DeviceNet™
CONFORMANCE TESTED

NI-DNET™ User Manual

NATIONAL INSTRUMENTS™

April 2001 Edition
Part Number 370375A-01

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 794 0100

Worldwide Offices

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 284 5011,
Canada (Calgary) 403 274 9391, Canada (Ottawa) 613 233 5949, Canada (Québec) 514 694 8521,
Canada (Toronto) 905 785 0085, China (Shanghai) 021 6555 7838, China (ShenZhen) 0755 3904939,
Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24, Germany 089 741 31 30,
Greece 30 1 42 96 427, Hong Kong 2645 3186, India 91805275406, Israel 03 6120092, Italy 02 413091,
Japan 03 5472 2970, Korea 02 596 7456, Malaysia 603 9596711, Mexico 5 280 7625, Netherlands 0348 433466,
New Zealand 09 914 0488, Norway 32 27 73 00, Poland 0 22 528 94 06, Portugal 351 1 726 9011,
Singapore 2265886, Spain 91 640 0085, Sweden 08 587 895 00, Switzerland 056 200 51 51,
Taiwan 02 2528 7227, United Kingdom 01635 523545

For further support information, see the *Technical Support Resources* appendix. To comment on the documentation, send e-mail to techpubs@ni.com.

Important Information

Warranty

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

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

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

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

Copyright

Trademarks

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.

Contents

About This Manual

Chapter 1
NI-DNET Software Overview

Chapter 2
Developing Your Application

Chapter 3
NI-DNET Programming Techniques

Chapter 4
Application Examples

Appendix A
DeviceNet Programming Overview

Appendix B
Uninstalling the NI-DNET Software

Appendix C
Technical Support Resources

Glossary

Index

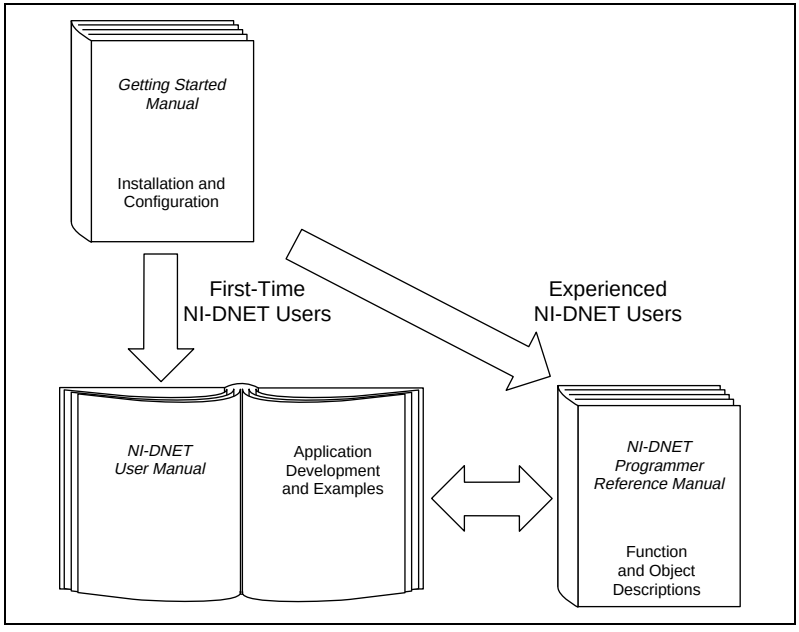
Figures

Tables

About This Manual

This manual describes the basics of DeviceNet and explains how to develop an application program. It also provides detailed examples you can use as models for your own applications. The NI-DNET software is meant to be used with either Windows 98/95 or Windows NT version 3.51 or later. This manual assumes that you are already familiar with the Windows system you are using.

How to Use the Manual Set



Use the getting started manual to install and configure your DeviceNet hardware and the NI-DNET software for Windows 98/95 or Windows NT.

Use this *NI-DNET User Manual* to learn the basics of DeviceNet and how to develop an application program. The user manual also contains detailed examples.

Use the *NI-DNET Programmer Reference Manual* for specific information about each NI-DNET function and object, including format, parameters, and possible errors.

Conventions Used in This Manual

The following conventions appear in this manual:

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

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

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

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

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

Related Documentation

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

- ANSI/ISO Standard 11898-1993, *Road Vehicles—Interchange of Digital Information—Controller Area Network (CAN) for High-Speed Communication*
- *DeviceNet Specification, Version 2.0*, Open DeviceNet Vendor Association
- LabVIEW online reference
- ODVA website, `www.odva.org`
- Microsoft Win32 Software Development Kit (SDK) online help

1

NI-DNET Software Overview

The DeviceNet software provided with National Instruments DeviceNet hardware is called NI-DNET. This section provides an overview of the NI-DNET software.

NI-DNET Objects

The NI-DNET software, like the DeviceNet Specification, uses object-oriented concepts to represent components in the DeviceNet system (for more information about object-oriented concepts in the DeviceNet Specification, refer to Appendix A, *DeviceNet Programming Overview*). However, whereas in the DeviceNet Specification objects represent a multitude of components in DeviceNet devices, NI-DNET objects represent components of the Windows NT/98/95 device driver software. The NI-DNET device driver objects do not correspond directly to objects contained in remote devices. To facilitate access to the DeviceNet network, the NI-DNET objects provide a more concise representation of various objects defined in the DeviceNet Specification.

Much like any other object-oriented system, NI-DNET device driver objects use the concepts of class, instance, attribute, and service to describe their features. The NI-DNET device driver software provides three classes of objects: Interface Objects, Explicit Messaging Objects, and I/O Objects. You can open an instance of an NI-DNET object using one of the three open functions (`ncOpenDnetExplMsg`, `ncOpenDnetIntf`, or `ncOpenDnetIO`). The services for an NI-DNET object are accomplished using the NI-DNET functions, which can be called directly from your programming environment (such as Microsoft C/C++ or LabVIEW). The essential attributes of an NI-DNET object are initialized using its open function; you can access other attributes using `ncGetDriverAttr` or `ncSetDriverAttr`. The attributes of NI-DNET device driver objects are called *driver attributes*, to differentiate them from actual attributes in remote DeviceNet devices.

For complete information on each NI-DNET object, including its driver attributes and supported functions (services), refer to your *NI-DNET Programmer Reference Manual*.

Interface Object

The Interface Object represents a DeviceNet interface (physical DeviceNet port on your DeviceNet board). Since this interface acts as a device on the DeviceNet network much like any other device, it is configured with its own MAC ID and baud rate.

Use the Interface Object to do the following:

- Configure NI-DNET settings that apply to the entire interface
- Start and stop communication for all NI-DNET objects associated with the interface

Explicit Messaging Object

The Explicit Messaging Object represents an explicit messaging connection to a remote DeviceNet device (physical device attached to your interface by a DeviceNet cable). Since only one explicit messaging connection is created for a given device, the Explicit Messaging Object is also used for features that apply to the device as a whole.

Use the Explicit Messaging Object to do the following:

- Execute the DeviceNet Get Attribute Single service on the remote device (`ncGetDnetAttribute`)
- Execute the DeviceNet Set Attribute Single service on the remote device (`ncSetDnetAttribute`)
- Send any other explicit message request to the remote device and receive the associated explicit message response (`ncWriteDnetExplMsg`, `ncReadDnetExplMsg`)
- Configure NI-DNET settings that apply to the entire remote device

I/O Object

The I/O Object represents an I/O connection to a remote DeviceNet device (physical device attached to your interface by a DeviceNet cable). The I/O Object usually represents I/O communication as a master with a remote slave device, but it can also be used for I/O communication as a slave.

The I/O Object supports as many master/slave I/O connections as currently allowed by the DeviceNet Specification. This means that you can use polled, strobed, and COS/cyclic I/O connections simultaneously for a given device. As specified by the DeviceNet Specification, only one master/slave I/O connection of a given type can be used for each device (MAC ID). For example, you cannot open two polled I/O connections for the same device.

Use the I/O Object to do the following:

- Read data from the most recent message received on the I/O connection (`ncReadDnetIO`)
- Write data for the next message produced on the I/O connection (`ncWriteDnetIO`)

Example

Figure 1-1 shows an example of how NI-DNET objects can be used to communicate on a DeviceNet network. This example shows three DeviceNet devices. The first device (at MAC ID 1) is the National Instruments DeviceNet interface. The second device (at MAC ID 5) uses NI-DNET to access a polled and a COS I/O connection simultaneously. The third device (at MAC ID 8) uses NI-DNET to access an explicit messaging connection and a strobed I/O connection.

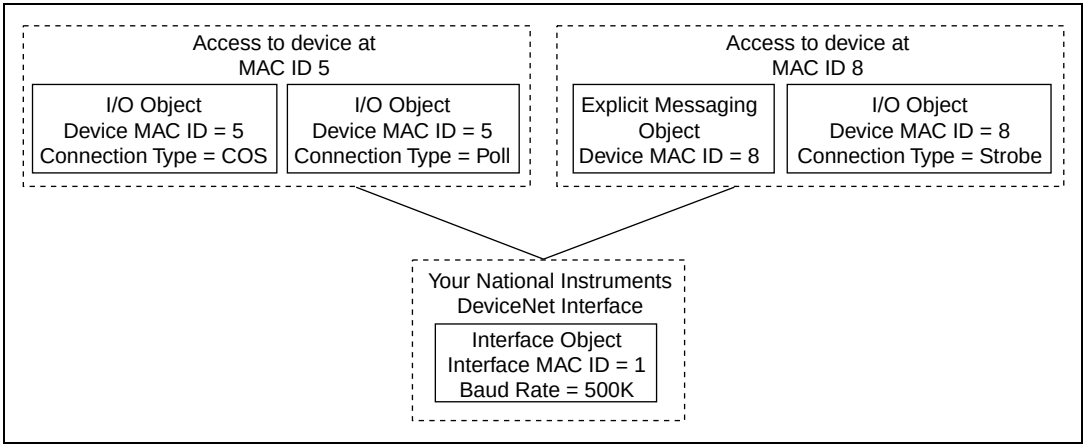


Figure 1-1. NI-DNET Objects for a Network of Three Devices

NI-DNET Software Components

The following section highlights important components included with the NI-DNET software for Windows NT/98/95.

Figure 1-2 illustrates how your NI-DNET software works with your system and your DeviceNet hardware.

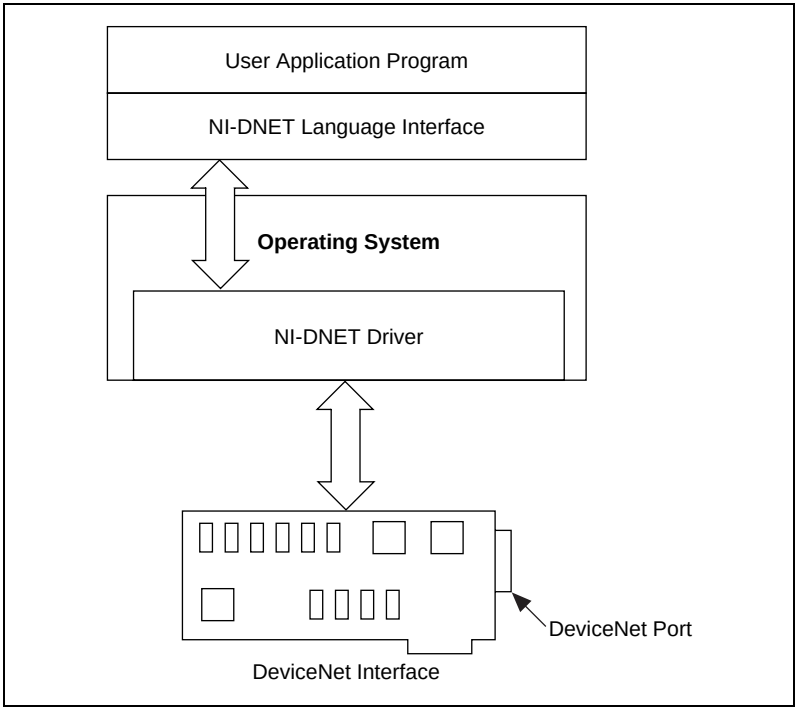


Figure 1-2. Structure of an NI-DNET System

NI-DNET Driver and Utilities

The NI-DNET software contains the following drivers and utilities:

- A documentation file, `readme.txt`, contains important information about the NI-DNET software and a description of any new features. Before you use the software, read this file for the most recent information about the NI-DNET software.
- A 32-bit, multitasking-aware device driver is used to interface with National Instruments DeviceNet hardware. Under Windows 98/95, this is a dynamically loadable, Plug and Play aware virtual device

driver (VxD). Under Windows NT, this is a native Windows NT kernel driver.

- A Win32 dynamic link library acts as the interface between all Windows NT/98/95 DeviceNet applications and the NI-DNET device driver.
- The NI-DNET Hardware Configuration utility is used to configure the DeviceNet hardware and verify proper operation of the DeviceNet hardware and software.
- The `SimpleWho` utility is used to determine basic information about the devices on your DeviceNet network. This information assists with determining the parameters needed for I/O configuration (`ncOpenDnetIO`).

Firmware Image Files

All National Instruments DeviceNet hardware products contain an on-board microprocessor. This microprocessor allows all time-critical aspects of the NI-DNET software to execute separately from your Windows NT/98/95 application. The firmware image which runs on the on-board microprocessor, `nidnet.nfw`, is loaded and executed automatically when your NI-DNET application starts.

Language Interface Files

- A documentation file, `readme.txt`, contains information about the NI-DNET language interface files.
- A 32-bit C language header file, `nidnet.h`, contains NI-DNET function prototypes, host data types, and various predefined constants.
- A 32-bit C language link library, `nidnetms.lib`, is used by Microsoft C/C++ applications to access NI-DNET functions.
- A 32-bit C language link library, `nidnetbo.lib`, is used by Borland C/C++ (5.0 or later) applications to access NI-DNET functions.
- NI-DNET language interface file for Microsoft Visual Basic (5.0 or later), `nidnet.bas`, when installed, provides complete integration in that environment.
- NI-DNET language interface files for LabWindows/CVI, when installed into your LabWindows/CVI directory, provide for complete integration in that environment.

- NI-DNET language interface files for LabVIEW/BridgeVIEW, when installed into your LabVIEW/BridgeVIEW directory, provide for complete integration in that environment.

Application Examples

The NI-DNET software includes example applications. For a detailed description of the example application files, refer to Chapter 4, *Application Examples*.

WinDnet Support Files

The NI-DNET software includes support files for the Allen-Bradley WinDnet architecture. With this software, you can use the National Instruments DeviceNet hardware with various Allen-Bradley DeviceNet software tools, such as the DeviceNet Manager software. The DeviceNet Manager software provides features to assists in the installation and configuration of devices in a DeviceNet system.

Using NI-CAN Software with DeviceNet

CAN (Controller Area Network) is the low-level protocol used for DeviceNet communications. In addition to the NI-DNET functions, your National Instruments DeviceNet interface can also be used for low-level access to CAN messages using the NI-CAN software. NI-CAN is intended primarily for applications that require direct access to CAN messages, such as test applications for automotive (non-DeviceNet) networks. When connecting to a DeviceNet network, the NI-CAN capabilities are useful only for the following applications:

- Low-level monitoring of CAN messages to determine conformance to DeviceNet specifications
- Implementation of sections of the DeviceNet Specification yourself, such as custom configuration tools

If you want to use the NI-CAN software, refer to the installation files and online manuals on the National Instruments FTP site, `ftp://ftp.ni.com/support/ind_comm/CAN`

NI-CAN 1.2 uses a different device driver infrastructure than NI-DNET 1.1. If you are using NI-CAN 1.2, you must uninstall NI-DNET before installing NI-CAN (and vice versa).

2

Developing Your Application

This chapter explains how to develop an application using the NI-DNET functions.

Accessing NI-DNET from your Programming Environment

Applications can access the NI-DNET driver software by using either Microsoft C/C++ Win32, Borland C/C++ 5.0 or later, LabWindows/CVI, LabVIEW, or Visual Basic. If you are using any other development environment, you must use direct entry. Each of these language interface techniques is summarized below.

Using LabVIEW or BridgeVIEW (G)

For applications written in LabVIEW (4.0 or later) or BridgeVIEW, NI-DNET provides a complete G language function library, front panel controls, and examples.

You can place NI-DNET functions into your diagram from the LabVIEW **Functions** palette, and you can place NI-DNET controls into your front panel from the LabVIEW **Controls** palette.

For online help for a specific NI-DNET function or control, select **Show Help** from the LabVIEW **Help** menu, then click on the NI-DNET function or control. You can click on a function within the NI-DNET palette or after placement into your diagram.

NI-DNET Palette Location

The NI-DNET Functions and Controls palettes are in LabVIEW's **Default** view. You do not have to set this palette view, unless you either select a different palette view or you are using LabVIEW 4.0. If you are using a different palette view than **Default**, select **Edit»Select Palette Set» Default** from the menu to view the NI-DNET controls and functions.

LabVIEW 4.0 Users

If you are using version 4.0 of LabVIEW, you should either contact National Instruments for a free upgrade to LabVIEW 4.1 or follow these steps to add the NI-DNET view to your existing palette set.

1. Select **Edit»Edit Controls & Functions Palette**. LabVIEW displays a dialog box with a list of palette sets and an option to add new setup.
2. Select **New Setup** from the **Palette Set** list and enter `NIDNET VIEW` for the setup name.
3. Right-click on the **Controls** box and select **Insert»Submenu** from the list. From the **Insert Menu** dialog box, select **Link to an existing menu file (.mnu)**.
4. Select **dir.mnu** from the `\LabVIEW\vi.lib\addons\nidnet\` directory. An icon box appears in your **Controls** set for NI-DNET controls.
5. Repeat Steps 3 and 4 for the **Functions** box.
6. Click on **Save Changes** on the **Edit Control and Functions Palette** box to make your view show up in the palette set.
7. To view the NI-DNET palette at any time in LabVIEW, select **Edit»Select Palette Set»NIDNET VIEW**.

Using LabWindows/CVI

For applications written in LabWindows/CVI, NI-DNET provides complete LabWindows/CVI function panels as well as a LabWindows/CVI header file (`nidnet.h`) and link library (`nidnet.lib`).

After you create a new project and a new C source file for your NI-DNET application, add the following line to the beginning of your code.

```
#include "nidnet.h"
```

You can now call NI-DNET functions within your LabWindows/CVI source code. You can enter NI-DNET function calls into your code by typing them directly, or by using the NI-DNET function panels. To open the NI-DNET function panels, select **NI-DNET** from the LabWindows/CVI **Library** menu.

The NI-DNET function panels are used much like any other LabWindows/CVI function panel set. They include online help for each function and each parameter.

When you compile your LabWindows/CVI application for NI-DNET, it is automatically linked with `nidnet.lib`, the link library for LabWindows/CVI. When NI-DNET is installed, the installation program checks to see which compatible C compiler you are using with LabWindows/CVI (Microsoft, Borland, Watcom, or Symantec), and copies an appropriate `nidnet.lib` for that compiler. If you re-install LabWindows/CVI to change your compatible compiler, you must also re-install your NI-DNET software.

Using Microsoft Visual Basic

To create an NI-DNET application in Visual Basic, add the `nidnet.bas` file to your project. This allows you to call any NI-DNET function declared in the `nident.bas` file from your code.

Using Microsoft C/C++ or Borland C/C++

For applications written in Microsoft Visual C/C++ (2.0 or later) or Borland C/C++ (5.0 or later), NI-DNET provides a header file (`nidnet.h`) and link library (`nidnetms.lib` or `nidnetbo.lib`). For LabWindows/CVI, support for NI-DNET is built-in (refer to the section *Using LabWindows/CVI*). For other C/C++ programming environments, you must access NI-DNET functions using direct entry (refer to the section *Direct Entry Access*, later in this chapter).

Follow these steps to use the Microsoft C/C++ or Borland C/C++ language interface files for your application:

1. Include the `nidnet.h` header file in your C source code.
   - For C applications (files with the `.c` extension), add the following line to the beginning of your source code.

     ```
     #include "nidnet.h"
     ```

   - For C++ applications (files with the `.cpp` extension), add the following lines to the beginning of your source code.

     ```
     #define _cplusplus
     #include "nidnet.h"
     ```

     The `_cplusplus` define allows `nidnet.h` to properly handle the transition from C++ to the C language NI-DNET functions.
2. You can now enter NI-DNET function calls into your source code by typing them directly.

3. Link your application to the NI-DNET link library.
   - For Microsoft C/C++, link your application with the NI-DNET link library for Microsoft C/C++ (2.0 or later), `nidnetms.lib`.
   - For Borland C/C++, link your application with the NI-DNET link library for Borland C/C++ (5.0 or later), `nidnetbo.lib`. For Borland C/C++ 4.5, you must use direct entry (see *Direct Entry Access*, later in this chapter).

For a complete description of how each NI-DNET function is used in C/C++ or LabVIEW, refer to the *NI-DNET Programmer Reference Manual*. For information on examples shipped with this software, refer to the main `readme.txt` file located in the `c:\nidnet\examples` directory. For language-specific details, refer to the `readme.txt` file located in that language folder.

Direct Entry Access

You can directly access NI-DNET from any programming environment that allows you to request addresses of functions that a dynamic link library (DLL) exports. The functions used to access a DLL in this manner are provided by the Microsoft Win32 functions of Windows NT/98/95. Using these Microsoft Win32 functions to access a DLL is often referred to as direct entry. To use direct entry with NI-DNET, complete the following steps:

1. Load the NI-DNET DLL, `nican.dll`.

   The following C language code fragment illustrates how to call the Win32 `LoadLibrary` function and check for an error.

```
#include <windows.h>
#include "nidnet.h"

HINSTANCE NidnetLib = NULL;

NidnetLib=LoadLibrary("nican.dll");
if (NidnetLib == NULL) {
   return FALSE; /*Error*/
}
```

2. Get the addresses for the NI-DNET DLL functions you will use.

   Your application must use the Win32 `GetProcAddress` function to get the addresses of the NI-DNET functions your application needs. For each NI-DNET function used by your application, you must define a direct entry prototype. For the prototypes for each function exported by `nican.dll`, refer to the *NI-DNET Programmer Reference Manual*.

The following code fragment illustrates how to get the addresses of the ncOpenDnetIO, ncCloseObject, and ncReadDnetIO functions.

```
static NCTYPE_STATUS (_NCFUNC_ *PncOpenDnetIO)
   (NCTYPE_STRING ObjName,
   NCTYPE_OBJH_P ObjHandlePtr);
static NCTYPE_STATUS (_NCFUNC_ *PncCloseObject)
   (NCTYPE_OBJH ObjHandle);
static NCTYPE_STATUS (_NCFUNC_ *PncReadDnetIO)
   (NCTYPE_OBJH ObjHandle, NCTYPE_UINT32 SizeofData,
   NCTYPE_ANY_P Data);

PncOpenDnetIO = (NCTYPE_STATUS (_NCFUNC_ *)
   (NCTYPE_STRING, NCTYPE_OBJH_P))
   GetProcAddress(NidnetLib,
   (LPCSTR)"ncOpenDnetIO");
PncCloseObject = (NCTYPE_STATUS (_NCFUNC_ *)

   (NCTYPE_OBJH))
   GetProcAddress(NidnetLib,
   (LPCSTR)"ncCloseObject");
PncRead = (NCTYPE_STATUS (_NCFUNC_ *)
   (NCTYPE_OBJH, NCTYPE_UINT32, NCTYPE_ANY_P))
   GetProcAddress(NidnetLib,
   (LPCSTR)"ncReadDnetIO");
```

If GetProcAddress fails, it returns a NULL pointer. The following code fragment illustrates how to verify that none of the calls to GetProcAddress failed.

```
if ((PncOpenDnetIO == NULL) ||
       (PncCloseObject  == NULL) ||
       (PncReadDnetIO == NULL)) {
   FreeLibrary(NidnetLib);
   printf("GetProcAddress failed");
}
```

3. Configure your application to de-reference the pointer to call an NI-DNET function, as illustrated by the following code.

```
NCTYPE_STATUS status;
NCTYPE_OBJH MyObjh;

status = (*PncOpenDnetIO) ("DNET0", &MyObjh);
if (status < 0) {
   printf("ncOpenDnetIO failed");
}
```

4. Free `nican.dll`.

   Before exiting your application, you need to free `nican.dll` with the following command.

   ```
   FreeLibrary(NidnetLib);
   ```

For the most recent information on NI-DNET support for other programming environments, refer to the `readme.txt` file. For more information on direct entry, refer to the Microsoft Win32 Software Development Kit (SDK) online help.

Programming Model for NI-DNET Applications

The following steps provide an overview of how to use the NI-DNET functions in your application. The steps are shown in Figure 2-1 in flowchart form. For examples of using NI-DNET for specific applications, refer to Chapter 4, *Application Examples*. The NI-DNET functions are described in detail in the *NI-DNET Programmer Reference Manual*.

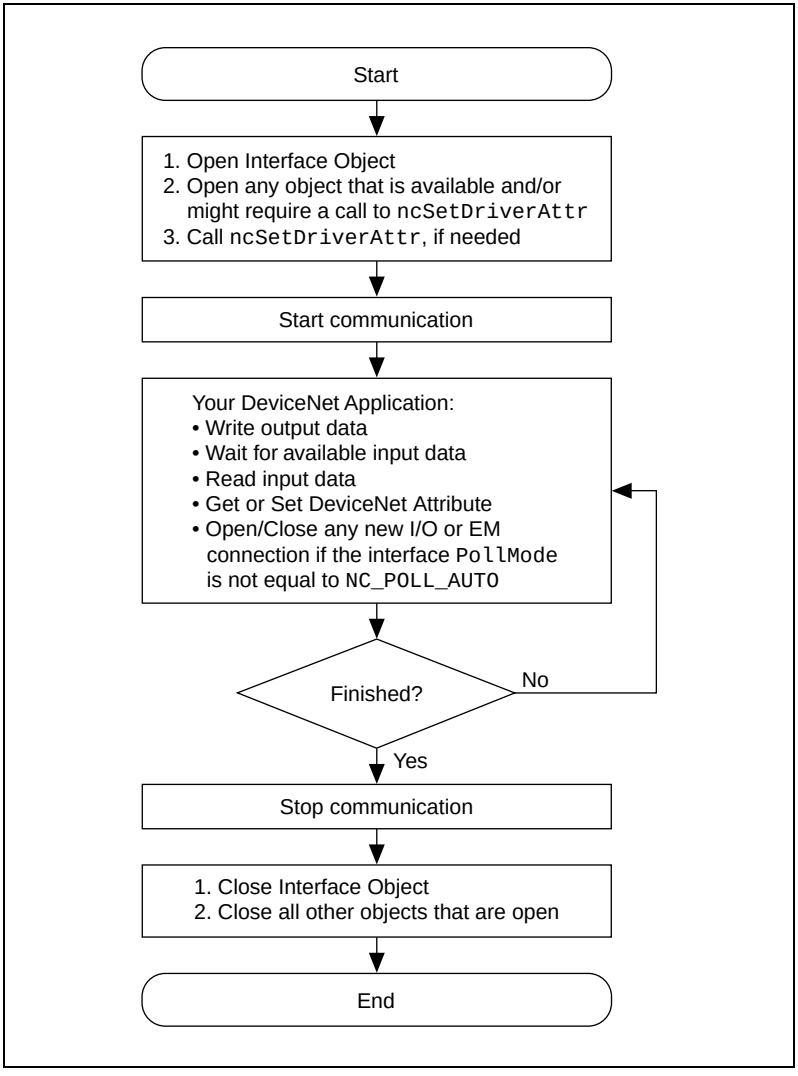


Figure 2-1. General Programming Steps for an NI-DNET Application

Step 1. Open Objects

Before you use an NI-DNET object in your application, you must configure and open it using either `ncOpenDnetIntf`, `ncOpenDnetExplMsg`, or `ncOpenDnetIO`. These open functions return a handle for use in all subsequent NI-DNET calls for that object.

The ncOpenDnetIntf function configures and opens an Interface Object. Your NI-DNET application uses this Interface Object to start and stop communication. The Interface Object must be the first NI-DNET object opened by your application.

The ncOpenDnetExplMsg function configures and opens an Explicit Messaging Object, and the ncOpenDnetIO function configures and opens an I/O Object.

Step 2. Start Communication

Start communication to initialize DeviceNet connections to remote devices. Use the Interface Object to call the ncOperateDnetIntf function with the Opcode parameter set to Start.

The following optional steps can be done before you start communication:

- For an I/O Object, if it is not acceptable to send output data of all zeros, call ncWriteDnetIO to provide valid output values for the initial transmission.
- For an I/O Object, if your application is multitasking, call the ncCreateNotification or ncCreateOccurrence function with the DesiredState parameter set to Read Available. This notifies your application when new input data is received from the remote device.
- For any NI-DNET object, if any of the Driver attributes needs to be changed, call ncSetDriverAttr with the attribute Id and attribute value. The ncSetDriverAttr function cannot be called after the communication has started.

Step 3. Run Your DeviceNet Application

After you open your NI-DNET objects and start communication, you are ready to interact with the DeviceNet network.

Use the following steps with an I/O Object:

1. Call the `ncWriteDnetIO` function to write output data for subsequent transmission on the DeviceNet network.
2. Call the `ncWaitForState` function with the `DesiredState` parameter set to `Read Available`. This function waits for output data to be transmitted and for new input data to be received. If your application is multitasking, you might have other tasks to do in your application while you wait for new input data. If so, use the `ncCreateNotification` or `ncCreateOccurrence` function instead of `ncWaitForState` (see *Step 2. Start Communication*).
3. Call the `ncReadDnetIO` function to read input data received from the DeviceNet network.
4. Loop back to Step 1 as needed.

Use the following steps with an Explicit Messaging Object:

1. Call the `ncWaitForState` function with the `DesiredState` parameter set to `Established`. This ensures that the explicit message connection is established before you send the first explicit message request.
2. To get an attribute from a remote DeviceNet device, call the `ncGetDnetAttribute` function.
3. To set the value of an attribute in a remote DeviceNet device, call the `ncSetDnetAttribute` function.
4. To invoke other explicit message services in a remote DeviceNet device, use the `ncWriteDnetExplMsg` function to write the service request, the `ncWaitForState` function to wait for the service response, and the `ncReadDnetExplMsg` function to read the service response.
5. Loop back to Step 2 as needed.

Addition of Slave Connections after Communication Start

If you need to add I/O and Explicit Messaging connections after the communication on the network has started, you can call `ncOpenDnetExplMsg` and `ncOpenDnetIO` as long as the Interface Object's poll mode had been configured to `NC_POLL_SCAN` (Scanned) or `NC_POLL_INDIV` (Individual). Since the Automatic poll mode (`NC_POLL_AUTO`) calculates the expected packet rate (EPR) based on the estimated network bandwidth, all the I/O connections have to be opened before you start the communication if the Automatic mode is selected. The EPR restrictions due to different values of the `PollMode` parameter still apply to the I/O objects. For details on these requirements, refer to `ncOpenDnetIO` and `ncOpenDnetIntf` function descriptions in the *NI-DNET Programmer Reference Manual*.

Step 4. Stop Communication

Before you exit your application, stop communication to shut down DeviceNet connections to remote devices. Use the Interface Object to call the `ncOperateDnetIntf` function with the `Opcode` parameter set to `Stop`.

Step 5. Close Objects

Before you exit your application, close all NI-DNET objects using the `ncCloseObject` function.

Multiple Applications on the Same Interface

The NI-DNET software allows multiple NI-DNET applications to use the same interface object simultaneously, as long as the interface configuration remains the same. For example, you can run both the `SingleDevice` example and `SimpleWho` on `"DNET0"` as long as the `Interface MacId` and `BaudRate` parameters are the same in both applications. Similarly, you can open up two copies of the `SingleDevice` example and communicate with two different devices as if it were through a single application. These same rules apply to the I/O Object and the Explicit Messaging Object.

As long as all the configuration attributes are the same, any object can be opened multiple times. You can enable only one notification or wait (through `ncWaitForState`, `ncCreateNotification`, or `ncCreateOccurrence` functions) for an object, no matter how many handles you have opened for that particular object. For example, if you are running two copies of the `SingleDevice` example on the same interface

with the same connection types, the notification triggers in only one application at a time.

The synchronization of events and the protection of the object I/O data is the responsibility of the application developer. Similarly, the application performance might change based on the number of objects open and the frequency of API calls made in each application. For example, several calls to `ncGetDnetAttribute` in one application might slow down another application running on the same interface.

To ensure proper clean up of all the objects, each open call to an object should be matched by a close call to the same object, and each call to `ncOperateDnetIntf` with `NC_OP_START` code should be matched by a call to the same function with `NC_OP_STOP` code.

Handling Status in G (LabVIEW/BridgeVIEW)

Checking Status

For applications written in G (LabVIEW/BridgeVIEW), status checking is handled automatically. For all NI-DNET functions, the lower left and right terminals provide status information using LabVIEW Error Clusters. LabVIEW Error Clusters are designed so that status information flows from one function to the next, and function execution stops when an error occurs. For more information, refer to the Error Handling section in the LabVIEW online reference.

Within your LabVIEW block diagram, you wire the `Error in` and `Error out` terminals of NI-DNET functions together in succession. When an error is detected in an NI-DNET function (`status` field true), all NI-DNET functions wired together are skipped except for `ncCloseObject`. The `ncCloseObject` function executes regardless of whether an error occurred, thus ensuring that all NI-DNET objects are closed properly when execution stops due to an error. Depending on how you want to handle errors, you can wire the `Error in` and `Error out` terminals together per-object (group a single open/close pair), per-device (group together Explicit Messaging and I/O Objects for a given device), or per-network (group all functions for a given interface).

The DeviceNet Error Handler function converts an NI-DNET Error Cluster into a descriptive string. If you display this string when an error or warning is detected, you can avoid interpretation of individual fields of the Error Cluster to debug the problem quickly and effectively. The Error in terminal of this function is normally wired from the Error out terminal of an ncCloseObject function.

To display an NI-DNET Error Cluster description without interrupting execution of other code, the Error out and Error String output terminals of the DeviceNet Error Handler are normally wired to front panel indicators. If you want to interrupt execution and display a dialog box that describes the error, set Show Error Dialog to true instead of using front panel indicators.

Figure 2-2 shows the Error Cluster of the ncCloseObject function wired into the DeviceNet Error Handler function. Instead of showing the dialog box when an error occurs, this diagram displays the error description using a front panel indicator.

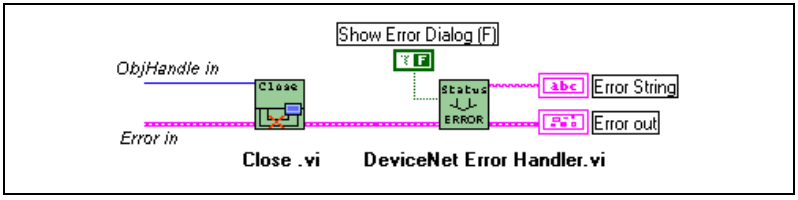


Figure 2-2. NI-DNET Error Cluster Example

Status Format

When you use the DeviceNet Error Handler function in your diagram, a description of the error is displayed either in a dialog box or on your front panel (assuming you wire Error String to an indicator). When you display the error string generated by DeviceNet Error Handler, you do not need to interpret the individual fields of the NI-DNET Error Cluster.

In the NI-DNET implementation of Error Clusters, each field has the following meaning.

Status

This boolean field is set to true when an error occurs and remains false when a warning or success occurs. An error occurs when a function does not perform the expected behavior. A warning occurs when the function performed as expected but a condition exists which might require your attention. Success indicates that the function performed normally.

Code

The 32 bits of the `code` field have the format shown in Figure 2-3.

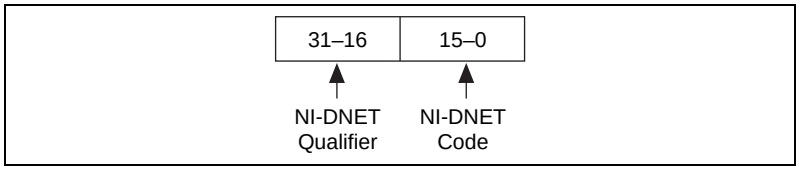


Figure 2-3. Error Cluster Code Field

The lower 16 bits indicate the primary status code used for warnings or errors. For example, if NI-DNET cannot initialize communication with a device, the code `NC_ERR_DEVICE_INIT` is returned. If no warning or error exists, the Error Cluster's `code` field has the value zero.

The upper 16 bits indicate a qualifier for the primary NI-DNET warning or error code. This NI-DNET qualifier is specific to individual values for the NI-DNET code and provides additional information useful for detailed debugging. For example, if the status code is `NC_ERR_DEVICE_INIT`, the qualifier indicates the exact cause of the initialization problem. If no qualifier exists, the NI-DNET qualifier field has the value zero.

Source

When an error or warning occurs, the `source` field (a string) of the Error Cluster provides the complete VI hierarchy for the NI-DNET function in which the error or warning occurred. If no error or warning occurs in your application, `source` remains blank.

The first line in `source` displays the NI-DNET function in which the error or warning occurred. The next line displays the name of the VI that called the NI-DNET function. Subsequent lines display the next highest VI in the call chain, up to the main VI for your application.

Handling Status in C

Checking Status

Each C language NI-DNET function returns a value that indicates the status of the function call. This status value is zero for success, greater than zero for a warning, and less than zero for an error.

After every call to an NI-DNET function, your program should check to see if the return status is nonzero. If so, call the `ncStatusToString` function to obtain an ASCII string which describes the error/warning. You can then use standard C function, such as `printf`, to display this ASCII string.

The following text shows C source code for handling the status returned from the `ncCloseObject` function. If an error or warning is detected, call `ncStatusToString` to obtain an error description.

```
NCTYPE_STATUS      status;
char               string[80];
. . .
status = ncCloseObject(objh);
if (status != NC_SUCCESS) {
   ncStatusToString(status, sizeof(string), string);
   printf("ncCloseObject: %s\n", string);
   . . .
}
. . .
```

When you access the NI-DNET code and qualifier within your application, you should use the constants defined in `nidnet.h`. These constants use the same names as described in the *NI-DNET Programmer Reference Manual*. For example, to check for a timeout after you call `ncWaitForState`, you would write C code similar to the following:

```
if (NC_STATCODE(status) == NC_ERR_TIMEOUT) {
   YourCodeToHandleTimeout();
}
```

Status Format

When you use the `ncStatusToString` function in your C source code, you can always obtain a complete description of the error, and you do not need to interpret the individual fields of the NI-DNET status.

To provide the maximum amount of information, the status returned by NI-DNET functions is encoded as a signed 32-bit integer. The format of this integer is shown in Figure 2-4.

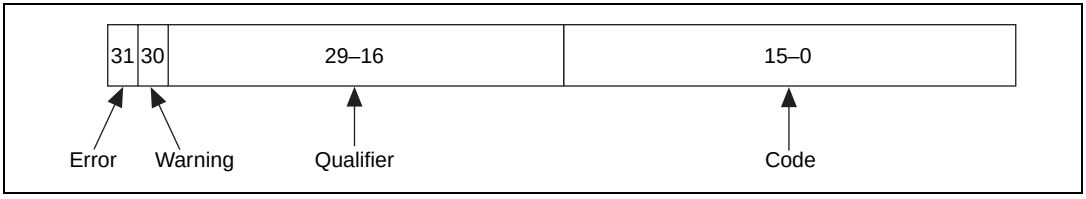


Figure 2-4. Status Format in C

Error/Warning Indicators (Severity)

The error and warning bits ensure that all NI-DNET errors generate a negative status and all NI-DNET warnings generate a positive status. The error bit sets when a function does not perform the expected behavior, resulting in a negative status. The warning bit sets when the function performed as expected but a condition exists that might require your attention. If no error or warning occurs, the entire status sets to zero to indicate success. Table 2-1 summarizes the behavior of NI-DNET status.

Table 2-1. Determining Severity of Status

Status	Result
Negative	Error. Function did not perform expected behavior.
Zero	Success. Function completed successfully.
Positive	Warning. Function performed as expected, but a condition arose that might require your attention.

Code

The code bits indicate the primary status code used for warning or errors. For example, if NI-DNET cannot initialize communication with a device, the code `NC_ERR_DEVICE_INIT` is returned. If no warning or error exists, this field has the value zero.

Qualifier

The qualifier bits hold a qualifier for the warning or error code. It is specific to individual values for the code field and provides additional information useful for detailed debugging. For example, if the status code is `NC_ERR_DEVICE_INIT`, the qualifier indicates the exact cause of the initialization problem. If no qualifier exists, this field has the value zero.

3

NI-DNET Programming Techniques

This chapter describes various techniques to help you program your NI-DNET application. The techniques include configuration of I/O connection timing, using I/O data (assemblies), using explicit messaging, and handling multiple devices. This chapter also includes information on the `SimpleWho` utility and DeviceNet network management tools.

Configuring I/O Connections

This section provides information on how I/O connections relate to one another and how your configuration of I/O connection timing can affect the overall performance of your DeviceNet system. The various types of I/O connections provided by DeviceNet are described in Chapter 1, *NI-DNET Software Overview*.

In a master/slave DeviceNet I/O system, the master determines the timing of all I/O communication. Within your NI-DNET application, the `ncOpenDnetIO` function configures the timing for I/O connections in which your application communicates as master. As you read this section, you might want to refer to the description of the `ncOpenDnetIO` function in the *NI-DNET Programmer Reference Manual*.

Expected Packet Rate

Each DeviceNet I/O connection contains an attribute called the expected packet rate, which specifies the expected rate (in milliseconds) of messages (*packets*) for the I/O connection. For NI-DNET, you use the `ExpPacketRate` parameter of the `ncOpenDnetIO` function to configure the expected packet rate.

After you start communication, the embedded microprocessor on your National Instruments DeviceNet interface transmits messages at the `ExpPacketRate`. This means that after the I/O connection is configured,

your NI-DNET application does not need to be concerned with the timing of messages on the DeviceNet network.

When you select an `ExpPacketRate` for an I/O connection, you must consider all I/O connections in your system. For example, although you might be able to configure an `ExpPacketRate` of 3 ms for a single I/O connection, you cannot configure a 3 ms `ExpPacketRate` for 40 I/O connections because DeviceNet's bandwidth capabilities cannot support 40 messages in a 3 ms time frame.

The following sections describe how to evaluate system considerations so that you can configure valid values for `ExpPacketRate`.

Strobed I/O

For strobed I/O connections, the master broadcasts a single strobe command message to all strobed slaves. Since all strobed I/O connections transfer data at the rate of this single strobe command message, the `ExpPacketRate` of each strobed I/O connection must be set to the same value.

The common `ExpPacketRate` for all strobed I/O connections should provide enough time for the strobe command and each strobed slave's response. You must also allow time for other I/O messages and explicit messages to occur in the `ExpPacketRate` time frame. If you do not know the time needed, let NI-DNET calculate a safe value for you (refer to the section *Automatic EPR Feature*, later in this chapter).

Figure 3-1 shows a timing example for four strobed devices at MAC ID 9, 11, 12, and 13. Notice that since MAC ID 11 is slow to respond, the `ExpPacketRate` is set to 20 ms to provide additional safety margin for other messages.

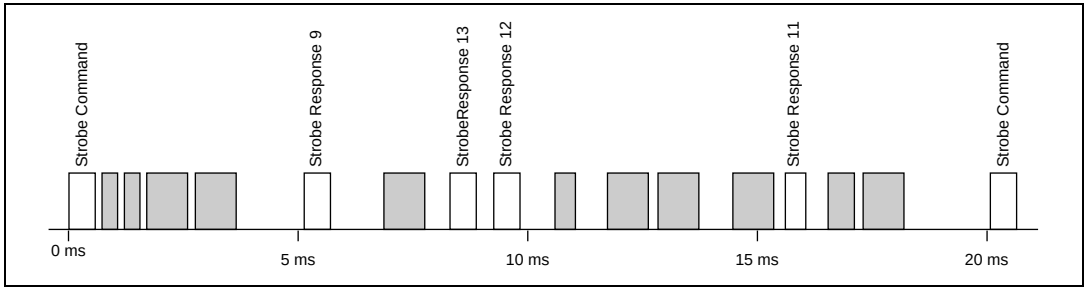


Figure 3-1. Strobed I/O Timing Example

Polled I/O

Polled I/O connections use a separate poll command and response message for each device.

The overall scheme that NI-DNET uses to time polled I/O connections is determined by the `PollMode` parameter of `ncOpenDnetIntf`. This `PollMode` parameter applies to all polled I/O connections (all calls to `ncOpenDnetIO` with `ConnectionType` of `Poll`).

The following sections describe different schemes you can use for polled I/O.

Scanned Polling

You can set the `ExpPacketRate` of each polled I/O connection to the same value used for all strobed I/O. Using a common `ExpPacketRate` for all strobed and polled I/O is referred to as scanned I/O. Scanned I/O is also referred to as scanned polling with respect to polled I/O connections. When you use scanned I/O, NI-DNET transmits all strobe and poll command messages onto the network in quick succession.

Scanned I/O is a simple, efficient way to handle I/O connections that require similar response rates. With scanned I/O, the master knows that all strobe and poll commands go out at the same time. Therefore, the master does not need to manage individual timers, thus optimizing processing overhead. Scanned I/O also provides overall consistency. If a given DeviceNet system uses only scanned I/O, you know that all higher level control algorithms can execute at the single common strobe/poll `ExpPacketRate`.

The common `ExpPacketRate` for all strobed and polled I/O connections should provide enough time for all strobe/poll commands and each slave's response. You must also allow time for other I/O messages and explicit messages to occur in the `ExpPacketRate` time frame.

NI-DNET provides two different methods you can use to configure scanned I/O:

- If you set the `PollMode` parameter of `ncOpenDnetIntf` to `Automatic`, NI-DNET automatically calculates a valid common `ExpPacketRate` value for each strobed and polled I/O connection. When you use this scheme, you do not need to specify a valid `ExpPacketRate` when you open your strobed/polled I/O connections. For more information, refer to the *Automatic EPR Feature* section, later in this chapter.

- If you set the `PollMode` parameter of `ncOpenDnetIntf` to `Scanned`, to configure scanned I/O you must specify the exact same `ExpPacketRate` when you open each of your strobed/polled I/O connections. Using this scheme, you must determine a valid `ExpPacketRate` for your DeviceNet system.

Figure 3-2 shows a scanned polling example for four polled devices at MAC ID 14, 17, 20, and 30. The shaded areas indicate other message traffic, such as the strobed I/O messages shown in Figure 3-1.

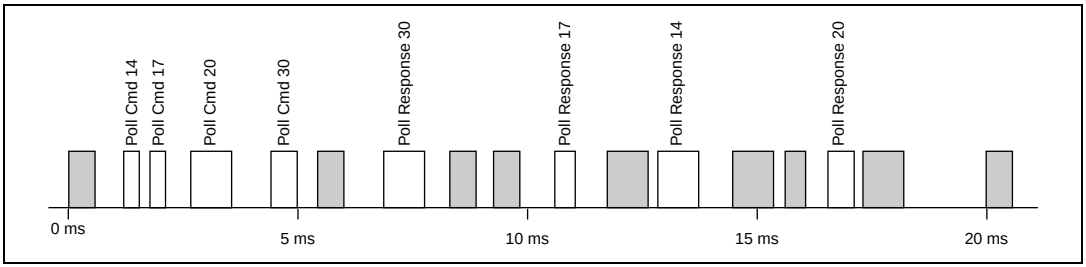


Figure 3-2. Scanned Polling Timing Example

Background Polling

Scanned polling can be less efficient when used with devices with significantly different response times or devices with significantly different rates of physical measurement. In the example above (Figure 3-2), consider what would happen if device 14 took 52 ms to respond and device 20 took 38 ms to respond. In this case, even though device 17 and device 30 respond well within 20 ms, the common `ExpPacketRate` would need to be at least 52 ms. This situation can often be avoided using a special case of scanned polling called *background polling*.

To configure background polling, you first set the `PollMode` parameter of `ncOpenDnetIntf` to `Scanned`. Then for each polled I/O connection you configure (`ncOpenDnetIO` with `ConnectionType` set to `Poll`), you must set `ExpPacketRate` to either a foreground rate or a background rate. The foreground poll rate is the same as the common `ExpPacketRate` used for all strobed I/O. Devices in this group generally respond quickly to poll commands or have data that changes relatively quickly. The background poll rate must be an exact multiple of the foreground poll rate. Devices in this group generally respond slowly to poll commands or have data that changes relatively slowly (such as temperature).

Background polling provides many of the same advantages as scanned polling. The handling of only two groups optimizes performance. Also, background polling maintains overall network consistency because NI-DNET evenly disperses all background poll commands among multiple foreground cycles. In other words, all background poll commands are not sent in quick succession and thus do not generate quick bursts of traffic on the network.

Figure 3-3 shows a background polling example which resolves the problem discussed previously. Devices at MAC ID 17 and 30 are foreground polled every 20 ms (as before). Devices at MAC ID 14 and 20 are background polled every 60 ms (3 times the 20 ms foreground rate). The shaded areas indicate other message traffic.

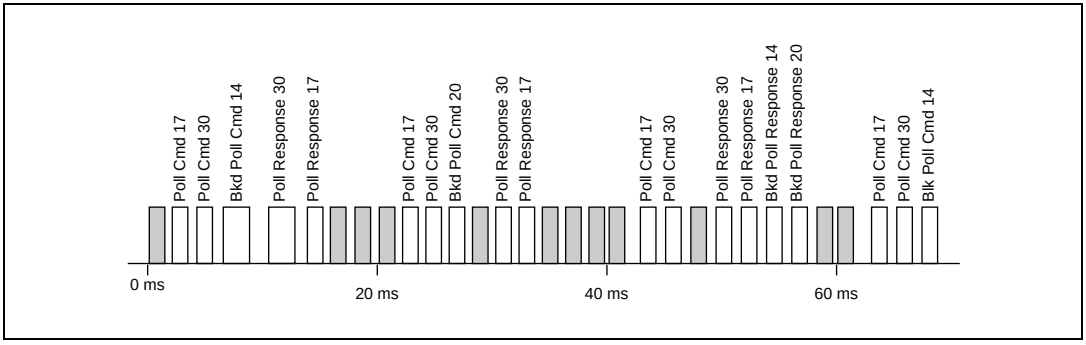


Figure 3-3. Background Polling Timing Example

Individual Polling

When the underlying response rates of all polled I/O devices do not fit into two clear groups, background polling can still be inefficient. For example, assume you have four different polled I/O sensors capable of updating measured input at 10 ms, 35 ms, 100 ms, and 700 ms respectively. Each device responds to its poll command within 1 ms but measures data at a different rate (such as a pushbutton for 10 ms and a temperature sensor for 700 ms). You could group these into a foreground rate of 10 ms and a background rate of 700 ms, but then much DeviceNet bandwidth would be wasted polling the 35 ms and 100 ms devices at the foreground rate. For this situation, the *individual polling* scheme is most appropriate.

To configure individual polling, first set the `PollMode` parameter of `ncOpenDnetIntf` to `Individual`. Then for each polled I/O connection you configure (`ncOpenDnetIO` with `ConnectionType` set to `Poll`), you must set `ExpPacketRate` to the rate desired for that device. Unlike the

scanned polling or background polling scheme, each poll command is no longer associated with the strobe command's rate, but instead is solely based on its `ExpPacketRate`.

Since the poll commands are not synchronized for individual polling, they can often be scattered relatively randomly. They can be evenly interspersed for a while, then suddenly occur in bursts of back-to-back messages. Because of this inconsistency, you should use smaller MAC IDs for smaller `ExpPacketRate` values. Since smaller MAC IDs in DeviceNet usually gain access to the network before larger MAC IDs, this helps to ensure that smaller rates can be maintained during bursts of increased traffic.

Figure 3-4 shows an individual polling example: MAC ID 3 is polled every 10 ms, MAC ID 10 every 35 ms, MAC ID 12 every 100 ms, and MAC ID 13 every 700 ms. Only the poll commands are shown (not poll responses or other messages).

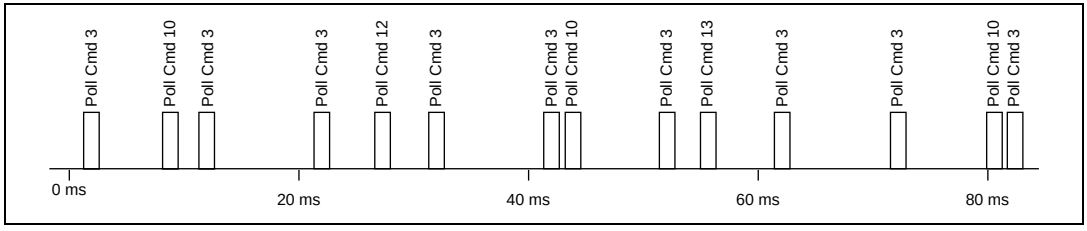


Figure 3-4. Individual Polling Timing Example

Cyclic I/O

Cyclic I/O connections essentially use the same timing scheme as individually polled I/O connections. Each cyclic I/O connection sends its data at the configured `ExpPacketRate`. The main difference is that cyclic I/O data is transferred from slave to master, rather than from master to slave.

In the DeviceNet Specification, a poll command message is exactly the same as a cyclic output message (master to slave data). Since cyclic data from master to slave can be handled using individual polling, cyclic I/O connections are more commonly used for input data from slave to master. For NI-DNET, this means that for cyclic I/O connections, `ncOpenDnetIO` is normally called with `InputLength` nonzero and `OutputLength` zero.

Just as for individually polled I/O, you should use smaller MAC IDs for smaller cyclic I/O `ExpPacketRate` values. Doing so ensures that cyclic I/O traffic is prioritized properly.

Change-of-State (COS) I/O

Change-of-State I/O connections use the same timing scheme as cyclic I/O connections, but in addition to the `ExpPacketRate`, COS I/O sends data to the master whenever a change is detected.

For COS I/O, the cyclic transmission is used solely to verify that the I/O connection still exists, so the `ExpPacketRate` is typically set to a large value, such as 10,000 (10 seconds). Given such a large `ExpPacketRate`, the main performance concerns for COS I/O are an appropriate MAC ID, and if needed, a nonzero `InhibitTimer`.

In many cases, a given COS I/O device cannot detect data changes very quickly. If a COS device is capable of detecting quickly changing data, there is a chance that it could transmit many COS messages back-to-back, precluding other I/O messages and thus dramatically impairing overall DeviceNet performance. This problem is demonstrated in Figure 3-5.

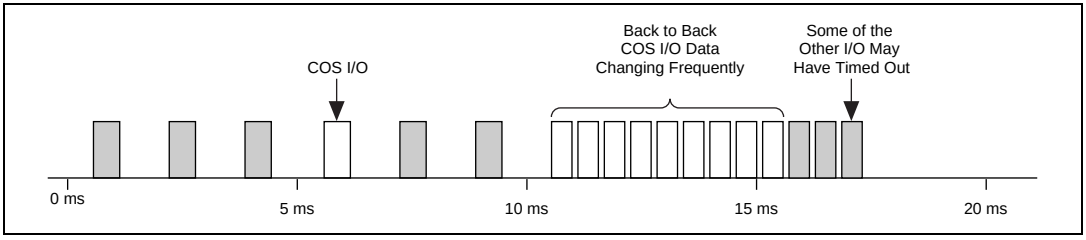


Figure 3-5. Congestion Due to Back-to-Back COS I/O

This problem can be prevented if you increase the MAC ID of the frequently changing COS I/O device. If the COS device has a higher MAC ID than other devices, it cannot preclude their I/O messages.

You can also prevent back-to-back COS I/O messages if you set the `InhibitTimer` driver attribute using `ncSetDriverAttr`. After transmitting COS data, the I/O connection must wait `InhibitTimer` before it can transmit COS data again. A reasonable value for `InhibitTimer` would be the smallest `ExpPacketRate` of an I/O connection with a larger MAC ID than the COS I/O device.

Automatic EPR Feature

For cyclic I/O connections, a valid `ExpPacketRate` is required for your call to `ncOpenDnetIO`. For COS I/O connections, a nonzero `ExpPacketRate` is recommended for your call to `ncOpenDnetIO` but can be set to a large value.

For strobed and polled I/O connections, determination of a valid ExpPacketRate can be somewhat complex. If you have trouble estimating an ExpPacketRate value for strobed/polled I/O, set the PollMode parameter of your initial call to ncOpenDnetIntf to Automatic. When you use this automatic EPR feature, the ExpPacketRate parameter of ncOpenDnetIO is ignored for strobed/polled I/O (ConnectionType of Strobe or Poll), and NI-DNET calculates a safe EPR value for you. This automatic EPR is the same for all strobed and polled I/O connections (scanned I/O).

After you start communication, you can use the ncGetDriverAttr function to determine the value calculated for ExpPacketRate. From that value, you can then experiment with other ExpPacketRate configurations using PollMode of Scanned or Individual.

The following information is used by NI-DNET to calculate a safe EPR:

- NI-DNET assumes that it is the only master in your DeviceNet system.
- The BaudRate parameter of ncOpenDnetIntf determines the time taken for each message.
- The InputLength and OutputLength parameters of each ncOpenDnetIO determine the time needed for each I/O message.
- NI-DNET assumes that each strobed/polled I/O device can respond to its command within 2 ms.
- NI-DNET sets aside a fixed amount of time for explicit messages. This time depends on the baud rate.

Using I/O Data in Your Application

Appendix A, *DeviceNet Programming Overview*, explains that the data transferred to and from a DeviceNet device on an I/O connection is usually processed by an Assembly Object within the slave device. Input assemblies represent the data received by NI-DNET from a remote device, and output assemblies represent data that NI-DNET transmits to a remote device.

To use a device's I/O data within your application, you need to understand the contents of its input and output assemblies. You can find this information in the following places:

- Printed documentation provided by the device's vendor.
- If the device conforms to a standard device profile, the I/O assemblies are defined within the DeviceNet Specification.

- Some device vendors provide comments about I/O assemblies in an Electronic Data Sheet (EDS). The EDS file is a text file whose format is defined by the DeviceNet Specification.
- Ask the device's vendor if they have filled out a DeviceNet compliance statement. This form is located at the front of the DeviceNet Specification, and it provides information about the device, including its I/O assemblies.

After you open an NI-DNET I/O Object and start communication, you use the `ncWriteDnetIO` function to write an output assembly for a device and the `ncReadDnetIO` function to read an input assembly received from a remote device. Both of these functions access the entire assembly as an array of bytes.

In most cases, the array of bytes for an input or output assembly contains more than one value. In DeviceNet terminology, an individual data value within an I/O assembly is referred to as a *member*.

Documentation for the members of an input or output assembly includes the position of each member in the assembly (often shown as a table with byte/bit offsets) and a listing of the attribute in the device that each member represents (often shown as class, instance, and attribute identifiers). For standard device profiles, the I/O assemblies are documented in the device profile's specification, and the actual attributes are documented in the individual object specifications. Attribute documentation includes the attribute's DeviceNet data type and a complete explanation of its meaning.

As an example of I/O assembly documentation, consider the standard AC Drive device profile. For this device profile, the DeviceNet Specification defines an output assembly called Basic Speed Control Output (Assembly Object instance 20). This output assembly is used to start/stop forward motion at a given speed and to reset faults in the device. The bytes of this output assembly are shown in Figure 3-6, and the attribute mapping is shown in Table 3-1.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	Fault Reset	0	Run Fwd
1	0	0	0	0	0	0	0	0
2	Speed Reference (Low Byte)							
3	Speed Reference (High Byte)							

Figure 3-6. AC Drive Output Assembly, Instance 20

Table 3-1. Attribute Mapping for Basic Speed Control Output Assembly

Member Name	Class Name	Class ID	Instance ID	Attribute Name	Attribute ID
Run Fwd	Control Supervisor	29 hex	1	Run1	3
Fault Reset	Control Supervisor	29 hex	1	FaultRst	12
Speed Reference	AC/DC Drive	2A hex	1	SpeedRef	8

By consulting the specifications for the Control Supervisor object and the AC/DC Drive object, you can determine that the DeviceNet data type for Run Fwd and Fault Reset is `BOOL` (boolean), and the DeviceNet data type for Speed Reference is `INT` (16-bit signed integer).

Accessing I/O Members in LabVIEW

Many fundamental differences exist between the encoding of a DeviceNet data type and its equivalent data type in LabVIEW. For example, for a 32-bit integer, the DeviceNet `DINT` data type uses Intel byte ordering (lowest byte first), and the equivalent LabVIEW `I32` data type uses Motorola byte ordering (highest byte first).

To make it easier for you to avoid these data type issues in your LabVIEW application, NI-DNET provides two functions to convert between LabVIEW data types and DeviceNet data types: `ncConvertForDnetWrite` and `ncConvertFromDnetRead`. These functions are used to access individual members of an I/O assembly using normal LabVIEW controls and indicators.

The following steps show an example of how you can use `ncConvertForDnetWrite` to access the Basic Speed Control Output Assembly described in the previous section:

1. Use the NI-DNET palette to place `ncConvertForDnetWrite` into your diagram.
2. Right-click on the `DnetData in` terminal and select **Create Constant**, then initialize the first 4 bytes of the array to zero.
3. Right-click on the `DnetType` terminal and select **Create Constant**, then select `BOOL` from the enumeration.
4. Right-click on the `ByteOffset` terminal and select **Create Constant**, then enter `0` as the byte offset.
5. Right-click on the `8[TF] in` terminal and select **Create Control**. In the front panel control that appears, you can use the button at index 0 to control Run Fwd and the button at index 2 to control Fault Reset.
6. Using the NI-DNET palette, place `ncConvertForDnetWrite` into your diagram.
7. Wire the `DnetData out` terminal from the previous `Convert` into the `DnetData in` terminal of this `Convert`.
8. Right-click on the `DnetType` terminal and select **Create Constant**, then select `INT` from the enumeration.
9. Right-click on the `ByteOffset` terminal and select **Create Constant**, then enter `2` as the byte offset.
10. Right-click on the `I32/I16/I8 in` terminal and select **Create Control**. You can use the front panel control that appears to change Speed Reference.
11. Using the NI-DNET palette, place `ncWriteDnetIO` into your diagram.
12. Wire the `DnetData out` terminal from the previous `Convert` into the `Data` terminal of `ncWriteDnetIO`.

For more information on the `ncConvertForDnetWrite` and `ncConvertFromDnetRead` functions, refer to the *NI-DNET Programmer Reference Manual*. For information on LabVIEW data types and their equivalent DeviceNet data types, refer to Chapter 1, *NI-DNET Data Types*, in the *NI-DNET Programmer Reference Manual*.

Accessing I/O Members in C

Since DeviceNet data types are very similar to C language data types, individual I/O members can be accessed in a straightforward manner. You can use the standard C language pointer manipulations to convert between C language data types and DeviceNet data types.

The following steps show an example of how standard C language can be used to access the Basic Speed Control Output Assembly described in the previous section:

1. Declare an array of 4 bytes, as in the following.

   ```
   NCTYPE_UINT8    OutputAsm[4];
   ```

2. Initialize the array to all zero.

   ```
   for (I = 0; I < 4; I++)
       OutputAsm [I] = 0;
   ```

3. Assume you have two boolean variables, `RunFwd` and `ResetFault`, of type `NCTYPE_BOOL`. For LabWindows/CVI, these variables could be accessed from front panel buttons. The following code inserts these boolean variables into `OutputAsm`.

   ```
   if (RunFwd)
       OutputAsm [0] |= 0x01;
   if (FaultReset)
       OutputAsm [0] |= 0x04;
   ```

4. Assume you have an integer variable `SpeedRef` of type `NCTYPE_INT16`. For LabWindows/CVI, this variable could be accessed from a front panel control. The following code inserts this integer variable into `OutputAsm`.

   ```
   *(NCTYPE_INT16 *)(&( OutputAsm[2])) = SpeedRef;
   ```

5. Write the output assembly to the remote device.

   ```
   status = ncWriteDnetIO(objh, sizeof(OutputAsm),
                OutputAsm);
   ```

For information on NI-DNET's C language data types and their equivalent DeviceNet data types, refer to Chapter 1, *NI-DNET Data Types*, of the *NI-DNET Programmer Reference Manual.*

Using Explicit Messaging Services

The NI-DNET Explicit Messaging Object represents an explicit messaging connection to a remote DeviceNet device. You use `ncOpenDnetExplMsg` to configure and open an NI-DNET Explicit Messaging Object.

The following sections describe how to use the Explicit Messaging Object.

Get and Set Attributes in a Remote DeviceNet Device

The two most commonly used DeviceNet explicit messages are the Get Attribute Single service and the Set Attribute Single service. These services are used to get or set the value of an attribute contained in a remote device. The easiest way to execute the Get Attribute Single service on a remote device is to use the NI-DNET `ncGetDnetAttribute` function. The easiest way to execute the Set Attribute Single service on a remote device is to use the NI-DNET `ncSetDnetAttribute` function.

For a given attribute of a DeviceNet device, you need the following information to use the `ncGetDnetAttribute` or `ncSetDnetAttribute` function:

- The class and instance identifiers for the object in which the attribute is located
- The attribute identifier
- The attribute's DeviceNet data type

You can normally find this information from the object specifications contained in the DeviceNet Specification, but many DeviceNet device vendors also provide this information in the device's documentation.

For the C programming language, the attribute's DeviceNet data type determines the corresponding NI-DNET data type you use to declare a variable for the attribute's value. For example, if the attribute's DeviceNet data type is `INT` (16-bit signed integer), you should declare a C language variable of type `NCTYPE_INT16`, then pass the address of that variable as the `Attr` parameter of the `ncGetDnetAttribute` or `ncSetDnetAttribute` function.

For LabVIEW, the attribute's DeviceNet data type determines the corresponding LabVIEW data type to use with the `ncConvertForDnetWrite` or `ncConvertFromDnetRead` functions. The `ncConvertFromDnetRead` function converts a DeviceNet attribute read using `ncGetDnetAttribute` into an appropriate LabVIEW data type. The `ncConvertForDnetWrite` function converts a LabVIEW data

type into an appropriate DeviceNet attribute to write using `ncSetDnetAttribute`. For more information on these LabVIEW conversion functions, refer to the previous section, *Using I/O Data in Your Application*.

Other Explicit Messaging Services

To execute services other than Get Attribute Single and Set Attribute Single, use the following sequence of function calls: `ncWriteDnetExplMsg`, `ncWaitForState`, `ncReadDnetExplMsg`. The `ncWriteDnetExplMsg` function sends an explicit message request to a remote DeviceNet device. The `ncWaitForState` function waits for the explicit message response, and the `ncReadDnetExplMsg` function reads that response.

Use `ncWriteDnetExplMsg` for such DeviceNet services as Reset, Save, Restore, Get Attributes All, and Set Attributes All. Although the DeviceNet Specification defines the overall format of these services, in most cases their meaning and service data are object-specific or vendor-specific. Unless your device requires such services and documents them in detail, you probably do not need them for your application.

You need the following information to use the `ncWriteDnetExplMsg` and `ncReadDnetExplMsg` functions for a given explicit messaging service:

- The class and instance identifiers for the object to which the service will be directed.
- The service code used to identify the service.
- The length and format of service request and response data. Some of data formats are defined by the service's overall specification (such as in Appendix G, *DeviceNet Explicit Services*, in the DeviceNet Specification), but many data formats are object-specific or vendor-specific. For example, for the Reset service, Appendix G defines the service's code for use with any object, but its actual data format is defined in the specification for the Identity Object.
- The error codes that can be returned in the service response. Error codes that are common to all services can be found in Appendix H, *DeviceNet Error Codes*, in the DeviceNet Specification, but many error codes are specific to the service, object, or vendor.

As with the `ncGetDnetAttribute` and `ncSetDnetAttribute` functions, the service data formats for the request and response are specified in terms of DeviceNet data types. These DeviceNet data types are converted to/from the data types of your programming environment (C or LabVIEW) as discussed in previous sections.

Handling Multiple Devices

This section describes techniques you can use to efficiently implement an application that communicates with a large number of DeviceNet devices. In such an application, there might be only one call to `ncOpenDnetIntf` (only one network), but there are usually multiple calls to `ncOpenDnetIO` (and possibly `ncOpenDnetExplMsg`).

Configuration

If the configuration parameters used with `ncOpenDnetIO` tend to change over time, you might want to organize them in data structures instead of using constants.

For the C programming language, you can declare a structure `typedef` to store the parameters of `ncOpenDnetIO`, similar to the following:

```
typedef struct {
    NCTYPE_UINT32        DeviceMacId;
    NCTYPE_CONN_TYPE     ConnectionType;
    NCTYPE_UINT32        InputLength;
    NCTYPE_UINT32        OutputLength;
    NCTYPE_UINT32        ExpPacketRate;
} OpenDnetIO_Struct;
```

For LabVIEW or BridgeVIEW, a cluster that contains these parameters is already defined for use with `ncOpenDnetIO`.

You can use this structure/cluster to declare an array that contains one entry for each call you make to `ncOpenDnetIO`. In LabVIEW, BridgeVIEW, and LabWindows/CVI, you can use front panel controls to index through this array and update configurations as needed.

In your code, write a For loop to index through the array and call `ncOpenDnetIO` once for each array entry. This simplifies your code because it does not contain a long list of sequential open calls, but instead all open calls are combined into a concise loop.

Object Handles

If you use an array to store configuration parameters for `ncOpenDnetIO`, you can use this same scheme to store the `ObjHandle` returned by `ncOpenDnetIO`. Within the For loop used for `ncOpenDnetIO`, you can store the resulting `ObjHandle` into an array of object handles. Throughout your code, you can index into this array to obtain the appropriate object handle.

Using an array of object handles is particularly useful in the LabVIEW programming environment because it eliminates confusing routing of individual object handle wires.

For applications with only a few object handles, another useful technique for LabVIEW is to store each object handle in an indicator, then create a local variable for each call that uses the handle. To create the indicator, right-click on the `ObjHandle out` terminal and select **Create Indicator**. To create a local variable, right-click on the indicator, select **Create»Local Variable**, right-click on the local variable, and select **Change To Read Local**. For more information on local variables, refer to the LabVIEW online reference.

Main Loop

If your application essentially accesses all DeviceNet input/output data as a single image, you would normally wait for read data to become available on one of the input connections (such as a strobed I/O connection), read all input data, execute your application code, then write all output data. The wait is important because it helps to synchronize your application with the overall DeviceNet network traffic.

In single-loop applications such as this, you normally set the `PollMode` parameter of `ncOpenDnetIntf` to `Automatic` or `Scanned` so that all poll command messages are sent out in quick succession.

Within a single-loop application, error handling is often done for the entire application as a whole. In the C programming language, this means that when an error is detected with any NI-DNET object, you display the error and exit the application. In LabVIEW, this means that you wire all error clusters of NI-DNET VIs together.

If your application uses different control code for different DeviceNet devices, you might want to split your application into multiple tasks. You can easily write a multitasking application by creating a notification for the NI-DNET `Read Avail` and `Error` states. This notification occurs when

either input data is available (to synchronize your code with each device's I/O messages), or an error occurs. In the C programming language, you create this notification callback using the `ncCreateNotification` function. In LabVIEW, you create this notification callback using the `ncCreateOccurrence` function.

In multiple-loop applications such as this, you normally set the `PollMode` parameter of `ncOpenDnetIntf` to `Individual` so that each poll command message can be sent out at its own individual rate.

Within a multiple-loop application, error handling is done separately for each task. In the C programming language, this means that when an error is detected, you handle it for the appropriate task, but you do not exit the application. In LabVIEW, this means that you only wire the error clusters of NI-DNET VIs that apply to each task, and thus you write different sub-diagrams that are not wired together in any way.

SimpleWho Utility

To provide valid parameters for the NI-DNET open functions (`ncOpenDnetIntf`, `ncOpenDnetExplMsg`, and `ncDnetOpenIO`), you need to determine some basic information about your DeviceNet devices. This information includes the MAC ID of each device, the I/O connections it supports, and the input/output lengths for those I/O connections.

In most cases, the vendor of each DeviceNet device provides this information, but if not, NI-DNET provides a utility that helps you determine this information. Searching a DeviceNet network to determine information about connected devices is often referred to as a *network who*, and thus the NI-DNET utility is called `SimpleWho`. This utility is not a complete network management or configuration utility. It provides read-only information about the DeviceNet devices connected to your National Instruments DeviceNet interface.

You can run the `SimpleWho` utility in one of the following ways:

- From the NI-DNET program group, select **Start»Programs» National Instruments DNET»SimpleWho**.
- Double-click on its name within the NI-DNET installation directory, such as `C:\nidnet\SimpleWho`.

When you run the `SimpleWho` utility, a dialog box titled **Setup Online Info** appears, as shown in Figure 3-7.

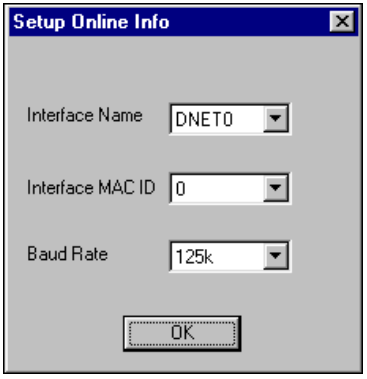


Figure 3-7. Setup Online Info Dialog Box

You use this dialog box to provide the configuration information needed to search your DeviceNet network. The **Setup Online Info** dialog box has the following controls:

- **Interface Name**—This name selects the DeviceNet interface to use. If you only have one National Instruments DeviceNet interface installed, the default `DNET0` is appropriate.
- **Interface MAC ID**—This selects the DeviceNet MAC ID to use for your National Instruments DeviceNet interface (it does not refer to a device). If you do not know of an unused MAC ID in your network, a MAC ID of `0` is often acceptable.
- **Baud Rate**—This selects the baud rate used by your DeviceNet devices, either `125000`, `250000`, or `500000` bits per second.

After you enter these values, select **OK**, and `SimpleWho` does a network who on your DeviceNet network.

For each device found, `SimpleWho` shows you the MAC ID of the device, followed by its product name. To see detailed information for each device, double-click on its MAC ID in the main list. A dialog box titled **Device Details** appears, as shown in Figure 3-8.

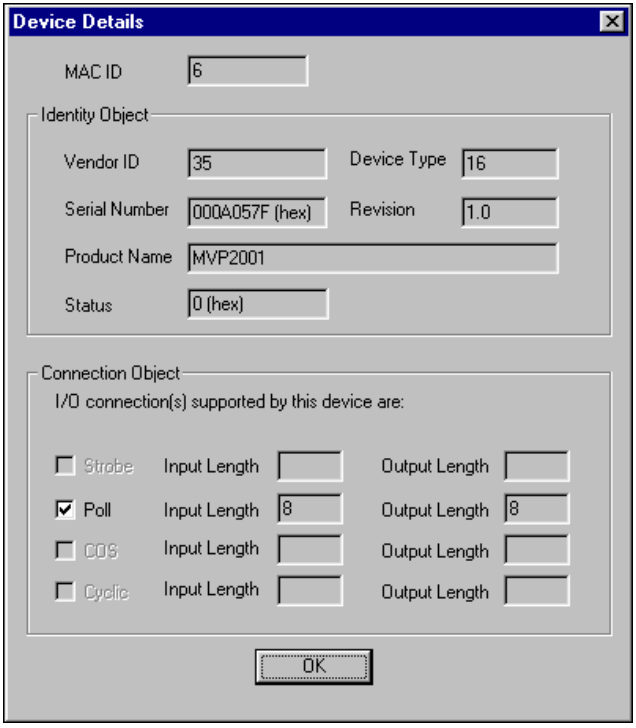


Figure 3-8. Device Details Dialog Box

This dialog box displays the following information:

- **MAC ID**—Address of the device (decimal)
- **Vendor ID**—Identifier assigned by ODVA to the device's vendor (decimal)
- **Device Type**—Identifies the standard device profile (decimal)
- **Serial Number**—Serial number assigned by the vendor (hexadecimal)
- **Revision**—Major and minor revision of the vendor's firmware
- **Product Name**—Text name for the device, assigned by the vendor

- **Status**—Bit mask for device status (hexadecimal)

  The following bits are defined for the DeviceNet status attribute:

  – **Owned** (bit 0, hex 0001)—A master is communicating with the slave device

  – **Configured** (bit 2, hex 0004)—Device is configured different than “out of box”

  – **Minor Recoverable Fault** (bit 8, hex 0100)—Device is still usable

  – **Minor Unrecoverable Fault** (bit 9, hex 0200)—Device is still usable

  – **Major Recoverable Fault** (bit 10, hex 0400)—Device is in a fault state

  – **Major Unrecoverable Fault** (bit 11, hex 0800)—Device is in a fault state

- **I/O Connections**—Provides information for use with `ncOpenDnetIO`

  The following fields are provided for each master/slave I/O connection type (strobe, poll, COS, and cyclic):

  – **Checkbox**—Indicates whether the slave device supports this connection type.

  – **Input Length**—Number to use as the `InputLength` parameter of `ncOpenDnetIO`. From the standpoint of the slave device itself, this length is referred to as the `produced_connection_size`.

  – **Output Length**—Number to use as the `OutputLength` parameter of `ncOpenDnetIO`. From the standpoint of the slave device itself, this length is referred to as the `consumed_connection_size`.

When `SimpleWho` is finished searching your network, you can select **Who Again** to perform the network who again (such as when you power on a new device), or **Exit** to exit the utility.

DeviceNet Network Management

Many DeviceNet devices have various internal attributes which must be configured properly prior to integration into your DeviceNet I/O system. One example is a device that does not provide an external switch for its MAC ID, and thus an appropriate MAC ID must be configured. Other examples include minimum/maximum detection distances for a

photoelectric sensor, minimum/maximum speed limits for an AC drive, and selecting voltage or current for an analog input.

You can handle these configuration tasks by integrating them into your NI-DNET application. Use the explicit messaging discussed in the previous section to execute Set Attribute Single to set a device's configuration attributes.

Another way to handle device configuration is to use a network management utility. A network management utility provides a user interface to set configuration attributes, as well as other features useful in managing a DeviceNet system.

NI-DNET supports Allen-Bradley's WinDnet device driver specification. If you use Allen-Bradley's DeviceNet Manager software to manage your DeviceNet system, you can use your National Instruments interface with that software package.

4

Application Examples

This chapter describes the example applications provided with your NI-DNET software.

The examples in this chapter illustrate basic NI-DNET programming and specific concepts and techniques that can help you write your own applications. Each example includes an overview, steps used to run the example, a program flowchart for the example's source code, and information on how the example can be extended.

The following example programs, included with your NI-DNET software, are discussed in detail in this chapter:

- `SingleDevice` does I/O communication as a DeviceNet master with a single slave device.
- `GetIdentityAttrs` uses explicit messaging services to get attributes from a device's Identity Object.

Example 1. SingleDevice

This example does I/O communication as a DeviceNet master with a single slave device. You can use this example with any type of DeviceNet slave device. The example should be a starting point for you to learn to use the NI-DNET software.

The LabVIEW front panel for this example is shown in Figure 4-1. The LabWindows/CVI front panel and the Visual Basic front panel are similar.

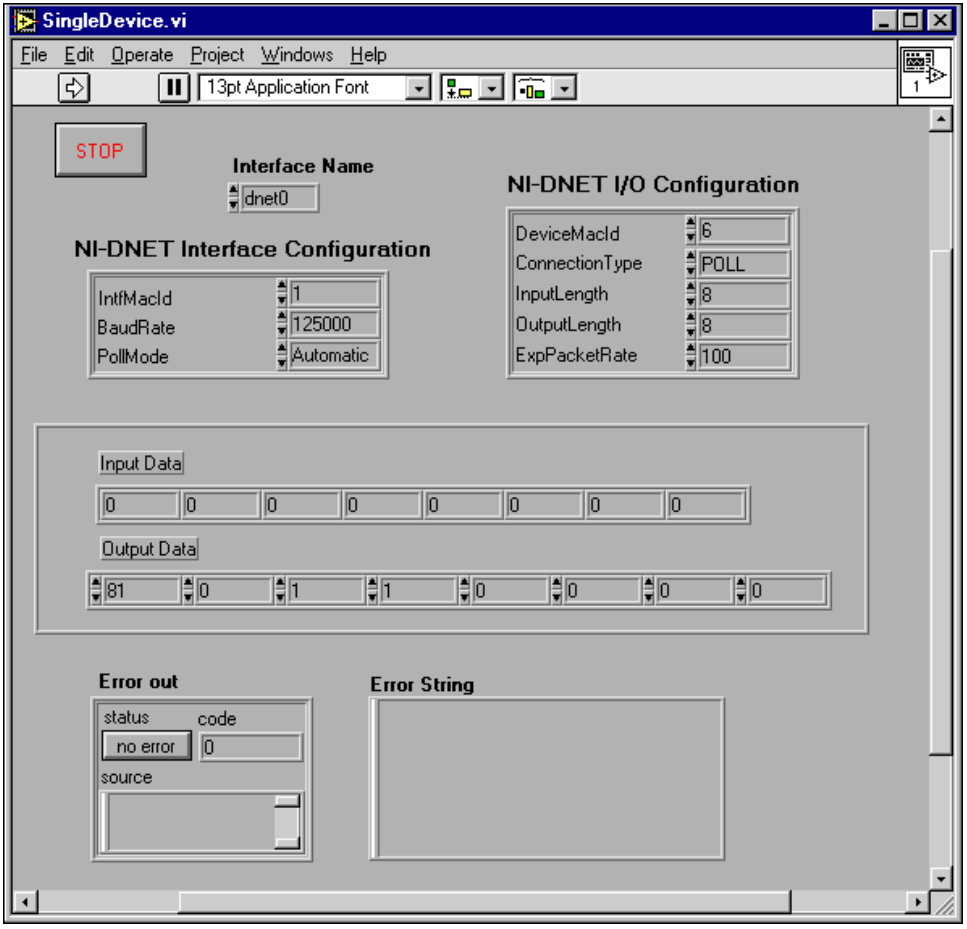


Figure 4-1. SingleDevice Front Panel

Run the Example

Follow these steps to run the example:

1. Connect a single DeviceNet slave device to your DeviceNet interface. For information on DeviceNet cabling, refer to your getting started manual.
2. Determine the baud rate and MAC ID used by your slave device.

   Many devices provide external switches for the MAC ID and baud rate. If this is the case, consult the documentation provided by your device's vendor.

If your device does not have baud rate and MAC ID external switches, the baud rate is often fixed at 125,000 (or determined by the device automatically), and the MAC ID is configured using a network management utility. For information on network management utilities, refer to Chapter 3, *NI-DNET Programming Techniques*.

If you know the baud rate of your device but not the MAC ID, NI-DNET provides a utility used to tell you the MAC ID of each connected device (`SimpleWho`). For information on `SimpleWho`, refer to the *SimpleWho Utility* section in Chapter 3, *NI-DNET Programming Techniques*.

3. Determine the type of I/O communication supported by your DeviceNet device. For example, most photoelectric sensors support strobed I/O with an input length of 1 byte. You need to know the I/O connection type (poll, strobe, and so on), number of input bytes, and number of output bytes.

   This information should be included in the documentation provided by the device vendor. If not, you can use the `SimpleWho` utility mentioned in Step 2 to find the information.

4. Load `SingleDevice` into your programming environment.

   - For LabVIEW or BridgeVIEW—Select **File»Open,** then find `SingleDevice.vi` in the LabVIEW or BridgeVIEW `Examples` directory.
   - For LabWindows/CVI—Select **File»Open»Project,** then find `SingleDevice.prj` in the LabWindows/CVI `Samples` directory.
   - For other programming environments, such as Microsoft C/C++, you can open `SingleDevice.c` in the NI-DNET `Examples` directory. Since the steps needed to edit, compile, and run `SingleDevice.c` are specific to your programming environment, the following steps provide information only for LabVIEW, BridgeVIEW, or LabWindows/CVI. For other development environments, refer to the section *Compiling and Linking the Example in Other Environments*.

5. Change the front panel controls to match the capabilities of your slave device.

   For LabVIEW or BridgeVIEW, you must change the front panel controls before you run the example.

   For LabWindows/CVI, you must first run the example to access the front panel controls.

   - **Interface Name**—This name selects the DeviceNet interface to use. If you only have one National Instruments DeviceNet interface installed, the default `DNET0` is appropriate.
   - **Interface MAC ID**—This selects the DeviceNet MAC ID to use for your National Instruments DeviceNet interface (it does not refer to a device). If you do not know of an unused MAC ID in your network, a MAC ID of `0` is often acceptable.
   - **Baud Rate**—This selects the baud rate used by your DeviceNet device. Enter the value you determined in Step 2.
   - **Poll Mode**—Leave this control set to `Automatic`.
   - **Device MAC ID**—This selects the MAC ID of your DeviceNet device. Enter the value you determined in Step 2.
   - **Connection Type**—This selects the type of I/O connection to use with your DeviceNet device. Enter the value you determined in Step 3.
   - **Input Length**—This selects the number of bytes to read from the device's I/O connection. Enter the value you determined in Step 3.
   - **Output Length**—This selects the number of bytes to write to the device's I/O connection. Enter the value you determined in Step 3.
   - **Exp Packet Rate**—This determines the rate of I/O communication. Since you set **Poll Mode** to `Automatic`, if you specified a **Connection Type** of `Strobe` or `Poll`, this rate is determined automatically by NI-DNET, and the value in this control is ignored (you can leave it zero). If you specified a **Connection Type** of `COS`, set this control to `10000`. If you specified a **Connection Type** of `Cyclic`, set this control to `100`.

6. Start I/O communication.

   For LabVIEW or BridgeVIEW, select the **Run** button (right arrow) on the LabVIEW/BridgeVIEW menu bar to run the example.

   For LabWindows/CVI, select the **Start** button on the example's front panel.

   After you start I/O communication, it takes up to six seconds for communication to initialize. After initialization is complete, you can view the slave's input bytes on the front panel **Input Data** indicators, and you can enter new output bytes using the front panel **Output Data** controls. You can use these input and output bytes to test and manipulate the physical capabilities of your slave device.

7. Stop the example.

   For LabVIEW or BridgeVIEW—Select the **Stop** button on the example's front panel. If needed, you can still view any errors or other information on the front panel.

   For LabWindows/CVI—Select the **Stop** button on the example's front panel. Although I/O communication has stopped, you can still view any errors or other information on the front panel. Select the **Exit** button to close the front panel and exit back to the LabWindows/CVI environment.

Congratulations on completing your first DeviceNet application! You should now view the source code of the example to understand its implementation. Refer to the next section for a program flow chart of the example's source code.

Program Flow Chart

Figure 4-2 shows the steps for the `SingleDevice` application example. For general information about programming an NI-DNET application, refer to Chapter 2, *Developing Your Application*.

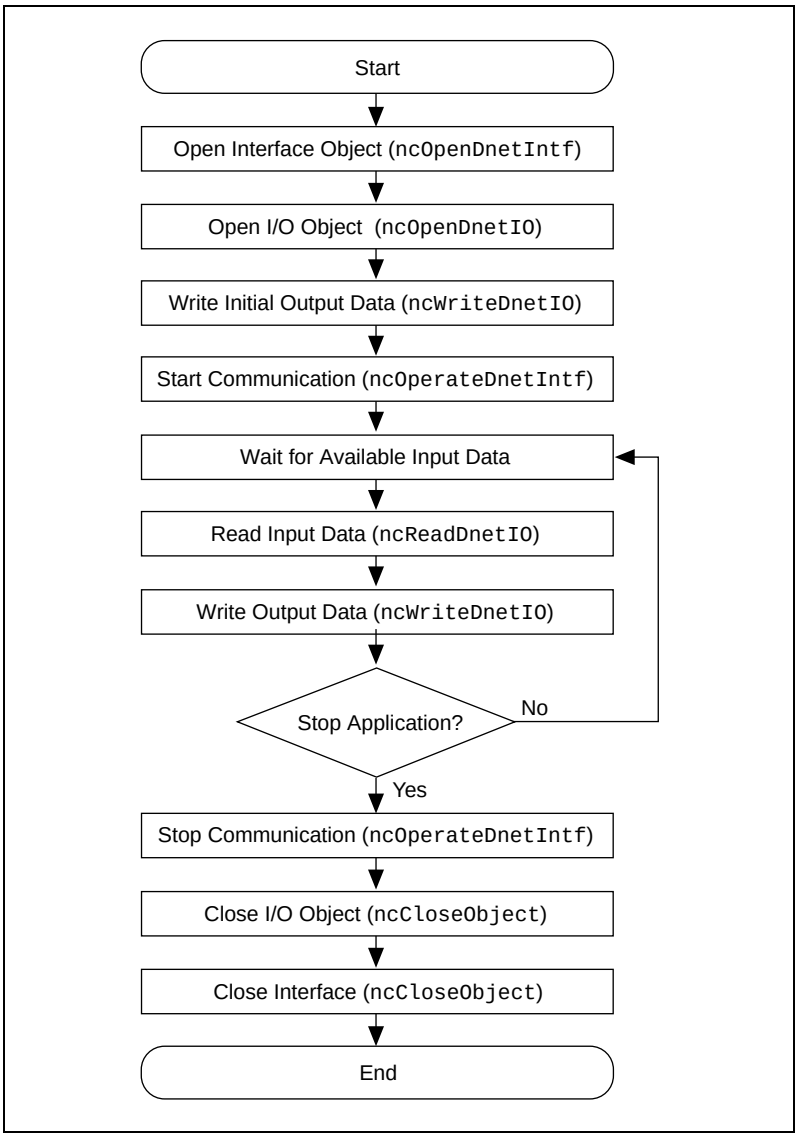


Figure 4-2. Programming Steps for SingleDevice Example

Extending the Example

The following list provides some ideas on how you might extend the `SingleDevice` example:

- Use your device's specific I/O data. For LabVIEW, BridgeVIEW, and LabWindows/CVI, this often involves use of various controls and indicators for each I/O item (instead of only byte arrays).

  For information on using a device's I/O data in your application, refer to the section *Using I/O Data in Your Application* in Chapter 3, *NI-DNET Programming Techniques*.

- Add additional slave devices. This involves duplicating all NI-DNET functions to communicate with the additional devices. For information on handling multiple slave devices, refer to the section *Handling Multiple Devices* in Chapter 3, *NI-DNET Programming Techniques*.

- Add explicit messaging support, such as for configuration. For information on explicit messaging, refer to the section *Using Explicit Messaging Services* in Chapter 3, *NI-DNET Programming Techniques*, and the next section, *Example 2. GetIdentityAttrs*.

- Implement a control algorithm. This involves reading inputs, executing control code for the physical environment, and writing outputs.

- Communicate as a DeviceNet slave. The master for this slave I/O can be a standalone controller such as a Programmable Logic Controller (PLC), or another National Instruments DeviceNet interface. Slave I/O can be used to integrate PC-based operator interface software, data acquisition, motion control, or image acquisition products into your master's control system. To enable slave I/O, set the I/O Object's `DeviceMacId` to the same value as `InterfaceMacId`. The `InputLength` is still used with `ncReadDnetIO`, and `OutputLength` is used with `ncWriteDnetIO`. You must start the `SingleDevice` example for slave I/O before you start the master.

Compiling and Linking the Example in Other Environments

This section describes how to compile and link the example using a variety of development environments.

Using Microsoft Visual C/C++ 2.0 or later, or Borland C/C++ 5.0 or later

The NI-DNET examples for C are written so that they can be compiled as Win32 console applications. A Win32 console application is a Win32 program which uses text-based input and output, not a graphical interface. This allows you to create a Win32 application by using simple input and output functions like `printf` and `scanf`.

Use the following steps to compile and link the example `SingleDevice.c` using the development environment of Microsoft Visual C/C++ version 4.0 or later (version 2.0 is similar):

1. Start the Microsoft Developer Studio.
2. Select **File»New**, then select **Project Workspace** and click on **OK**.
3. In the dialog box that appears, enter the following, then click on **Create**.

   Type: Console Application

   Name: SingleDevice

   Platforms: Win32

   Location: `c:\nidnet\examples` (NI-DNET examples directory)
4. Select **Insert»Files into Project**, then select `SingleDevice.c` and click on **Add**.
5. Select **Insert»Files into Project**, then select `nidnetms.lib` (located in `c:\nidnet\langint`) and click on **Add**. This adds the NI-DNET link library for Microsoft C/C++.
6. Select **Build»Build SingleDevice.EXE**.
7. After the build is complete, you can run the example by selecting **Build»Run SingleDevice.EXE**.

   You can also compile and link the example `SingleDevice.c` from the standard DOS shell command line using Microsoft Visual C/C++ (version 2.0 or later, `CL.EXE` version 10.0 or later) by typing the following:

   ```
   cl Single~1.c nidnetms.lib
   ```

For complete information, refer to your Microsoft Visual C/C++ documentation.

The steps required to compile and link an NI-DNET example for Borland C/C++ 5.0 are similar to Microsoft C/C++.

Using Microsoft Visual Basic

The following steps can be used to compile, link, and run the example SingleDevice using Visual Basic 5.0 or later:

1. Start Visual Basic.
2. Select **File»New Project»Standard EXE**. If any default form file is created for the project, right-click on that form name in the project window and select the **Remove Form** option from the menu.
3. Select **Project»Add Form**, and in the dialog box that appears, select **Existing**. Go to `C:\nidnet\examples\vb` (NI-DNET examples directory) and add `SingleDevice.frm` and `SingleDeviceHelp.frm` to your project.
4. Select **Project»Add Module** and in the dialog box that appears, select **Existing**. Go to `C:\nidnet\langint` (NI-DNET language interface directory) and add `nidnet.bas` to your project.
5. Select **Project»Properties** from the menu. In the dialog box that appears, select the **General** tab and under **StartUp Object** select **frmMain**.
6. Select **Run»Start (or Start with Full Compile)** from the menu to start your example.
7. On the front panel, enter the configuration for your DeviceNet interface (Interface Name, Interface MAC ID, Baud Rate).
8. On the front panel, enter the configuration for communication with the DeviceNet device you have connected (Device MAC ID, Connection Type, Input Length, Output Length, Exp Packet Rate).
9. On the front panel, enter the initial Output Data bytes (in hex) for the DeviceNet device you have connected.
10. Click on the **Start** button from the front panel to start communication.
11. When you are done viewing the incoming Input Data and experimenting with changes to the Output Data, select the **Stop** button to stop communication and view any error information.
12. To view help at any time during the execution of the program, select the **Help** button from the front panel.
13. When you are finished, select the **Exit** button to exit the example application and return to Visual Basic.
14. To view the source code, select **View»Code** and go to `SingleDevice.frm` window.

Example 2. GetIdentityAttrs

This example uses explicit messaging services to get attributes from a remote device's Identity Object. The DeviceNet Identity Object provides attributes which identify the device's vendor, device profile, and product name. The LabVIEW front panel for this example is shown in Figure 4-3. The LabWindows/CVI front panel and the Visual Basic front panel are similar.

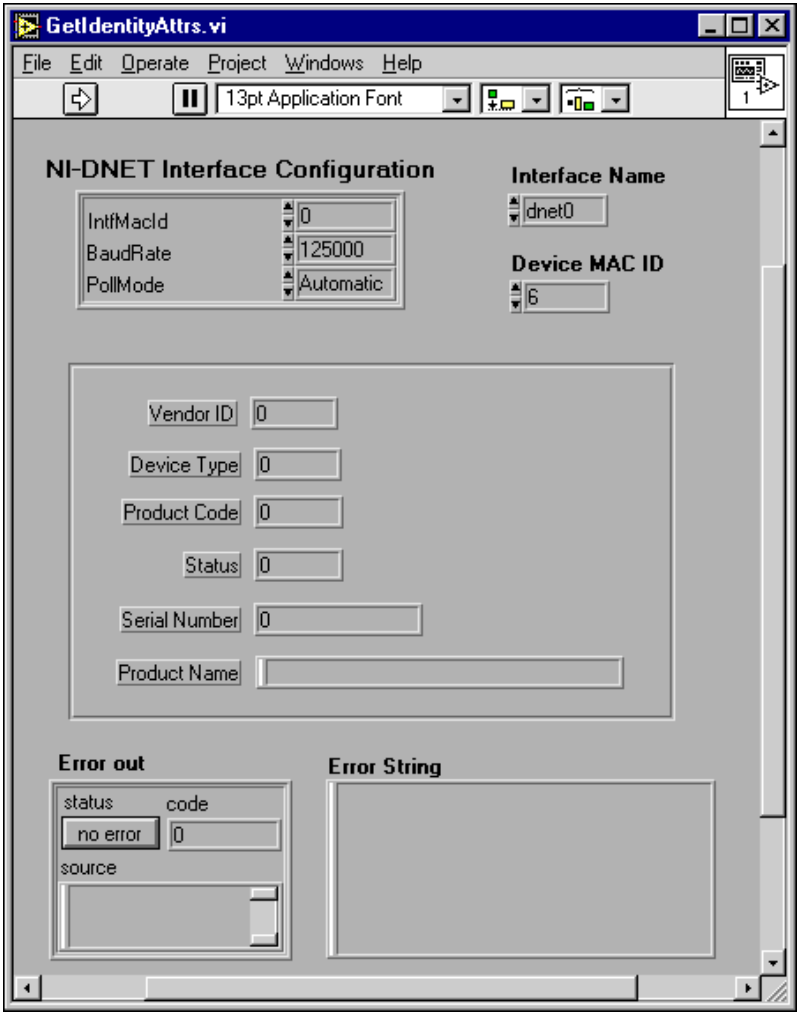


Figure 4-3. GetIdentityAttrs Front Panel

Run the Example

Follow these steps to run the example:

1. Connect a single DeviceNet slave device to your DeviceNet interface. For information on DeviceNet cabling, refer to your getting started manual.
2. Determine the baud rate and MAC ID used by your slave device.

   Many devices provide external switches for the MAC ID and baud rate. If this is the case, consult the documentation provided by your device's vendor.

   If your device does not have baud rate and MAC ID external switches, the baud rate is often fixed at 125,000 (or determined by the device automatically), and the MAC ID is configured using a network management utility. For information on network management utilities, refer to Chapter 3, *NI-DNET Programming Techniques*.

   If you know the baud rate of your device but not the MAC ID, NI-DNET provides a simple utility used to tell you the MAC ID of each connected device (`SimpleWho`). For information on `SimpleWho`, refer to the section *SimpleWho Utility* in Chapter 3, *NI-DNET Programming Techniques*.
3. Load `GetIdentityAttrs` into your programming environment.
   - For LabVIEW or BridgeVIEW—Select **File»Open**, then find `GetIdentityAttrs.vi` in the LabVIEW or BridgeVIEW `Examples` directory.
   - For LabWindows/CVI—Select **File»Open»Project**, then find `GetIdentityAttrs.prj` in the LabWindows/CVI `Samples` directory.
   - For other programming environments, such as Microsoft C/C++, you can open `GetIdentityAttrs.c` in the NI-DNET `Examples` directory. Since the steps needed to edit, compile, and run `GetIdentityAttrs.c` are specific to your programming environment, the following steps provide information only for LabVIEW, BridgeVIEW, or LabWindows/CVI.

4. Change the front panel controls to match the capabilities of your slave device.

   For LabVIEW or BridgeVIEW, you must change the front panel controls prior to running the example.

   For LabWindows/CVI, you must first run the example to access the front panel controls.

   - **Interface Name**—This name selects the DeviceNet interface to use. If you only have one National Instruments DeviceNet interface installed, the default `DNET0` is appropriate.
   - **Interface MAC ID**—This selects the DeviceNet MAC ID to use for your National Instruments DeviceNet interface (it does not refer to a device). If you do not know of an unused MAC ID in your network, a MAC ID of `0` is often acceptable.
   - **Baud Rate**—This selects the baud rate used by your DeviceNet device. Enter the value you determined in Step 2.
   - **Poll Mode**—Leave this control set to `Automatic`.
   - **Device MAC ID**—This selects the MAC ID of your DeviceNet device. Enter the value you determined in Step 2.

5. Get the identity attributes.

   For LabVIEW or BridgeVIEW, select the **Run** button (right arrow) on the LabVIEW/BridgeVIEW menu bar to run the example.

   For LabWindows/CVI, select the **Start** button on the example's front panel.

   When you run the example, it calls `ncGetDnetAttribute` to get each Identity Object attribute, displays those attributes on the front panel, then stops.

   For LabWindows/CVI, select the **Exit** button to close the front panel and exit back to the LabWindows/CVI environment.

You should now view the source code of the example to understand its implementation. Refer to the next section for a program flow chart of the example's source code.

Program Flow Chart

Figure 4-4 shows the steps for the `GetIdentityAttrs` application example. For general information about programming an NI-DNET application, see Chapter 2, *Developing Your Application*.

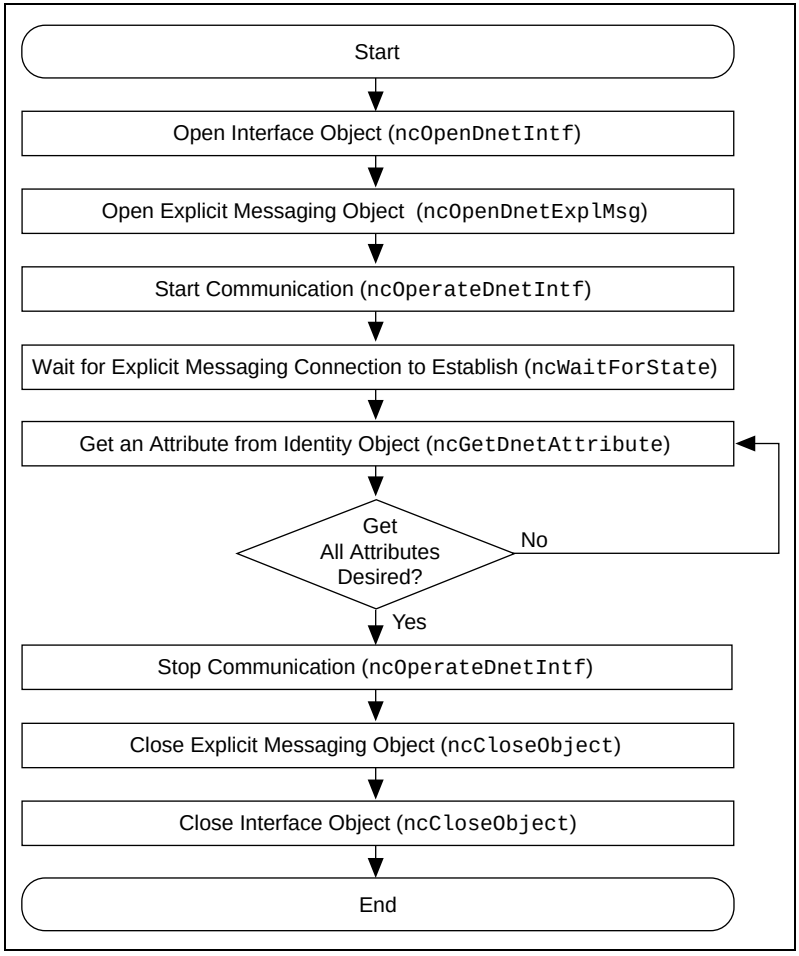


Figure 4-4. Programming Steps for GetIdentityAttrs Example

Extending the Example

The following list provides some ideas on how you might extend the `GetIdentityAttrs` example:

- Get attributes from other objects in the device. This involves changing the class ID, instance ID, and attribute ID used in `ncGetDnetAttribute`.
- Use the DeviceNet Set Attribute Single service to support device configuration. The NI-DNET `ncSetDnetAttribute` function is used to set an attribute in a remote DeviceNet device.
- Add additional slave devices. This involves duplicating all NI-DNET functions to communicate with the additional devices. For information on handling multiple slave devices, refer to the section *Handling Multiple Devices* in Chapter 3, *NI-DNET Programming Techniques*.
- Use explicit messaging services other than Get Attribute Single and Set Attribute Single.

Other Examples

NI-DNET ships with two other example applications that demonstrate the use of NI-DNET functions. The general steps to run these examples are similar to the ones discussed earlier. A brief description of the purpose and use of each example is given below.

MultipleDevice.vi (LabVIEW only)

This example does I/O communication with one or more slave device(s) using the `Easy IO Config` and `Easy IO Close` VIs. It can be used with any type and any number of DeviceNet slave devices. It is intended to demonstrate the ease with which multiple devices (and connections) can be added to your LabVIEW application. You can easily modify this example to add the `Create Occurence.vi` and other NI-DNET VIs to meet the requirements of your application.

WriteReadExplMsg

This example demonstrates how to execute a DeviceNet service on a remote DeviceNet device. It sends an explicit message request, waits for the response from the device, and then reads the response. This example uses the Get Attribute Single service to get the first attribute (Vendor ID) of the Identity Object (Class ID = 1, Instance ID = 1) that is present in the remote device. You can perform any other DeviceNet services (such as Set Attribute Single, Reset, and Start) with this example.

To learn more about the service codes, class codes, instance Ids, and service data bytes, refer to the DeviceNet Specification.

A

DeviceNet Programming Overview

This appendix gives an overview of DeviceNet.

History of DeviceNet

The Controller Area Network (CAN) was developed in the early 1980s by Bosch, a leading automotive equipment supplier. CAN was developed to overcome the limitations of conventional automotive wiring harnesses. CAN connects devices such as engine controllers, anti-lock brake controllers, and various sensors and actuators on a common serial bus. By using a common pair of signal wires, any device on a CAN network can communicate with any other device.

As CAN implementations became widespread throughout the automotive industry, CAN was standardized internationally as ISO 11898, and major semiconductor manufacturers such as Intel, Motorola, and Philips began producing CAN chips. With these developments, many manufacturers of industrial automation equipment began to consider other applications of CAN technology. Automotive and industrial device networks showed many similarities, including the transition away from dedicated signal lines, low cost, resistance to harsh environments, and excellent real-time capabilities.

In response to these similarities, Allen-Bradley developed DeviceNet, an industrial networking protocol based on CAN. DeviceNet built on CAN's communication facilities to provide higher-level features which allow industrial devices from different vendors to operate on the same network.

Soon after DeviceNet was developed, Allen-Bradley transferred the specification to an independent organization called the Open DeviceNet Vendor's Association (ODVA). ODVA formally manages the DeviceNet Specification and provides services to facilitate development of DeviceNet devices and tools by various vendors. Due in large part to the efforts of ODVA, hundreds of different vendors now provide DeviceNet products for a wide range of applications.

Physical Characteristics of DeviceNet

The list below summarizes the physical characteristics of DeviceNet.

- Trunkline-dropline cabling—main trunk cable with a drop cable for each device
- Selectable baud rates of 125K, 250K, and 500K

Table A-1. DeviceNet Baud Rates and Wiring Lengths

Baud Rate	Trunk Length	Drop Length Maximum	Drop Length Cumulative
125Kb/s	500 m (1640 ft)	6 m (20 ft)	156 m (512 ft)
250Kb/s	250 m (820 ft)	6 m (20 ft)	78 m (256 ft)
500Kb/s	100 m (328 ft)	6 m (20 ft)	39 m (128 ft)

- Support for up to 64 devices—each device identifies itself using a MAC ID (Media Access Control Identifier) from 0-63
- Device removal/insertion without severing the network
- Simultaneous support for both network-powered and self-powered devices
- Various connector styles

For complete information on how to connect your National Instruments hardware onto the DeviceNet network, refer to your getting started manual.

General Object Modeling Concepts

The DeviceNet Specification uses object-oriented modeling to describe the behavior of different components in a device, how those components relate to one another, and how network communication takes place. The following paragraphs briefly describe object-oriented modeling and how these concepts are used within the DeviceNet Specification.

In object-oriented terminology, a classification of components with similar qualities is called a *class*. For example, different classes of geometric shapes could include squares, circles, and triangles. Figure A-1 shows various classes and instances of geometric shapes.

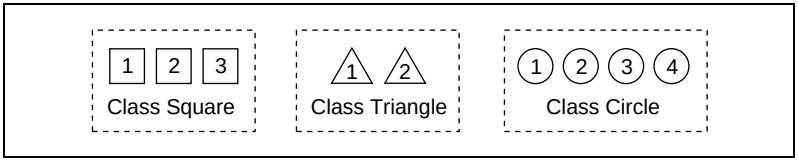


Figure A-1. Classes of Geometric Shapes

All squares belong to the same class because they all have similar qualities, such as four equal sides. The term instance refers to a specific instance of a given class. For example, a blue square of 4 inches per side would be one instance of the class square, and a red square of 5 inches per side would be another instance. The term object is often used as a synonym for the term instance, although in some contexts it might also refer to a class.

Each class defines a set of attributes which represent its externally visible characteristics. The set of attributes defined by a class is common to all instances within that class. For the class square, attributes could include length of each side and color. For the class circle, attributes could include radius and color. Each class also defines a set of services (or methods) which is used to perform an operation on an instance. For the class square, services could include resize, rotate, or change color.

Object Modeling in the DeviceNet Specification

Figure A-2 illustrates the object modeling used within the DeviceNet Specification.

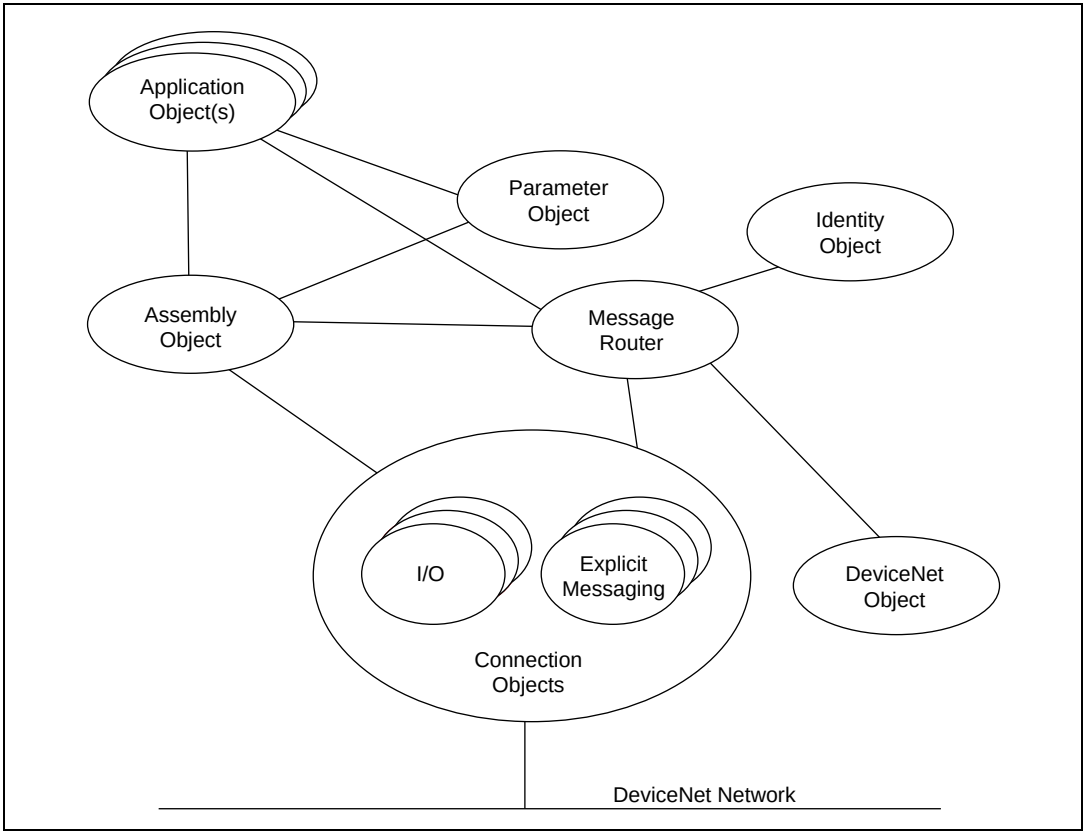


Figure A-2. Object Modeling Used in DeviceNet Specification

Every DeviceNet device contains at least one instance (instance one) of the Identity Object. The Identity Object instance defines attributes which describe the device, including the device's vendor, product name, and serial number. The Identity Object also defines services which apply to the entire device. For example, if you use the Reset service on instance one of the Identity Object, the device resets to its power on state.

Another class of object contained in every DeviceNet device is the Connection Object. Each instance of the Connection Object represents a communication path to one or more devices. Attributes of each Connection

Object instance include the maximum number of bytes produced on the connection, the maximum number of bytes consumed, and the expected rate at which data is transferred.

In Figure A-2, the term Application Object(s) refers to objects within the device which are used to perform its fundamental behavior. For example, within a photoelectric sensor, an instance of the Presence Sensing object (an Application Object) represents the physical photoelectric sensor hardware. Within a position controller device, an instance of the Position Controller object (an Application Object) is provided for every axis (motor) which can be controlled using the device.

For more information on the classes, instances, attributes, and services provided by DeviceNet, refer to the DeviceNet Specification. You can find additional information on the specific classes and instances supported by a given device in the documentation that came with the device.

Although the NI-DNET driver software provides object instances which are used to access the DeviceNet network, these objects do not correspond directly to the objects defined by the DeviceNet Specification, and the NI-DNET functions do not directly correspond to the services defined by DeviceNet. To facilitate access to your DeviceNet network, the features provided by the NI-DNET driver are a simplification of the objects and services defined in the DeviceNet Specification.

Explicit Messaging Connections

Each device on the DeviceNet network supports at least one explicit messaging connection. Explicit messaging connections provide a general-purpose communication path used to execute services on a particular object in a device.

For a given explicit messaging connection between two DeviceNet devices, the device requesting execution of the service is called the *client*, and the device to which the service request is directed is called the *server*. Your NI-DNET software can be used as an explicit messaging client with any number of DeviceNet server devices.

Using an explicit messaging connection, the client device sends an explicit message request to the server device. This request indicates the service to perform and the object to which the service is directed. When the server receives the explicit message request, it executes the service and sends an explicit message response to the client device. If the service executed successfully, this response contains information requested by the client.

The MAC ID (address) of the explicit message client and server is contained in the header of the DeviceNet explicit messages.

The following tables describe the general format of DeviceNet explicit message requests and responses as they appear on the DeviceNet network.

Table A-2. Explicit Message Request

Field	Description
Service Code	This number identifies the service requested by the client. The DeviceNet Specification defines valid service codes.
Class ID	This number identifies the class to which the service is directed. The DeviceNet Specification defines valid class IDs.
Instance ID	This number identifies the instance to which the service is directed. If the instance ID is zero, the service is directed to the entire class. If the instance ID is one or greater, the service is directed to a specific instance within the class.
Service Data	Data bytes specific to the Service Code. The number and format of these data bytes is defined by the specification for the service.

Table A-3. Explicit Message Response

Field	Description
Service Code	This number indicates success or failure for execution of the service. If this number is the same as the Service Code of the request, the service executed successfully. If this number is 14 hex, the service failed to execute due to an error.
Service Data	If the service executed successfully, this field contains data bytes which are specific to the Service Code. The number and format of these data bytes are defined by the specification for the service. If the service failed to execute, the first byte of Service Data contains a General Error Code which describes the error, and the second byte contains an Additional Error Code which qualifies the error. The DeviceNet Specification defines valid values for the General Error Code and Additional Error Code.

The DeviceNet Specification defines a set of services supported in a common way by different devices. These common services include Reset, Save, Restore, Get Attribute Single, and Set Attribute Single.

The Get Attribute Single service obtains the value of a specific attribute within a device's object, and the Set Attribute Single service sets the value of an attribute. These Get and Set services are the most commonly used explicit messaging services. Since these two services are used often, NI-DNET provides functions for these services: `ncGetDnetAttribute` and `ncSetDnetAttribute`.

Other services defined by DeviceNet are used less often. For these services, NI-DNET provides general purpose functions to send an explicit message request (`ncWriteDnetExplMsg`) and receive an explicit message response (`ncReadDnetExplMsg`). These NI-DNET functions use parameters which are similar to the explicit message request/response listed above. For more information on DeviceNet common services other than Get/Set Attribute Single, refer to the DeviceNet Specification.

I/O Connections

In addition to explicit messaging connections, DeviceNet devices provide another type of Connection Object called an I/O connection. I/O connections provide a communication path for the exchange of physical input/output (sensor/actuator) data as well as other control-oriented data. I/O connections are useful for transferring data at regular intervals.

Since many DeviceNet devices do not begin their normal operation until an I/O connection is established, explicit messaging is often used for configuration and initialization. For example, for a device with an analog input, the I/O connection is normally used to read the analog input measurement, and explicit messages are used for configuration such as setting the measurement range and units (such as –10 to +10 V versus 4 to 20 mA).

The DeviceNet Specification defines two types of I/O connections: master/slave and peer-to-peer. In master/slave I/O connections, a master device uses an I/O connection to communicate with one or more slave devices, and those slave devices can only communicate with the master and not one another. In peer-to-peer I/O connections, each device on the network can communicate as a peer, and communication paths between peer devices are established as needed. The NI-DNET software currently supports only master/slave I/O connections because the procedure used to

establish these I/O connections is more well defined. For this reason, almost all existing DeviceNet devices only implement master/slave I/O connections.

The DeviceNet Specification defines four types of master/slave I/O connections: polled, bit strobed, change-of-state (COS), and cyclic. A slave device can support at most one polled, one strobed, and one COS or cyclic connection (COS and cyclic connections cannot be used simultaneously).

Polled I/O

The polled I/O connection uses a request/response scheme for each device. The master sends a poll command (request) message to the slave device with any amount of output data. The slave then sends a poll response message back to the master with any amount of input data. The poll command/response messages are handled individually for each slave which supports polled I/O connections. Polled I/O is typically used for devices which provide both input and output data, such as position controllers and modular I/O devices.

Figure A-3 shows an example of four polled slave devices.

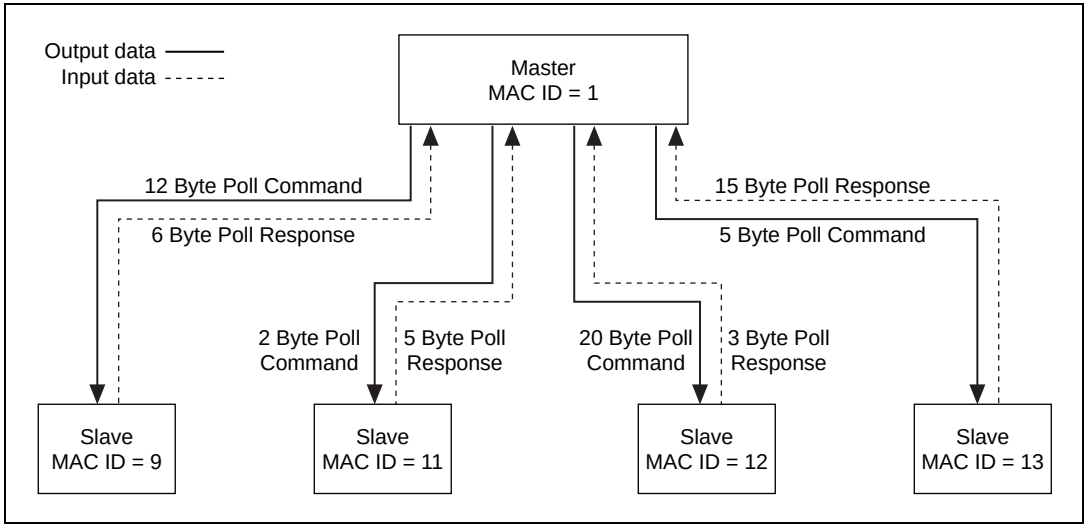


Figure A-3. Polled I/O Example

Bit Strobed I/O

The (bit) strobed I/O connection is designed to move small amounts of input data from the slave to its master. Strobed I/O is typically used for simple sensors, such as photoelectric sensors and limit switches. Strobed I/O is also called bit strobed I/O since the master sends a 64-bit (8-byte) message containing a single bit of output data for each strobed slave. This strobe command (request) message is received by all slave devices simultaneously and can be used to trigger simultaneous measurements (such as to take multiple photoelectric readings simultaneously).

When a strobed slave receives the strobe command, it uses the output data bit that corresponds to its own MAC ID (for example, the slave with MAC ID 5 uses bit 5). Regardless of the value of its output bit, each strobed slave responds to the command message by sending an individual strobe message back to the master. The slave's strobe response contains from 0 to 8 bytes of input data.

Figure A-4 shows an example of four strobed slave devices.

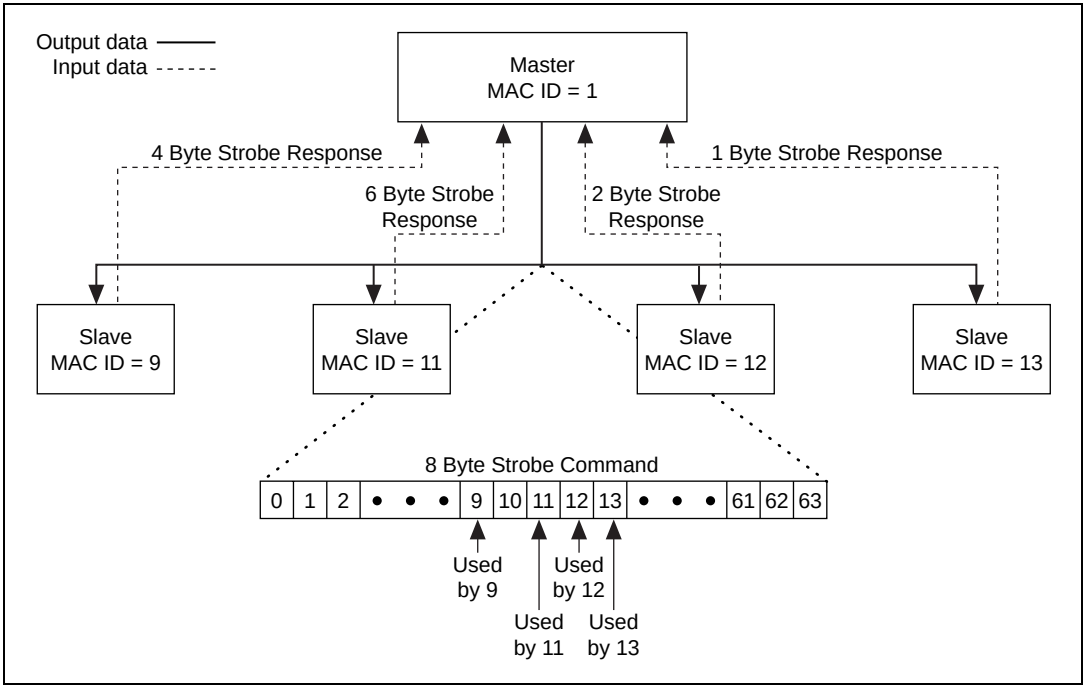


Figure A-4. Strobed I/O Example

Change-of-State and Cyclic I/O

The change-of-state (COS) and cyclic I/O connections both use the same underlying communication mechanisms. Both transmit data at a fixed interval called the expected packet rate (EPR). Since COS and cyclic I/O connections use the same messaging on the DeviceNet network, they are often referred to as a single I/O connection called COS/cyclic I/O.

The cyclic I/O connection enables a slave device to send input data to its master at the configured EPR interval. You normally configure the EPR to be consistent with the rate at which the device measures its physical input sensors. For example, if a temperature sensor can take a measurement at most once every 500 ms, you would configure the cyclic I/O connection's EPR as 500 ms. Cyclic I/O can be configured to send output data from master to slave, but this configuration is seldom used since it is essentially the same as polled I/O. Cyclic I/O messages can contain any amount of data.

The COS I/O connection enables a slave device to send input data to its master when a change is detected on its physical inputs. In addition to sending input data when a change is detected, the COS slave also sends its input data at a slower EPR interval that lets the master know it is still functioning. COS I/O is typically used for devices with physical inputs that can change frequently but can have the same input value for a long time. For example, if a pushbutton device supports COS I/O, you might configure its EPR as 3 seconds since the device sends a message immediately if a button is pressed. COS I/O can be configured to send output data from master to slave. Although master-to-slave COS output is seldom used, it can be useful for things like front-panel pushbuttons which are sent to a slave's discrete outputs (such as LEDs and simple motors). COS I/O messages can contain any amount of data.

When using COS/cyclic I/O connections, you can configure the device that receives data to send an acknowledgment so that the transmitting device can verify that the data was received successfully. For example, if you configure slave-to-master COS I/O (input length nonzero), the master sends an acknowledgment to the slave each time it receives an input message. Since the acknowledgment message is used for verification only, it does not contain data. If this verification can be handled using other means (such as using strobed I/O to verify device status), the acknowledgment message can be suppressed. For information on how to suppress COS/cyclic acknowledgments using NI-DNET, refer to the description of the I/O Object in the *NI-DNET Programmer Reference Manual*.

Since COS and cyclic I/O use the same messages on the DeviceNet network, they cannot be used simultaneously for a given slave device. Also, polled I/O uses the same messages on the DeviceNet network as master-to-slave output messages of COS/cyclic I/O. This means that a slave device can use slave-to-master COS/cyclic I/O simultaneously with polled I/O, but not master-to-slave COS/cyclic I/O.

Figure A-5 shows an example of four COS/cyclic I/O connections.

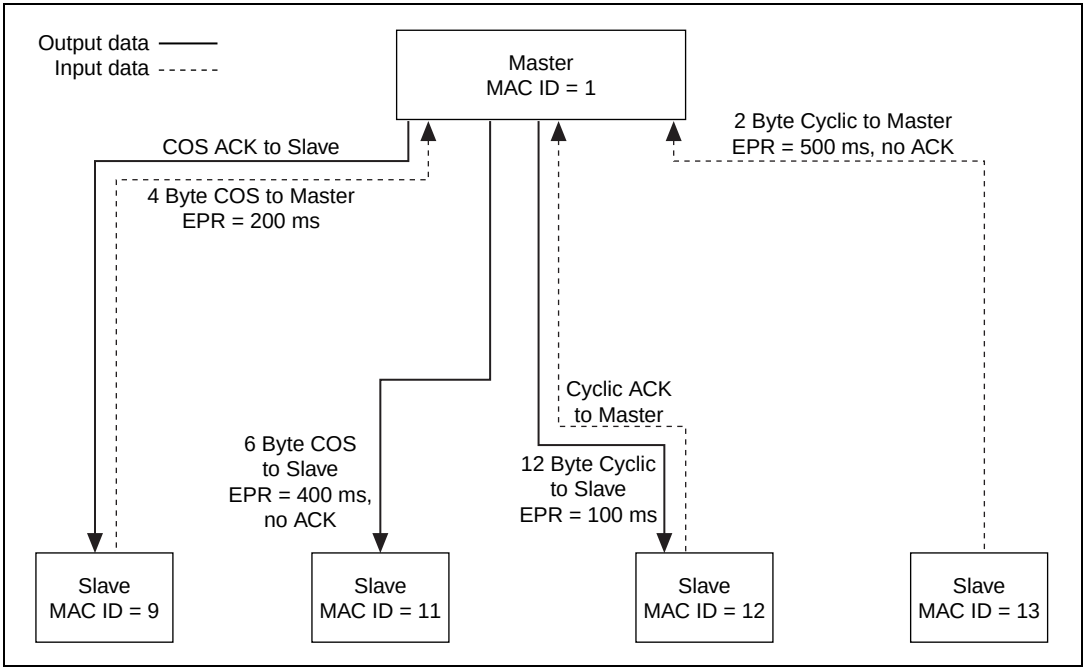


Figure A-5. COS/Cyclic I/O Example

Assembly Objects

One of the more important objects in the DeviceNet Specification is the Assembly Object. There are two types of Assembly Object: input assemblies and output assemblies. Assembly objects act like a switchboard, routing incoming and outgoing data to its proper location within the device. Output assemblies receive an output message from an I/O connection and distribute its contents to multiple attributes within the slave. Input assemblies gather multiple attributes within the slave for transmission on an I/O connection.

Figure A-6 shows the operation of input and output assemblies.

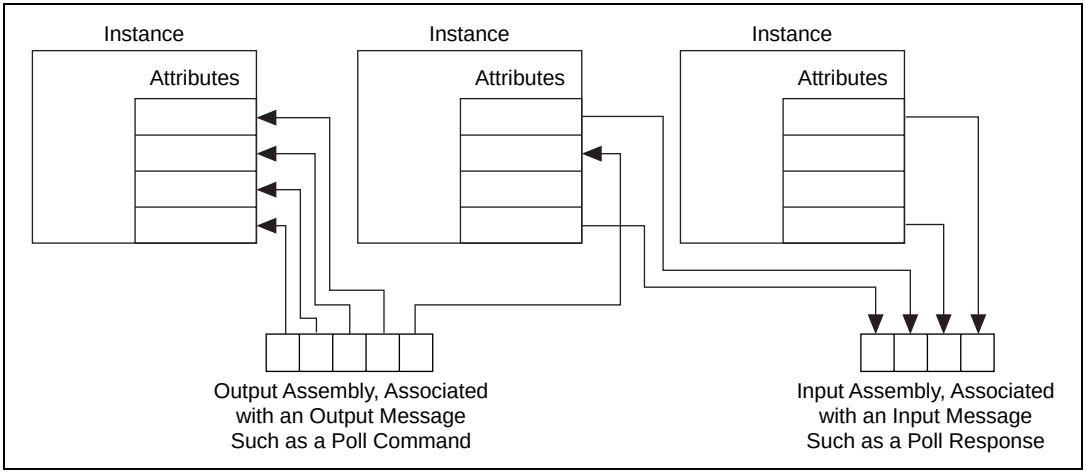


Figure A-6. Input and Output Assemblies

As a more specific example, consider a DeviceNet photoelectric sensor (photoeye) or a limit switch. These devices contain a single instance of a class called the Presence Sensing object. This instance has attributes for the Output Signal (on/off) and Diagnostic Status (good/fault). These two attributes are often routed through a single input assembly consisting of a single byte.

Figure A-7 shows an example of a Presence Sensing instance and its input assembly.

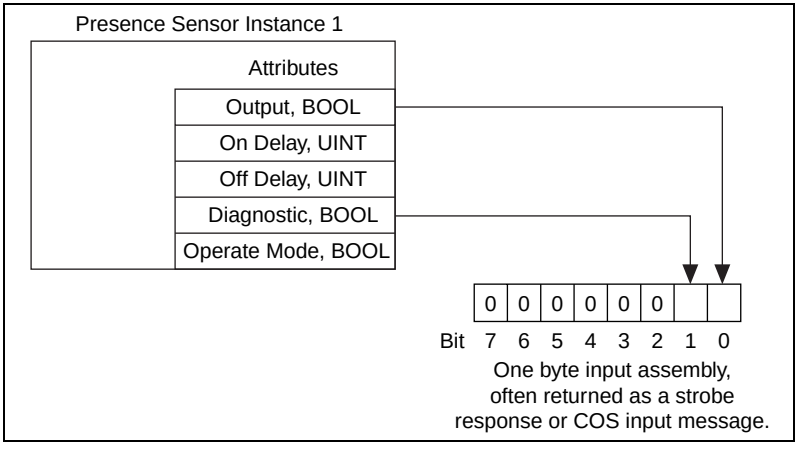


Figure A-7. Input Assembly for Photoeye or Limit Switch

As you can see, to use the data bytes contained in I/O messages, it is important to know the format of a device's internal input and output assemblies. For more information on I/O assemblies, refer to the section *Using I/O Data in Your Application* in Chapter 3, *NI-DNET Programming Techniques*.

Device Profiles

To provide interoperability for devices of the same type, the DeviceNet Specification defines various device profiles. The goal behind device profiles is that for a given type of device, such as a photoelectric sensor, it should be relatively straightforward to replace a sensor from one vendor with a sensor from another vendor.

All devices which conform to a given profile must do the following:

- Exhibit the same behavior
- Use the same object model (certain instances are required)
- Contain the same input and output assemblies
- Contain the same set of configurable attributes

In addition to required features, most device profiles define a variety of optional features. When an optional feature is supported by a vendor, it must be supported as defined by the DeviceNet Specification. Device profiles also allow for vendor-specific features.

The DeviceNet Specification provides device profiles for such devices as photoelectric sensors, limit switches, motor starters, position controllers, and mass-flow controllers.

Open DeviceNet Vendors Association (ODVA)

This chapter provides only a short summary of DeviceNet. For additional information, such as a list of DeviceNet products and how to purchase the DeviceNet Specification, refer to the ODVA web site at `www.odva.org`.

B

Uninstalling the NI-DNET Software

This appendix describes how to uninstall the NI-DNET software.

Before you uninstall the NI-DNET software, you should remove all interfaces that use NI-DNET from your computer.

The uninstall program removes only items that the installation program installed. If you add anything to a directory that was created by the installation program, the uninstall program does not delete that directory because the directory is not empty after the uninstallation. You must remove any remaining components yourself.

If you want to reinstall the software, refer to your getting started manual.

Windows 98/95/NT 4.0

Complete the following steps to remove the NI-DNET software from Windows 98/95/NT 4.0:

Note Windows NT users must first log in as `Administrator` or as a user with Administrator privileges.

1. Select the **Add/Remove Programs** icon under **Start»Settings» Control Panel**. A dialog box appears that lists the software available for removal.
2. Select the NI-DNET software you want to remove, and click on the **Add/Remove** button. The uninstall program runs and removes all folders, utilities, device drivers, DLLs, and registry entries associated with the NI-DNET software.

Windows NT 3.51

Complete the following steps to remove the NI-DNET software from Windows NT 3.51:

Note Windows NT users must first log in as `Administrator` or as a user with Administrator privileges.

1. Open the program group **National Instruments DNET**.
2. Double-click on the **Uninstall** icon.
3. The uninstall program runs and removes all folders, utilities, device drivers, DLLs, and registry entries associated with the NI-DNET software.

C

Technical Support Resources

Web Support

National Instruments Web support is your first stop for help in solving installation, configuration, and application problems and questions. Online problem-solving and diagnostic resources include frequently asked questions, knowledge bases, product-specific troubleshooting wizards, manuals, drivers, software updates, and more. Web support is available through the Technical Support section of `ni.com`

NI Developer Zone

The NI Developer Zone at `ni.com/zone` is the essential resource for building measurement and automation systems. At the NI Developer Zone, you can easily access the latest example programs, system configurators, tutorials, technical news, as well as a community of developers ready to share their own techniques.

Customer Education

National Instruments provides a number of alternatives to satisfy your training needs, from self-paced tutorials, videos, and interactive CDs to instructor-led hands-on courses at locations around the world. Visit the Customer Education section of `ni.com` for online course schedules, syllabi, training centers, and class registration.

System Integration

If you have time constraints, limited in-house technical resources, or other dilemmas, you may prefer to employ consulting or system integration services. You can rely on the expertise available through our worldwide network of Alliance Program members. To find out more about our Alliance system integration solutions, visit the System Integration section of `ni.com`

Worldwide Support

National Instruments has offices located around the world to help address your support needs. You can access our branch office Web sites from the Worldwide Offices section of `ni.com`. Branch office Web sites provide up-to-date contact information, support phone numbers, e-mail addresses, and current events.

If you have searched the technical support resources on our Web site and still cannot find the answers you need, contact your local office or National Instruments corporate. Phone numbers for our worldwide offices are listed at the front of this manual.

Glossary

Prefix	Meanings	Value
m-	milli-	10^{-3}
k-	kilo-	10^{3}

A

A	amperes
AC	alternating current
actuator	A device that uses electrical, mechanical, or other signals to change the value of an external, real-world variable. In the context of device networks, actuators are devices that receive their primary data value from over the network; examples include valves and motor starters. Also known as *final control element.*
ANSI	American National Standards Institute
Application Programming Interface (API)	A collection of functions used by a user application to access hardware. Within NI-DNET, you use API functions to make calls into the NI-DNET driver.
ASCII	American Standard Code for Information Exchange
Assembly Object	Objects in DeviceNet devices which route I/O message contents to/from individual attributes in the device.
attribute	The externally visible qualities of an object; for example, an instance square of class Geometric Shapes could have the attributes length of sides and color, with the values 4 in. and blue.
automatic polling	A polled I/O mode in which NI-DNET automatically determines an appropriate *scanned polling* rate for your DeviceNet system.

B

b	Bits
background polling	A polled I/O communication scheme in which all polled slaves are grouped into two different communication rates: a foreground rate and a slower background rate.
bit strobed I/O	Master/slave I/O connection in which the master broadcasts a single strobe command to all strobed slaves then receives a strobe response from each strobed salve.

C

CAN	Controller Area Network
change-of-state I/O	Master/slave I/O connection which is similar to cyclic I/O but data can be sent when a change in the data is detected.
class	A classification of things with similar qualities.
client	In explicit messaging connections, the client is the device requesting execution of the service.
common services	Services defined by the DeviceNet specification such that they are largely interoperable.
connection	An association between two or more devices on a network that describes when and how data is transferred.
controller	A device that receives data from sensors and sends data to actuators to hold one or more external, real-world variables at a certain level or condition. A thermostat is a simple example of a controller.
COS I/O	*See* change-of-state I/O.
cyclic I/O	Master/slave I/O connection in which the slave (or master) sends data at a fixed interval.

D

DC	direct current
device	A physical assembly, linked to a communication line (cable), capable of communicating across the network according to a protocol specification.
device network	Multi-drop digital communication network for sensors, actuators, and controllers.
device profiles	DeviceNet specifications which provide interoperability for devices of the same type.
direct entry	Microsoft Win 32 functions used to directly access the functions of a Dynamic Link Library (DLL).
DLL	Dynamic Link Library
driver attributes	Attributes of the NI-DNET driver software.

E

EDS	Electronic Data Sheet. Text file that describes DeviceNet device features electronically.
expected packet rate	The rate (in milliseconds) at which a DeviceNet connection is expected to transfer its data.
Explicit messaging connection	General-purpose connection used for executing services on a particular object in a DeviceNet device.

F

FCC	Federal Communications Commission
ft	feet
FTP	File transfer protocol

H

hex	Hexadecimal
Hz	Hertz

I

in.	inches
individual polling	A polled I/O communication scheme in which each polled slave communicates at its own individual rate.
instance	A specific instance of a given class. For example, a blue square of 4 inches per side would be one instance of the class Squares.
I/O connection	Connection used for exchange of physical input/output (sensor/activator) data, as well as other control-oriented data.
ISO	International Standards Organization

K

KB	Kilobytes of memory

L

LabVIEW	Laboratory Virtual Instrument Engineering Workbench
LED	light-emitting diode
local	Within NI-DNET, anything that exists on the same host (personal computer) as the NI-DNET driver.

M

m	meter
MAC ID	Media access control layer identifier. In DeviceNet, a device's MAC ID represents its address on the DeviceNet network.

master/slave	DeviceNet communication scheme in which a master device allocates connections to one or more slave devices, and those slave devices can only communicate with the master and not one another.
MB	Megabytes of memory
member	Individual data value within a DeviceNet I/O Assembly.
method	*See* service.
multi-drop	A physical connection in which multiple devices communicate with one another along a single cable.

N

network interface	A device's physical connection onto a network.
network management utility	Utility used to manage configuration of DeviceNet devices.
network who	A search of a DeviceNet network to determine information about its devices.
NI-DNET driver	Device driver and/or firmware that implement all the specifics of a National Instruments DeviceNet interface.
notification	Within NI-DNET, an operating system mechanism that the NI-DNET driver uses to communicate events to your application. You can think of a notification of as an API function, but in the opposite direction.

O

object	*See* instance.
object-oriented	A software design methodology in which classes, instances, attributes, and methods are used to hide all of the details of a software entity that do not contribute to its essential characteristics.
ODVA	Open DeviceNet Vendor's Association

P

PC	personal computer
peer-to-peer	DeviceNet communication scheme in which each device communicates as a peer and connections are established among devices as needed.
PLC	Programmable Logic Controller
polled I/O	Master/slave I/O connection in which the master sends a poll command to a slave, then receives a poll response from that slave.
protocol	A formal set of conventions or rules for the exchange of information among devices of a given network.

R

RAM	Random-access memory
remote	Within NI-DNET, anything that exists in another device of the device network (not on the same host as the NI-DNET driver).
resource	Hardware settings used by National Instruments DeviceNet hardware, including an interrupt request level (IRQ) and an 8 KB physical memory range (such as `D0000` to `D1FFF` hex).

S

s	seconds
scanned polling	A polled I/O communication scheme in which all poll commands are sent out at the same rate, in quick succession.
sensor	A device that measures electrical, mechanical, or other signals from an external, real-world variable; in the context of device networks, sensors are devices that send their primary data value onto the network; examples include temperature sensors and presence sensors. Also known as transmitter.
server	In explicit messaging connections, the server is the device to which the service is directed.

service	An action performed on an instance to affect its behavior; the externally visible code of an object. Within NI-DNET, you use NI-DNET functions to execute services for objects. Also known as method and operation.
strobed I/O	*See* bit strobed I/O.

V

V	volts
VI	Virtual Instrument
VxD	Virtual device driver

Index

A

B

C

D

E

F

G

H

I

L

M

N

T

W