manager API.

The NI-FBUS Communications Manager is interface-independent because this tool does not require you to specify which Fieldbus interface to use in NI-FBUS Communications Manager calls. It determines the interface over which to send certain Fieldbus messages. The NI-FBUS Communications Manager lets you write applications that are as independent as possible of the actual configuration of your Fieldbus interfaces.

The NI-FBUS Communications Manager API is useful for developing host applications. Typical examples are function block tuning software packages and applications for monitoring a function block, diagnosing a network, and developing interfaces to Human-Machine Interface (HMI) packages.

Installing the OPC NI-FBUS Server

The NI-FBUS installer automatically installs the OPC NI-FBUS server. However, it also can be installed manually. To do this, open a DOS command prompt and run the following commands from the target directory:

```
regsvr32 opccomn_ps.dll
regsvr32 opcproxy.dll
nifb_opcda.exe /regserver
```

NI-FBUS Functions Overview

The NI-FBUS functions are classified into four categories:

- Administrative functions
- Core functions
- Alert and trend functions
- Device description functions

All NI-FBUS functions are described in detail in the *NI-FBUS Communications Manager Function Reference Manual*.

Administrative Functions

You can use the administrative functions to get the list of physical devices in a link, get a list of virtual field devices in a physical device, and get a list of blocks (resource, function, transducer) from a virtual field device. The administrative functions include `nifGetDeviceList`, `nifGetVfdList`, and `nifGetBlockList`. Typically, you must call these before you call a core, alert, or any other administrative function.

Because you can use the NI-FBUS Communications Manager to communicate with each of the FOUNDATION Fieldbus entities, such as links, physical devices, virtual field devices, and blocks, there are `nifOpen` calls for you to open and get a descriptor to each of these entities.

Example: Using Administrative Functions

Suppose you want to get a descriptor to a block with `nifOpenBlock` before you read or write the block parameters. Then you want to open a block using the block's tag.

To open a block with the tag `TI101_Analog_Input`, invoke `nifOpenBlock(sessionDesc, "TI101_Analog_Input", &blockDesc)`, where `sessionDesc` is the descriptor of the session that you established with the NI-FBUS Communications Manager. The NI-FBUS Communications Manager returns the descriptor of the block that you opened in `blockDesc`. From then on, you can use this descriptor for calls associated with this block.

Core Functions

Core NI-FBUS functions are the functions that deal with processing function block parameters—primarily the `nifReadObject` and `nifWriteObject` functions, which read and write block parameters. The NI-FBUS Communications Manager encapsulates the device description services with the core function `nifGetObjectAttributes`, which gives you the device description attributes of any parameter.

Function blocks contain view or display objects. As the name implies, these objects are a collection of parameters in function blocks that are typically displayed in an operator console. Four view objects are defined for each of the ten standard function blocks in the FOUNDATION Fieldbus specification.

The following examples are a summary of the NI-FBUS Communications Manager because they demonstrate that details such as VCRs, indices, and connections are hidden by the *TAG.PARAMETER* access provided by the NI-FBUS Communications Manager. However, to correctly write an application using the NI-FBUS Communications Manager, you must be familiar with the *FOUNDATION Specification: Function Block Application Process, Parts 1 and 2* document—the standard blocks, their parameters, and their syntax—and have an idea of the architecture of Fieldbus. Refer to the *FOUNDATION™ Fieldbus Overview* document for an outline of Fieldbus architecture.

Example: Using Core Functions

Suppose the object `VIEW_1` for a PID function block consists of `GAIN`, `RATE`, `SP`, `CAS_IN`, `MODE`, and `ALARM_SUM` parameters of the PID function block. You want to get the values of all these parameters using a single read of the `VIEW_1` object. If the tag of a PID function block is `TIC101_PID`, you can read the `VIEW_1` object by executing the following function call:

```
nifReadObject(sessionDesc, "TIC101_PID.VIEW_1", buffer, &cnt)
```

Notice that it is not necessary to have a block descriptor to read the parameters of an object. If you *do* have the block descriptor, you can read the object with the following call:

```
nifReadObject(blockDesc, "VIEW_1", buffer, &cnt)
```

You can get the block descriptor using `nifOpenBlock`, which returns **`blockDesc`**.

If you wanted to change the setpoint of the preceding PID block, you can do so with the following call:

```
nifWriteObject(sessionDesc, "TIC101_PID.SP", buffer, cnt)
```

Alert and Trend Functions

When a properly configured device detects an alarm condition, the device broadcasts the data. A host device receives the alarm, then sends a communication acknowledgment and an operator acknowledgment to the field device. The field device also can collect trends based on a configured sample type and interval. When the field device collects 16 samples, it broadcasts the

trend data on the Fieldbus. Any number of interested hosts can collect this data. For more details, refer to the *Foundation Specification: Function Block Application Process, Part 1* document.

With a program such as the NI-FBUS Configurator, you can configure the FOUNDATION Fieldbus field devices to broadcast alert and trend data.

The NI-FBUS Communications Manager has functions to receive trends and alerts from configured devices and to perform operator acknowledgment on alerts. `nifWaitAlert` and `nifWaitAlert2` lets you wait for an alert from any device in a link, any function block in a physical device, or a specific function block, depending on the type of descriptor that you pass to it. When the NI-FBUS Communications Manager receives an alert, it returns a structure containing information about the alert. The NI-FBUS Communications Manager sends the communication acknowledgment to the device automatically. The NI-FBUS Communications Manager provides a separate function, `nifAcknowledgeAlarm`, to send the operator acknowledgment.

Similarly, `nifWaitTrend` lets you wait for a trend from any device in a link, any function block in a physical device, or a specific function block, depending on the type of descriptor you pass to it. When the NI-FBUS Communications Manager receives a trend, it returns a structure containing information about the trend along with the trend data itself.

`nifWaitAlert`, `nifWaitAlert2`, and `nifWaitTrend` wait until an alert or trend is received before returning, so it might be preferable to have separate threads invoke these functions.

Device Description Functions

The NI-FBUS Communications Manager gives your applications access to device descriptions, which are binary files that describe the characteristics of blocks and parameters. Your application can use the NI-FBUS function `nifGetObjectAttributes` to decode attributes of parameters including data type, data size, help strings, and other attributes defined in the *Device Description Language Specification* document. In addition, device description symbol files are used automatically to assist in allowing your applications to access parameters by name.

The NI-FBUS Communications Manager ships with device descriptions for all standard FOUNDATION Fieldbus function blocks. The NI-FBUS Communications Manager provides attributes for the parameters of all *standard* function blocks, even if the device manufacturers for your devices did not provide device descriptions. However, to get the attributes of parameters of nonstandard (not FOUNDATION Fieldbus-defined) blocks, the NI-FBUS Communications Manager requires that the device manufacturer provide the device description.

NI-FBUS supports device description menus and methods. When NI-FBUS attempts to locate a device description file (`.ffo` and `.sym`) for a device, it uses the file with the latest device description revision for a given `MANUFAC_ID`, `DEV_TYPE`, and `DEV_REV`. For more information about device descriptions, refer to the *FOUNDATION™ Fieldbus Overview* document or your *Getting Started* manual.

Using the NI-FBUS Communications Manager Process

For any of your NI-FBUS Communications Manager applications to run correctly, you must successfully launch the NIFB process. The NIFB process is the medium by which your application communicates with the devices on the Fieldbus network. The NIFB process receives requests from your application and passes them on to the specified Fieldbus device through the Fieldbus interface connected to your machine. Refer to the *Start the NIFB Process* chapter in your Getting Started manual for instructions on how to start the NIFB process.

At startup, the NIFB process downloads the FOUNDATION Fieldbus communication stack file `ffstack.bin` or `ffstack.usb.bin` to the Fieldbus interfaces connected to your machine. It then downloads the communication stack configuration parameters, such as the Fieldbus network address for the interface device and so on, to each interface device. You can edit these parameters using the NI-FBUS Interface Configuration utility by clicking the **Advanced** button on the dialog box for the **Port** information. The advanced parameters affect the operation of the communication stack and should only be changed if you are aware of the effect of your changes on the stack.

You must make sure to specify a unique, non-default Fieldbus network address for the NI-FBUS Communications Manager to work properly. You can use a default address if another entity on the Fieldbus assigns your interface a non-default address. You can change the address from the NI-FBUS Interface Configuration utility in the **Port** dialog box. You must restart the NI-FBUS Communications Manager for any changes you make to take effect.

The NI-FBUS Communications Manager process features non-volatile storage of all network parameters, including the last known Link Active Schedule. After network parameters (including the Link Active Schedule) are stored, the NI-FBUS Communications Manager automatically reloads them to the interface on startup.

At installation time, the non-volatile copy of the schedule is empty, but you can make the NI-FBUS Communications Manager store the non-volatile Link Active Schedule by downloading it to your Fieldbus interface. To download a Link Active Schedule to your Fieldbus interface, you can use the NI-FBUS Dialog utility. Refer to the *Configuring the Link Active Schedule File* section for an example of how to download the Link Active Schedule to your Fieldbus interface. You also can use the NI-FBUS Configurator to download a Link Active Schedule to your Fieldbus interface.

Developing Your NI-FBUS Communications Manager Application

This section contains information to help you develop your NI-FBUS Communications Manager application.

Choose Your Level of Communication

While a few functions require a specific type of descriptor (for example, `nifGetDeviceList` requires a link descriptor), many functions (such as the core, alert, and trend functions) let you communicate using any type of descriptor. With these functions, the descriptor type you choose depends on what is most convenient for you in designing your application, because there is no significant difference in performance between the different types.

For example, if it is convenient for your application to use only a session descriptor to keep track of tags for each block (so that you refer to all parameters in *BLOCKTAG.PARAMNAME* format), you should write your application this way. If it is easier for you to keep track of a descriptor for each block rather than a tag for each block, you should open a block descriptor for each block you are communicating with, keep track of that descriptor value, and access parameters by *PARAMNAME* using the block descriptor.

Choose to Access by Name or Index

The NI-FBUS Communications Manager supports access by name or by index for all block parameters. National Instruments recommends that you access all variables by name. Although access by index might be slightly faster in some cases, an application cannot always reliably determine indices.

The NI-FBUS Communications Manager may convert the parameter name you specify to the final index that FOUNDATION Fieldbus protocols must use to access the parameter over the network. The NI-FBUS Communications Manager converts the name to an index using standard FOUNDATION Fieldbus-specified methods, which include a check to the device at run time to verify the index. If you hard-code indices, you will have to modify them when the devices they are accessing become replaced, upgraded, or have new blocks created on them.

Choose to Write Single-Thread or Multi-Thread Applications

All NI-FBUS functions are synchronous, meaning that the calling function is blocked until the NI-FBUS call completes. A Fieldbus device usually takes tens of milliseconds to respond to a block parameter read or write. It takes longer if any communication errors occur. The NI-FBUS Communications Manager uses the protocol connections to communicate with the devices. If a connection is lost, the NI-FBUS Communications Manager tries to reestablish the connection. When a connection is lost, an NI-FBUS read or write call may take several seconds to complete.

Single-Thread Applications

If potential delays like the ones discussed in the previous paragraph are acceptable for your application, you can write your application or the Fieldbus access part of your application as a single thread. Single-threaded applications are easier to develop, debug, and test because you do not have to consider exclusion between threads. If you are writing an application for testing, monitoring, or configuring a single device, a single-threaded application might be adequate.

Multi-Thread Applications

If your application monitors or tests several devices at a time, communication delays might affect the throughput of your application and therefore be unacceptable. If so, you can develop a multi-threaded application to improve the performance of your application. There are several ways to multi-thread your application.

If you are accessing information from function blocks or transducer blocks, you might want to create a thread for each block. Each block's thread reads and writes information for that block. If creating a thread for each block is excessive, you might consider an architecture in which you have a set of threads dedicated to Fieldbus I/O. Your application can then interface with I/O threads through a shared queue in which threads put their I/O requests. When the I/O completes, the I/O threads can inform the application by passing a message or some other synchronization scheme.

If your application performs trending or alarm handling, you might want to have separate threads that perform these functions. You can make a thread wait for a trend or alarm with the `nifWaitTrend` or `nifWaitAlert` or `nifWaitAlert2` function and then process the trend or alarm when it arrives. If you are monitoring the live list (the current list of devices on the bus), you may have a dedicated thread that calls `nifGetDeviceList` because the call will not return until the live list changes.

Access Object Dictionary Entries

If you want to access object dictionary entries that do not reside in a block, you can access them with an object dictionary index along with a virtual field device descriptor. You can access trend and linkage objects by name using a virtual field device descriptor. To access trend objects by name, either from an application program or from the NI-FBUS Dialog utility, use the name `TREND.`*`X`*, where *`X`* is a number from 1 to the number of trend objects in the virtual field device. To access linkage objects, use `LINKAGE.`*`X`*, where *`X`* is a number from 1 to the number of linkage objects in the virtual field device. If *`X`* exceeds the number of linkage objects or trend objects in the virtual field device, the NI-FBUS Communications Manager returns the **`E_ORDINAL_NUM_OUT_OF_RANGE`** error code.

Access Management Information Base (MIB) Parameters

To access Management Information Base parameters directly, either from a program or from the NI-FBUS Dialog utility, open the physical device you want to communicate with and open a virtual field device on the device with the tag `MIB`. You can use the resulting virtual field device descriptor to access the MIB parameters by index or by their names (as described in the *FOUNDATION™ Fieldbus Specification*). For example, to write the macrocycle duration, access the MIB parameter `MACROCYCLE_DURATION`, and to read the live list, access the object named `LIVE_LIST_STATUS`. This method works both on local interface devices and on remote devices over the Fieldbus.

Some MIB parameters are elements of a list (such as the list of function block schedule entries or VCR entries). You can use the name for these items with a *.X* appended, where *X* is the element in the list you want to access. For example, the first function block schedule entry in the MIB is named `FB_START_ENTRY.1`, and the first VCR static entry in the MIB is named `VCR_STATIC_ENTRY.1`. If *X* exceeds the number of objects of that type in the MIB, the NI-FBUS Communications Manager returns the **`E_ORDINAL_NUM_OUT_OF_RANGE`** error code.

Because most of these parameters have to do with network configuration, a network configurator, such as the NI-FBUS Configurator, can best set these parameters.

Keep in mind that the NI-FBUS Communications Manager manages some MIB objects internally. For instance, the NI-FBUS Communications Manager builds up internal data structures for some MIB objects, especially VCRs, and so on. Manually changing the existing VCRs through an MIB descriptor can lead to problems with using the NI-FBUS Communications Manager.

H1 Device MIB List Parameters

`FB_START_ENTRY`

`MAX_TOKEN_HOLD_TIME`

`SCHEDULE_DESCRIPTOR`

`VCR_STATIC_ENTRY`

`VFD_REF_ENTRY`

H1 Device MIB Parameters

`AP_CLOCK_SYNC_INTERVAL`

`BOOT_OPERAT_FUNCTIONAL_CLASS`

`CHANNEL_STATES`

`CONFIGURED_LINK_SETTING`

`CURRENT_LINK_SETTING`

`CURRENT_TIME`

DEV_ID
DLME_BASIC_CHARACTERISTICS
DLME_BASIC_INFO
DLME_LINK_MASTER_INFO
LINK_SCHEDULE_ACTIVATION
LINK_SCHEDULE_LIST_CHARACTERISTICS
LIVE_LIST_STATUS
LOCAL_TIME_DIFF
MACROCYCLE_DURATION
OPERATIONAL_POWERUP
PD_TAG
PLME_BASIC_CHARACTERISTICS
PLME_BASIC_INFO
PRIMARY_AP_TIME_PUBLISHER
PRIMARY_LINK_MASTER_FLAG
SM_SUPPORT
STACK_CAPABILITIES
T1
T2
T3
TIME_LAST_RCVD
TIME_PUBLISHER_ADDR
VCR_LIST_CHARACTERISTICS
VERSION_OF_SCHEDULE

HSE Device MIB List Parameters

SCHEDULE_DESCRIPTOR
VFD_REF_ENTRY
CONFIGURED_SESSION_ENTRY
AUTOMATIC_SESSION_ENTRY
HSE_CONFIGURED_VCR_ENTRY
HSE_AUTOMATIC_VCR_ENTRY

HSE Device MIB Parameters

SM_SUPPORT
OPERATIONAL_POWERUP
LIST_OF_VERSION_NUMBERS
OPERATIONAL_IP_ADDRESS

LOCAL_IP_ADDRESS_ARRAY

SYNC_AND_SCHEDULING

LAST_SNTP_MESSAGE

SNTP_TIMESTAMPS

DEVICE_IDENTIFICATION

SCHEDULE_ACTIVATION_VARIABLE

SCHEDULE_LIST_CHARACTERISTICS

NM_CHARACTERISTICS

CONFIGURED_SESSION_LIST_HEADER

AUTOMATIC_SESSION_LIST_HEADER

HSE_CONFIGURED_VCR_LIST_HEADER

HSE_AUTOMATIC_VCR_LIST_HEADER

BRIDGE_CHARACTERISTICS

CURRENT_NMA_CONFIGURATION_ACCESS

PREVIOUS_NMA_CONFIGURATION_ACCESS

INTERFACE_ADDRESS_ARRAY

INTERFACE_DESIRED_STATE_ARRAY

INTERFACE_ACTUAL_STATE_ARRAY

Use the NI-FBUS Dialog Utility to Communicate with Devices

The NI-FBUS Dialog utility helps you perform simple tests of your whole Fieldbus setup, including the NI-FBUS Communications Manager, your interface board(s), and any devices you have. The NI-FBUS Dialog utility has dialog boxes that call the NI-FBUS Communications Manager API, allowing you to specify parameters and make NI-FBUS calls. For example, you can use the NI-FBUS Dialog utility to get a list of devices on your network, as well as view and set parameters in each device. For more information on using the NI-FBUS Dialog utility, refer to the *Configuring the Link Active Schedule File* section.

Write Your Application

Use the following guidelines to make sure your application uses the NI-FBUS Communications Manager interface properly.

- Always call `nifOpenSession` early in your program and check the return value of the call. This check verifies that the NI-FBUS Communications Manager process is running, which is a prerequisite for your application to access the Fieldbus network. If this call fails, your application should inform the user that the Fieldbus is currently inaccessible.
- Always close any descriptors that you open before your program exits, including session descriptors. The NI-FBUS Communications Manager requires that your application close all descriptors that it opens.

- Always check the return values from NI-FBUS calls. The NI-FBUS Communications Manager is a high-level API and performs many operations that can fail because of incorrect parameters, incorrect bus configuration, or communication failures. An application that fails to check return values might use output parameters from NI-FBUS calls that are NULL or uninitialized, leading to incorrect behavior or a program crash.
- If you plan to call any of the indefinitely-blocking functions including `nifGetDeviceList`, `nifWaitAlert`, `nifWaitAlert2`, and `nifWaitTrend`, you should probably use a separate descriptor for these calls. To terminate these calls early, you have to close the descriptor. Having a separate descriptor will ensure that terminating these calls does not affect any other NI-FBUS calls your application has pending.
- If the NI-FBUS Communications Manager stops for any reason, any outstanding calls in your application complete with the error `E_SERVER_CONNECTION_LOST`. At this point, all of the descriptors that you have (including the session) are invalid. If you restart the NI-FBUS Communications Manager, your application should recover by opening a new session to the NI-FBUS Communications Manager and opening all new descriptors. After this recovery procedure, your application should be fully operational.

Compile, Link, and Run Your Application

To compile, link, and execute your application, you must complete the following:

- Add the line `#include "nifbus.h"` to any of your source files that make NI-FBUS calls. The `nifbus.h` file is located in the `includes` subdirectory of your installation. Also, make sure that the `includes` subdirectory is included in your project's settings.
- Link your application with `nifb.lib`, which is located in the `MS Visual C` subdirectory of your installation.
- Ensure that `nifb.dll` is present in your Windows directory. `nifb.dll` is an interface DLL required to interface to the NIFB process. `nifb.dll` must be present when your application runs.
- Ensure that the NI-FBUS Communications Manager (NIFB process) has started and is entirely initialized before your application makes its first NI-FBUS call.
- Ensure your compiler has the structure padding or alignment parameter set to eight bytes. This will allow proper communication of data structures.
- The `nifbus.h` header file and `nifb.lib` library have been compiled and linked with Microsoft Visual C/C++ version 6.0 or later.

Note NI-FBUS software supports 64-bit since version 4.0.1. To build a 64-bit application, you must link your application with `nifb64.lib`. `nifb64.dll` should be automatically installed in your `Windows` system directory.

Sample Programs

The NI-FBUS Communications Manager software includes four sample programs: `nifbtest.c`, `nifb_mt.c`, `nifbdd.c`, and `nifb_list.c`. These files provide you with some examples of NI-FBUS Communications Manager API usage.

Because NI-FBUS uses a device description library from the FOUNDATION Fieldbus, the header files from the device description library also are part of the NI-FBUS `includes` directory.

Configuring the Link Active Schedule File

If you want to do scheduling and use publishers and subscribers, you must follow the instructions in this section. You may ignore this section if there is no schedule, if the schedule is downloaded over the network to your Fieldbus interface, or if you are using software such as the NI-FBUS Configurator.

Introduction to the Link Active Schedule File

You must download the Link Active Schedule file to your Fieldbus interface before the board can have Link Active Scheduler functionality on the Fieldbus network.

Save the Link Active Schedule file as an `.ini` file. You can download this file to your interface board using the NI-FBUS Dialog utility.

For detailed information about the parameters in the Link Active Schedule file, refer to the *Data Link Layer* section of the *Final Specification* version of the *FOUNDATION™ Fieldbus Specification* document.

Format of the Link Active Schedule File

Create your Link Active Schedule file with the following format. The names of the sections of the Link Active Schedule file are:

```
[Schedule Summary]
...
[Subschedule 1]
...
[Sequence 1-1]
...
[Sequence 1-n]
...
[Subschedule x]
...
[Sequence x-1]
...
```

```
[Sequence x-y]
...
```

The general line format for all other lines is:

VARIABLE=VALUE

where the valid variable names and values are defined in Tables 3-1 to 3-4.

Table 3-1. Valid Variable Names and Values for the Schedule Summary Section

Variable Name	Valid Values	Implied Units	Default
encodingVersionNumber	0–7	none	none
versionNumber	0x0–0xffff	none	none
builderIdentifier	0x100–0xfff	none	none
numSubSchedules	0–255	none	none
maxSchedulingOverhead	0x0–0x3f	octets	none
macroCycle	0x0–0xffffffff	1/32 ms	none

Table 3-2. Valid Variable Names and Values for the Subschedule Section

Variable Name	Valid Values	Implied Units	Default
period	0x0–0xffffffff	1/32 ms	none
numSequence	0–255	none	none

Table 3-3. Valid Variable Names and Values for the Sequence Section

Variable Name	Valid Values	Implied Units	Default
maxDuration	0x0–0xffff	1/32 ms	none
numElement	0–255	none	none

For the variables in Table 3-4, *N* is an integer between 1 and `numElement`. Repeat these variables within this subschedule section exactly `numElement` times.

Table 3-4. Valid Variable Names Including the Variable *N* and Values for the Sequence Section

Variable Name	Valid Values	Implied Units	Default

`priority`*N*	`TIMEAVAILABLE`
	`URGENT`
	`NORMAL`
		none	none

`address`*N*	Parameter name in *`TAG.PARAM`* format or DLCEP (Data Link Connection End Point) in `0x`*NNNN* format	none	none

4

Developing The Application

This chapter explains how to develop your Fieldbus applications using the NI-FBUS APIs and Libraries.

LabVIEW

Use the Foundation Fieldbus VIs available in LabVIEW to interact with the Foundation Fieldbus devices. You also can use the NI-FBUS Configurator or the Tag Editor to view links, devices, blocks, and parameters of the FF network.

Use the FF Set Device Address VI or the FF Set Tag VI in LabVIEW to change the device address or the block tag. You also can customize the tag names using the NI-FBUS Configurator.

Note The NI-FBUS VIs in LabVIEW do not support downloading the Link Active Schedule. To download the schedule, use the NI-FBUS Configurator.

Refer to the *NI-FBUS VI Help* for more information about using these VIs.

Visual C++

The NI-FBUS software supports Microsoft Visual C/C++ version 6 or later.

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

To use the NI-FBUS C API, include the `nifbus.h` header file in the code, and set the folder `MS Visual C\includes` as include path, then link with the `nifb.lib` library file.

Note The NI-FBUS C API supports only the NI-FBUS Communication Manager on the local computer. The NI-FBUS C API does not support changing the device address or the block tag.

The reference for each NI-FBUS API function is in Chapter 5, *NI-FBUS Function Reference*.

You can find examples for the C language in the `MS VIsual C\examples` subfolder of the `NI-FBUS` folder. There are four sample programs: `nifbtest.c`, `nifb_mt.c`, `nifbdd.c`, and `nifb_list.c`. These files provide you with some examples of the NI-FBUS Communications Manager API usage. A description of each example is in comments at the top of the `.c` file.

Visual Basic

The NI-FBUS software support Microsoft Visual Basic 6.0 version.

To create an application in Visual Basic, add the `Declares.bas` to your project. The `Declares.bas` defines standard API calls to NI-FBUS Communications Manager.

The `Declares.bas` are located in the `MS Visual Basic` folder of the `NI-FBUS` folder. The typical path to this folder is `\Program Files\National Instruments\NI-FBUS\MS Visual Basic`.

Note The NI-FBUS software does not support changing the device address or the block tag using the Visual Basic applications.

The reference for each NI-FBUS API function is in Chapter 5, *NI-FBUS Function Reference*.

You can find example for Visual Basic in the `example` subfolder of `MS Visual Basic` folder. The `nifbusVBInterface.vbp` file is the Visual Basic project of the example.

.NET Class Libraries

This section provides general information about the .NET class libraries included with the NI-FBUS software, you can use the .NET class libraries to develop complete FOUNDATION Fieldbus applications in Visual Basic .NET and Visual C#.

Use the `NationalInstruments.Fieldbus.dll` to create these applications. If you want to run the application on the 64-bit operating system, choose the `.dll` file in either the `library32` or `library64` folder of the `MS.Net` folder. If you want to run the application on the 32-bit operating system, choose the `.dll` file in the `library32` folder of the `MS.Net` folder. The typical path to the folder is `\Program Files\National Instruments\NI-FBUS\MS.Net.`

NI-FBUS Software includes the following .NET class libraries:

- Alert
- Block
- Device
- FBDate
- FBObject
- FBTime
- HseDevice
- Link
- Mib
- Session

- TimeOfDay
- Trend
- Vfd

The Visual Basic .NET example can be found in the `examples/VBExample` subfolder of the `MS .NET` folder. The `VBExample.vbproj` file is the Visual Basic .NET project of the example.

Note The NI-FBUS software does not support changing the device address or the block tag using the .NET class libraries.

The Visual C# example can be found in the `examples/CsharpExample` subfolder of the `MS .NET` folder. The `CsharpExample.csproj` file is the Visual C# project of the example.

Another .NET example can be found in `examples/AdvDemo` subfolder of the `MS .NET` folder. The `AdvDemo.csproj` file is the Visual C# project of the example.

OPC Server

NI-FBUS software includes a separate OPC Data Access Server, which is compliant with the *OPC Data Access 2.0 and 3.0 Specification*.

Any OPC client program can easily access NI-FBUS OPC Server through standard OPC DA interfaces. The FF data types are mapped to OPC data types as below.

OPC Data Type Mapping Rule

The **SIMPLE** type and **ARRAY** type variables are regarded as leaf nodes in the OPC address space. The **RECORD** type variables are regarded as branch nodes, you need to access each of its member variable through this branch node.

Table 4-1 shows the data type-mapping rule.

Table 4-1. OPC Data Type Mapping Rule

Meta Type	FMS Standard Data Types	OPC Data Type
Simple	Boolean	VT_BOOL
	Integer8	VT_I1
	Integer16	VT_I2
	Integer32	VT_I4
	Unsigned8	VT_UI1
	Unsigned16	VT_UI2
	Unsigned32	VT_UI4
	Floating Point	VT_R4
	Visible String	VT_BSTR
	Octet String	VT_ARRAY | VT_UI1
	Date	VT_DATE
	Time of Day	VT_DATE
	Time Difference	VT_DATE
	Bit String	VT_ARRAY | VT_UI1
	Time Value	VT_DATE
Array	Boolean	VT_ARRAY | VT_BOOL
	Integer8	VT_ARRAY | VT_I1
	Integer16	VT_ARRAY | VT_I2
	Integer32	VT_ARRAY | VT_I4
	Unsigned8	VT_ARRAY | VT_UI1
	Unsigned16	VT_ARRAY | VT_UI2
	Unsigned32	VT_ARRAY | VT_UI4
	Floating Point	VT_ARRAY | VT_R4
	Visible String	VT_ARRAY | VT_BSTR

Table 4-1. OPC Data Type Mapping Rule (Continued)

Meta Type	FMS Standard Data Types	OPC Data Type
Array (continued)	Octet String	—
	Date	VT_ARRAY | VT_DATE
	Time of Day	VT_ARRAY | VT_DATE
	Time Difference	VT_ARRAY | VT_DATE
	Bit String	—
	Time Value	VT_ARRAY | VT_DATE

The NI-FBUS OPC Server has passed OPC Foundation Compliance Test, for more information, please visit OPC Foundation web site `www.opcfoundation.org`.

5

NI-FBUS Function Reference

This chapter provides function reference for the NI-FBUS Communications Manager software. You must have a general knowledge of the Fieldbus architecture to write programs for the NI-FBUS Communications Manager, and you must understand how your code will work with your Microsoft operating system.

Administrative Functions

For details on how NI-FBUS functions are classified and how to use them, refer to Chapter 3, *NI-FBUS CM Software*.

List of Administrative Functions

Table 5-1. List of Administrative Functions

Function	Purpose
nifClose	Closes an open descriptor.
nifDownloadDomain	Downloads data to the virtual field device (VFD) domain.
nifGetBlockList	Returns a list of information for all blocks of the type specified in the VFD.
nifGetDeviceList	Returns the list of information for all active devices on the network.
nifGetInterfaceList	Reads the list of interface names from the NI-FBUS Communications Manager.
nifGetVFDList	Gathers VFD information on a specified physical device.
nifOpenBlock	Returns a descriptor representing a block.
nifOpenLink	Returns a descriptor representing a Fieldbus link.
nifOpenPhysicalDevice	Returns a descriptor representing a physical device.
nifOpenSession	Returns a descriptor for an NI-FBUS session.
nifOpenVfd	Returns a descriptor representing a VFD.
nifShutdownCM	Closes NI-FBUS Communications Manager.
nifStartupCM	Starts NI-FBUS Communications Manager.

nifClose

Purpose

Closes an open descriptor.

Format

```
nifError_t          nifClose(
                    nifDesc_t ud);
```

Input

`ud`	The descriptor from an `nifOpen` call.

Output

Not applicable.

Context

Block, VFD, physical device, link, session.

Description

`nifClose` closes the specified descriptor. The descriptor is invalid after it is closed. Ensure that your application closes all of the descriptors it opens. Your application should always close a descriptor if it no longer needs the descriptor.

If you close a descriptor with calls pending on it, the calls complete within the usual time, but an error code is returned indicating that you closed the descriptor prematurely. If you make more synchronous wait calls that wait on the closing descriptor, such as `nifWaitTrend`, `nifWaitAlert`, `nifWaitAlert2`, and `nifGetDeviceList`, the NI-FBUS Communications Manager aborts these functions and returns an error code indicating that you closed the descriptor. Since calls that wait on a closed descriptor return an error message, you should have a separate descriptor for these synchronous wait calls.

Note A *session* is a connection between your application and an NI-FBUS entity. If you close a session, you close the communication channel between your application and the NI-FBUS entity associated with the session. Ensure that you close all descriptors opened under this session before closing a session descriptor.

Return Values

`E_OK`	The call was successful.
`E_INVALID_DESCRIPTOR`	The descriptor is invalid.
`E_SERVER_CONNECTION_LOST`	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifDownloadDomain

Purpose

Downloads data to the virtual field device (VFD) domain.

Format

```
nifError_t          nifDownloadDomain(
                    nifDesc_t ud,
                    uint32 index,
                    char *fileName);
```

Input

`ud`	The descriptor of the VFD you are accessing with `index`.
`index`	The absolute VFD index value of the domain you specified to download the data.
`fileName`	The name of the file where the download data is stored.

Context

VFD, physical device, link, session.

Description

`nifDownloadDomain` is used to download the data or parameter values to the specified VFD domain. The domain is specified by `index`.

To determine the appropriate `index` value, consult the documentation of the device to which you are trying to download the domain. If the device supports the Domain Download feature, the `index` for download should be specified in the documentation.

Return Values

`E_OK`	The call was successful.
`E_INVALID_DESCRIPTOR`	The descriptor specified is not valid.
`E_SERVER_CONNECTION_LOST`	The session established with the NI-FBUS Communication Manager, under which the descriptor was opened, has been lost or closed.
`E_RESOURCE`	The NI-FBUS Communications Manager is unable to allocate a system resource. This is usually a memory problem.
`E_DEVICE_CHANGED`	The device you specified has changed.
`E_VFD_CHANGED`	The VFD you specified has changed.

nifGetBlockList

Purpose

Returns a list of information for all blocks of the specified type present in the VFD.

Format

```
nifError_t          nifGetBlockList(
                    nifDesc_t ud,
                    uint8 whichTypes,
                    nifBlockInfo_t *info,
                    uint16 *numBlocks)
```

Input

`ud`	The descriptor of a VFD.
`whichTypes`	Specifies what types of blocks to return (function, transducer, or physical).
`numBlocks`	The number of buffers allocated in the `info` list.

Output

`info`	The list of information associated with each block.
`numBlocks`	The number of blocks actually in the VFD.

Context

VFD.

Description

`nifGetBlockList` returns information about all the blocks in the specified VFD. A *`block`* can be a resource block, transducer block, or function block residing within a VFD. Only blocks of the types specified by `whichTypes` are returned.

To determine how many list items are to be returned, call the function twice. The first time you call the function, set the `numBlocks` parameter to 0. The function will return an error stating that there were not enough buffers configured, and it will return a new number for `numBlocks`. Use this new `numBlocks` parameter to allocate memory for the data. When you call the function the second time, use this new parameter. By doing so you will allocate only as much memory as necessary.

`nifBlockInfo_t` is defined as follows:

```
typedef struct {
    char fbTag[TAG_SIZE + 1];
    uint16 startIndex;
    uint32 ddName;
    uint32 ddItem;
```

```
    uint16 ddRev;
    uint16 profile;
    uint16 profileRev;
    uint32 executionTime;
    uint32 periodExecution;
    uint16 numParams;
    uint16 nextFb;
    uint16 startViewIndex;
    uint8 numView3;
    uint8 numView4;
    uint16 ordNum;
    uint8 blockType;
} nifBlockInfo_t;
```

The `blockType` field in `nifBlockInfo_t` can be `FUNCTION_BLOCK`, `TRANSDUCER_BLOCK`, or `RESOURCE_BLOCK`.

The `whichTypes` parameter must be a bit combination of `FUNCTION_BLOCK`, `TRANSDUCER_BLOCK`, and `RESOURCE_BLOCK`.

Return Values

`E_OK`	The call was successful.
`E_INVALID_DESCRIPTOR`	The descriptor was invalid or of the wrong type.
`E_COMM_ERROR`	The NI-FBUS Communications Manager failed to communicate with the device.
`E_BUF_TOO_SMALL`	The buffer does not contain enough entries to hold all the information for the blocks. If you receive this error, buffer entries that you allocated do not contain valid block information when the call returns.
`E_OBSOLETE_DESC`	The input descriptor is no longer valid. It was closed before `nifGetBlockList` completed.
`E_BAD_ARGUMENT`	The `whichtypes` value is something other than `FUNCTION_BLOCK`, `TRANSDUCER_BLOCK`, or `RESOURCE_BLOCK`.
`E_RESOURCES`	A system resource problem occurred. The resource problem is usually a memory shortage.
`E_BAD_DEVICE_DATA`	The device returned some inconsistent information.
`E_SERVER_CONNECTION_LOST`	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifGetDeviceList

Purpose

Returns the list of information for all active devices on the network.

Format

```
nifError_t          nifGetDeviceList(nifDesc_t link,
                    nifDeviceInfo_t *devInfo,
                    uint16 *numDevices,
                    uint16 *revision)
```

Input

`link`	The link descriptor for which to return information.
`numDevices`	The number of allocated list entries.
`revision`	The revision number from the last `nifGetDeviceList` call, or zero (refer to the *Description* section for usage).

Output

`devInfo`	The list of device information.
`numDevices`	The number of devices present in the link.
`revision`	Current revision number of the live list that the NI-FBUS Communications Manager reads from the Fieldbus interface to the specified link.

Context

Link.

Description

`nifGetDeviceList` returns a list of information describing each device on the link. A *link* is a group of Fieldbus devices connected across a single wire pair with no intervening bridges. Before `nifGetDeviceList` returns the list of information, it waits until the revision argument passed in differs from the live list revision number the Fieldbus interface keeps for the specified link. The revision numbers the Fieldbus interface keeps start at one, so if you pass in a zero for `revision`, you can force `nifGetDeviceList` to immediately return the current device list. To use `nifGetDeviceList` most effectively, you should pass in the `revision` parameter output from the previous call to `nifGetDeviceList` in subsequent calls to it. Using the revision parameter output from the previous call forces `nifGetDeviceList` to wait until the device list has actually changed before returning the list of information.

If a device on the bus is unresponsive, its entry in the device information list has the tag and device ID `unknown device`, but its address field is correct. Also, the flag bit `NIF_DEV_NO_RESPONSE` is set.

The device list includes devices in the fixed, temporary, and visitor address ranges.

If there are too few input buffers, `nifGetDeviceList` returns an error code, but the `numDevices` parameter is set to the total number of devices available. In this case, the buffers you pass in do *not* contain valid data, but the revision number is set to the correct value.
If a device is an interface device, then the flag bit `NIF_DEV_INTERFACE` is set. You can abort a pending `nifGetDeviceList` call by closing the link descriptor on which the call was made.

To determine how many list items are to be returned in the call, call the function twice. The first time you call the function, set the `numDevices` parameter to 0. The function will return an error stating that there were not enough buffers configured, and it will return a new number for `numDevices`. Use this new `numDevices` parameter to allocate memory for the data. When you call the function the second time use this new parameter. By doing so you will allocate only as much memory as necessary.

`nifHseDeviceInfo_t` is defined as follows.

```
typedef struct {
    uint32 IpAddress;
        uint16 deviceIndex;
        uint16 maxDeviceIndex;
        uint32 hseRepeatTime;
        uint8  state;
        uint8  type;
        uint8  deviceRedundancyState;
        uint8  duplicateDetectionState;
        uint16 lanRedundancyPort;
        uint16 reserved;
        uint32 annunciationVersionNumber;
        uint32 hseDeviceVersionNumber;
        uint32 numH1Ports;
        uint32 *h1VersionList;
    } nifHseDeviceInfo_t;
```

`nifDeviceInfo_t` is defined as follows.

```
typedef struct {
    char deviceID[DEV_ID_SIZE + 1];
    char pdTag[TAG_SIZE + 1];
    uint8 nodeAddress;
    uint32 flags;
    nifHseDeviceInfo_t* hseDeviceInfo;
} nifDeviceInfo_t;
```

Return Values

`E_OK`	The call was successful.
`E_INVALID_DESCRIPTOR`	The link descriptor is invalid.

E_BUF_TOO_SMALL	There are not enough buffers allocated. If you receive this error, your input buffers do not contain valid data.
E_COMM_ERROR	The NI-FBUS Communications Manager failed to communicate with the device.
E_OBSOLETE_DESC	The input descriptor is no longer valid. It was closed before `nifGetDeviceList` completed.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifGetInterfaceList

Purpose

Reads the list of interface names from the NI-FBUS Communications Manager configuration.

Format

```
nifError_t          nifGetInterfaceList(
                    nifDesc_t ud,
                    int16 *numIntf,
                    nifInterfaceInfo_t *info)
```

Input

`ud`	A valid session descriptor.
`numIntf`	The number of buffers for interface information reserved in `info`.

Output

`numIntf`	The actual number of names returned.
`info`	An array of structures containing the interface name and device ID for each interface.

Context

Not applicable.

Description

`nifGetInterfaceList` returns the interface name and device ID of each Fieldbus interface. The `numIntf` parameter is an IN/OUT parameter. On input, it must contain the number of buffers that info allocates and points to, and on output it contains the total number of interface information entries available. If enough buffers were not allocated, or if the `info` buffer is NULL, the NI-FBUS Communications Manager returns an error and does not copy any data to the buffers. In this case, the `numIntf` parameter is still valid.

To determine how many list items are to be returned in the call, call the function twice. The first time you call the function, set the `numIntf` parameter to 0. The function will return an error stating that there were not enough buffers configured, and it will return a new number for `numIntf`. Use this new `numIntf` parameter to allocate memory for the data. When you call the function the second time, use this new parameter. By doing so, you will allocate only as much memory as necessary.

The `nifInterfaceInfo_t` structure is defined as follows:

```
typedef struct nifInterfaceInfo_t{
    char interfaceName[NIF_NAME_LEN];
    char deviceID[DEV_ID_SIZE +1];
} nifInterfaceInfo_t;
```

Note `nifGetInterfaceList` is an internal function for the NI-FBUS Communications Manager and does not cause Fieldbus activity.

Return Values

`E_OK`	The call was successful.
`E_BUF_TOO_SMALL`	The buffer does not contain enough entries to hold all the interface information.
`E_CONFIG_ERROR`	Some configuration information, such as registry information or network configuration information, is incorrect.
`E_NOT_FOUND`	Some interfaces are missing in the bus.

nifGetVFDList

Purpose

Gathers VFD information on a specified physical device.

Format

```
nifError_t          nifGetVFDList(
                    nifDesc_t ud,
                    nifVFDInfo_t *info,
                    uint16 *numBuffers)
```

Input

`ud`	The descriptor of the physical device for which to get the VFD list.
`numBuffers`	The number of buffers allocated in the `info` list.

Output

`numBuffers`	The number of VFDs actually in the device.
`info`	The VFD information.

Context

Physical device.

Description

`nifGetVFDList` gathers function block application VFD information from the specified physical device.

If there are too few input buffers, or if the input buffer pointer is NULL, an error code is returned, but the `numBuffers` parameter is set to the total number of VFDs in the device. In this case, no buffers contain valid data on output.

To determine how many list items are to be returned in the call, call the function twice. The first time you call the function, set the `numBuffers` parameter to 0. The function will return an error stating that there were not enough buffers configured, and it will return a new number for `numBuffers`. Use this new `numBuffers` parameter to allocate memory for the data. When you call the function the second time, use this new parameter. By doing so, you will allocate only as much memory as necessary.

The `info` parameter has the following format:

```
typedef struct {
    char vfdTag[TAG_SIZE +1];
    char vendor[TAG_SIZE +1];
    char model[TAG_SIZE +1];
```

```
    char revision[TAG_SIZE +1];
    int16 ODVersion;
    uint16 numTransducerBlocks;
    uint16 numFunctionBlocks;
    uint16 numActionObjects;
    uint16 numLinkObjects;
    uint16 numAlertObjects;
    uint16 numTrendObjects;
    uint16 numDomainObjects;
    uint16 totalObjects;
    uint32 flags;
} nifVFDInfo_t;
```

Return Values

E_OK	The call was successful.
E_COMM_ERROR	The NI-FBUS Communications Manager failed to communicate with the device.
E_INVALID_DESCRIPTOR	The input descriptor does not correspond to a physical device.
E_BUF_TOO_SMALL	There were not enough allocated buffers. Your specified input buffers do *not* contain valid data.
E_SM_NOT_OPERATIONAL	The device is present, but cannot respond because it is at a default address.
E_OBSOLETE_DESC	The input descriptor is no longer valid. It was closed before `nifGetVFDList` completed.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.
E_BAD_DEVICE_DATA	The device returned some inconsistent information.

nifOpenBlock

Purpose

Returns a descriptor representing a block.

Format

```
nifError_t          nifOpenBlock (
                    nifDesc_t ud,
                    char *blockTag,
                    nifDesc_t *out_ud)
nifError_t          nifOpenBlock (
                    nifDesc_t ud,
                    NIFB_ORDINAL(n),
                    nifDesc_t *out_ud)
```

Input

`ud`	A valid session, link, physical device, or VFD descriptor.
`blockTag`	The tag of the block. To access a block by ordinal number within a VFD, use the `NIFB_ORDINAL` macro in the `nifbus.h` header file. You can only access a block by ordinal number for VFD descriptors.

Output

`out_ud`	A descriptor for the block you request.

Context

VFD, physical device, link, session.

Description

`nifOpenBlock` returns a descriptor for the block you specify. You must pass a valid session, link, physical device, or VFD descriptor to this function.

There are two ways to specify the block: by tag and by ordinal number. To open the block by its tag, you must set `blockTag` to the current tag of the block. The NI-FBUS Communications Manager returns an error if it finds more than one block with the same tag. You can obtain the list of block tags within a specified VFD with a call to `nifGetBlockList`.

To open the block by its ordinal number, use the `NIFB_ORDINAL` macro. This macro is valid only if `ud` is a VFD descriptor. The first block in a VFD has the ordinal number zero. Notice that the first block in a VFD is always the resource block.

Return Values

E_OK	The call was successful.
E_INVALID_DESCRIPTOR	The input descriptor is invalid.
E_MULTIPLE	There are identical block tags.
E_ORDINAL_NUM_OUT_OF_RANGE	The ordinal number is out of the device range.
E_COMM_ERROR	An error occurred when the NI-FBUS Communications Manager communicated with the device.
E_NOT_FOUND	There is no such block in the device or VFD with the specified tag.
E_OBSOLETE_DESC	The input descriptor is no longer valid. It was closed before `nifOpenBlock` completed.
E_RESOURCES	A system resource problem occurred. The resource problem is usually a memory shortage.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.
E_BAD_DEVICE_DATA	The device returned some inconsistent information.

nifOpenLink

Purpose

Returns a descriptor representing a Fieldbus link.

Format

```
nifError_t          nifOpenLink (
                    nifDesc_t session,
                    uint8 interfaceOrDevID,
                    char *name,
                    uint16 linkID,
                    nifDesc_t *out_ud)
```

Input

`session`	A valid session descriptor on which to open the link.
`interfaceOrDevID`	How to specify the link: zero if by interface name, one if by local device ID.
`name`	The interface name or local device ID.
`linkID`	The link ID.

Output

`out_ud`	A descriptor for the link you request.

Context

Session.

Description

`nifOpenLink` returns a descriptor for the link you specify. You must pass a valid session descriptor to this function.

There are two ways you can specify the link. If the `interfaceOrDevID` parameter is zero, then `name` specifies the name of the interface the link is connected to. The list of valid interface names is contained in a configuration source which the NI-FBUS Communications Manager has access to, and can be obtained by a call to `nifGetInterfaceList`. If `interfaceOrDevID` is one, then the `name` specifies the device ID of an interface device to which the NI-FBUS Communications Manager is attached.

In both cases, `linkID` is the Fieldbus link ID number for the specified link. For single-link Fieldbus networks, you can set `linkID` to zero.

Return Values

E_OK	The call was successful.
E_INVALID_DESCRIPTOR	The input descriptor is invalid.
E_CONFIG_ERROR	Some configuration information, such as registry information or network configuration information, is incorrect.
E_NOT_FOUND	The interface name, device ID, or link ID you specified is not found.
E_RESOURCES	A system resource problem occurred. The resource problem is usually a memory shortage.
E_BAD_ARGUMENT	The `interfaceOrDevID` value is not valid.
E_OBSOLETE_DESC	The input descriptor is no longer valid. It was closed before `nifOpenLink` completed.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifOpenPhysicalDevice

Purpose

Returns a descriptor representing a physical device.

Format

```
nifError_t          nifOpenPhysicalDevice (
                    nifDesc_t ud,
                    uint8 tagOrDevID,
                    char *name,
                    nifDesc_t *out_ud)
```

Input

`ud`	A valid session or link descriptor on which to open the device.
`tagOrDevID`	How to specify the device: zero if by physical device tag, one if by device ID.
`name`	The tag or device ID.

Output

`out_ud`	A descriptor for the device you request

Context

Link, session.

Description

`nifOpenPhysicalDevice` returns a descriptor for the physical device you specify. You must pass a valid session or link descriptor to this function. If you pass a link descriptor, the NI-FBUS Communications Manager searches only that link for the specified device.

There are two ways you can specify the device. If the `tagOrDevID` parameter is zero, then the `name` specifies the tag of the physical device. If `tagOrDevID` is one, then `name` is the device ID of the device you specify. You can obtain the list of physical device tags and device IDs of devices on the network with a call to `nifGetDeviceList`.

Return Values

`E_OK`	The call was successful.
`E_INVALID_DESCRIPTOR`	The input descriptor is invalid.
`E_BAD_ARGUMENT`	The `tagOrDevID` value is not valid.
`E_NOT_FOUND`	No attached physical device has the specified device ID or physical device tag.

E_MULTIPLE	There is more than one device with the same tag or device ID on the same Fieldbus network.
E_COMM_ERROR	An error occurred when the NI-FBUS Communications Manager communicated with the device.
E_RESOURCES	A system resource problem occurred. The resource problem is usually a memory shortage.
E_OBSOLETE_DESC	The input descriptor is no longer valid. It was closed before `nifOpenPhysicalDevice` completed.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifOpenSession

Purpose

Returns a descriptor for an NI-FBUS Communications Manager session.

Format

```
nifError_t          nifOpenSession (
                    void *reserved,
                    nifDesc_t *out_ud)
```

Input

`reserved` Reserved for future use. You must set this value to NULL.

Output

`out_ud` A descriptor for the NI-FBUS Communications Manager communications entity you request.

Context

Not applicable.

Description

`nifOpenSession` returns a descriptor for the NI-FBUS Communications Manager session. When you open a session, the NI-FBUS Communications Manager establishes a communication channel between your application and the NI-FBUS entity. All subsequent descriptors you open are associated with this session, and all the NI-FBUS calls on these descriptors communicate with the NI-FBUS entity through the communication channel established during the `nifOpenSession` call.

The `reserved` argument is reserved for future use. You must set `reserved` to NULL.

Return Values

`E_OK` The call was successful.

`E_SERVER_NOT_RESPONDING`
Either the NI-FBUS Communications Manager server has not been started or the server, in its current state, cannot respond to the request.

`E_RESOURCES` A system resource problem occurred. The resource problem is usually a memory shortage or a failure of file access functions.

nifOpenVfd

Purpose

Returns a descriptor representing a Virtual Field Device (VFD).

Format

```
nifError_t          nifOpenVfd (
                    nifDesc_t ud,
                    char *vfdTag,
                    nifDesc_t *out_ud)
nifError_t          nifOpenVfd (
                    nifDesc_t ud,
                    NIFB_ORDINAL(n),
                    nifDesc_t *out_ud)
```

Input

`ud`	A valid physical device descriptor.
`vfdTag`	The tag of the VFD. To access by ordinal number within a physical device, use the `ORDINAL` macro in the `nifbus.h` header file.

Output

`out_ud`	A descriptor for the VFD you request.

Context

Physical device.

Description

`nifOpenVfd` returns a descriptor for the VFD you specify. More than one VFD can reside within a physical device. You must pass a valid physical device descriptor to this function.

There are two ways to specify the VFD: by tag and by ordinal number. To open the VFD by its tag, you must set the `vfdTag` parameter to the current tag of the VFD. The NI-FBUS Communications Manager returns an error if it finds more than one VFD with the same tag. You can obtain the list of VFD tags within a specified physical device with a call to `nifGetVFDList`.

To open the VFD by its ordinal number, use the `NIFB_ORDINAL` macro. The first VFD of your application in a physical device has the ordinal number zero. Notice that the Management VFDs are not included in the ordinal numbering scheme.

Return Values

E_OK	The call was successful.
E_INVALID_DESCRIPTOR	The input descriptor is invalid.
E_MULTIPLE	There are identical VFD tags.
E_ORDINAL_NUM_OUT_OF_RANGE	The ordinal number is out of the device range.
E_COMM_ERROR	An error occurred when the NI-FBUS Communications Manager communicated with the device.
E_NOT_FOUND	No VFD in the device has the specified VFD tag.
E_RESOURCES	A system resource problem occurred. The resource problem is usually a memory shortage.
E_SM_NOT_OPERATIONAL	The device is present, but cannot respond because it is at a default address.
E_OBSOLETE_DESC	The input descriptor is no longer valid. It was closed before `nifOpenVfd` completed.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.
E_BAD_DEVICE_DATA	The device returned some inconsistent information.

nifShutdownCM

Purpose

Closes the NI-FBUS Communications Manager.

Format

```
nifError_t          nifShutdownCM (
                    uint32 interval);
```

Input

`interval` — The maximum waiting time in milliseconds for closing the NI-FBUS Communications Manager process. If `interval` is set to 0, the default timeout value of 2000 milliseconds is used. If `interval` is set to `NIFB_TIMEOUT_INFINITE`, the function will return only when the NI-FBUS Communications Manager has been cleanly closed. If the interval time is exceeded, the NI-FBUS Communications Manager process will be forcefully closed.

Output

None.

Context

Not applicable.

Description

`nifShutdownCM` closes the NI-FBUS Communications Manager. The return value indicates whether the NI-FBUS Communications Manager has been forcibly closed. If the NI-FBUS Communications Manager cannot be closed normally within the `interval` time, it will be closed forcefully. The normal close can ensure all the system resources are cleaned up. The forceful close can't ensure that.

Return Values

`E_OK` — The NI-FBUS Communications Manager has been closed normally. The call was successful.

`E_SHUTDOWN_FORCE` — The NI-FBUS Communications Manager has been forcefully closed.

nifStartupCM

Purpose

Starts the NI-FBUS Communications Manager.

Format

```
nifError_t          nifStartupCM (
                    uint32 windowStyle,
                    uint32 interval);
```

Input

`windowStyle` Specifies the style of how the NI-FBUS Communications Manager main window is displayed. This parameter may be one of the following:

`NIFB_WND_STYLE_NORMAL` Activate and display the main window of the NI-FBUS Communications Manager.

`NIFB_WND_STYLE_MINIMIZE` Minimize the main window of the NI-FBUS Communications Manager.

`interval` The maximum waiting time in milliseconds for the NI-FBUS Communications Manager to complete the initialization. If `interval` is set to 0, this function will immediately return after the NI-FBUS Communications Manager process is created. If `interval` is set to `NIFB_TIMEOUT_INFINITE`, the function will return only when the NI-FBUS Communications Manager has completed initialization or an error has occurred. The total time of completing initialization depends on the number and the type of the FBUS interface cards.

Output

None.

Context

Not applicable.

Description

`nifStartupCM` launches the NI-FBUS Communications Manager. Depending on the `windowStyle` parameter, The NI-FBUS Communications Manager will be launched in normal style or minimized style.

Note `nifStartupCM` reads the NI-FBUS installation information from the registry to find the path of the NI-FBUS Communications Manager. If the specific

NI-FBUS system registry information cannot be found or is corrupt, this function will return an error code.

Return Values

E_OK	The NI-FBUS Communications Manager has launched successfully.
E_FILE_NOT_FOUND	The NI-FBUS Communications Manager binary cannot be found or is corrupt.
E_REGKEY_NOT_FOUND	The NI-FBUS system registry information cannot be found or is corrupt.
E_TIMEOUT	The NI-FBUS Communications Manager has started but the initialization procedure has not completed within the timeout period.
E_SERVER_CONNECTION_LOST	The NI-FBUS Communications Manager has encountered an error during initialization.

Core Fieldbus Functions

You can use the NI-FBUS core functions to access Fieldbus block parameters using any type of descriptor. Because there are several ways to identify the Fieldbus block parameters, the NI-FBUS core functions accept special interface macros for the `name` argument, as well as the standard *`TAG.PARAM`* identifier format. Refer to the *Using Interface Macros* section for tips on using the interface macros.

List of Core Functions

Table 5-2. List of Core Functions

Function	Purpose
`nifFreeObjectAttributes`	Frees an `nifAttributes_t` structure allocated during a previous call to `nifGetObjectAttributes`.
`nifFreeObjectType`	Frees an `nifObjTypeLinst_t` structure allocated during a previous call to `nifGetObjectType`.
`nifGetObjectAttributes`	Reads a single set of object attributes from the Device Description (DD).
`nifGetObjectName`	Returns the Object Dictionary symbol name of the specified object.
`nifGetObjectSize`	Returns the size in bytes of an object's value.
`nifGetObjectType`	Returns the Object Dictionary type of the specified object.
`nifReadObject`	Reads an object's value from a device.
`nifReadObjectList`	Reads the values of several objects from a device or several devices.
`nifWriteObject`	Writes a parameter value to a device.

nifFreeObjectAttributes

Purpose

Frees an `nifAttributes_t` structure allocated during a previous call to `nifGetObjectAttributes`.

Format

```
nifError_t          nifFreeObjectAttributes(
                    nifAttributes_t *attr)
```

Input

`attr`	Object attribute values your application reads using `nifGetObjectAttributes`.

Output

Not applicable.

Context

Session, block, VFD, physical device, link.

Description

`nifFreeObjectAttributes` frees up the memory associated with the `nifAttributes_t` structure specified by `attr`. `attr` must have been filled in by a successful call to `nifGetObjectAttributes`. Once this function has been called, the contents of `attr` are no longer valid.

If your application does not call this function after calling `nifGetObjectAttributes`, your application will not free up memory properly.

Return Values

`E_OK`	The call was successful.
`E_BAD_ARGUMENT`	`attr` was not a valid `nifAttributes_t` structure.

nifFreeObjectType

Purpose

Frees the `nifObjTypeList_t` structure allocated during a previous call to `nifGetObjectType`.

Format

```
nifError_t          nifFreeObjectType(
                    nifObjTypeList_t *typeData)
```

Input

`typeData`	Object Type values to be freed. These values were previously read with the `nifGetObjectType` function call.

Output

Not applicable.

Context

Session, block, VFD, physical device, link.

Description

`nifFreeObjectType` frees up the memory associated with the `nifObjTypeList_t` structure specified by `typeData`. `typeData` must have been filled in by a successful call to `nifGetObjectType`. Once this function has been called, the contents of `typeData` are no longer valid.

If your application does not call this function after calling `nifGetObjectType`, your application will not free up memory properly.

Refer to `nifGetObjectType` to get more details about the `nifObjTypeList_t` structure.

Return Values

`E_OK`	The call was successful.
`E_BAD_ARGUMENT`	`typeData` was not a valid `nifObjTypeList_t` structure.

nifGetObjectAttributes

Purpose

Reads a single set of object attributes from the Device Description (DD).

Format

```
nifError_t          nifGetObjectAttributes(
                    nifDesc_t ud,
                    char *name,
                    nifAttributes_t *attr)
nifError_t          nifGetObjectAttributes(
                    nifDesc_t ud,
                    NIFB_INDEX(uint32 idx),
                    nifAttributes_t *attr)
nifError_t          nifGetObjectAttributes(
                    nifDesc_t ud,
                    NIFB_INDEX_SUBINDEX(uint32 idx, uint32
                    subidx),
                    nifAttributes_t *attr)
nifError_t          nifGetObjectAttributes(nifDesc_t ud,
                    NIFB_ITEM(uint32 item),
                    nifAttributes_t *attr)
nifError_t          nifGetObjectAttributes(
                    nifDesc_t ud,
                    NIFB_ITEM_SUBINDEX(uint32 item, uint32
                    subidx),
                    nifAttributes_t *attr)
nifError_t          nifGetObjectAttributes(
                    nifDesc_t ud,
                    NIFB_BLOCK_ITEM(char *blocktag, uint32 item),
                    nifAttributes_t *attr)
nifError_t          nifGetObjectAttributes(
                    nifDesc_t ud,
                    NIFB_BLOCK_ITEM_SUBINDEX(char *blocktag,
                    uint32 item,
                    uint32 subidx),
                    nifAttributes_t *attr)
nifError_t          nifGetObjectAttributes(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX(char *blocktag, uint32 idx),
                    nifAttributes_t *attr)
```

```
nifError_t          nifGetObjectAttributes(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX_SUBINDEX(char *blocktag,
                    uint32 idx,
                    uint32 subidx),
                    nifAttributes_t *attr)
nifError_t          nifGetObjectAttributes(
                    nifDesc_t ud,
                    NIFB_NAME_SUBINDEX(char *name, uint32 subidx),
                    nifAttributes_t *attr)
nifError_t          nifGetObjectAttributes(
                    nifDesc_t ud,
                    NIFB_BLOCK_NAME_SUBINDEX(char *blocktag, char
                    *name,
                    uint32 subidx), nifAttributes_t *attr)
```

Input

`ud` The descriptor (of any type if by name; VFD or block if by index).

`name` Name of the object you need the device description attributes of, in *`BLOCKTAG.PARAM`* form. To specify a structure element by name, specify the name in *`BLOCKTAG.STRUCT.ELEMENT`* format. Refer to Table 5-5 for an explanation of how to use macros to specify the object.

Output

`attr` Object attribute values read from the DDOD (Device Description Object Dictionary). The type `nifAttributes_t` consists of a data structure including a type code which selects from a list of structures, one for each type of object. Other information, including whether individual attributes were successfully evaluated and whether individual attributes are dynamic (meaning they could change) also is provided. The structure is too long to be included in this chapter. You can find it in the NI-FBUS Communications Manager header files.

Context

Session, block, VFD, physical device, link.

Description

The NI-FBUS Communications Manager reads the device description object attributes identified in the call from the DDOD associated with `ud` and returned in `attr`. Notice that the object attributes describe certain characteristics of the object, but do not contain the object value. The device description object attributes also differ in content from the FMS Object Description of the object.

For block, VFD, physical device, or link descriptors, the object name may refer to a variable or a variable list. You normally would use `nifGetObjectAttributes` to read the type description of a certain data type.

Refer to Table 5-5 for an explanation of how to use macros to specify the object.

For more detailed information concerning the `nifAttributes_t` structure, refer to Chapter 3, *Using ddi_get_item*, of the *Fieldbus Foundation Device Description Services User Guide*.

Note After a successful call to `nifGetObjectAttributes`, your application must call `nifFreeObjectAttributes` when it is done using the `attr` structure. Your application will not free up memory correctly if it does not perform this operation.

Return Values

`E_OK`	The call was successful.
`E_CONFIG_ERROR`	Some configuration information, such as registry information or network configuration information, is incorrect.
`E_INVALID_DESCRIPTOR`	The device descriptor does not correspond to a VFD or block.
`E_SYMBOL_FILE_NOT_FOUND`	The NI-FBUS Communications Manager could not find the symbol file.
`E_SM_NOT_OPERATIONAL`	The device is present, but cannot respond because it is at a default address.
`E_NOT_FOUND`	The referred object does not exist, or it does not have object attributes.
`E_MULTIPLE`	The NI-FBUS Communications Manager found more than one identical tag; the function failed.
`E_ORDINAL_NUM_OUT_OF_RANGE`	The ordinal number is out of the device range.
`E_OBSOLETE_DESC`	The input descriptor is no longer valid. It was closed before `nifGetObjectAttributes` completed.
`E_SERVER_CONNECTION_LOST`	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifGetObjectName

Purpose

Returns the Object Dictionary symbol name of the specified object.

Format

```
nifError_t          nifGetObjectName(
                    nifDesc_t ud,
                    char *inName,
                    char *outName)
nifError_t          nifGetObjectName(
                    nifDesc_t ud,
                    NIFB_INDEX(uint32 idx),
                    char *outName)
nifError_t          nifGetObjectName(
                    nifDesc_t ud,
                    NIFB_INDEX_SUBINDEX(uint32 idx, uint32
                    subidx),
                    char *outName)
nifError_t          nifGetObjectName(
                    nifDesc_t ud,
                    NIFB_ITEM(uint32 item),
                    char *outName)
nifError_t          nifGetObjectName(
                    nifDesc_t ud,
                    NIFB_ITEM_SUBINDEX(uint32 item, uint32
                    subidx),
                    char *outName)
nifError_t          nifGetObjectName(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX(char *blocktag, uint32 idx),
                    char *outName)
nifError_t          nifGetObjectName(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX_SUBINDEX(char *blocktag,
                    uint32 idx, uint32 subidx),
                    char *outName)
nifError_t          nifGetObjectName(
                    nifDesc_t ud,
                    NIFB_NAME_SUBINDEX(char *name, uint32 subidx),
                    char *outName)
```

```
nifError_t          nifGetObjectName(
                    nifDesc_t ud,
                    NIFB_BLOCK_NAME_SUBINDEX (char *blocktag, char
                    *name, uint32 subidx),
                    char *outName)
```

Input

`ud` The descriptor of the session, link, physical device, VFD or block if you are accessing by name. If you are accessing by index, `ud` must be a VFD or block.

`inName` The name of the parameter you want to read the OD symbol name in *`BLOCKTAG.PARAM`* form. Refer to Table 5-5 for an explanation of how to use macros to specify the parameter. To specify a named structure element, supply name in *`BLOCKTAG.STRUCT.ELEMENT`* format.

Output

`outName` The Object symbol name read from the Object Dictionary in the device.

Context

Session, block, VFD, DDOD, physical device, link.

Description

`nifGetObjectName` is used to read the Object Dictionary symbol names of objects such as block, VFD, MIB objects, or communication objects from devices.

- If `ud` is the descriptor of a link, then `inName` must be in *`BLOCKTAG.PARAM_NAME`* format.
- If `ud` is a session descriptor, then all links are searched for the given *`BLOCKTAG.PARAM_NAME`*. The call fails if identical *`BLOCKTAG.PARAM_NAME`* tags are found on the bus. Index access is not allowed for session descriptors.
- If `ud` is the descriptor of a general function block application VFD, and you use the `NIFB_INDEX` macro, the index specified is the index of the object in the VFD.
- If `ud` is the descriptor of a function block, name must be in *`PARAM_NAME`* format.
- If `ud` is the descriptor of a function block, and you use the `NIFB_INDEX` or `NIFB_INDEX_SUBINDEX` macro, the index specified is the relative index of the parameter within the block. Relative indices start at one for the first parameter. Index zero retrieves the object dictionary symbol name of the block itself.
- In all cases, you can expand *`PARAM_NAME`* to *`STRUCT.ELEMENT`* format to represent a named element of a named structure.

Refer to Table 5-5 for an explanation of how to use macros to specify the parameter.

Return Values

E_OK	The call was successful.
E_INVALID_DESCRIPTOR	The descriptor you specified is not valid.
E_NOT_FOUND	The NI-FBUS Communication Manager could not find the specified object.
E_SYMBOL_FILE_NOT_FOUND	The NI-FBUS Communication Manager could not find the symbol file.
E_BAD_ARGUMENT	The object specified by index was that of a simple data type, which must already be known to you.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communication Manager, under which the descriptor was opened, has been lost or closed.
E_DEVICE_CHANGED	The device you specified is changed.
E_VFD_CHANGED	The VFD you specified is changed.
E_COMM_ERROR	An error occurred when the NI-FBUS Communication Manager tried to communicate with the device.
E_RESOURCE	The NI-FBUS Communications Manager is unable to allocate some system resource; this is usually a memory problem.
E_OBSOLETE_BLOCK	The block you specified is no longer valid.

nifGetObjectSize

Purpose

Returns the size (in bytes) of an object's value.

Format

```
nifError_t          nifGetObjectSize(
                    nifDesc_t ud,
                    char *name,
                    int16 *size_in_bytes)
nifError_t          nifGetObjectSize(
                    nifDesc_t ud,
                    NIFB_INDEX(uint32 idx),
                    int16 *size_in_bytes)
nifError_t          nifGetObjectSize(
                    nifDesc_t ud,
                    NIFB_INDEX_SUBINDEX(uint32 idx, uint32
                    subidx),
                    int16 *size_in_bytes)
nifError_t          nifGetObjectSize(
                    nifDesc_t ud,
                    NIFB_ITEM(uint32 item),
                    int16 *size_in_bytes)
nifError_t          nifGetObjectSize(
                    nifDesc_t ud,
                    NIFB_ITEM_SUBINDEX(uint32 item, uint32
                    subidx),
                    int16 *size_in_bytes)
nifError_t          nifGetObjectSize(
                    nifDesc_t ud,
                    NIFB_BLOCK_ITEM(char *blocktag, uint32 item),
                    int16 *size_in_bytes)
nifError_t          nifGetObjectSize(
                    nifDesc_t ud,
                    NIFB_BLOCK_ITEM_SUBINDEX(char *blocktag,
                    uint32 item,
                    uint32 subidx),
                    int16 *size_in_bytes)
nifError_t          nifGetObjectSize(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX(char *blocktag, uint32 idx),
                    int16 *size_in_bytes)
nifError_t          nifGetObjectSize(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX_SUBINDEX(char *blocktag,
```

```
                    uint32 idx, uint32 subidx),
                    int16 *size_in_bytes)
nifError_t          nifGetObjectSize(
                    nifDesc_t ud,
                    NIFB_NAME_SUBINDEX(char *name, uint32 subidx),
                    int16 *size_in_bytes)
nifError_t          ni1fGetObjectSize(
                    nifDesc_t ud,
                    NIFB_BLOCK_NAME_SUBINDEX(char *blocktag, char
                    *name, uint32 subidx),
                    int16 *size_in_bytes)
```

Input

`ud` The descriptor of a block.

`name` Character string name of the object you need the size of, in *`BLOCKTAG.PARAM`* form. To specify a structure element by name, specify the name in *`BLOCKTAG.STRUCT.ELEMENT`* format. Refer to Table 5-5 for an explanation of how to use macros to specify the character string name.

Output

`size_in_bytes` The size of the object.

Context

Session, block, VFD, physical device, link.

Description

This function returns the size of the specified Object Value. You have to pass a buffer of the returned size to `nifReadObject` to hold the value of the object.

Refer to Table 5-5 for an explanation of how to use macros to specify the character string name.

Return Values

`E_OK` The call was successful.

`E_INVALID_DESCRIPTOR`
The specified descriptor is invalid.

`E_SYMBOL_FILE_NOT_FOUND`
The NI-FBUS Communications Manager could not find the symbol file.

`E_NOT_FOUND` The named object does not exist.

`E_MULTIPLE` Multiple identical tags were found; the function failed.

`E_OBSOLETE_DESC` The input descriptor is no longer valid. It was closed before `nifGetObjectSize` completed.

`E_ORDINAL_NUM_OUT_OF_RANGE`

The ordinal number is out of the device range.

`E_SERVER_CONNECTION_LOST`

The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifGetObjectType

Purpose

Returns the Object Dictionary type of the specified object.

Format

```
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    char *objName,
                    nifObjTypeList_t *typeData)
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    NIFB_INDEX(uint32 idx),
                    nifObjTypeList_t *typeData)
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    NIFB_INDEX_SUBINDEX(uint32 idx, uint32
                    subidx),
                    nifObjTypeList_t *typeData)
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    NIFB_ITEM(uint32 item),
                    nifObjTypeList_t *typeData)
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    NIFB_ITEM_SUBINDEX(uint32 item, uint32
                    subidx),
                    nifObjTypeList_t *typeData)
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    NIFB_BLOCK_ITEM(char *blocktag, uint32 item),
                    nifObjTypeList_t *typeData)
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    NIFB_BLOCK_ITEM_SUBINDEX(char *blocktag,
                    uint32 item, uint32 subidx),
                    nifObjTypeList_t *typeData)
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX(char *blocktag, uint32 idx),
                    nifObjTypeList_t *typeData)
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX_SUBINDEX(char *blocktag,
                    uint32 idx, uint32 subidx),
                    nifObjTypeList_t *typeData)
```

```
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    NIFB_NAME_SUBINDEX(char *name, uint32 subidx),
                    nifObjTypeList_t *typeData)
nifError_t          nifGetObjectType(
                    nifDesc_t ud,
                    NIFB_BLOCK_NAME_SUBINDEX(char *blocktag, char
                    *name, uint32 subidx),
                    nifObjTypeList_t *typeData)
```

Input

`ud` The descriptor of the session, link, physical device, VFD, or block if you are accessing by name. If you are accessing by index, `ud` must be a VFD or block.

`objName` The name of the parameter you want to read the OD type of, in *`BLOCKTAG.PARAM`* form. Refer to Table 5-5 for an explanation of how to use macros to specify the parameter. To specify a named structure element, supply `name` in *`BLOCKTAG.STRUCT.ELEMENT`* format. To specify a type index returned by a previous call to `nifGetObjectType`, use the `NIFB_TYPE_INDEX` macro.

Output

`typeData` Object Type value read from the object dictionary in the device. The `nifObjTypeList_t` data structure is a record consisting of an object type code, the number of elements, the `blocktag` to which this object belongs (if applicable), and a pointer to a list of elements of type `nifObjElem_t`. The `nifObjElem_t` type is a structure which consists of two elements: the OD `typeIndex` of the element and the OD length of the element.

Context

Session, block, VFD, DDOD, physical device, link.

Description

`nifGetObjectType` is used to read the Object Dictionary type values of objects such as block parameters, MIB objects, or communication parameters from devices.

- If `ud` is the descriptor of a link, then `objName` must be in *`BLOCKTAG.PARAM_NAME`* format.
- If `ud` is a session descriptor, then all links are searched for the given *`BLOCKTAG.PARAM_NAME`*. The call fails if identical *`BLOCKTAG.PARAM_NAME`* tags are found on the bus. Index access is not allowed for session descriptors.

- If ud is the descriptor of a general function block application VFD, and you use the NIFB_INDEX macro, the index specified is the index of the object in the VFD.
- If ud is the descriptor of a function block, name must be in *PARAM_NAME* format.
- If ud is the descriptor of a function block, and you use the NIFB_INDEX or NIFB_INDEX_SUBINDEX macro, the index specified is the relative index of the parameter within the block. Relative indices start at one for the first parameter. Index zero retrieves the OD type of the block itself.
- In all cases, you can expand *PARAM_NAME* to *STRUCT.ELEMENT* format to represent a named element of a named structure.

Refer to Table 5-5 for an explanation of how to use macros to specify the parameter.

The nifObjTypeList_t data structure is defined as follows:

```
typedef struct {
    uint8 objectCode;
    uint16 numElems;
    char blockTag[TAG_SIZE + 1];
    nifObjElem_t *allElems;
    } nifObjTypeList_t;
```

The nifObjElem_t data type is defined as follows:

```
typedef struct {
    uint16 objTypeIndex;
    uint16 objSize;
    } nifObjElem_t;
```

The objectCode returned in the data structure nifObjTypeList_t is as specified in the *FMS Specifications* section of the *Fieldbus Foundation Specifications* document and is listed in Table 5-3, for your convenience.

Table 5-3. Object Codes for the nifObjTypeList_t Data Structure

Object	Object Code in fbtypes.h
Domain	ODT_DOMAIN
Program Invocation	ODT_PI
Event	ODT_EVENT
Data Type	ODT_SIMPLETYPE
Data Type Structure Description	ODT_STRUCTTYPE
Simple Variable	ODT_SIMPLEVAR
Array	ODT_ARRAY

Table 5-3. Object Codes for the `nifObjTypeList_t` Data Structure (Continued)

Object	Object Code in `fbtypes.h`
Record	ODT_RECORD
Variable List	ODT_VARLIST

For object codes `ODT_STRUCTTYPE`, `ODT_SIMPLEVAR`, `ODT_ARRAY`, and `ODT_RECORD`, the list of elements in `allElements` contains the `typeIndex` and the size of each component element. For example, the following fragment of pseudocode gets the type information for a structured object and does something with the type information for each element:

```
nifObjTypeList_t typeInfo;
nifDesc_t aiBlock;
int loop;
...
nifGetObjectType(aiBlock, "OUT", &typeInfo);
for (loop=0; loop < typeInfo.numElems; loop++)
{
    doSomethingWithElement(typeInfo.allElems[loop]);
}
```

For variable list objects (type `ODT_VARLIST`), you must call `nifGetObjectType` for each element in the list of elements with the `typeIndex` of the element returned in the list with the first `nifGetObjectType` call. The `typeIndex` of the element returned in the list in this case is the relative index of the element within the block, whose name is returned by `blockTag`. These subsequent calls to `nifGetObjectType` should use the `NIFB_INDEX` macro to specify the `typeIndex` returned by the first call.

For example, the following fragment of pseudocode gets the type information for a variable list object and does something with the type information for each variable:

```
nifObjTypeList_t typeInfo, varTypeInfo;
nifDesc_t aiBlock;
int loop;
...
nifGetObjectType(aiBlock, "VIEW_1", &typeInfo);
if (typeinfo.objectCode == ODT_VARLIST)
{
    for (loop=0; loop < typeInfo.numElems; loop++)
    {
        nifGetObjectType(aiBlock,
            NIFB_INDEX(typeInfo.allElems[loop].objTypeIndex),
            &varTypeInfo);
        doSomethingWithVariable(varTypeInfo);
    }
}
```

For all successful calls to `nifGetObjectType`, you must call `nifFreeObjectType` to clean up memory allocated within these structures.

For objects with the object codes `ODT_DOMAIN`, `ODT_PI`, `ODT_EVENT`, and `ODT_SIMPLETYPE`, only the object type is returned, and the list of elements `allElems` in the structure `nifObjTypeList_t` is empty. The list of standard data types for an object which has the object code `ODT_SIMPLETYPE` also is as specified in the *FMS Specifications* section of the *Fieldbus Foundation Specifications* document.

Table 5-4. Object Codes for the `nifObjTypeList_t` Data Structure

Data Type	**objTypeIndex in fbtypes.h**	Number of Octets (Size)
Boolean	FF_BOOLEAN	1
Integer8	FF_INTEGER8	1
Integer16	FF_INTEGER16	2
Integer32	FF_INTEGER32	4
Unsigned8	FF_UNSIGNED8	1
Unsigned16	FF_UNSIGNED16	2
Unsigned32	FF_UNSIGNED32	4
Floating Point	FF_FLOAT	4
Visible String	FF_VISIBLE_STRING	1, 2, 3, ...
Octet String	FF_OCTET_STRING	1, 2, 3, ...
Date	FF_DATE	7
Time of Day	FF_TIMEOFDAY	4 or 6
Time Difference	FF_TIME_DIFF	4 or 6
Bit String	FF_BIT_STRING	1, 2, 3, ...
Time Value	FF_TIME_VALUE	8

Return Values

`E_OK`	The call was successful.
`E_INVALID_DESCRIPTOR`	The descriptor you specified is not valid.
`E_TIMEOUT`	The device containing the object is present but did not respond within the timeout period.
`E_MULTIPLE`	More than one identical tag was found. The function failed.

E_NOT_FOUND	The NI-FBUS Communications Manager could not find the specified object.
E_BAD_ARGUMENT	The object specified by index was that of a simple data type, which must already be known to you.
E_RESOURCES	The NI-FBUS Communications Manager is unable to allocate some system resource. This is usually a memory problem.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager, under which the descriptor was opened, has been lost or closed.

nifReadObject

Purpose

Reads an object's value from a device.

Format

```
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    char *name,
                    void *buffer,
                    uint8 *length)
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    NIFB_INDEX(uint32 idx),
                    void *buffer, uint8 *length)
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    NIFB_INDEX_SUBINDEX(uint32 idx, uint32
                    subidx),
                    void *buffer, uint8 *length)
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    NIFB_ITEM(uint32 item),
                    void *buffer,
                    uint8 *length)
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    NIFB_ITEM_SUBINDEX(uint32 item, uint32
                    subidx),
                    void *buffer,
                    uint8 *length)
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    NIFB_BLOCK_ITEM(char *blocktag, uint32 item),
                    void *buffer, uint8 *length)
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    NIFB_BLOCK_ITEM_SUBINDEX(char *blocktag,
                    uint32 item, uint32 subidx),
                    void *buffer,
                    uint8 *length)
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX(char *blocktag, uint32 idx),
                    void *buffer,
                    uint8 *length)
```

```
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX_SUBINDEX(char *blocktag,
                    uint32 idx,
                    uint32 subidx),
                    void *buffer,
                    uint8 *length)
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    NIFB_NAME_SUBINDEX(char *name, uint32 subidx),
                    void *buffer,
                    uint8 *length)
nifError_t          nifReadObject(
                    nifDesc_t ud,
                    NIFB_BLOCK_NAME_SUBINDEX(char *blocktag, char
                    *name, uint32 subidx),
                    void *buffer,
                    uint8 *length)
```

Input

`ud`	The descriptor of the session, link, physical device, VFD or block if reading by name. If reading by index, `ud` must be a VFD or block.
`name`	Name of the parameter your application reads, in *`BLOCKTAG.PARAM`* format. To specify a structure element by name, specify the name in `BLOCKTAG.STRUCT.ELEMENT` format. Refer to Table 5-5 for an explanation of how to use macros to specify the parameter.
`length`	The size of the buffer to hold the result, in bytes.

Output

`buffer`	The value that the NI-FBUS Communications Manager reads.
`length`	The actual size of the resulting data, in bytes.

Context

Session, block, VFD, physical device, link.

Description

`nifReadObject` reads the values of objects such as block parameters or communications parameters from devices.

- If `ud` is the descriptor of a link, then `name` must be in the format *`BLOCKTAG.PARAM_NAME`*.

- If ud is a session descriptor, then all links are searched for the given *BLOCKTAG.PARAM_NAME*. The call fails if multiple identical *BLOCKTAG.PARAM_NAME* tags are located on the bus. Index access is not allowed for session descriptors.
- If ud is the descriptor of a general function block application VFD, then name must be in the format *BLOCKTAG.PARAM_NAME*.
- If ud is the descriptor of a function block, name must be in the format *PARAM_NAME*.
- If ud is the descriptor of a function block, and the NIFB_INDEX or NIFB_INDEX_SUBINDEX macro is used, the index specified is the relative index of the parameter within the block. Relative indices start at 1 for the first parameter.
- In all descriptor cases, you can expand *PARAM_NAME* itself to *STRUCT.ELEMENT* format to represent a named element of a named structure.

In each case, name can represent either a variable or a variable list object. You should determine the size of the object beforehand, possibly with a call to nifGetObjectSize. If the object is larger than the buffer size specified in length, the NI-FBUS Communications Manager returns an error, and none of the data in the buffer is valid.

Refer to Table 5-5 for an explanation of how to use macros to specify the parameter.

The data nifReadObject returns is in Fieldbus Foundation FMS Application format. You must accomplish conversion of the data to the internal format of your processor and compiler.

Return Values

E_OK	The call was successful.
E_INVALID_DESCRIPTOR	The descriptor does not correspond to a VFD or function block. This descriptor is no longer valid.
E_NOT_FOUND	The referred object does not exist.
E_OBJECT_ACCESS_DENIED	The NI-FBUS Communications Manager interface does not have the required privileges. The access group you belong to is not allowed to acknowledge the event, or the password you used is wrong.
E_MULTIPLE	The NI-FBUS Communications Manager found more than one identical tag. The function failed.
E_BUF_TOO_SMALL	The object is larger than your buffer.
E_SM_NOT_OPERATIONAL	The device is present, but cannot respond because it is at a default address.
E_SYMBOL_FILE_NOT_FOUND	The NI-FBUS Communications Manager could not find the symbol file.
E_OBSOLETE_DESC	The input descriptor is no longer valid. It was closed before nifReadObject completed.

E_COMM_ERROR	The NI-FBUS Communications Manager failed to communicate with the device.
E_PARAMETER_CHECK	The device reported a violation of parameter-specific checks.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifReadObjectList

Purpose

Reads the values of several objects from a device or several devices.

Format

```
nifError_t          nifReadObjectList (
                    nifDesc_t ud,
                    char **blkParamList,
                    uint16 numObjects,
                    void *buffer,
                    uint16 *length,
                    nifError_t *errArray)
```

Input

`ud`	The descriptor of the session, link, physical device, VFD, or block.
`blkParamList`	The list of parameter names your application reads in the form of *`BLOCKTAG.PARAM`*. To specify any parameter by index use the `NIFB_INDEX` macro. To specify any parameter that is an array or structure element by index and subindex, use the `NIFB_INDEX_SUBINDEX` macro. To specify a named structure element, supply the parameter name in the form of *`BLOCKTAG.STRUCT.ELEMENT`*.
`numObjects`	The number of parameter names specified in `blkParamList`. (The maximum number of objects that can be specified in `blkParamList` is given by the constant `MAX_LIST_ELEMS`.)
`length`	The size of the buffer to hold the result of all the parameter reads, in bytes.

Output

`buffer`	The values of all the parameters read, stored as a continuous string of bytes.
`length`	The cumulative size of the actual resulting data in bytes.
`errArray`	The error codes resulting from each parameter read. The error codes have a one-to-one correspondence with the order in which the parameters are specified in `blkParamList`.

Context

Session, link, device, VFD, block.

Description

nifReadObjectList reads the values of objects specified in the list, which may include block parameters or communication parameters from devices.

- If ud is the descriptor of a link, each name in blkParamList must be in the format *BLOCKTAG.PARAM_NAME*.
- If ud is a session descriptor, then all links are searched for any given name specified by the blocktag.param format in blkParamList. The read of this particular object fails if identical *BLOCKTAG.PARAM_NAME* tags are located on the bus. Index access is not allowed for session descriptors.
- If ud is the descriptor of a general function block application VFD, any name in blkParamList must be in the format blocktag.param_name.
- If ud is the descriptor of a function block, any name in blkParamList must be in the format *PARAM_NAME*.
- If ud is the descriptor of a function block and the NIFB_INDEX or NIFB_INDEX_SUBINDEX macro is used to specify a name in blkParamList, the index specified is the relative index of the parameter within the block. Relative indices start at 1 for the first block parameter.
- In all descriptor cases, any PARAM_NAME specified in blkParamList can be expanded to *STRUCT.ELEMENT* format to represent a named element of a named structure.

For each name specified in blkParamList, the name can either represent a variable or a variable list object. You should determine the size of each object specified in blkParamList beforehand, possibly with a call to nifGetObjectSize. If the cumulative size of all the objects specified in the list is larger than the buffer size specified in length, the NI-FBUS Communications Manager returns an error. The data in the buffer is valid for however many objects were successfully read. The success or failure of the read for every object specified in blkParamList is indicated in errArray, the array in which error codes are returned. The error code in the first element of errArray is the error code indicating success or failure upon read of the first object specified in blkParamList, and so on.

Refer to Table 5-5 for an explanation of how to use macros to specify the parameters in blkParamList.

The data nifReadObjectList returns is in Fieldbus Foundation FMS Application format. You must accomplish conversion of the data to the internal format of your processor and compiler.

Return Values

E_OK	The call was successful.
E_INVALID_DESCRIPTOR	The descriptor is no longer valid.
E_BUF_TOO_SMALL	The size of the data resulting from the read of all objects specified in the list is larger than your buffer.

E_RESOURCES	A system resource problem occurred. The resource problem is usually a memory shortage.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifWriteObject

Purpose

Writes a parameter value to a device.

Format

```
nifError_t          nifWriteObject(
                    nifDesc_t ud,
                    char *name,
                    void *buffer,
                    uint8 length)
nifError_t          nifWriteObject(
                    nifDesc_t ud,
                    NIFB_INDEX(uint32 idx),
                    void *buffer,
                    uint8 length)
nifError_t          nifWriteObject(
                    nifDesc_t ud,
                    NIFB_INDEX_SUBINDEX(uint32 idx, uint32
                    subidx),
                    void *buffer,
                    uint8 length)
nifError_t          nifWriteObject(
                    nifDesc_t ud,
                    NIFB_ITEM(uint32 item),
                    void *buffer,
                    uint8 length)
nifError_t          nifWriteObject(
                    nifDesc_t ud,
                    NIFB_ITEM_SUBINDEX(uint32 item, uint32
                    subidx),
                    void *buffer,
                    uint8 length)
nifError_t          nifWriteObject(
                    nifDesc_t ud,
                    NIFB_BLOCK_ITEM(char *blocktag, uint32 item),
                    void *buffer,
                    uint8 length)
nifError_t          nifWriteObject(
                    nifDesc_t ud,
                    NIFB_BLOCK_ITEM_SUBINDEX(char *blocktag,
                    uint32 item, uint32 subidx),
                    void *buffer,
                    uint8 length)
nifError_t          nifWriteObject(
                    nifDesc_t ud,
```

```
                    NIFB_BLOCK_INDEX(char *blocktag, uint32 idx),
                    void *buffer,
                    uint8 length)
nifError_t          nifWriteObject(
                    nifDesc_t ud,
                    NIFB_BLOCK_INDEX_SUBINDEX(char *blocktag,
                    uint32 idx,
                    uint32 subidx),
                    void *buffer,
                    uint8 length)
nifError_t          nifWriteObject(
                    nifDesc_t ud,
                    NIFB_NAME_SUBINDEX(char *name, uint32 subidx),
                    void *buffer,
                    uint8 length)
nifError_t          nifWriteObject(
                    nifDesc_t ud,
                    NIFB_BLOCK_NAME_SUBINDEX(char *blocktag, char
                    *name, uint32 subidx),
                    void *buffer,
                    uint8 length)
```

Input

`ud`	The descriptor of the session, link, physical device, VFD, or block, if writing by name. If writing by index, `ud` must be a VFD or block.
`name`	Name of the parameter you want the NI-FBUS Communications Manager to write, in *`BLOCKTAG.PARAM`* form. To specify a structure element by name, specify the name in *`BLOCKTAG.STRUCT.ELEMENT`* format. Refer to Table 5-5 for an explanation of how to use macros to specify the parameter.
`buffer`	The value you want the NI-FBUS Communications Manager to write.
`length`	The size of the data buffer, in bytes.

Output

Not applicable.

Context

Block, VFD, physical device, link, session.

Description

nifWriteObject writes the values of a function block parameter to a device.

- If ud is the descriptor of a session or link, then name must be in the format *BLOCKTAG.PARAM_NAME*.
- If ud is a session descriptor, then all links are searched for the given *BLOCKTAG.PARAM_NAME*. The function fails if more than one identical *BLOCKTAG.PARAM_NAME* match is found.
- If ud is a physical device descriptor, a parameter is written by *BLOCKTAG.PARAM_NAME*.
- If ud is the descriptor of a general Virtual Field Device, then name must be in the format *BLOCKTAG.PARAM_NAME*.
- If ud is the descriptor of a function block, name must be in the format *PARAM_NAME*.
- If ud is the descriptor of a function block, and you use the NIFB_INDEX or NIFB_INDEX_SUBINDEX macro, the index specified is the relative index of the parameter within the block. Relative indices start at one for the first parameter.
- In all descriptor cases, you can expand *PARAM_NAME* itself to *STRUCT.ELEMENT* format to represent a named element of a named structure.

Refer to Table 5-5 for an explanation of how to use macros to specify the parameter.

Return Values

E_OK	The call was successful.
E_INVALID_DESCRIPTOR	The device descriptor does not correspond to a VFD.
E_SYMBOL_FILE_NOT_FOUND	The NI-FBUS Communications Manager could not find the symbol file.
E_ORDINAL_NUM_OUT_OF_RANGE	The parameter is out of the device range.
E_OBJECT_ACCESS_UNSUPPORTED	The device does not support write access to this object.
E_MULTIPLE	The NI-FBUS Communications Manager found more than one identical tag. The function failed.
E_SM_NOT_OPERATIONAL	The device is present, but cannot respond because it is at a default address.
E_COMM_ERROR	The NI-FBUS Communications Manager failed to communicate with the device.
E_PARAMETER_CHECK	The device reported a violation of parameter-specific checks.
E_EXCEED_LIMIT	The device reported that the value exceeds the limit.
E_WRONG_MODE_FOR_REQUEST	The device reported that the current function block mode does not allow you to write to the parameter.

E_WRITE_IS_PROHIBITED

The device reported that the WRITE_LOCK parameter value is set. The WRITE_LOCK parameter prohibits writing to the name parameter.

E_DATA_NEVER_WRITABLE

The specified object is read-only.

E_SERVER_CONNECTION_LOST

The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

Using Interface Macros

This section contains tips for using the NI-FBUS Communications Manager interface macros. These macros are defined in the header file `nifbus.h`.

Table 5-5. Core Function Macros

Descriptor Type You Have	Parameter Information You Have	Macro to Use
Block Descriptor	Name	Normal Access by Name
	Name and Subindex	`NIFB_NAME_SUBINDEX`
	Relative Index within the Block	`NIFB_INDEX`
	Relative Index and Subindex	`NIFB_INDEX_SUBINDEX`
	Device Description Item ID	`NIFB_ITEM`
	Device Description Item ID and Subindex	`NIFB_ITEM_SUBINDEX`
Non-Block Descriptor	Name	Normal Access Using *`BLOCKTAG.PARAM`* Format
	Name and Subindex	`NIFB_BLOCK_NAME_SUBINDEX`
	Relative Index within the Block	`NIFB_BLOCK_INDEX`
	Relative Index and Subindex	`NIFB_BLOCK_INDEX_SUBINDEX`
	Device Description Item ID	`NIFB_BLOCK_ITEM`
	Device Description Item ID and Subindex	`NIFB_BLOCK_ITEM_SUBINDEX`

As shown in Table 5-5, you can specify the parameter your application reads in the `name` parameter in the following ways:

- To specify an object by index, use the `NIFB_INDEX` macro in the `nifbus.h` header file.
- To specify an array or structure element by index and subindex, use the `NIFB_INDEX_SUBINDEX` macro.
- If you already have a block descriptor, you can specify an object by its item ID with the `NIFB_ITEM` macro, or you can specify a subelement by its item ID with the `NIFB_ITEM_SUBINDEX` macro.

- If you do not have a block descriptor, you have the following choices:
  - You can use the NIFB_BLOCK_ITEM macro to specify an item.
  - You can use the NIFB_BLOCK_ITEM_SUBINDEX macro to specify a subelement.
  - You can use the NIFB_BLOCK_INDEX macro specify an object by index.
  - You can use the NIFB_BLOCK_INDEX_SUBINDEX macro to specify a subindex.

You can find all these macros in the nifbus.h header file.

Alert and Trend Functions

The following tables list the alert and trend functions.

Table 5-6. Alert Functions

Function	Purpose
nifAcknowledgeAlarm	Acknowledges an alarm received
nifWaitAlert	Waits for an alert (an event or an alarm) from a specific device or from *any* device
nifWaitAlert2	Waits for an alert (an event or an alarm) from a specific device or from *any* device. This function supports Standard Diagnostic Alert.

Table 5-7. Trend Function

Function	Purpose
nifWaitTrend	Waits for a trend from a specific device or from *any* device

nifAcknowledgeAlarm

Purpose

Acknowledges an alarm received.

Format

```
nifError_t          nifAcknowledgeAlarm(
                    nifDesc_t ud,
                    char *alarmName)
```

Input

`ud` — A session, link, physical device, VFD, or block descriptor for the alarm.

`alarmName` — The name of the alarm object that you want the NI-FBUS Communications Manager to acknowledge. If `ud` is a block descriptor, `alarmName` should be the parameter name, otherwise `alarmName` should be in *`BLOCKTAG.PARAMNAME`* format.

Context

Block, VFD, physical device, link, session.

Description

`nifAcknowledgeAlarm` acknowledges an alarm notification from a device. The NI-FBUS Communications Manager clears the `unacknowledged` field associated with the alarm object `alarmName`.

If `ud` is a block descriptor, the `alarmName` is the same as the `alarmOrEventName` field of the alert data you get in the `nifWaitAlert` or `nifWaitAlert2` call. If `ud` is a session, link, VFD, or physical device descriptor, then `alarmName` is in *`BLOCKTAG.PARAMNAME`* format, where `blockTag` is the same as the `blockTag` field of the alert data in the `nifWaitAlert` or `nifWaitAlert2` function.

Return Values

`E_OK` — The call was successful.

`E_INVALID_DESCRIPTOR` — The device descriptor is not a valid descriptor.

`E_OBJECT_ACCESS_DENIED` — The NI-FBUS Communications Manager interface does not have the required privileges. The access group you belong to is not allowed to acknowledge the event, or the password you used is wrong.

E_COMM_ERROR	An error occurred when the NI-FBUS Communications Manager tried to communicate with the device.
E_ALARM_ACKNOWLEDGED	The alarm has already been acknowledged.
E_MULTIPLE	There are identical block tags.
E_NOT_FOUND	There is no such block in the device or VFD with the specified tag.
E_SYMBOL_FILE_NOT_FOUND	The NI-FBUS Communications Manager could not find the symbol file.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifWaitAlert

Purpose

Waits for an alert (an event or an alarm) from a specific device or from *any* device.

Format

```
nifError_t          nifWaitAlert(
                    nifDesc_t ud,
                    nifAlertData_t *aldata,
                    uint8 alertPriority)
```

Input

`ud`	The descriptor of the session, link, physical device, VFD, block, or link the alert comes from.
`alertPriority`	Lowest priority of the alert coming in that you want to wait on.

Output

`aldata`	The information about the specific alert.

Context

Block, VFD, physical device, link, session.

Description

`nifWaitAlert` only supports normal alert types and does not support Standard Diagnostics Alert. It is recommended to use `nifWaitAlert2` instead.

`ud` represents a descriptor of a session, link, a physical device, a VFD, or a block. If `ud` is a VFD descriptor, then the NI-FBUS Communications Manager waits for an alert from any block in the Virtual Field Device. If `ud` is a block, the NI-FBUS Communications Manager waits for an alarm or event from the block `ud` refers to. If `ud` represents a link, `nifWaitAlert` completes when an event is received from any device connected to that link. If the descriptor is a session descriptor, the function waits on any event from any attached link.

`nifWaitAlert` waits indefinitely until the NI-FBUS Communications Manager receives an alert with a priority greater than or equal to the input alert priority. Your application can have a dedicated thread which does `nifWaitAlert` only.

When the NI-FBUS Communications Manager interface receives an alert, the `aldata` parameter is filled in with the information about the `aldata`. The form of `aldata->alertData` depends on the value of `aldata->alertType`. `aldata->alarmOrEventName` is the name of the alarm parameter or event parameter that caused the alert. `aldata->deviceTag` and `aldata->blockTag` are the tags of the device and the block of the alarm, respectively.

nifWaitAlert sends a confirmation to the device, informing the alerting device that the alert was received. Note that this is a separate step from alert acknowledgment, which must be carried out for alarms using nifAcknowledgeAlarm.

If you have multiple threads waiting to receive the same alert, the NI-FBUS Communications Manager sends a copy of the alert to all the waiting threads. Your application must ensure that only one thread acknowledges any one alarm with a call to nifAcknowledgeAlarm. You can abort a pending nifWaitAlert call by closing the descriptor on which the call was made.

The alertType parameter can be ALERT_ANALOG, ALERT_DISCRETE, or ALERT_UPDATE.

nifAlertData_t is defined as follows:

```
typedef struct nifAlertData_t{
    uint8 alertType;
    char deviceTag[TAG_SIZE + 1];
    char blockTag[TAG_SIZE + 1];
    char alarmOrEventName [TAG_SIZE + 1];
    uint8 alertKey;
    uint8 standardType;
    uint8 mfrType;
    uint8 messageType;
    uint8 priority;
    nifTime_t timeStamp;
    uint16 subCode;
    uint16 unitIndex;
    union {
        float floatAlarmData;
        uint8 discreteAlarmData;
        uint16 staticRevision;
    } alertData;
} nifAlertData_t;
```

Return Values

E_OK	The call was successful.
E_INVALID_DESCRIPTOR	The descriptor you gave is invalid.
E_OBSOLETE_DESC	The input descriptor is no longer valid. It was closed before nifWaitAlert completed.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifWaitAlert2

Purpose

Waits for an alert (an event or an alarm) from a specific device or from *any* device. `nifWaitAlert2` supports Standard Diagnostics Alert.

Format

```
nifError_t          nifWaitAlert2(
                    nifDesc_t ud,
                    nifAlertData2_t *aldata,
                    uint8 alertPriority)
```

Input

`ud`	The descriptor of the session, link, physical device, VFD, block, or link the alert comes from.
`alertPriority`	Lowest priority of the alert coming in that you want to wait on.

Output

`aldata`	The information about the specific alert (supports Standard Diagnostics Alert).

Context

Block, VFD, physical device, link, session.

Description

`nifWaitAlert2` is compatible with all of the alert types of `nifWaitAlert`, and `nifWaitAlert2` is able to support Standard Diagnostics Alert. It is recommended that you use `nifWaitAlert2` instead of `nifWaitAlert`.

`ud` represents a descriptor of a session, link, a physical device, a VFD, or a block. If `ud` is a VFD descriptor, then the NI-FBUS Communications Manager waits for an alert from any block in the Virtual Field Device. If `ud` is a block, the NI-FBUS Communications Manager waits for an alarm or event from the block `ud` refers to. If `ud` represents a link, `nifWaitAlert2` completes when an event is received from any device connected to that link. If the descriptor is a session descriptor, the function waits on any event from any attached link.

`nifWaitAlert2` waits indefinitely until the NI-FBUS Communications Manager receives an alert with a priority greater than or equal to the input alert priority. Your application can have a dedicated thread which does `nifWaitAlert2` only.

When the NI-FBUS Communications Manager interface receives an alert, the aldata parameter is filled in with the information about the aldata. The form of aldata->alertData depends on the value of aldata->alertType. aldata->alarmOrEventName is the name of the alarm parameter or event parameter that caused the alert. aldata->deviceTag and aldata->blockTag are the tags of the device and the block of the alarm, respectively.

nifWaitAlert2 sends a confirmation to the device, informing the alerting device that the alert was received. Note that this is a separate step from alert acknowledgment, which must be carried out for alarms using nifAcknowledgeAlarm.

If you have multiple threads waiting to receive the same alert, the NI-FBUS Communications Manager sends a copy of the alert to all the waiting threads. Your application must ensure that only one thread acknowledges any one alarm with a call to nifAcknowledgeAlarm. You can abort a pending nifWaitAlert2 call by closing the descriptor on which the call was made.

The alertType parameter can be ALERT_ANALOG, ALERT_DISCRETE, ALERT_UPDATE, or or ALERT_STANDARD_DIAGNOSTICS.

nifAlertData2_t is defined as follows:

```
typedef struct nifAlertData2_t{
    uint8 alertType;
    char deviceTag[TAG_SIZE + 1];
    char blockTag[TAG_SIZE + 1];
    char alarmOrEventName [TAG_SIZE + 1];
    uint8 alertKey;
    uint8 standardType;
    uint8 mfrType;
    uint8 messageType;
    uint8 priority;
    uint8 reserved[3];
    nifTime_t timeStamp;
    union {
        nifAlertAnalogData_t analog;
        nifAlertDiscreteData_t discrete;
        nifAlertUpdateData_t update;
        nifAlertStandardDiagnosticsData_t stdDiag;
    } alertData;
} nifAlertData2_t;
typedef struct nifAlertAnalogData_t {
        uint16  subCode;
        float   value;
        uint32  relativeIndex;
        uint16  unitIndex;
} nifAlertAnalogData_t;
```

```
typedef struct nifAlertDiscreteData_t {
       uint16  subCode;
       uint8   value;
       uint32  relativeIndex;
       uint16  unitIndex;
} nifAlertDiscreteData_t;
typedef struct nifAlertUpdateData_t {
       uint16  staticRevision;
       uint32  relativeIndex;
} nifAlertUpdateData_t;
typedef struct nifAlertStandardDiagnosticsData_t {
       uint32  subCode;
       uint8   value;
       uint32  relativeIndex;
       uint16  sourceBlockIndex;
} nifAlertStandardDiagnosticsData_t;
```

Return Values

E_OK	The call was successful.
E_INVALID_DESCRIPTOR	The descriptor you gave is invalid.
E_OBSOLETE_DESC	The input descriptor is no longer valid. It was closed before nifWaitAlert2 completed.
E_SERVER_CONNECTION_LOST	The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

nifWaitTrend

Purpose

Waits for a trend from a specific device or from any device.

Format

```
nifError_t          nifWaitTrend(
                    nifDesc_t ud,
                    nifTrendData_t *trend)
```

Input

`ud` The descriptor of the session, physical device, VFD, block, or link that the trend comes from.

Output

`trend` The information about the specific trend.

Context

Block, VFD, physical device, link, session.

Description

`ud` represents a descriptor of a session, link, physical device, VFD, or block. If `ud` is a VFD descriptor, then the NI-FBUS Communications Manager waits for a trend from any block in the Virtual Field Device. If `ud` is a block, the NI-FBUS Communications Manager waits for a trend from the block `ud` identifies. If `ud` represents a link, the call completes when a trend is received from any device connected to that link. If the descriptor is a session descriptor, `nifWaitTrend` waits on any trend from any attached link.

`nifWaitTrend` waits indefinitely until the NI-FBUS Communications Manager interface receives a trend. Your application can have a dedicated thread which does `nifWaitTrend` only.

When a trend comes in, the `trend` parameter is filled in with the information about the trend. The form of `trend->trendData` depends on the value `of trend->trendType`. There are three trend types: `TREND_FLOAT`, `TREND_DISCRETE`, and `TREND_BITSTRING`. If the trend type is `TREND_FLOAT`, the `trend->trendData` is a 16-element array of floating point numbers. If the trend type is `TREND_DISCRETE`, the `trend->trendData` is a 16-element array of 1-byte integers. If the trend type is `TREND_BITSTRING`, the `trend->trendData` is a 16-element array of 2-byte bit strings, which is equivalent to a 32-element array of 1-byte integers. `deviceTag` and `blockTag` are the device and block tags of the parameter that has the trend; `paramName` is the name of the parameter.

If you have multiple threads waiting to receive the same trend, the NI-FBUS Communications Manager sends a copy of the trend to all the waiting threads. You can abort a pending `nifWaitTrend` call by closing the descriptor on which the call was made.

The `trend` type can be `TREND_FLOAT`, `TREND_DISCRETE`, or `TREND_BITSTRING`. The sample type can be `SAMPLE_INSTANT` or `SAMPLE_AVERAGE`.

`nifTrendData_t` is defined as follows:

```
typedef struct nifTrendData_t {
    uint8 trendType;
    char deviceTag[TAG_SIZE + 1];
    char blockTag[TAG_SIZE + 1];
    char paramName[TAG_SIZE + 1];
    uint8 sampleType;
    uint32 sampleInterval;
    nifTime_t lastUpdate;
    uint8 status[16];
    union {
        float f[16];
        uint8 d[16];
        uint8 bs[32];
    } trendData;
} nifTrendData_t;
```

Return Values

`E_OK` The call was successful.

`E_INVALID_DESCRIPTOR`
The descriptor you gave is not valid.

`E_SERVER_CONNECTION_LOST`
The session established with the NI-FBUS Communications Manager for this descriptor has been closed or lost.

A

Specifications

This appendix lists the hardware specifications and interface cabling information for the PCI-FBUS, PCMCIA-FBUS, and USB-8486.

PCI-FBUS/2

Power Requirement

PCI-FBUS/2 ..... 820 mA Typical

Physical

Dimensions ..... 10.67 × 17.46 cm (4.2 × 6.88 in.)
I/O connector ..... 9-pin male D-SUB (1 per Fieldbus link)
Altitude ..... 2,000 m
Pollution Degree 2
Indoor use only.

Environment

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

Safety

This product meets the requirements of the following standards of safety for electrical equipment for measurement, control, and laboratory use:

- IEC 60950-1, EN 60950-1
- UL 60950-1, CSA 60950-1

Note For UL and other safety certifications, refer to the product label or the *Online Product Certification* section.

Electromagnetic Compatibility

This product meets the requirements of the following EMC standards for electrical equipment for measurement, control, and laboratory use:

- EN 61326-1 (IEC 61326-1): Class A emissions; Basic immunity
- EN 55011 (CISPR 11): Group 1, Class A emissions
- AS/NZS CISPR 11: Group 1, Class A emissions
- FCC 47 CFR Part 15B: Class A emissions
- ICES-001: Class A emissions

Note For EMC declarations and certifications, refer to the *Online Product Certification* section.

Note For EMC compliance, operate this device with shielded cables and accessories.

CE Compliance CE

This product meets the essential requirements of applicable European Directives as follows:

- 2006/95/EC; Low-Voltage Directive (safety)
- 2004/108/EC; Electromagnetic Compatibility Directive (EMC)

Online Product Certification

To obtain product certifications and the Declaration of Conformity (DoC) for this product, visit `ni.com/certification`, search by model number or product line, and click the appropriate link in the Certification column.

Environmental Management

NI is committed to designing and manufacturing products in an environmentally responsible manner. NI recognizes that eliminating certain hazardous substances from our products is beneficial to the environment and to NI customers.

For additional environmental information, refer to the *Minimize Our Environmental Impact* web page at `ni.com/environment`. This page contains the environmental regulations and directives with which NI complies, as well as other environmental information not included in this document.

Waste Electrical and Electronic Equipment (WEEE)

EU Customers At the end of the product life cycle, all products *must* be sent to a WEEE recycling center. For more information about WEEE recycling centers, National Instruments WEEE initiatives, and compliance with WEEE Directive 2002/96/EC on Waste and Electronic Equipment, visit `ni.com/environment/weee`.

电子信息产品污染控制管理办法 （中国 RoHS）

中国客户 National Instruments 符合中国电子信息产品中限制使用某些有害物质指令 (RoHS)。关于 National Instruments 中国 RoHS 合规性信息，请登录 `ni.com/environment/rohs_china`。 (For information about China RoHS compliance, go to `ni.com/environment/rohs_china`.)

PCMCIA-FBUS

Note The PCMCIA-FBUS here stands for PCMCIA-FBUS Series 2 card, and the PCMCIA-FBUS/2 below stands for PCMCIA-FBUS/2 Series 2 card.

Power Requirement

+5 VDC (±5%)

PCMCIA-FBUS.....350 mA typical; active

PCMCIA-FBUS/2.....350 mA typical; active

Physical

Dimensions .....8.56 × 5.40 × 0.5 cm (3.4 × 2.1 × 0.2 in.)

I/O connector .....PCMCIA-FBUS cable with 9-pin male D-SUB and pluggable screw terminal for each port

Altitude .....2,000 m

Pollution Degree 2

Indoor use only.

Environment

Operating Environment

Ambient temperature .....0 to 55 °C

Relative humidity.....10 to 90%, noncondensing (tested in accordance with IEC-60068-2-1, IEC-60068-2-2, EC-60068-2-56)

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, EC-60068-2-56)

Safety

This product meets the requirements of the following standards of safety for electrical equipment for measurement, control, and laboratory use:

- IEC 60950-1, EN 60950-1
- UL 60950-1, CSA 60950-1

Note For UL and other safety certifications, refer to the product label or the *Online Product Certification* section.

Electromagnetic Compatibility

This product meets the requirements of the following EMC standards for electrical equipment for measurement, control, and laboratory use:

- EN 61326-1 (IEC 61326-1): Class A emissions; Basic immunity
- EN 55011 (CISPR 11): Group 1, Class A emissions
- AS/NZS CISPR 11: Group 1, Class A emissions
- FCC 47 CFR Part 15B: Class A emissions
- ICES-001: Class A emissions

Note For EMC declarations and certifications, refer to the *Online Product Certification* section.

Note For EMC compliance, operate this device with shielded cables and accessories.

CE Compliance CE

This product meets the essential requirements of applicable European Directives as follows:

- 2006/95/EC; Low-Voltage Directive (safety)
- 2004/108/EC; Electromagnetic Compatibility Directive (EMC)

Online Product Certification

To obtain product certifications and the Declaration of Conformity (DoC) for this product, visit `ni.com/certification`, search by model number or product line, and click the appropriate link in the Certification column.

Environmental Management

NI is committed to designing and manufacturing products in an environmentally responsible manner. NI recognizes that eliminating certain hazardous substances from our products is beneficial to the environment and to NI customers.

For additional environmental information, refer to the *Minimize Our Environmental Impact* web page at `ni.com/environment`. This page contains the environmental regulations and directives with which NI complies, as well as other environmental information not included in this document.

Waste Electrical and Electronic Equipment (WEEE)

EU Customers At the end of the product life cycle, all products *must* be sent to a WEEE recycling center. For more information about WEEE recycling centers, National Instruments WEEE initiatives, and compliance with WEEE Directive 2002/96/EC on Waste and Electronic Equipment, visit `ni.com/environment/weee`.

电子信息产品污染控制管理办法 （中国 RoHS）

中国客户 National Instruments 符合中国电子信息产品中限制使用某些有害物质指令 (RoHS)。关于 National Instruments 中国 RoHS 合规性信息，请登录 `ni.com/environment/rohs_china`。 (For information about China RoHS compliance, go to `ni.com/environment/rohs_china`.)

USB-8486

This section lists specifications for the USB-8486 hardware.

Bus Interface

USB ..... USB 2.0 High-Speed or Full-Speed[1]

FOUNDATION Fieldbus ..... Standard H1 interface[2]

Power Requirement

USB High-power Bus-powered Device

Working Mode Current ..... 300 mA maximum (full temperature range)
180 mA typical (at 25 °C)

Suspend Current ..... 2.5 mA maximum(full temperature range)

Physical

USB-8486 without Screw Retention and Mounting Options

Dimensions ..... 7.87 × 6.35 × 2.54 cm (3.1 × 2.5 × 1.0 in.)

Weight ..... 165 g (5.82 oz)

Captive USB cable length ..... 2 m

I/O connector

- USB ..... Standard series A plug
- FOUNDATION Fieldbus
  H1 Interface ..... 9-pin male D-SUB

Altitude ..... 2,000 m

Pollution Degree 2

Indoor use only.

USB-8486 with Screw Retention and Mounting Options

Dimensions ..... 8.61 × 6.35 × 2.98 cm (3.39 × 2.5 × 1.18 in.)

Weight ..... 175 g (6.17 oz)

Captive USB cable length ..... 1 m

I/O connector

- USB ..... Standard series A plug with retention thumbscrew
- FOUNDATION Fieldbus
  H1 Interface ..... 9-pin male D-SUB

[1] Using the USB-8486 in full-speed mode reduces device performance.

[2] Galvanically isolated.

Altitude ..... 2,000 m
Pollution Degree 2

Indoor use only.

Environment

Operating Environment

Ambient temperature ..... 0 to 55 °C
Relative humidity ..... 10 to 90%, noncondensing
(tested in accordance with IEC-60068-2-1, IEC-60068-2-2, EC-60068-2-56)

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, EC-60068-2-56)

Safety

This product meets the requirements of the following standards of safety for electrical equipment for measurement, control, and laboratory use:

- IEC 60950-1, EN 60950-1
- UL 60950-1, CSA 60950-1

Note For UL and other safety certifications, refer to the product label or the *Online Product Certification* section.

Electromagnetic Compatibility

This product meets the requirements of the following EMC standards for electrical equipment for measurement, control, and laboratory use:

- EN 61326-1 (IEC 61326-1): Class A emissions; Basic immunity
- EN 55011 (CISPR 11): Group 1, Class A emissions
- AS/NZS CISPR 11: Group 1, Class A emissions
- FCC 47 CFR Part 15B: Class A emissions
- ICES-001: Class A emissions

Note For EMC declarations and certifications, refer to the *Online Product Certification* section.

Note For EMC compliance, operate this device with shielded cables and accessories.

CE Compliance CE

This product meets the essential requirements of applicable European Directives as follows:

- 2006/95/EC; Low-Voltage Directive (safety)
- 2004/108/EC; Electromagnetic Compatibility Directive (EMC)

Online Product Certification

To obtain product certifications and the Declaration of Conformity (DoC) for this product, visit `ni.com/certification`, search by model number or product line, and click the appropriate link in the Certification column.

Environmental Management

NI is committed to designing and manufacturing products in an environmentally responsible manner. NI recognizes that eliminating certain hazardous substances from our products is beneficial to the environment and to NI customers.

For additional environmental information, refer to the *Minimize Our Environmental Impact* web page at `ni.com/environment`. This page contains the environmental regulations and directives with which NI complies, as well as other environmental information not included in this document.

Waste Electrical and Electronic Equipment (WEEE)

EU Customers At the end of the product life cycle, all products *must* be sent to a WEEE recycling center. For more information about WEEE recycling centers, National Instruments WEEE initiatives, and compliance with WEEE Directive 2002/96/EC on Waste and Electronic Equipment, visit `ni.com/environment/weee`.

电子信息产品污染控制管理办法 （中国 RoHS）

中国客户 National Instruments 符合中国电子信息产品中限制使用某些有害物质指令 (RoHS)。关于 National Instruments 中国 RoHS 合规性信息，请登录 `ni.com/environment/rohs_china`。(For information about China RoHS compliance, go to `ni.com/environment/rohs_china`.)

B

Troubleshooting and Common Questions

This appendix describes how to troubleshoot common problems that occur while getting started with Fieldbus hardware and software products.

Interface Board—USB, PCI, and PCMCIA

Error Messages

Utility could not access or locate the registry. Make sure you are logged in to Windows with administrator privileges.

Your registry entries for NI-FBUS may have been deleted or corrupted. Uninstall the NI-FBUS software, then reinstall the software.

Board cannot be found.

- Launch the Interface Configuration Utility and ensure that your board appears under the list of interfaces.
- Ensure the NIFB driver is started by verifying the settings in the Windows **Device Manager**. Select **Start»Control Panel»System»Hardware»Device Manager** to launch the **Device Manager**. Select **National Instruments FieldBus Interfaces»USB-8486** or **PCI-FBUS/2** or **PCMCIA-FBUS**. Right-click the interface and select **Properties**. Ensure no conflicts appear.

When using a USB-8486, Nifb returns an error message stating that the configured board does not exist.

Ensure that the USB-8486 has not been unplugged.

If you want to use the USB-8486 again, complete the following steps.

1. Connect the USB-8486 to an available USB port on the system.
2. Launch the Interface Configuration Utility
3. Right-click the USB-8486 to enable it.

If you want to use other interfaces in the system without this USB-8486, complete the following steps.

1. Launch the Interface Configuration Utility.
2. Right-click the USB-8486 to delete it.

VCR_FULL_ERROR.

Delete the board from the Interface Configuration Utility, then reinstall.

Interface Configuration Problems

When using the NI-FBUS Interface Configuration Utility, the error message `utility could not access or locate the registry` appears.

- Ensure that you are logged in to Windows with administrator privileges.
- Your registry entries for NI-FBUS may have been deleted or corrupted. Uninstall the NI-FBUS software, then reinstall the software.

In the Interface Configuration Utility, I see more boards than what physically exist in the machine.

Select **Edit** for the extra board. In the next window, select **Delete**.

Caution You should *not* attempt to make unguided changes in the Windows registry. Doing so can cause many problems with your system.

NIFB Problems

When a Fieldbus device is connected to the bus, the NIFB process often hangs when the title bar reads Waiting for Startup Completion. If I disconnect the cables, it starts fine.

This is probably due to a device address conflict. In the NI-FBUS Interface Configuration Utility, ensure that the interface is not at the same address as anything else on the link. You also can temporarily give the interface a visitor address to troubleshoot this problem.

The NIFB process hangs, does not start up, or never shows that it is running.

- The Fieldbus network address is not unique. Remove the cable from the board. Restart the NIFB process. If it runs successfully, there is probably a Fieldbus network address conflict. You can try to change the board address. In the Interface Configuration Utility, select the port and click **Edit**. Ensure that the port does not have an address that conflicts with another device on the bus. You also can set the interface to a visitor address. In this case, the board will find and take an unused address. If this corrects the problem, find and change the address of one of the conflicting devices. Return the board to a fixed address.
- Check for multiple copies of `nifb.dll` on the machine. If multiple copies are found, NI-FBUS was incorrectly reinstalled. Uninstall NI-FBUS, search for any remaining copies of `nifb.dll`, delete them, then reinstall the software.
- Check to see how many boards are showing up in the Interface Configuration Utility. Ensure that this matches the number of boards in the system. Also check that the number of ports match the physical hardware (one port versus two port boards).

- Link masters do not always work well together (if you have another link master on the link). Try setting the board to be a basic device in the Interface Configuration Utility.

  If a board interface is configured as a basic device, another link master device must be present on this link before the NI-FBUS process will start up. For more about Basic and Link Master devices, refer to the *FOUNDATION™ Fieldbus Overview* document.

  1. Launch the Interface Configuration Utility.
  2. In the Interface Configuration window, select the icon of the board you want to change and click the **Edit** button. If you are adding a board, click the **Add Interface** button.

Problems Using Manufacturer-Defined Features

NI-FBUS uses identifying information in the actual device to locate the device description for the device. The identifying information includes four resource block parameters: `MANUFAC_ID`, `DEV_TYPE`, `DEV_REV`, and `DD_REV`. If the identifying information is incorrect, NI-FBUS will not be able to locate the device description for the device. When it has located the device description, NI-FBUS matches the block types in the device description with the actual blocks in the device by using the Item ID of the block characteristics record.

If the blocks in the device do not match the blocks in the description, or if there is no appropriate device description for the manufacturer, device type, device revision, and device description revision being returned by the device, then there is a device description mismatch. In either case, NI-FBUS uses only the standard dictionary (`nifb.dct`) and you will be unable to use any manufacturer-supplied functionality.

These parameters can be read from the device resource block. The following procedure will help you troubleshoot a `DD_SIZE_MISMATCH_ERROR` by finding out if there is a device description available on your computer that matches what your device expects.

Complete the following steps to use the NI-FBUS Dialog utility to check device description files.

1. Start the NIFB process. Wait until the process has finished initializing.
2. Select the Dialog utility.
3. Right-click **Open Descriptors** and select **Expand All**.
4. After the expansion is complete, click **Cancel** to close the Expand All window.
5. Right-click the resource block for your device (it should be under **Open Descriptors» Session»Interface Name»Device Name»VFD Name»Resource Block Name**). Select **Read Object**.
6. Select the **Read by Name** radio button and enter *`MANUFAC_ID`* as the name. Click the **Read** button. Write down the hexidecimal number found in parenthesis (0xnumber) in the name column of Table B-1.
7. Repeat step 6 for the name *`DEV_TYPE`*.
8. Repeat step 6 for the name *`DEV_REV`*.
9. Repeat step 6 for the name *`DD_REV`*.

10. Repeat steps 5 through 9 for each device, then close the NI-FBUS Dialog utility.

Table B-1. Device Names

Resource Block Parameter	Name
MANUFAC_ID	
DEV_TYPE	
DEV_REV	
DD_REV	

11. In the Interface Configuration Utility, click the **DD Info** button. Write down the base directory specified for device descriptions. Close the Interface Configuration Utility.
12. Use Windows Explorer to view the contents of the base directory specified in the Interface Configuration Utility. The Fieldbus specification defines the directory hierarchy for storing device descriptions. There is a different subdirectory for each device manufacturer. Under the base directory, you should see a directory with the number from step 6 for the first device.
13. Under the appropriate manufacturer directory, there is a directory for each device type that you have from that manufacturer. Check to make sure that you see a directory with the number from step 7.
14. Under the appropriate device type directory, there are the individual device descriptions. The device description file name is a combination of the device revision (the number from step 8) and the device description revision (the number from step 9). The device revision is the first two digits, and the device description revision is the second two digits. For example, if your number from step 8 was 2 and from step 9 was 1, you should see files called `0201.ffo` and `0201.sym`. Device descriptions are backward-compatible. This means that instead of seeing 0201, you might see 0202. This is allowed by the Fieldbus specification. Also, having additional files in this directory is not a problem. The NI-FBUS Configurator will use the most recent device description revision for a given device revision. If you do not have the appropriate `.ffo` and `.sym` files, you must obtain them from the device manufacturer. Be sure to properly import them by clicking **DD Info** and using the **Import DD** button in the Interface Configuration Utility.
15. Repeat steps 12 through 14 for each device.

The second cause for this problem is when the contents of the file do not accurately describe the device characteristics, even if the device identification information matches the file identification information. This problem is caused when a device manufacturer makes a change to the firmware of the device without incrementing the device revision, in violation of the FOUNDATION Fieldbus recommendation. If this is the case, you must contact your device manufacturer for a resolution.

USB-8486 Troubleshooting

The H1 Fieldbus LED flashes red.

The USB-8486 encountered an error during the Power-On Self-Test (P.O.S.T.). Complete the following steps to correct the issue.

1. Remove the USB-8486 from the computer and close NI-FBUS Communications Manager.
2. Connect the USB-8486 to another USB port in the system.
3. Start the NI-FBUS Communication Manager.

If the H1 Fieldbus LED still flashes red, contact National Instruments through the information provided in Appendix C, *Technical Support and Professional Services*.

The H1 Fieldbus LED is solid red.

The USB-8486 H1 Fieldbus port encountered a fatal network error. Complete the following steps to correct the issue.

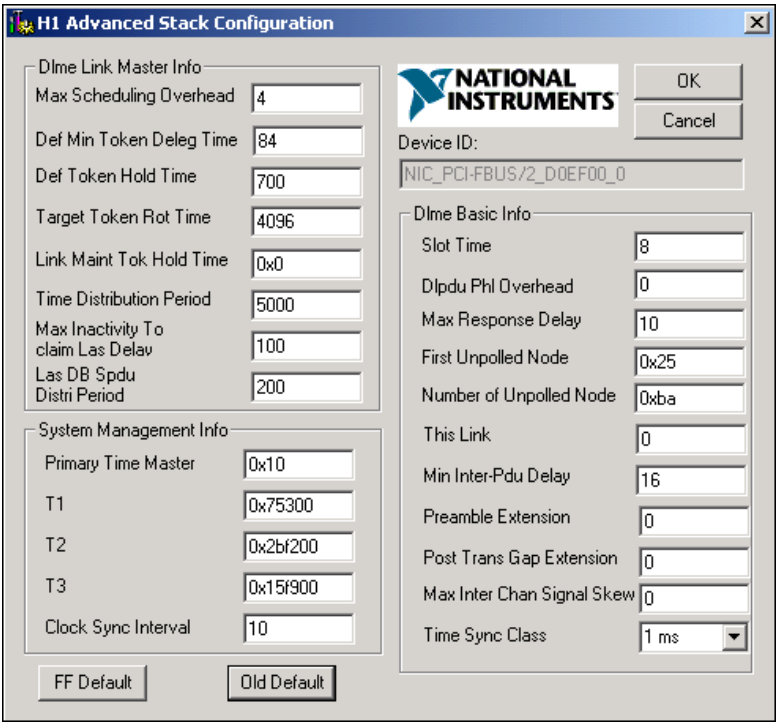
1. Remove the USB-8486 from the computer and close NI-FBUS Communications Manager
2. Re-connect the USB-8486 to the USB port of the system again.
3. Restart the NI-FBUS Communication Manager and check the H1 Fieldbus LED state.

Configuring Advanced Parameters

Caution Do *not* modify the **Advanced** parameters without good reason. If you must modify parameters for certain devices, the device manufacturer will recommend settings. Modifying these parameters can have an adverse affect on data throughput rates. If settings are incorrectly modified, some devices might disappear off the bus.

In the NI-FBUS Interface Configuration Utility, click the **Advanced** button on the dialog box for the port you want to configure. The **Advanced Stack Configuration** dialog box is shown in Figure B-1.

Figure B-1. Advanced Stack Configuration Dialog Box



The parameters involved in setting addresses are T1 and T3. These parameters represent delay time values that your board uses to compensate for the delays inherent in the device and in the set address protocol itself. T1 describes the expected response delay of the device at a given address. T3 describes the expected time for the device to respond at its new address.

Uninstalling the Software

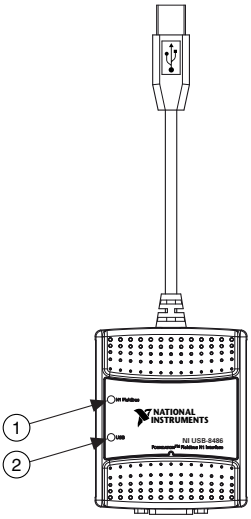
If you are only using the Communications Manager, uninstall the NI-FBUS Communications Manager. If you are using the NI-FBUS Configurator, uninstall the NI-FBUS Configurator.

The uninstall utility does not remove the NI-FBUS directory itself or any files in the `\Data\Nvm` directory. To completely uninstall the software, manually remove the files in the `\Data\Nvm` directory and the NI-FBUS directory structure.

USB-8486 LED Indicators

The USB-8486 has two LED indicators on the front panel, as shown in Figure B-2.

Figure B-2. LEDs on the USB-8486



1 H1 Fieldbus Port Status 2 USB Status

The USB LED is located on the front of the USB-8486, as shown in Figure B-2. It indicates whether the USB-8486 is powered, configured, and operating properly. Table B-2 shows how to interpret the USB LED states.

Table B-2. Interpretation of USB-8486 USB STATUS LED

LED State	Meaning
Off	There is no power on the USB port , the USB-8486 is disabled, or an error has occurred.
Solid green	The USB-8486 is working in USB 2.0 full speed mode.
Solid amber	The USB-8486 is working in USB 2.0 high speed mode.

The H1 Fieldbus port on the USB-8486 has an LED to indicate the functional states of the port. Table B-3 describes each state.

Table B-3. Interpretation of USB-8486 H1 Fieldbus Status LED

LED State	Meaning
Off	The USB-8486 has not been initialized.
Solid green	The Fieldbus port is disconnected from the network or receiving nothing.
Slow flashing green	The Fieldbus port is only receiving/transmitting network maintenance packets.
Fast flashing green	The Fieldbus port is receiving/ transmitting payload traffic packets.
Flashing red	The USB-8486 encountered an error during the P.O.S.T.
Solid red	The Fieldbus port encountered a fatal network error.

For more information about error handling, refer to the *USB-8486 Troubleshooting* section of this appendix.

NI-FBUS Software

This section contains information about how to identify and solve problems with the NI-FBUS Communications Manager software.

Startup Problems

If the NIFB process is unable to find the information it needs to start up, error messages will appear. You may ignore these messages and continue; however, this will result in your application not being able to communicate with the interface devices for which the error messages appeared. These messages tell you the information that the NIFB process is looking for but cannot find.

If NI-FBUS is unable to connect to and initialize an interface device, and you decide to continue NI-FBUS startup, NI-FBUS will not try to reconnect to that interface again. This is true of all interface types supported by this software.

If a USB-8486, PCMCIA-FBUS, or PCI-FBUS interface is configured as a basic device, a link master device must be present on this link before NI-FBUS will start up.

Call to Open Session Fails

If the call fails, ensure that your NI-FBUS Communications Manager process is running and that it has not displayed any error message boxes during startup. You can check this by maximizing and looking at the `nifb.exe` console window. If the title bar does not end in "(Running)," NI-FBUS did not start up correctly.

Set Address Problems

If you are having trouble setting the address of your device, you may need to change some of the System Management Info parameters in the **Advanced** settings of your interface port in the NI-FBUS Interface Configuration utility. The parameters involved in setting addresses are T1 and T3. These parameters represent delay time values that your interface card uses to compensate for the delays inherent in the device and in the set address protocol itself.

T1 is a parameter that describes the expected response delay of the device at a given address. Normally, you will not need to increase this parameter; however, if it appears that your interface card is not seeing the device responses related to setting addresses, you can increase this value. The correct value for this parameter can be dependent on the number of devices on the link. For example, if you are using a bus monitor, you might see a `WHO_HAS_PD_TAG` request going to the device to start the Set Address sequence, and an `IDENTIFY` response coming back, but with the host never continuing on to the next step of the protocol (the `SET_ADDRESS` packet). This probably means that your T1 value is too small and should be increased.

T3 is a parameter that describes the expected time for the device to respond at its new address. This parameter is highly dependent on the number of devices on the link, and the number of addresses being polled. Refer to the *Setting Number of Polled Addresses* section for instructions on how to set the number of polled addresses. If you are using a bus monitor, you may be able to see the host identify a device (with the `IDENTIFY` packet) at the new address, before the devices has sent its probe response (PR) packet to the host. This is an error that is indicative of a T3 value that is too small; if this occurs, increase your T3 value until the `IDENTIFY` to the new address occurs after the PR.

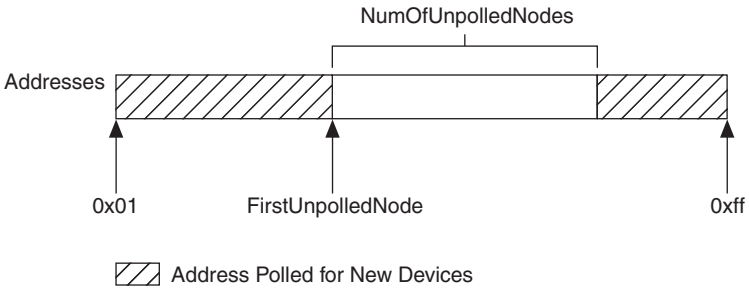
All of the System Management Info timers are in units of 1/32 of a millisecond; for instance, T3 = 32000 units means that T3 = 1 second.

Setting Number of Polled Addresses

The Fieldbus specification describes how a Link Active Scheduler device (LAS device) probes a list of addresses to allow devices to come online during normal operation. The LAS sends a Probe Node (PN on the bus monitor) packet to each address in its list of addresses during operation, and the length of time between Probe Nodes depends on the number of devices on the link and the setting of the Link Maintenance Token Hold Time parameter.

The Fieldbus specification describes how to tell the LAS to skip probing certain addresses in the range to speed up how long it takes to detect new devices on the bus (or devices that are having their addresses changed). The two parameters involved in maintaining the list are called

`FirstUnpolledNode` and `NumOfUnpolledNodes`, and they can be found in the NI-FBUS Interface Configuration utility advanced settings for a port, in the DLME Master Info section. The following diagram shows how the LAS determines the list.



In other words, `FirstUnpolledNode` tells the LAS the beginning of a region of addresses to not probe, and `NumOfUnpolledNodes` tells the LAS the length of that region. So if `FirstUnpolledNode` were 0x25, and `NumOfUnpolledNodes` were 0xba, then no addresses from 0x25 to 0xdf would be probed. That means that if a device with an address of 0x25 were placed on this bus, the LAS would not probe it, and it would never be able to send or receive packets on the bus.

The reason to have a `NumOfUnpolledNodes` whose value is nonzero is as follows. The LAS probes every address in the list, then starts over again at the beginning. Because a device cannot come on the bus until its address is probed, if the LAS is probing all 255 - 16 + 1 = 240 possible addresses and each probe node request goes out every *T* milliseconds, it might take 240*T* milliseconds for a device to get on the bus. If, however, the LAS probed only the first 16 addresses and the last 16 addresses, it might take 32*T* milliseconds for the device to get on the bus. This results in the new device being recognized almost eight times faster.

These parameters also affect the Set Address protocol, because recognizing a device at a new address is really the same as recognizing a completely new device, as the new address must be probed for the device to come online. In this way, the `NumOfUnpolledNodes` parameter can affect the value of the Set Address protocol parameter `T3`, which is described in the *Set Address Problems* section. For example, increasing the `NumOfUnpolledNodes` parameter might fix a SetAddress T3 problem because it takes the device less time to be recognized at the new address.

Using Fieldbus with OPC

Starting with version 3.1, NI-FBUS has a separate OPC Data Access Server, which is compliant with the *OPC Data Access 2.0 Specification*. This OPC server supports VIEW-oriented I/O operations, and has better performance.

An OPC client utility is provided with the NI-FBUS software to let you browse Fieldbus OPC tags. Follow the instructions listed in the *Visual Basic* section of Chapter 4, *Developing The Application*, to make the OPC server operational.

OPC Data Type Mapping Rule

The **SIMPLE** type and **ARRAY** type variables are regarded as leaf nodes in the OPC address space. The **RECORD** type variables are regarded as branch nodes, you need to access each of its member variable through this branch node.

Table B-4 shows the data type-mapping rule.

Table B-4. OPC Data Type Mapping Rule

Meta Type	FMS Standard Data Types	OPC Data Type
Simple	Boolean	VT_BOOL
	Integer8	VT_I1
	Integer16	VT_I2
	Integer32	VT_I4
	Unsigned8	VT_UI1
	Unsigned16	VT_UI2
	Unsigned32	VT_UI4
	Floating Point	VT_R4
	Visible String	VT_BSTR
	Octet String	VT_ARRAY | VT_UI1
	Date	VT_DATE
	Time of Day	VT_DATE
	Time Difference	VT_DATE
	Bit String	VT_ARRAY | VT_UI1
	Time Value	VT_DATE

Table B-4. OPC Data Type Mapping Rule (Continued)

Meta Type	FMS Standard Data Types	OPC Data Type
Array	Boolean	VT_ARRAY | VT_BOOL
	Integer8	VT_ARRAY | VT_I1
	Integer16	VT_ARRAY | VT_I2
	Integer32	VT_ARRAY | VT_I4
	Unsigned8	VT_ARRAY | VT_UI1
	Unsigned16	VT_ARRAY | VT_UI2
	Unsigned32	VT_ARRAY | VT_UI4
	Floating Point	VT_ARRAY | VT_R4
	Visible String	VT_ARRAY | VT_BSTR
	Octet String	—
	Date	VT_ARRAY | VT_DATE
	Time of Day	VT_ARRAY | VT_DATE
	Time Difference	VT_ARRAY | VT_DATE
	Bit String	—
	Time Value	VT_ARRAY | VT_DATE

Lookout

1. Create one or more `OPCClient` objects in your Lookout process.
2. Select the NIFB_OPCDA.3 server from the drop-down list of OPC servers.
3. Set the Activate member of the OPC client object to FALSE using one of the following methods:
   - Edit the connections for the `OPCClient` object and set the Activate member to FALSE.
   - Create a switch object on the Control Panel with the position source set to Remote. Then, set the Remote source to the Activate member of the `OPCClient` object. Leave edit mode, then set the switch to the off (FALSE) position.

4. Add all the items you are interested in to the `OPCClient` object(s).
5. Set the Activate member of the OPC client object to TRUE using one of the following methods:
   - Edit the connections for the `OPCClient` object and set the Activate member to TRUE.
   - Set the switch to the on (TRUE) position.

A similar deactivation/activation procedure will have to be followed while opening a previously saved `.lkp` process file. The Lookout process will always go live immediately when it is loaded. The OPC client object Activate member is always set to TRUE at startup, even though the switch position may indicate off/FALSE.

Server Explorer

1. Launch the Server Explorer.
2. Create an inactive OPC client group.
   a. Right-click **NIFB_OPCDA.3** and select **Add/Edit Groups**.
   b. Create a group with the appropriate parameters. Ensure there is no checkmark in the **Active** box.
3. Add all items.
4. Select **File»OPC»Save** to save the file.
5. Activate the group by right-clicking the group and selecting **Activate Group**.
6. When you open the saved file and want to go live, right-click **NIFB_OPCDA.3** and select **Connect to Server**. After Server Explorer has connected to the server, activate the group as described in step 5.

LabVIEW DSC

Stop (but do not quit) the LabVIEW DSC engine before you add any items to your current configuration. Allow the engine two to five minutes to shut down, especially if your tag configuration file has a large number of items. When you are done adding items, restart the engine.

Problems Using Fieldbus with Lookout

Fieldbus Objects Do Not Appear in Lookout

If you want to use the native Fieldbus objects in Lookout, you have to delete the `lookout.dat` file in the Lookout directory. This file is an index file that tells Lookout what objects it has available. Fieldbus objects are not available by default. Lookout will regenerate the `lookout.dat` file the next time it is started. When it regenerates the file, it will see that Fieldbus software has been installed and will make the Fieldbus objects available.

Fieldbus Alarms in Lookout

In Lookout, there is a separate alarms window for Fieldbus alarms. Under the **Options** menu, select **Fieldbus** to show this window. The window also can be shown using traditional Lookout datamember ShowAlarms. Refer to the entry for National Instruments Fieldbus in the *Lookout Object Reference Manual* (also available from the Help menu within Lookout).

If you want alarms to appear in the main alarm window (rather than the Fieldbus alarms window), you need to create Lookout alarm objects.

C

Technical Support and Professional Services

Log in to your National Instruments ni.com User Profile to get personalized access to your services. Visit the following sections of ni.com for technical support and professional services:

- **Support**—Technical support at ni.com/support includes the following resources:
  - **Self-Help Technical Resources**—For answers and solutions, visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support, as well as exclusive access to self-paced online training modules at ni.com/self-paced-training. All customers automatically receive a one-year membership in the Standard Service Program (SSP) with the purchase of most software products and bundles including NI Developer Suite. NI also offers flexible extended contract options that guarantee your SSP benefits are available without interruption for as long as you need them. Visit ni.com/ssp for more information.

    For information about other technical support options in your area, visit ni.com/services, or contact your local office at ni.com/contact.
- **Training and Certification**—Visit ni.com/training for training and certification program information. You can also 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, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.
- **Declaration of Conformity (DoC)**—A DoC is our claim of compliance with the Council of the European Communities using the manufacturer's declaration of conformity. This system affords the user protection for electromagnetic compatibility (EMC) and product safety. You can obtain the DoC for your product by visiting ni.com/certification.
- **Calibration Certificate**—If your product supports calibration, you can obtain the calibration certificate for your product at ni.com/calibration.

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

Symbol	Prefix	Value
p	pico	10^{-12}
n	nano	10^{-9}
μ	micro	10^{-6}
m	milli	10^{-3}
k	kilo	10^{3}
M	mega	10^{6}
G	giga	10^{9}
T	tera	10^{12}

Numbers

4 to 20 mA system Traditional control system in which a computer or control unit provides control for a network of devices controlled by 4 to 20 mA signals.

A

Address Character code that identifies a specific location (or series of locations) in memory.

Administrative Function An NI-FBUS function that deals with administrative tasks, such as returning descriptors and closing descriptors.

Alarm A notification the NI-FBUS Communications Manager software sends when it detects that a block leaves or returns to a particular state.

Alarm condition A notification that a Fieldbus device sends to another Fieldbus device or interface when it leaves or returns to a particular state.

Alert An alarm or event.

Alert function A function that receives or acknowledges an alert.

Analog	A description of a continuously variable signal or a circuit or device designed to handle such signals.
API	*See* Application Programmer Interface.
Application	Function blocks.
Application Programmer Interface	A message format that an application uses to communicate with another entity that provides services to it.
Array	Ordered, indexed list of data elements of the same type.
Attribute	Properties of parameters.

B

Basic device	A device that can communicate on the Fieldbus, but cannot become the LAS.
Bitstring	A data type in the object description.
Block	A logical software unit that makes up one named copy of a block and the associated parameters its block type specifies. The values of the parameters persist from one invocation of the block to the next. It can be a resource block, transducer block, or function block residing within a virtual field device.
Block tag	A character string name that uniquely identifies a block on a Fieldbus network.
Buffer	Temporary storage for acquired or generated data.
Bus	The group of conductors that interconnect individual circuitry in a computer. Typically, a bus is the expansion vehicle to which I/O or other devices are connected. Examples of PC buses are the ISA and PCI buses.

C

Cable	A number of wires and shield in a single sheath.
Channel	A pin or wire lead to which you apply or from which you read the analog or digital signal.

Client	A device that sends a request for communication on the bus.
Communication stack	Performs the services required to interface the user application to the physical layer.
Connection Management	The service the NI-FBUS Communications Manager provides by handling Virtual Communication Relationships.
Control loop	A set of connections between blocks used to perform a control algorithm.
Controller	An intelligent device (usually involving a CPU) that is capable of controlling other devices.
ControlNet	A 5 Mbit/s communications protocol based on Producer/Consumer technology.
Core Function	The basic functions that the NI-FBUS Communications Manager software performs, such as reading and writing block parameters.

D

Data Link Layer	The second-lowest layer in the ISO seven-layer model (layer two). The Data Link Layer splits data into frames to send on the physical layer, receives acknowledgment frames, and re-transmits frames if they are not received correctly. It also performs error checking to maintain a sound virtual channel to the next layer.
DD	*See* Device Description.
Descriptor	A number returned to the application by the NI-FBUS Communications Manager, used to specify a target for future NI-FBUS calls.
Device	A sensor, actuator, or control equipment attached to the Fieldbus.
Device Description	A machine-readable description of all the blocks and block parameters of a device.
Device Description Language	A formal programming language that defines the parameters of the blocks. It also defines attributes of parameters and blocks like help strings in different languages, ranges of values for parameters, and so on.

Device Description Service	A set of functions that applications use to access Device Descriptions.
Device tag	A name you assign to a Fieldbus device.
Directory	A structure for organizing files into convenient groups. A directory is like an address showing where files are located. A directory can contain files or subdirectories of files.
DLL	*See* Dynamic Link Library.
Driver	Device driver software installed within the operating system.
Dynamic Link Library	A library of functions and subroutines that links to an application at run time.

E

EMI	Electromagnetic interference.
Event	An occurrence on a device that causes a Fieldbus entity to send the Fieldbus event message.

F

FB	Function Block.
Field device	A Fieldbus device connected directly to a Fieldbus.
Fieldbus	An all-digital, two-way communication system that connects control systems to instrumentation. A process control local area network defined by ISA standard S50.02.
Fieldbus cable	Shielded, twisted pair cable made specifically for Fieldbus that has characteristics important for good signal transmission and are within the requirements of the Fieldbus standard.
Fieldbus Foundation	An organization that developed a Fieldbus network specifically based upon the work and principles of the ISA/IEC standards committees.

Fieldbus Messaging Specification	The layer of the communication stack that defines a model for applications to interact over the Fieldbus. The services FMS provides allow you to read and write information about the OD, read and write the data variables described in the OD, and perform other activities such as uploading/downloading data and invoking programs inside a device.
Fieldbus Network Address	Location of a board or device on the Fieldbus; the Fieldbus node address.
FMS	*See* Fieldbus Messaging Specification.
FOUNDATION Fieldbus Specification	The communications network specification that the Fieldbus Foundation created.
Function block	A named block consisting of one or more input, output, and contained parameters. The block performs some control function as its algorithm. Function blocks are the core components you control a system with. The Fieldbus Foundation defines standard sets of function blocks. There are ten function blocks for the most basic control and I/O functions. Manufacturers can define their own function blocks.
Function Block Application	The block diagram that represents your control strategy.

H

H1	The 31.25 kbit/second type of Fieldbus.
Hard code	To permanently establish something that should be variable in a program.
Header file	A C-language source file containing important definitions and function prototypes.
HMI	Human-Machine Interface. A graphical user interface for the process with supervisory control and data acquisition capability.
Host device	A computer or controller on a Fieldbus network.
Hz	Hertz.

I

I/O	Input/output.
IEC	International Electrotechnical Commission. A technical standards committee which is at the same level as the ISO.
in.	Inches.
Index	An integer that the Fieldbus specification assigns to a Fieldbus object or a device that you can use to refer to the object. A value in the object dictionary used to refer to a single object.
ISA	Industry Standard Architecture.

K

Kbits	Kilobits.
Kernel	The set of programs in an operating system that implements basic system functions.
Kernel mode	The mode in which device drivers run on Windows.

L

LabVIEW DSC	The LabVIEW Datalogging and Supervisory Control (DSC) Module builds on the power of LabVIEW for high channel count and distributed applications. It adds easy networking, channel and I/O management, alarm and event management, historical datalogging, real-time trending, and OPC integration to the LabVIEW environment.
LAS	*See* Link Active Schedule.
Link	A FOUNDATION Fieldbus network is made up of devices connected by a serial bus. This serial bus is called a link (also known as a segment).
Link Active Schedule	A schedule of times in the macrocycle when devices must publish their output values on the Fieldbus.

Link Active Scheduler	The Fieldbus device that is currently controlling access to the Fieldbus. A device that is responsible for keeping a link operational. The LAS executes the link schedule, circulates tokens, distributes time, and probes for new devices.
Link master device	A device that is capable of becoming the LAS.
Linkage	A connection between function blocks.
Linkage object	An object resident in a device that defines connections between function block input and output across the network. Linkage objects also specify trending connections.
Live list	The list of all devices that are properly responding to the Pass Token.
LM	Link Master.
Lookout	National Instruments Lookout is a full-featured object-based automation software system that delivers unparalleled power and ease of use in demanding industrial measurement and automation applications.

M

Macrocycle	The least common multiple of all the loop times on a given link, or one iteration of a the process control loop.
Manufacturer's identification	An identifier used to correlate the device type and revision with its device description and device description revision.
Menu	An area accessible from the command bar that displays a subset of the possible command choices. In the NI-FBUS Configurator, refers to menus defined by the manufacturer for a given block.
Method	Methods describe operating procedures to guide a user through a sequence of actions.
Mode	Type of communication.

N

Network address	The Fieldbus network address of a device.
Network Management	A layer of the FOUNDATION Fieldbus communication stack that contains objects that other layers of the communication stack use, such as Data Link, FAS, and FMS. You can read and write SM and NM objects over the Fieldbus using FMS Read and FMS Write services.
NI-FBUS API	The function calls provided by NI-FBUS Communication Manager.
NI-FBUS Communications Manager	Software shipped with National Instruments Fieldbus interfaces that lets you read and write values. It does not include configuration capabilities.
NI-FBUS Configurator	National Instruments Fieldbus configuration software. With it, you can set device addresses, clear devices, change modes, and read and write to the devices.
NI-FBUS Fieldbus Configuration System	*See* NI-FBUS Configurator.
NI-FBUS process	Process that must be running in the background for you to use the NI-FBUS interface boards (USB-8486, PCMCIA-FBUS, or PCI-FBUS) to communicate between the application and Fieldbus.
`Nifb.exe`	The NIFB process that must be running in the background for you to use your USB-8486, PCMCIA-FBUS, or PCI-FBUS interface to communicate between the board and the Fieldbus.
Node	Junction or branch point in a circuit.

O

Object	An element of an object dictionary.
Object Dictionary	A structure in a device that describes data that can be communicated on the Fieldbus. The object dictionary is a lookup table that gives information such as data type and units about a value that can be read from or written to a device.

Octet	A single 8-bit value.
OD	*See* Object Dictionary.
OPC	OLE for Process Control.
Output parameter	A block parameter that sends data to another block.

P

Parameter	One of a set of network-visible values that makes up a function block.
PC	Personal Computer.
PCMCIA	Personal Computer Memory Card International Association.
PD	Proportional Derivative.
Physical device	A single device residing at a unique address on the Fieldbus.
PID	Proportional/Integral/Derivative. A common control function block algorithm that uses proportions, integrals, and derivatives in calculation.
PN	Probe Node.
Poll	To repeatedly inspect a variable or function block to acquire data.
Port	A communications connection on a computer or remote controller.
PR	Probe Response.
Program	A set of instructions the computer can follow, usually in a binary file format, such as a `.exe` file.
Publisher	A device that has at least one function block with its output value connected to the input of another device.

R

Repeater	Boost the signals to and from the further link.
Resource block	A special block containing parameters that describe the operation of the device and general characteristics of a device, such as manufacturer and device name. Only one resource block per device is allowed.
Roundcard	A hardware interface for developing FOUNDATION Fieldbus-compliant devices.

S

s	Seconds.
Sample type	Specifies how trends are sampled on a device, whether by averaging data or by instantaneous sampling.
Segment	*See* Link.
Sensor	A device that responds to a physical stimulus (heat, light, sound, pressure, motion, flow, and so on), and produces a corresponding electrical signal.
Server	Device that receives a message request.
Service	Services allow user applications to send messages to each other across the Fieldbus using a standard set of message formats.
Session	A communication path between an application and the NI-FBUS Communications Manager.
Shield	Metal grounded cover used to protect a wire, component or piece of equipment from stray magnetic and/or electric fields.
Signal	An extension of the IEEE 488.2 standard that defines a standard programming command set and syntax for device-specific operations.
Spur	A secondary route having a junction to the primary route in a network.

Stack	A set of hardware registers or a reserved amount of memory used for calculations or to keep track of internal operations.
Static library	A library of functions/subroutines that you must link to your application as one of the final steps of compilation, as opposed to a Dynamic Link Library, which links to your application at run time.
Stub	*See* Spur.
Subscriber	A device that has at least one function block with its input value connected to the output of another device.
Surge	Large, unwanted voltage or current on wires. Generally caused by lightning or nearby heavy electrical power use.
Surge suppressor	A device used to discharge surges to ground.
Symbol file	A Fieldbus Foundation or device manufacturer-supplied file that contains the ASCII names for all the objects in a device.
System Management	A layer of the FOUNDATION Fieldbus communication stack that assigns addresses and physical device tags, maintains the function block schedule for the function blocks in that device, and distributes application time. You also can locate a device or a function block tag through SM.

T

Tag	A name you can define for a block, virtual field device, or device.
Thread	An operating system object that consists of a flow of control within a process. In some operating systems, a single process can have multiple threads, each of which can access the same data space within the process. However, each thread has its own stack and all threads can execute concurrently with one another (either on multiple processors, or by time-sharing a single processor).

Transducer block	A block that is an interface to the physical, sensing hardware in the device. It also performs the digitizing, filtering, and scaling conversions needed to present input data to function blocks, and converts output data from function blocks. Transducer blocks decouple the function blocks from the hardware details of a given device, allowing generic indication of function block input and output. Manufacturers can define their own transducer blocks.
Trend	A Fieldbus object that allows a device to sample a process variable periodically, then transmit a history of the values on the network.
Trend function	An NI-FBUS call related to trends.

U

USB	Universal Serial Bus.
USB-8486	NI USB-8486 FOUNDATION Fieldbus interface.

V

VCR	*See* Virtual Communication Relationship.
VFD	*See* Virtual Field Device.
View objects	Predefined groupings of parameter sets that HMI applications use.
Virtual Communication Relationship	Preconfigured or negotiated connections between virtual field devices on a network.

Virtual Field Device — The virtual field device is a model for remotely viewing data described in the object dictionary. The services provided by the Fieldbus Messaging Specification allow you to read and write information about the object dictionary, read and write the data variables described in the object dictionary, and perform other activities such as uploading/downloading data and invoking programs inside a device. A model for remotely viewing data described in the object dictionary.

W

Waveform — Multiple voltage readings taken at a specific sampling rate.

Index

E

F

H

I

K

L

M

N

O

P

R

S

T

U

W