NI-DAQ®
Software Reference Manual for Macintosh

Version 4.8

Data Acquisition Software for the Macintosh

February 1996 Edition

Part Number 371345A-01

National Instruments Corporate Headquarters
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
Technical support fax: (512) 418-1111

Branch Offices:
Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Canada (Ontario) 519 622 9310,
Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 90 527 2321, France 1 48 14 24 24,
Germany 089 741 31 30, Hong Kong 2645 3186, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456,
Mexico 95 800 010 0793, Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085,
Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, U.K. 01635 523545

Limited 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 manual 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 MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

Chapter 7
SCXI Functions 7-1

Figures

Tables

About This Manual

The *NI-DAQ Software Reference Manual for Macintosh* is for users of the NI-DAQ software for Macintosh version 4.8, which contains low-level interfaces for developing data acquisition applications with the National Instruments data acquisition devices. This manual describes how to configure the data acquisition interface boards for use with this software, how to install the NI-DAQ software for Macintosh, and how to communicate with the device drivers using C, Pascal, or FutureBASIC.

Assumption of Previous Knowledge

The material in this manual is for users who are familiar with Macintosh computers.

Organization of This Manual

The *NI-DAQ Software Reference Manual for Macintosh* is organized as follows:

- Chapter 1, *Getting Started*, will help you get started using NI-DAQ for Macintosh to build your data acquisition application for National Instruments DAQ hardware.
- Chapter 2, *Board-Specific Functions*, describes the functions for configuring and calibrating the boards.
- Chapter 3, *Analog Input Functions*, describes the functions for single A/D conversions.
- Chapter 4, *Analog Output Functions*, describes the functions for single D/A conversions.
- Chapter 5, *Digital I/O Functions,* describes the functions used to read from and write to digital ports, which can be addressed as a single entity or as individual digital lines.
- Chapter 6, *Data Acquisition Functions*, explains the functions used for performing data acquisition operations. Single-channel acquisition, multiple-channel scan acquisition, interval scanning, pretrigger mode, posttrigger mode, double-buffered mode, and AMUX-64T multiplexer mode are all documented.
- Chapter 7, *SCXI Functions*, describes functions used to configure and communicate with SCXI modules and chassis.
- Chapter 8, *Counter/Timer Functions*, describes the functions that perform timing I/O and counter operations such as pulse generation, frequency generation, and event counting.
- Chapter 9, *RTSI Bus Trigger Functions*, describes the functions used to connect and disconnect signals over the RTSI bus trigger lines for the NB Series boards.
- Chapter 10, *Waveform Generation Functions*, describes the functions for generating analog waveform signals at the analog output channels for the NB Series boards.
- Chapter 11, *NI-DAQ for Macintosh Examples*, describes the examples included in the NI-DAQ for Macintosh software. These examples show you how you can use various NI-DAQ for Macintosh functions in actual applications.
- Appendix A, *NI-DAQ for Macintosh Function and Board Compatibility*, contains a table of the National Instruments boards that work with NI-DAQ for Macintosh and the functions that work with each board, as well as a table of National Instruments SCXI chassis and modules that work with NI-DAQ for Macintosh and the functions that work with each element of SCXI hardware.

- Appendix B, *Error Codes*, lists the error codes NI-DAQ for Macintosh returns, including the error number, name, and description. Each function returns an error code that indicates whether the function was performed successfully.
- Appendix C, *Using an External Multiplexer*, contains information on using the AMUX-64T.
- Appendix D, *Transducer Conversion Routines*, describes the transducer conversion routines included in NI-DAQ for Macintosh. You can use these routines to convert analog input voltages read from thermocouples, RTDs, and strain gauges into units of temperature or strain.
- Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, lists the valid channel and gain settings for DAQ boards, describes how NI-DAQ calculates voltage, and describes the measurement of offset and gain measurement.
- Appendix F, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics used in this manual, including the page where each one can be found.

Each chapter is introduced with a list of the National Instruments boards that you can use for the operations under discussion and a brief description of individual board features. This introduction is followed by a list of the functions that perform these operations. The remainder of each chapter contains a detailed description of each function.

Conventions Used in This Manual

The following conventions are used in this manual.

bold	Denotes menus, menu items, options, parameters, or data types.
bold italic	Denotes a note, caution, or warning.
DIO boards	Refers to the DAQCard-DIO-24, NB-DIO-24, NB-DIO-96, NB-DIO-32F, and PCI-DIO-96.
DMA board	Refers to the NB-DMA-8-G or NB-DMA2800.
E Series device	Refers to the PCI-MIO-16XE-50.
italic	Denotes emphasis, a cross reference, or an introduction to a key concept.
Lab and 1200 boards	Refers to the Lab-NB, Lab-LC, PCI-1200, and DAQCard 1200.
MIO boards	Refers to the NB-MIO-16, NB-MIO-16X, and PCI-MIO-16XE-50.
`monospace`	Text in this font denotes text or characters that are to be literally input 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, variables, filenames, and extensions, and for statements and comments taken from program code.
`italic monospace`	Italic text in this font denotes that you must supply the appropriate words or values in the place of these items.

< >	Angle brackets enclose the name of a key on the keyboard–for example, <enter>.
<enter>	Key names are lowercase.
Macintosh	Macintosh refers to all Macintosh II, Macintosh Quadra, Macintosh Centris, Macintosh LC, and Power Macintosh computers.
NI-DAQ	NI-DAQ refers to the NI-DAQ software for Macintosh.

Abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms are listed in the *Glossary*.

About the National Instruments Documentation Set

The *NI-DAQ Software Reference Manual for Macintosh* is one piece of the documentation set for your data acquisition system. You could have any of several types of manuals, depending on the hardware and software in your system. Use the manuals you have as follows:

- *Getting Started with SCXI*—If you are using SCXI, this is the first manual you should read. It gives an overview of the SCXI system and contains the most commonly needed information for the modules, chassis, and software.
- Your SCXI user manuals—If you are using SCXI, read these manuals for detailed information about signal connections and module configuration. They also explain in greater detail how the module works and contain application hints.
- Your DAQ hardware user manuals—These manuals have detailed information about the DAQ hardware that plugs into your computer. Use these manuals for hardware installation and configuration instructions, specification information about your DAQ hardware, and application hints.
- Software manuals—Examples of software manuals you may have are the LabVIEW manual sets and the NI-DAQ manual. After you set up your hardware system, use either the application software (LabVIEW) manuals or the NI-DAQ manual to help you write your application. If you have a large and complicated system, it is worthwhile to look through the software manuals before you configure your hardware.
- Accessory installation guides or manuals—If you are using accessory products, read the terminal block and cable assembly installation guides or accessory board user manuals. These are the terminal block and cable assembly installation guides. They explain how to physically connect the relevant pieces of the system. Consult these guides when you are making your connections.
- SCXI chassis manuals—If you are using SCXI, read these manuals for maintenance information on the chassis and installation instructions.

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix F, *Customer Communication*, at the end of this manual.

Chapter 1
Getting Started

This chapter will help you get started using NI-DAQ for Macintosh to build your data acquisition application for National Instruments DAQ hardware. NI-DAQ for Macintosh contains programming language interfaces for the following environments:

- MPW C/C++ for Macintosh
- MPW C/C++ for Power Macintosh
- THINK C/Symantec C++ for Macintosh
- Symantec C/C++ for Power Macintosh
- Metrowerks C/C++ for Macintosh
- Metrowerks C/C++ for Power Macintosh
- MPW Pascal for Macintosh
- THINK Pascal for Macintosh
- Metrowerks Pascal for Macintosh
- Metrowerks Pascal for Power Macintosh
- Zedcor FutureBASIC for Macintosh

Figure 1-1 shows the steps to install your hardware and software, configure your hardware, and begin using NI-DAQ in your application programs.

If you will be accessing the NI-DAQ device drivers through LabVIEW, you should read the *NI-DAQ Installation for LabVIEW* section, then use your *LabVIEW Data Acquisition VI Reference Manual* to help you get started using the data acquisition VIs in LabVIEW.

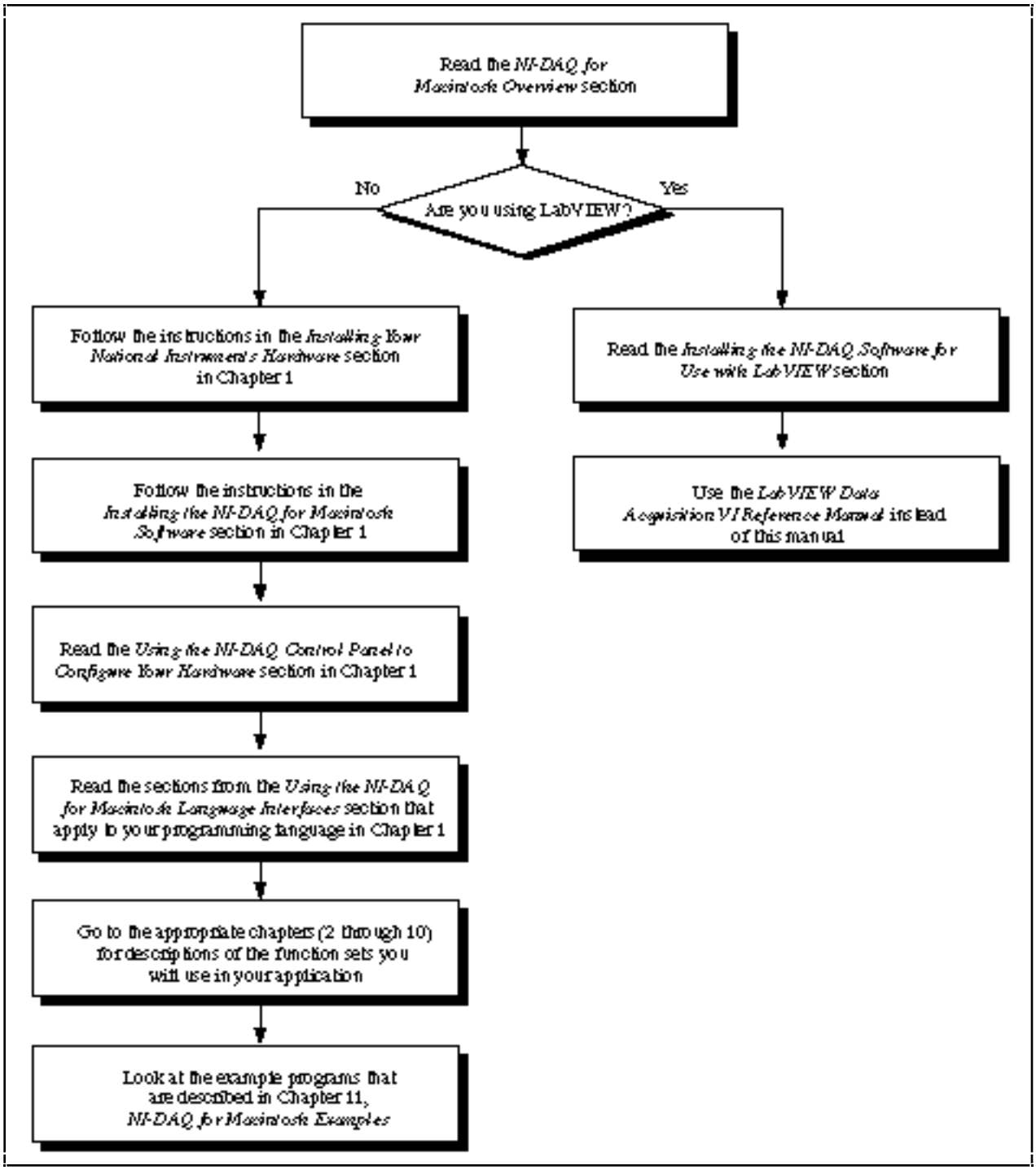

Figure 1-1. Steps to Begin Using NI-DAQ

NI-DAQ for Macintosh Overview

NI-DAQ for Macintosh Hardware Compatibility

Table 1-1. NI-DAQ for Macintosh Hardware Compatibility

Plug-in Boards	External Devices	SCXI
DAQCard-500	AMUX-64T	SCXI-1000
DAQCard-700	SC-2040	SCXI-1001
DAQCard-1200	SC-2042	SCXI-1100
DAQCard-DIO-24	SC-2043	SCXI-1102
DAQCard-AO-2DC	SC-2070	SCXI-1120
Lab-LC	SC-2071	SCXI-1121
Lab-NB		SCXI-1122
NB-A2000		SCXI-1124
NB-A2100		SCXI-1140
NB-A2150		SCXI-1141
NB-AO-6		SCXI-1160
NB-DIO-24		SCXI-1161
NB-DIO-32F		SCXI-1162
NB-DIO-96		SCXI-1162HV
NB-DMA-8-G		SCXI-1163
NB-DMA2800		SCXI-1163R
NB-MIO-16		
NB-MIO-16X		
NB-PRL		
NB-TIO-10		
PCI-1200		
PCI-DIO-96		
PCI-MIO-16XE-50		

NI-DAQ for Macintosh also works with all Second Wave Expansion Chassis that support NuBus-to-NuBus or PCI-to-NuBus conversion. Furthermore, NI-DAQ for Macintosh supports the Newer Technology NuBus-to-PC Card or PCI-to-PC Card expansion modules.

NI-DAQ for Macintosh Clones

In order for NI-DAQ to identify the NuBus devices installed in your computer, the NI-DAQ NuBus interface requires information about the computer. By default, NI-DAQ may not recognize the presence of the NuBus on certain Macintosh clone machines. However, you can configure NI-DAQ to request the needed information from your Macintosh clone. To activate the configuration sequence, install NI-DAQ, disconnect *all* expansion chassis, restart your computer, and press and hold <command> <+> or <command> <shift> <=> until the boot sequence is complete. Once the boot sequence is complete, you will see your NuBus devices in the control panel.

NI-DAQ for Macintosh Function Summary

The NI-DAQ for Macintosh software contains the following groups of functions:

- Board-Specific
- Analog Input
- Analog Output
- Digital I/O
- Data Acquisition
- SCXI
- Counter/Timer
- RTSI Bus Trigger
- Waveform Generation

An overview of the individual functions within each group is at the beginning of chapters 2 through 10.

NI-DAQ for Macintosh can use only non-GPIB functions; that is, NI-DAQ for Macintosh does not work with the NB-GPIB board or the GPIB interface on the NB-DMA-8-G and NB-DMA2800 boards. The National Instruments NI-488 or LabVIEW software have GPIB functions; if you use GPIB functions, you can install the NI-488 software in addition to NI-DAQ for Macintosh. These two packages run cooperatively on Macintosh computers.

Installing the NI-DAQ Software for Use with LabVIEW

The LabVIEW installation program installs the NI-DAQ software for you. However, the NI-DAQ software that is included with your DAQ hardware may be a more recent version than the NI-DAQ software that LabVIEW installed.

To ensure that the correct NI-DAQ version is installed, follow these steps:

1. If you had a previous version of NI-DAQ or LabVIEW installed, and you had configuration information entered in the `NI-DAQ Config`, `NI-DAQ Utilities`, or `NI-DAQ` control panel, open that control panel and record your configuration information. You will have to enter that information again once you have installed the new software.

   Any new versions of NI-DAQ *after* version 4.8 will have the ability to read configuration information from the previous control panel; you will not have to enter the configuration information again after you have entered it in NI-DAQ version 4.8.

2. Install LabVIEW first. Follow the LabVIEW installation instructions that came with your LabVIEW package.

3. Insert the NI-DAQ for Macintosh disk 1 into your disk drive. There is one file on that disk–the `NI-DAQ Installer`. Open the `NI-DAQ Installer` by double-clicking on its icon.

4. Click on the **Read Me** button; it will display important late-breaking information that is not included in this manual. The **Read Me** information may also contain important installation instructions. If the instructions there differ from the instructions here, you should follow the instructions displayed by the **Read Me** button.

5. Drag the NI-DAQ icon to your startup drive to update NI-DAQ.

6. The `NI-DAQ Installer` will check the NI-DAQ version that was installed by LabVIEW. If LabVIEW installed a newer NI-DAQ version than the one included with this package, the `NI-DAQ Installer` will not install anything. Remove the NI-DAQ for Macintosh disk and restart your machine to load the NI-DAQ drivers if you have not already done so after the LabVIEW installation. Continue by reading the *LabVIEW for Macintosh Data Acquisition VI Reference Manual.* You no longer need this manual.

7. If the NI-DAQ version included with this package is newer than the one installed by LabVIEW, the `NI-DAQ Installer` will install the new NI-DAQ version and remove the old version.

   The `NI-DAQ Installer` will install two files:

   - The `NI-DAQ` file in the Control Panels folder contains all of the data acquisition and SCXI drivers that are required to run your DAQ hardware.

   - The `NI-DMA/DSP` file in the Extensions folder contains DMA and DSP drivers that are shared by NI-DAQ, NI-488, and NI-DSP.

   You should continue by reading your *LabVIEW for Macintosh Data Acquisition VI Reference Manual*. If that manual makes reference to the `NI-DAQ Utilities` control panel, use the `NI-DAQ` control panel instead. To open the `NI-DAQ` control panel, double-click on the `NI-DAQ` icon.

Installing Your National Instruments Hardware

1. Turn off your Macintosh computer.

2. Check the user manuals that came with your plug-in boards to determine if you need to change any jumper settings. Some DAQ boards have jumpers to set analog input polarity, input mode, analog output reference, and so on. Be sure to record any jumper settings you change so that you can enter the information correctly in the configuration utility later.

3. Insert your plug-in DAQ boards and your DMA board, if any, into your Macintosh computer.

4. If you have a DMA board (NB-DMA-8-G or NB-DMA2800) you must connect it to your other DAQ boards using a RTSI cable. NI-DAQ will detect the DMA board automatically and will attempt to use DMA for data transfers if possible. The DMA operation will not function properly if you do not connect your boards with the RTSI cable. If you are using a Second Wave expansion chassis and/or two or more DMA boards, you can use the NI-DAQ control panel to help you cable your RTSI buses properly. See the *Devices* section later in this chapter for more information.

   If your DAQ board does not support DMA (it does not have a RTSI connector) or if you do not have a DMA board in your system, NI-DAQ will use interrupts for data transfers instead of DMA.

5. If you have SCXI hardware, read the next section for SCXI installation instructions. If you do not have SCXI hardware, turn on your computer and go to the *Installing the NI-DAQ for Macintosh Software* section.

Installing Your SCXI Hardware

The *Getting Started with SCXI* manual that came with your SCXI hardware contains step-by-step detailed instructions for assembling your SCXI system, including module jumper settings, cable assemblies, and terminal blocks. The basic steps are as follows:

1. Check the jumpers on your modules. You should almost always leave the jumpers in their default positions. The *Getting Started with SCXI* manual has a section for each module type that lists the cases where you may want to change the jumper settings. The SCXI-1120, SCXI-1121, and SCXI-1140 modules have jumper-selectable gains for each channel.

2. Make sure the chassis power is turned off. Plug your modules in through the front of the chassis. You can put the modules in any slot; for simplicity start with slot 1 on the left side of the chassis and move right with additional modules. Be sure to screw the modules tightly into the chassis frame.

3. If you are using an SCXI-1180 feedthrough panel, you must install the SCXI-1180 in the slot immediately to the right of the module that you will cable to the DAQ board. Otherwise, the cable connectors may not fit together conveniently.

4. Plug the appropriate terminal blocks into the front of each module and screw them tightly into the chassis frame.

5. If you have more than one chassis, select a unique jumpered address for each additional chassis by using the jumpers directly behind the front panel of the chassis.

6. Connect the mounting bracket of the SCXI-134*x* cable assembly to the back of one of the modules and screw it into the chassis frame. Connect the other end of the cable to a DAQ board in your computer. In Multiplexed mode, you need to cable only one module to the DAQ board, and in most cases it does not matter which module. There are two special cases:

   - If you are using SCXI-1140 modules along with other types of modules, you need to cable one of the SCXI-1140 modules to a DAQ board.

   - If you are using analog input modules along with other types of modules, you need to cable one of the analog input modules to a DAQ board.

   Refer to the *Getting Started with SCXI* manual if you need more in-depth information on related topics, such as multichassis cabling.

7. Turn on your chassis power.

8. Turn on your computer.

Installing the NI-DAQ for Macintosh Software

Warning: ***Do not use the NI-DAQ for Macintosh 4.8 language interface and examples with earlier versions of NI-DAQ for Macintosh or NB LabDriver installed, and vice versa. To use applications created with earlier versions of NI-DAQ for Macintosh or NB LabDriver, make sure to replace the language interfaces from the earlier version with the NI-DAQ for Macintosh 4.8 language interfaces.***

Follow these steps to install your new NI-DAQ for Macintosh software:

1. If you had a previous version of NI-DAQ installed, and you had configuration information entered in the `NI-DAQ Config`, `NI-DAQ Utilities`, or `NI-DAQ` control panel, open that control panel and record your configuration information. You will have to enter that information again once you have installed the new software.

   Any new versions of NI-DAQ *after* version 4.8 will have the ability to read configuration information from the previous control panel; you will not have to enter the configuration information again after you have entered it in NI-DAQ version 4.8.

2. Insert the NI-DAQ for Macintosh disk 1 into your disk drive. There is one file on that disk–the `NI-DAQ Installer`. Open the `NI-DAQ Installer` by double-clicking on its icon.

3. Click on the **Read Me** button; it will display important late-breaking information that is not included in this manual. The **Read Me** information may also contain important installation instructions. If the instructions there differ from the instructions here, you should follow the instructions displayed by the **Read Me** button.

4. Drag the `NI-DAQ` icon to your startup drive to install or update NI-DAQ.

5. After the NI-DAQ driver installation is complete, you can install one or more language interfaces by clicking on the **Show Other Installations** button in the top left corner of the installer window, selecting one or more interface icons, and dragging them to the desired disk.

6. If you have not done so already, restart your computer to install your device drivers.

7. The `NI-DAQ Installer` will install two files:

   - The `NI-DAQ` file in the Control Panel folder contains all of the data acquisition and SCXI drivers that are required to run your DAQ hardware.

   - The `NI-DMA/DSP` file in the Extensions folder contains DMA and DSP drivers that are shared by NI-DAQ, NI-488, and NI-DSP.

Using the NI-DAQ Control Panel to Configure Your Hardware

Double-click on the `NI-DAQ` icon in the Control Panels folder. Upon opening, the `NI-DAQ` Control Panel displays a list of all the devices in your Macintosh.

Devices

Once you have restarted your computer and NI-DAQ has installed successfully, you can use the `NI-DAQ` control panel to view information associated with the devices in your computer, as shown in Figure 1-2.

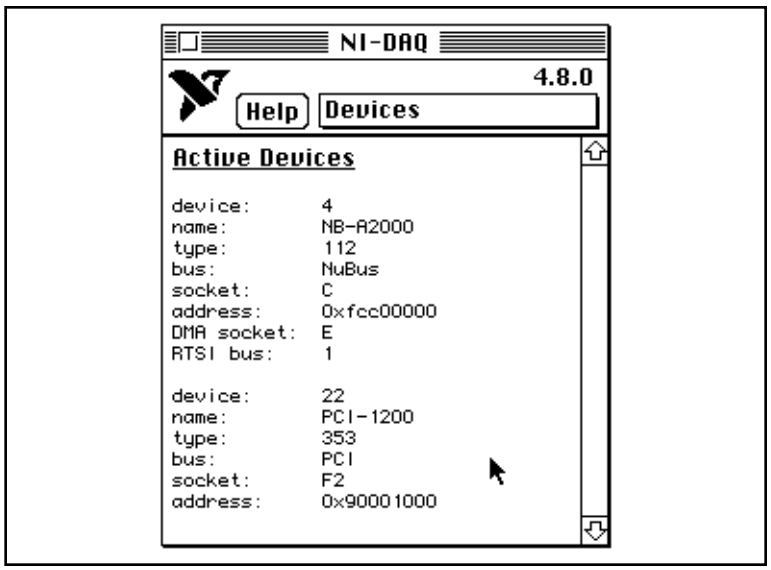

Figure 1-2. The NI-DAQ Control Panel

When the **Devices** portion of the control panel is active, you will see a document that provides the following information for each device:

- device—The logical device number associated with the device. Use this number in your function calls or in LabVIEW.

- name—The name of the device.

- type—The value returned by `Board_ID` or `Get_DAQ_Device_Info`.
- bus—The bus the device belongs to.
- socket—The name of the socket or slot in which the device is installed.
- address—The base address of the device. This field may not be accurate if the device is not a National Instruments device.
- DMA socket (NuBus only)—The location of the DMA device that will service this device.
- RTSI bus (NuBus only)—The number of the RTSI bus for this device. This value is not important unless you are using a Second Wave Expansion Chassis and/or two or more DMA devices. In either of these cases, cable together all devices with the same RTSI bus number, and make sure to use separate cables for each distinctly numbered bus.

Device Configuration

Select the **Device Configuration** option from the menu shown in Figure 1-3.

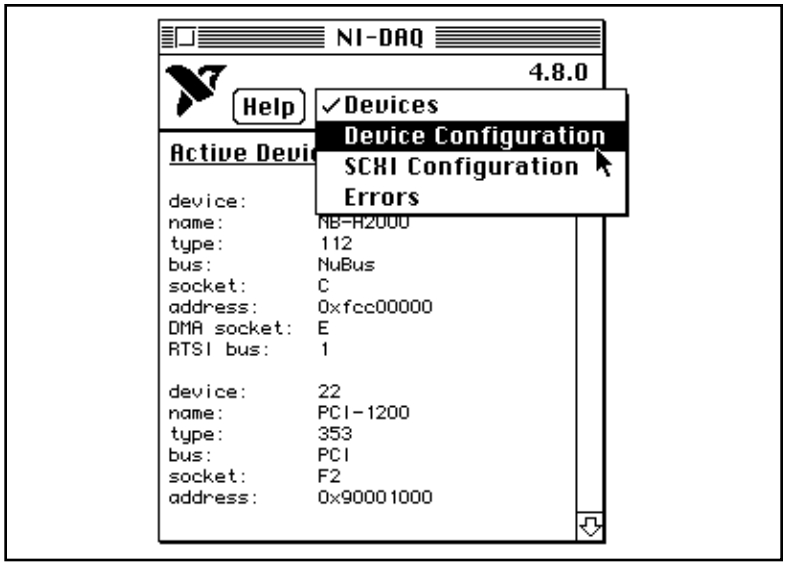

Figure 1-3. Selecting Device Configuration in the NI-DAQ Control Panel

From the **Device Configuration** window, you can enter any jumper settings you changed when you installed your board. For example, if you changed the jumper for analog input polarity on your NB-MIO-16 board, select the appropriate setting from the **Polarity** menu. You can also edit the default settings for any parameters shown that are software configurable. If you are using an accessory board with your DAQ board, choose the appropriate setting in the **Accessories** menu.

Click on the device name to display the I/O connector pinout for the device, as shown in Figure 1-4. Click in the display window to return to the control panel.

Use the **I/O Subsystem** menu to select which functional section of the board you are editing (ADC, DAC, DIO port, and so on).

Figure 1-4 shows the `NI-DAQ` device configuration window.

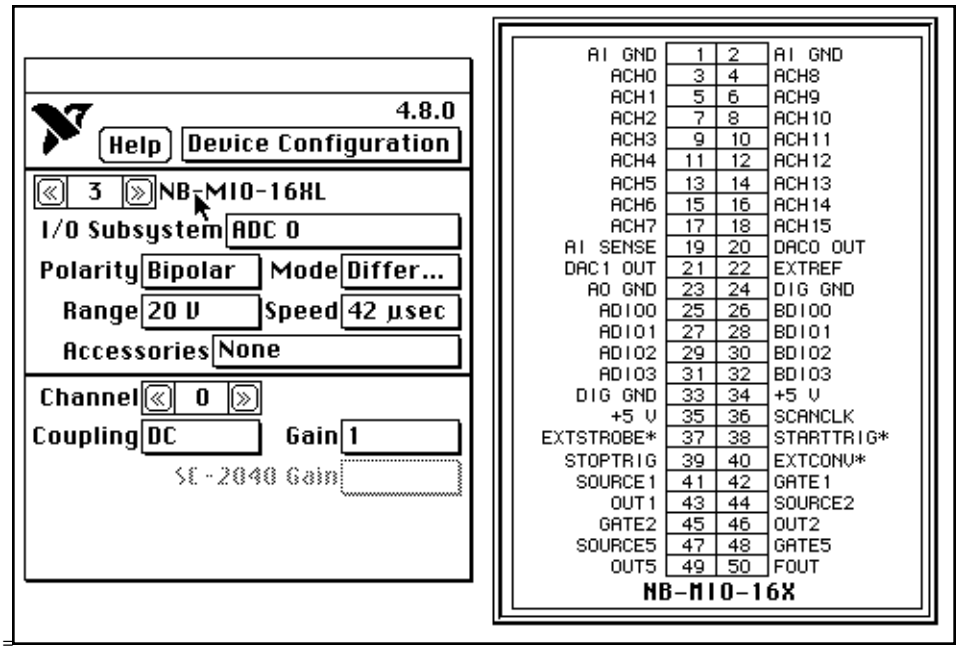

Figure 1-4. The NI-DAQ Device Configuration Window

Note: ***NI-DAQ contains function calls that you can use to change any of these parameters programmatically from your application.***

If you have SCXI hardware, follow the instructions in the next section to configure your SCXI system with the `NI-DAQ` Control Panel.

If you do not have SCXI hardware, continue by reading the appropriate sections in the *Using the NI-DAQ for Macintosh Language Interfaces* section later in this chapter.

SCXI Configuration

To use SCXI with NI-DAQ, you must enter the configuration for each SCXI chassis using the `NI-DAQ` Control Panel. Select the **SCXI Configuration** window from the menu shown in Figure 1-5.

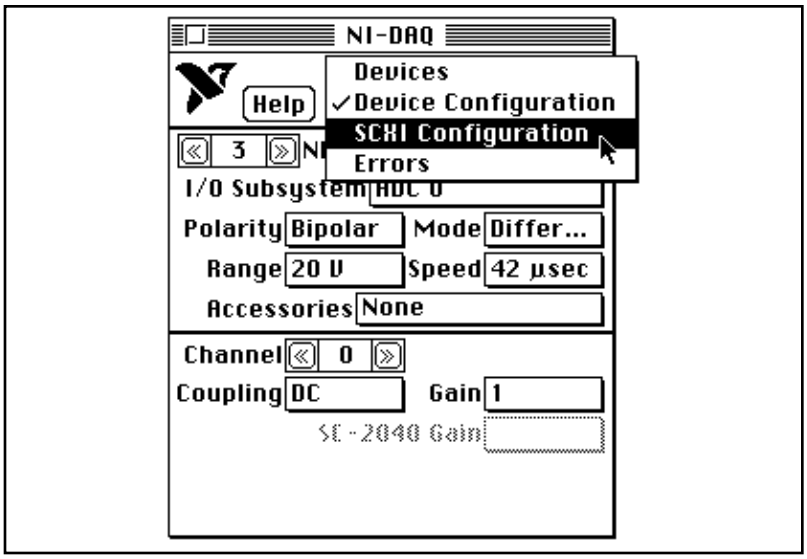

Figure 1-5. Selecting the NI-DAQ SCXI Configuration Window

1. Leave the **Chassis** set to one. You will use this number to access the SCXI chassis from your application. If you have multiple chassis, advance the **Chassis** to configure the next chassis after you finish configuring the first chassis.

2. Select the appropriate chassis type for your chassis; this enables the remaining fields on the panel.

3. If you have additional chassis, you need to select a unique hardware-jumpered address for each chassis and enter it in the **Address** field. If you have only one chassis, leave this field and the address jumpers on your chassis set to zero.

4. Leave the **Method** set to **Serial**, which means that NI-DAQ communicates with the chassis serially using a DIO port of the plug-in DAQ board. The **Path** automatically sets itself to the device number of the appropriate DAQ board when you enter the **Cabled Device** information in step 5b.

5. Enter the configuration for each slot in the chassis. The fields in the bottom two sections of the window reflect the settings for the selected **Module** number. For each SCXI module you install, you must set the following fields:

   a. **Module Type**–Select the correct module type for the module that is installed in the current slot. If the current slot has no module in it, leave this field set to **None** and advance the **Module** number to the next slot.

   b. **Cabled Device**–If the module in the current slot is *directly cabled* to a DAQ board in your computer, set this field to the device number of that DAQ board. Leave the **Cabled Device** field at **None** if the module in the current slot is not directly cabled to a DAQ board. If you are operating your modules in Multiplexed mode, you need to cable only one module in each chassis to your plug-in DAQ board. If you are not using Multiplexed mode, refer to the operating modes discussion in Chapter 7, *SCXI Functions*, for instructions about module cabling and the **Cabled Device** field.

   c. **Operating Mode**–Multiplexed mode is the default operating mode—it is recommended for almost all SCXI applications. The operating modes available for each SCXI module type are discussed in Chapter 7, *SCXI Functions*.

   d. NI-DAQ does not use the menus in the bottom section of the window, the module configuration settings. Only LabVIEW 3.0 and higher uses those settings. You can, however, use this section to record your

settings for easy reference. You must set and keep track of your SCXI gain and filter settings using the SCXI functions in your NI-DAQ application.

You can command-click on any input field to pop up a help window. You can also refer to Chapter 7, *SCXI Functions*, if you need more detailed information on alternative SCXI configurations. Figure 1-6 shows the `NI-DAQ` Control Panel with the SCXI Configuration window selected.

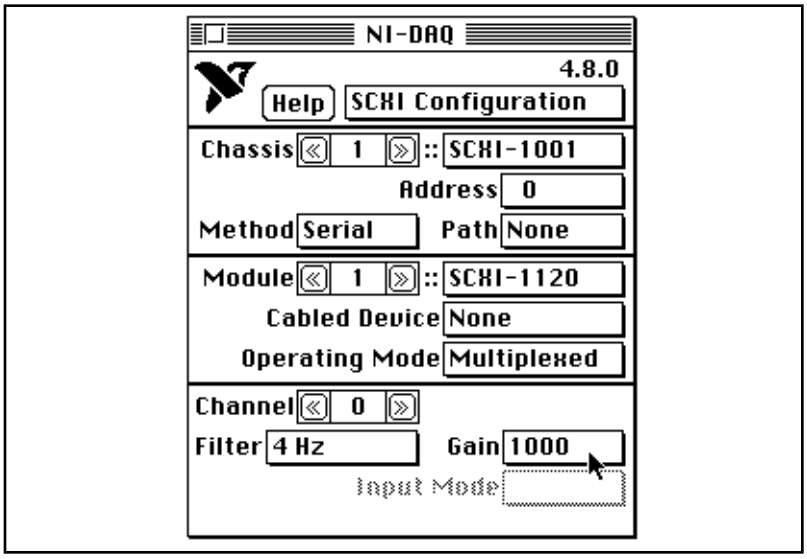

Figure 1-6. The NI-DAQ SCXI Configuration Window

Continue by reading the appropriate sections from the following *Using NI-DAQ for Macintosh Language Interfaces* section.

Using the NI-DAQ for Macintosh Language Interfaces

This section presents an introduction to the NI-DAQ for Macintosh Language Interface portion of this manual. The following sections discuss the various libraries used by the compiler environments, the organization of the include files, the various data types NI-DAQ uses, and the changes in the error-handling scheme. The final section details language-specific information.

Libraries

The NI-DAQ language-interface libraries are organized in the following hierarchy:

- `680x0 Libraries`
  - `Apple`
  - `Metrowerks`
  - `National Instruments`
    - `LabVIEW 3.1.x`
    - `LabVIEW 4.0.0`
  - `shared`

  - Symantec
    - C/C++
    - Pascal
  - Zedcor
- PowerPC Libraries
  - National Instruments
    - LabVIEW 3.1.x
    - LabVIEW 4.0.0
  - shared
  - static

The `National Instruments` libraries are updated versions of the DAQ VI libraries for LabVIEW 3.1.x and LabVIEW 4. If you need to upgrade your LabVIEW 3.1.x or LabVIEW 4 installation, install the NI-DAQ interfaces and use the Finder to move the entire contents of the appropriate library folder to the `DAQ` folder in LabVIEW's `vi.lib` folder.

To use the shared libraries for 680x0-based Macintoshes, you must have CFM-68K installed (these libraries are included in the `shared` folder); to install the libraries, use the Finder to move the contents of the `shared` folder to the `System Folder»Extensions` folder on your startup disk.

To use the shared libraries for PowerPC-based Macintoshes, use the Finder to move the contents of the `shared` folder to the `System Folder»Extensions` folder on your startup disk.

Whether you use the shared libraries or any of the static libraries, you will need to link the libraries with your object code. For PowerPC-based Macintoshes, select the library you want. For 680x0-based Macintoshes, if you use a static library with the Metrowerks, Symantec, or Zedcor environments, you will need to choose between A5-relative and A4-relative libraries; use A5-relative libraries for applications, and A4-relative libraries for *all* other cases. In the Metrowerks and Symantec environments, you need to initialize A4 yourself when you use A4-relative libraries; consult your development environment manuals for information on how to configure A4.

Include Files

The NI-DAQ language-interface include files are located in the following folders:

- `preferred headers`
- `shared headers`
- `compatibility headers`

If you are doing new development, you should use the files in the `preferred headers` folder. If you are updating older code, you will probably want to use the files in the `compatibility headers` folder. In either case, you will also need to use the files in the `shared headers` folder.

If you are using C or C+, it is *very* important that you use the new header files because the libraries have been converted to use Pascal calling conventions.

If you are using Pascal or BASIC, you will need to call the function `setSourceLanguage` with the constant appropriate to your language; consult the header files for prototypes and language-selector constants. If you do not call this function, certain functions that deal with 8-bit data will not work properly.

Data Types

The NI-DAQ interface libraries and headers now use the following types:

- signed, 8-bit integer data
- unsigned, 8-bit integer data
- signed, 16-bit integer data
- unsigned, 16-bit integer data
- signed, 32-bit integer data
- unsigned, 32-bit integer data
- 32-bit floating-point data
- 64-bit floating-point data
- 32-byte string data

Consult the language-specific sections for more information on which types are supported in your environment.

Error Codes

NI-DAQ now returns platform-independent error codes. If you are upgrading older programs, you may need to change any values you searched for with hardwired values; in other words, if you did not use the error constants defined in the interface headers, you will need to modify your source code. However, if you did use the constants defined in the interface headers, you can use the compatibility headers to help you reduce the number of modifications you need to make to your source code.

Also, the entry points for NI-DAQ no longer return system errors. Consequently, the global variable **LDSysError** no longer exists.

Using NI-DAQ for Macintosh with C/C++

To use NI-DAQ with C/C++, include one of two files in your source file. If you are updating an existing program, use `ni_daq_mac.h`. If you are writing a new program, include `nidaq.h`.

The data types used by NI-DAQ are defined in the file `platform.h`. The following types are used:

- i8 signed, 8-bit integer data
- u8 unsigned, 8-bit integer data
- i16 signed, 16-bit integer data
- u16 unsigned, 16-bit integer data

- i32 signed, 32-bit integer data
- u32 unsigned, 32-bit integer data
- f32 32-bit floating-point data
- f64 64-bit floating-point data
- string32 32-byte string data

If you are using Metrowerks for 68K or THINK C, you must configure your project to use 8-byte doubles, or the format of the f64 type will not be correct. In the THINK C environment, you may find it necessary to recompile certain standard libraries in order to use 8-byte doubles.

The prototype for each NI-DAQ function is shown in the following chapters. As an example, consider the following prototype:

```
locus i32   DAQ_VScale(u32 deviceNumber, u32 channel, u32 gain,
               f64 gainAdjust, f64 offset, u32 count, i16 *readings,
               f64 *voltages);
```

Notice the qualifier *locus*, which means *location or place of origin*. This qualifier shows a very important point—*all NI-DAQ functions now use Pascal calling conventions* because we now provide unified libraries that you can use with C/C++, Pascal, and BASIC. It is very important that you always include the NI-DAQ header files in your source code—*never* use an NI-DAQ function without a prototype. If you are using THINK C, you must configure your project to use THINK C language extensions in order for the `pascal` keyword to operate.

Also, notice that all parameters except **readings** and **voltages** are passed by value. In this case, both **readings** and **voltages** are arrays; however, other functions may pass scalar values by reference. Consult the chapters that follow for more information concerning the actual type of a pass-by-reference parameter. You can use the C/C++ operator `&` to generate a pointer to a scalar or array item.

Using NI-DAQ for Macintosh with Pascal

To use NI-DAQ with Pascal, include the file `nidaq.p` in your source or project. Before using any of the NI-DAQ functions, you should call the function `setSourceLanguage` with a value of `kSourceIsPascal`; this function is not discussed in the chapters that follow, but its prototype can be found in `nidaq.p`, as can a definition for `kSourceIsPascal`. The purpose of this function is to inform the library that the calling language is Pascal so that strings are placed in Pascal format and 8-bit arrays are manipulated properly (because Pascal does not support 8-bit data types).

The data types used by NI-DAQ are defined in the file `nidaq.p`. The following types are used:

- i16 signed, 16-bit integer data
- pi16 a pointer to an i16
- ppi16 a pointer to a pointer to an i16
- i32 signed, 32-bit integer data
- pi32 a pointer to an i32
- f32 32-bit floating-point data
- pf32 a pointer to an f32

- f64 64-bit floating-point data
- pf64 a pointer to an f64
- string32 32-byte string data

The prototype for each NI-DAQ function is shown in the following chapters. As an example, consider the following prototype:

```
function DAQ_VScale(deviceNumber : i32; channel : i32; gain : i32;
            gainAdjust : f64; offset : f64; count : i32; readings : pi16;
            voltages : pf64) : i32;
```

Notice that all parameters except **readings** and **voltages** are scalars. In this case, both **readings** and **voltages** are arrays that are passed by value using a pointer; however, other functions may pass scalar or array values by reference. You can use the Pascal operator `@` to generate a pointer to a scalar or array item.

Using NI-DAQ for Macintosh with BASIC

To use NI-DAQ with BASIC, specifically, FutureBASIC, include the file `nidaq.bas` in your project. Before using any of the NI-DAQ functions, you should call the function `setSourceLanguage` with a value of `kSourceIsBASIC`; this function is not discussed in the chapters that follow, but its interface can be found in `nidaq.bas`, as can a definition for `kSourceIsBASIC`. The prototype for this function is as follows:

```
FN        setSourceLanguage(languageType&)
```

This function informs the library that the calling language is BASIC so that strings are placed in BASIC format and 8-bit arrays are manipulated properly (because BASIC does not support 8-bit data types).

The following types are used:

- % signed, 16-bit integer data
- & signed, 32-bit integer data
- ! 32-bit floating-point data
- # 64-bit floating-point data
- $ 32-byte string data
- & a pointer to %, &, !, or #

You must configure your project to use six significant digits for single-precision floating-point data and 12 significant digits for double-precision floating-point data; if you use any other size, the NI-DAQ interface will not function properly.

The prototype for each NI-DAQ function is shown in the following chapters. As an example, consider the following prototype:

```
FN        DAQ_VScale(deviceNumber&, channel&, gain&, gainAdjust#,
            offset#, count&, readings&, voltages&)
```

For this function, all parameters except **readings** and **voltages** are passed by value. However, because of the manner in which BASIC passes parameters, it is not intuitive that **readings** and **voltages** are arrays. Because the parameter types for BASIC are very similar to those used by Pascal, you can use the Pascal prototype to help you

discern the actual type of a parameter. You can use the BASIC operator `@` to generate a pointer to a scalar or array item.

Chapter 2
Board-Specific Functions

This chapter describes the functions for configuring and calibrating the boards.

Board-Specific Functions

The Board-Specific functions are used for configuring and calibrating boards.

`A2000_Calibrate`	Calibrates the NB-A2000 A/D gain and offset values or restores them to the original factory-set values. The gain and offset values calculated during calibration adjust the accuracy of the readings from the four analog input channels.
`A2000_Config`	Configures some special NB-A2000 features: selects the source of the sample clock and whether or not to drive the SAMPCLK* line, chooses whether or not to add dithering to the input signal, and chooses whether or not to use block-mode transfers with the NB-A2000.
`A2100_Calibrate`	Selects the desired calibration reference and performs an offset calibration cycle on the ADCs on the NB-A2100 or the NB-A2150.
`A2100_Config`	Selects the signal source used to provide data to the DACs and lets you configure the external digital trigger so that it is shared both by data acquisition and waveform generation operations on the NB-A2100.
`A2150_Config`	Selects whether or not an internally generated trigger should be driven to the I/O connector. Also determines whether the board's sampling clock signal should be driven over the RTSI bus to other boards for multiple-board synchronized data acquisition.
`Board_ID`	Returns the National Instruments-assigned board ID for the selected board.
`Board_Reset`	Stops any ongoing operation and resets the board in the specified slot to its system startup default configuration.
`Calibrate_1200`	Calibrates the gain and offset values for the DAQCard-1200 and PCI-1200 ADCs and DACs. You can perform a new calibration or use an existing set of calibration constants by copying the constants from their storage location in the onboard EEPROM. You can store up to six sets of calibration constants. NI-DAQ automatically loads the calibration constants stored in EEPROM user area 5 the first time you call a function pertaining to the device.
`Calibrate_E_Series`	Use this function to calibrate your E Series device and to select a set of calibration constants for NI-DAQ to use.
`Get_DAQ_Device_Info`	Retrieves parameters pertaining to the device operation.
`Master_Slave_Config`	Configures one board as a master board and one or more other boards as slave boards. Currently used only by the NB-A2000 and the NB-A2150, this function ensures that, in a multiple frame acquisition, the slave boards are always re-enabled before the master board.
`MIO_Config`	Turns dithering (the addition of Gaussian noise to the analog input signal) on and off, for an E Series device (except the PCI-MIO-16XE-50), PCI-1200, and DAQCard-1200.

`MIO_16X_Config`	Configures the oscillator frequency for the ADC selected when using external timing sources on the NB-MIO-16X.
`SC_2040_Config`	Informs NI-DAQ that an SC-2040 Track-and-Hold accessory is attached to the device and communicates to NI-DAQ gain settings for one or all channels.
`Select_Signal`	Chooses the source and polarity of a signal that the board uses (E Series devices only).
`Set_DAQ_Device_Info`	This function can be used to change the data transfer mode (interrupts and DMA) for certain classes of data acquisition operations, some settings for an SC-2040 Track-and-Hold accessory and an SC-2043-SG strain-gauge accessory, as well as the source for the CLK1 signal on the DAQCard-700.

A2000_Calibrate

Function

Calibrates the NB-A2000 A/D gain and offset values or restores them to the original factory-set values. The gain and offset values calculated during calibration adjust the accuracy of the readings from the four analog input channels.

Warning: ***Read the calibration chapter in the*** *NB-A2000 User Manual* ***before using*** `A2000_Calibrate`**.**

Synopsis

C Syntax	`locus i32 A2000_Calibrate(u32 deviceNumber, u32 saveNewValues, u32 calibrationMethod, u32 channel, f64 extRefVoltage);`
Pascal Syntax	`function A2000_Calibrate(deviceNumber : i32; saveNewValues : i32; calibrationMethod : i32; channel : i32; extRefVoltage : f64) : i32;`
BASIC Syntax	`FN A2000_Calibrate(deviceNumber&, saveNewValues&, calibrationMethod&, channel&, extRefVoltage#)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

saveNewValues selects the values used for calibration. The gain and offset calibration values are stored in an EEPROM on the NB-A2000 board, which does not lose its data even when there is no power to the board. These values are read from the EEPROM and loaded into the NB-A2000 calibration circuitry when the board is initialized (at power-up) or reset (`Board_Reset`) and saved for use during data acquisition. When you calibrate the NB-A2000, you can choose to replace the permanent copies of the gain and offset values in the EEPROM and use the new values until the next calibration, even if the board is re-initialized, or you can elect not to replace the EEPROM values but use the new values until the next calibration or initialization.

Set **saveNewValues** as follows:

0: do not write new values to EEPROM.
1: do write new values to EEPROM.

For example, if you get consistently inaccurate readings from one or more input channels, even after resetting the board, you can calibrate and save the new gain and offset calibration values as permanent copies in the EEPROM. However, if acquisition results are accurate after initialization but start to drift after a few hours of operation when the board's temperature has increased, you can calibrate the board at this operating temperature, but retain the current EEPROM values to use after the next initialization.

calibrationMethod selects the method for calibration as follows:

0: use internal reference to calibrate.
1: use external reference to calibrate.
2: reload factory calibration values.

channel determines the input channel connected to the external reference source. For greatest accuracy in the calibration, connect the reference to more than one channel, and set channel to -1. All channels with input values close to the given **extRefVoltage** are averaged to find the reference voltage. If you have the reference voltage connected to only one input channel, set the channel to that channel number. The channel settings are given as follows:

-1: Source channels are determined automatically and values averaged.
0: channel 0.
1: channel 1.
2: channel 2.
3: channel 3.

extRefVoltage is the voltage of the external reference.

A2000_Config

Function

Configures some special NB-A2000 features: selects the source of the sample clock and whether or not to drive the SAMPCLK* line, chooses whether or not to add dithering to the input signal, and chooses whether or not to use block-mode transfers with the NB-A2000.

Synopsis

C Syntax	`locus i32 A2000_Config(u32 deviceNumber, u32 sClockSource, u32 sClockDrive, u32 dither, u32 memoryType);`
Pascal Syntax	`function A2000_Config(deviceNumber : i32; sClockSource : i32; sClockDrive : i32; dither : i32; memoryType : i32) : i32;`
BASIC Syntax	`FN A2000_Config(deviceNumber&, sClockSource&, sClockDrive&, dither&, memoryType&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

sClockSource sets the sampling timing to be controlled either by the onboard sample clock or by the external signal at the SAMPCLK* input.

0: Onboard sample clock.
1: External sample clock.

By setting **sClockSource** to 1, the NB-A2000 can receive the sample clock from the external SAMPCLK* line or from the CLOCKI line of the RTSI bus. To receive the sample clock from the RTSI CLOCKI line, call `RTSI_Conn` (see Chapter 9, *RTSI Bus Trigger Functions*).

sClockDrive sets the sample clock signal to drive or not drive the SAMPCLK* line.

0: Sample clock signal does not drive SAMPCLK* line.
1: Sample clock signal drives SAMPCLK* line.

It is not possible to receive the sample clock from the SAMPCLK* line and drive it at the same time.

dither determines whether or not to add approximately 0.5 LSB RMS of white Gaussian noise to the input signal. This is useful for applications that involve averaging to increase the resolution of the NB-A2000 to more than 12 bits. For high-speed applications that do not involve averaging, dithering is not recommended and should be disabled.

0: Dither disabled.
1: Dither enabled.

memoryType determines whether the memory allocated for the acquisition buffer is capable of performing block-mode transfers. If the acquisition buffer is on a memory-expansion board or in main memory that is capable of performing block-mode transfers and you are using an NB-DMA2800, faster DMA performance can be achieved by setting **memoryType** to 1. If the memory allocated for the acquisition buffer is not capable of performing block-mode transfers, or if you are not using an NB-DMA2800, set **memoryType** to 0.

0: no block-mode capability.
1: block-mode capability.

After system startup, the NB-A2000 is configured for the following:

sClockSource = 0: onboard sample clock.
sClockDrive = 0: sample clock signal does not drive SAMPCLK* line.
dither = 0: dither disabled.
memoryType = 0: no block mode capability.

As mentioned in the description of **sClockDrive**, it is not possible to receive the sample clock signal from the SAMPCLK* line and drive the SAMPCLK* line simultaneously. However, because setting **sClockSource** to 1 indicates that the sample clock is received from an external source that can be either the SAMPCLK* line or the RTSI bus, conflicts are detected by `MAI_Arm` and `MDAQ_Start` when the source of the sample clock is known.

A2100_Calibrate

Function

Selects the desired calibration reference and performs an offset calibration cycle on the ADCs on the NB-A2100 or the NB-A2150.

Synopsis

C Syntax	`locus i32 A2100_Calibrate(u32 deviceNumber, u32 adcGroup, u32 reference);`
Pascal Syntax	`function A2100_Calibrate(deviceNumber : i32; adcGroup : i32; reference : i32) : i32;`
BASIC Syntax	`FN A2100_Calibrate(deviceNumber&, adcGroup&, reference&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

adcGroup selects the A/D channels that should be calibrated.

Valid values for the NB-A2100: 0. On the NB-A2100, both the A/D channels, channel 0 and channel 1, belong to **adcGroup** 0.

Valid values for the NB-A2150: 0, 1, 2. On the NB-A2150, all four A/D channels belong to **adcGroup** 0, A/D channels 0 and 1 belong to **adcGroup** 1, and A/D channels 2 and 3 belong to **adcGroup** 2.

reference selects the calibration reference to be used during the offset calibration cycle.

0: calibrate the channels using the analog input ground as the reference for each channel.
1: calibrate the channels using the external signal connected to each channel as the reference for that channel.

The two A/D channels are calibrated using the analog input ground as the reference for each channel when the computer is powered up.

A2100_Config

Function

Selects the signal source used to provide data to the DACs and lets you configure the external digital trigger so that it is shared both by data acquisition and waveform generation operations on the NB-A2100.

Synopsis

C Syntax	`locus i32 A2100_Config(u32 deviceNumber, u32 dacSource, u32 triggerMode);`
Pascal Syntax	`function A2100_Config(deviceNumber : i32; dacSource : i32; triggerMode : i32) : i32;`
BASIC Syntax	`FN A2100_Config(deviceNumber&, dacSource&, triggerMode&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

dacSource selects the source to be used to supply data to the DACs.

0: use the data in the D/A FIFO. This is the default setting at startup.
1: use the data being sampled by the ADCs. With this setting, you can send the data sampled by the ADCs directly to the DACs.

triggerMode disables or enables subsequent data acquisition and waveform generation operations to share the external digital trigger.

0: disable subsequent data acquisition and waveform generation operations to share the external trigger. This indicates that `MDAQ` and `BWF` Functions should execute independently of each other.
1: enable subsequent data acquisition and waveform generation operations to share the external trigger. This indicates that the software should recognize the external trigger when both data acquisition and waveform generation operations are ready to receive the trigger. In other words, any trigger applied when only one operation has been initiated is ignored, and any trigger applied when both operations have been initiated are simultaneously accepted by both operations. After the shared trigger is enabled, any subsequent calls to `MDAQ_Start` and `BWF_Start` must be made with the external trigger enabled for the operation being initiated. A typical function sequence to use shared trigger would be as follows:

`A2100_Config` to enabled shared trigger
`MDAQ_Setup` to set up acquisition buffer
`MDAQ_Trig_Config` to enable external trigger
`MDAQ_Start` to start data acquisition
`BWF_BufLoad` to set up waveform buffer
`BWF_Start` to start waveform generation

The last `BWF_Start` call in this case would enable the recognition of the external trigger.

If multiple data acquisition frames are being acquired and multiple waveform cycles are being generated with a trigger required at the beginning of each cycle, then the recognition of the external trigger is synchronized so that each trigger simultaneously initiates the acquisition of the next data frame and the output of the next waveform cycle.

A2150_Config

Function

Selects whether or not an internally generated trigger should be driven to the I/O connector. Also determines whether the board's sampling clock signal should be driven over the RTSI bus to other boards for multiple-board synchronized data acquisition.

Synopsis

C Syntax	`locus i32 A2150_Config(u32 deviceNumber, u32 triggerDrive, u32 masterClock, u32 slaveCount, u16 *slaveList);`
Pascal Syntax	`function A2150_Config(deviceNumber : i32; triggerDrive : i32; masterClock : i32; slaveCount : i32; slaveList : pi16) : i32;`
BASIC Syntax	`FN A2150_Config(deviceNumber&, triggerDrive&, masterClock&, slaveCount&, slaveList&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

triggerDrive selects whether the trigger signal received over the RTSI bus or the internally generated analog level trigger signal should be connected to EXTTRIG* line at the I/O connector.

0: do not drive the EXTTRIG* line at the I/O connector.
1: drive the EXTTRIG* line a the I/O connector.

masterClock selects whether or not the sampling clock signal of the selected **deviceNumber** should be configured to be driven over the RTSI bus to other NB-A2150 boards.

0: do not change the configuration of the sampling clock drive circuitry.
1: configure the sampling clock circuitry according to the given **slaveList**.

slaveCount selects the number of slave boards to be configured and the number of elements in **slaveList**.

slaveList is an array that contains the slot numbers of the boards that should accept the sampling clock signal over the RTSI bus from **deviceNumber**. **slaveList** is ignored if **masterClock** is 0 or if **slaveCount** is 0.

You should enable **triggerDrive** only if you have executed `RTSI_Conn` to receive the RTSITRIG* signal over the RTSI bus or if you have executed `MDAQ_Trig_Config` to enable the analog level trigger. In these cases, after you execute `MDAQ_Start`, you can monitor the signal being sent to the A/D trigger circuitry at the EXTTRIG* line of the I/O connector. A high-to-low edge of the signal triggers the data acquisition.

The NB-A2150 uses signals over the RTSI bus for sampling clock synchronization between two or more NB-A2150 boards. The sampling clock synchronization circuitry makes simultaneous sampling possible on more than four channels using additional NB-A2150 boards. If **masterClock** is 1, **slaveList** should contain the list of boards in **slaveList** that will accept the sampling clock from **deviceNumber**. After you execute `A2150_Config` with **masterClock** as 1 and **slaveCount** greater than zero, `MDAQ_ScanRate` ignores the parameters for boards in **slaveList** until you execute `A2150_Config` again on **deviceNumber** with **masterClock** as 1 and **slaveCount** as 0. Executing `A2150_Config` with **masterClock** as 1 and **slaveCount** as 0 deconfigures the boards previously in the **slaveList** and sets them up to use their own sampling clock signal.

Board_ID

Function

Returns the National Instruments-assigned board ID for the selected device.

Synopsis

C Syntax	`locus i32  Board_ID(u32 deviceNumber, i16 *deviceType);`
Pascal Syntax	`function  Board_ID(deviceNumber : i32; var deviceType : i16) : i32;`
BASIC Syntax	`FN Board_ID(deviceNumber&, deviceType&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

deviceType returns the board ID used by NI-DAQ. A return value of negative one (-1) indicates that the slot corresponding to the selected **deviceNumber** is either empty or does not contain a device NI-DAQ recognizes.

`Board_ID` can be executed to find out programmatically which board exists at the slot position of the selected **deviceNumber**. This information can then be used to determine if a particular operation on a board is possible. The following are the decimal values of the devices recognized by NI-DAQ for Macintosh.

DAQCard-500:	49
DAQCard-700:	31
DAQCard-1200:	48
DAQCard-DIO-24:	35
DAQCard-AO-2DC:	47
Lab-LC:	110
Lab-NB:	111
NB-A2000:	112
NB-A2100:	118
NB-A2150C:	115
NB-A2150F:	116
NB-A2150S:	117
NB-AO-6:	114
NB-DIO-24:	107
NB-DIO-32F:	108
NB-DIO-96:	109
NB-DMA-8-G:	266
NB-DMA2800:	458
NB-DSP2300	125
NB-DSP2301	126
NB-DSP2305	127
NB-MIO-16H-9:	104
NB-MIO-16H-15:	105
NB-MIO-16H-25:	106
NB-MIO-16L-9:	100
NB-MIO-16L-15:	101
NB-MIO-16L-25:	102
NB-MIO-16XH-18:	121
NB-MIO-16XH-42:	122
NB-MIO-16XL-18:	119
NB-MIO-16XL-42:	120
NB-PRL:	107
NB-TIO-10:	113
PCI-1200:	353
PCI-DIO-96:	352
PCI-MIO-16XE-50:	354

Board_Reset

Function

Stops any ongoing operation and resets the specified board to its system startup default configuration.

Synopsis

C Syntax	`locus i32  Board_Reset(u32 deviceNumber);`
Pascal Syntax	`function  Board_Reset(deviceNumber : i32) : i32;`
BASIC Syntax	`FN Board_Reset(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`Board_Reset` stops any ongoing operation on the board specified by **deviceNumber** and reinitializes the board to its system startup default state.

If any board resources have been reserved for SCXI use when a call to `Board_Reset` is made, those resources will still be reserved after the function call is made. The only exception is the Mux Counter (Counter 1) on the NB-MIO-16 or NB-MIO-16X; after `Board_Reset` is called, Counter 1 is unreserved. Please refer to Chapter 7, *SCXI Functions*, for listings of the different board resources that may be reserved for SCXI.

- The default state for the NB-MIO-16 and the NB-MIO-16X is as follows:

Analog Input:

number of channels:	8
input mode:	differential
polarity:	bipolar
input range:	-10 V to 10 V
gains:	1 for all channels
multiplexer:	internal
external gate:	disabled
A/D timing:	onboard
external trigger:	disabled
pretrigger:	disabled

Analog Output:

number of channels:	2
output mode:	bipolar
output range:	10 V
group configuration:	disabled

Digital I/O:

number of lines:	8
input channels:	all 8 channels
latching:	disabled

Counters:

number of channels:	3
counter channels:	1, 2, 5
output state:	high impedance
latching:	disabled

4-Bit Programmable Frequency:

number of channels:	1
output frequency:	disabled

- The default state for the NB-DMA-8-G and the NB-DMA2800 is as follows:

  Counters:
  - number of channels: 5
  - counter channels: 1 through 5
  - output state: high impedance

  4-Bit Programmable Frequency:
  - number of channels: 1
  - output frequency: 6.25 kHz

  RTSI Lines: clear

  DMA Channels: all available

- The default state for the NB-DIO-24 is as follows:

  Digital I/O:
  - number of lines: 24
  - latching: disabled

- The default state for the NB-DIO-96 is as follows:

  Digital I/O:
  - number of lines: 96
  - latching: disabled

- The default state for the NB-DIO-32F is as follows:

  Digital I/O:
  - number of lines: 32
  - latching: disabled
  - group configuration: disabled

- The default state for the NB-AO-6 is as follows:

  Analog Output:
  - number of channels: 6
  - output mode: bipolar
  - output range: 10 volts
  - update mode: immediate update
  - group configuration: disabled
  - external update edge: falling edge

- The default state for the Lab and 1200 series is as follows:

  ADCs:
  - number of channels: 8
  - input mode: single-ended
  - gains: 1 for all channels
  - polarity: bipolar
  - input range: -5 V to 5 V
  - A/D timing: onboard
  - triggering: software trigger

  DACs:
  - number of channels: 2
  - output mode: bipolar
  - output range: -5 V to 5 V
  - update mode: immediate update
  - group configuration: disabled

- The default state for the NB-A2000 is as follows:

  Analog Input:

number of channels:	4
input channels:	0, 1, 2, 3
sample clock source:	onboard
trigger mode:	posttrigger
analog trigger:	disabled
digital trigger:	disabled
coupling:	all channels AC coupled
dithering:	disabled

  Note: `Board_Reset` ***clears the previously defined master slave configuration. If*** `Board_Reset` ***is called on a master board or a slave board that is the only slave to its master, the whole master slave configuration is cleared. Otherwise, only the board on which*** `Board_Reset` ***is called is taken out of the master slave configuration.*** `Board_Reset` ***also reads the current user values from the EEPROM and loads them into the NB-A2000 calibration circuitry.***

- The default state for the NB-A2100 is as follows:

  Analog Input:

number of channels:	2
input channels:	0, 1
sample clock frequency:	48 kHz
trigger mode:	posttrigger
analog trigger:	disabled
digital trigger:	disabled
coupling:	both channels DC coupled

  Note: `Board_Reset` ***does not calibrate the ADC on the NB-A2100.***

  Analog Output:

output data source:	D/A FIFO
update clock frequency:	48 kHz
digital trigger:	disabled
coupling:	jumper dependent

- The default state for the NB-A2150 is as follows:

  Analog Input:

number of channels:	4
input channels:	0, 1, 2, 3
sample clock frequency:	51.2 kHz for NB-A2150F 48 kHz for NB-A2150C 24 kHz for NB-A2150S
trigger mode:	posttrigger
analog trigger:	disabled
digital trigger:	disabled
coupling:	all channels DC coupled

- The default state for the NB-TIO-10 is as follows:

  Digital I/O:

number of lines:	16
input channels:	all 16 channels
latching:	disabled

  Counters:

number of counters:	10
counter channels:	1 through 10
output state:	high impedance

  4-Bit Programmable Frequency:

number of channels:	2
output frequency:	disabled

- The default state for the DAQCard-500 is as follows:

  Analog Input:

number of channels:	8
input mode:	single-ended
gains:	1 for all channels
input range:	-5 to +5 V
polarity:	bipolar
A/D timing:	onboard
triggering:	software trigger

  Digital I/O:

number of output lines:	4
number of input lines:	4
latching:	disabled

- The default state for the DAQCard-700 is as follows:

  Analog Input:

number of channels:	16
input mode:	single-ended
gains:	1 for all channels
input range:	-5 to +5 V
polarity:	bipolar
A/D timing:	onboard
triggering:	software trigger

  Digital I/O:

number of output lines:	8
number of input lines:	8
latching:	disabled

- The default state for the DAQCard-AO-2DC is as follows:

  Analog Output:

number of channels:	2
output mode:	unipolar
output range:	10 V

  Digital I/O:

number of lines:	16
input channels:	all 16 channels
latching:	disabled

- The default state for the PCI-DIO-96 is as follows:

  Digital I/O:

number of lines:	96
power-on state:	high

- The default state for the PCI-MIO-16XE-50 is as follows:

  Analog Output:

number of channels:	2
output mode:	bipolar
output range:	10 V
group configuration:	disabled

Calibrate_1200

Function

The PCI-1200 and DAQCard-1200 come fully equipped with accurate factory calibration constants. However, if you feel that the device is not performing either analog input or output accurately and suspect the device calibration to be in error, you can use `Calibrate_1200` to obtain a user defined set of new calibration constants.

A complete set of calibration constants consists of ADC constants for all gains at one polarity plus DAC constants for both DACs, again at the same polarity setting. It is important to understand the polarity rules. The polarity your device was in when a set of calibration constants was created must match the polarity your device is in when those calibration constants are used. For example, calibration constants created when your ADC is in unipolar must only be used for data acquisition when your ADC is also in unipolar.

You can store up to six sets of user defined calibration constants. These are stored in the EEPROM on your device in places called user calibration areas. You may also at any time use the calibration constants created at the factory. These are stored in write protected places in the EEPROM called factory calibration areas. There are two of these. One holds constants for bipolar operation and the other for unipolar. One additional area in the EEPROM important to calibration is called the default load table. This table contains four pointers to sets of calibration constants; one pointer each for ADC unipolar constants, ADC bipolar constants, DAC unipolar and DAC bipolar. This table is used by NI-DAQ for calibration constant loading.

It is important to understand the calibration constant loading rules. The first time a function requiring use of the ADC or DAC is called in an application, NI-DAQ automatically loads a set of calibration constants. At that time the polarities of your ADC and DACs are examined and the appropriate pointers in the default load table are used.

Synopsis

C Syntax	`locus i32 Calibrate_1200(u32 deviceNumber, u32 calOp, u32 saveNewCal, u32 EEPROMloc, u32 calRefChan, u32 groundRefChan, u32 DAQ0chan, u32 DAQ1chan, f64 calRefVolts, f64 gain);`
Pascal Syntax	`function Calibrate_1200(deviceNumber : i32; calOp : i32; saveNewCal : i32; EEPROMloc : i32; calRefChan : i32; groundRefChan : i32; DAQ0chan : i32; DAQ1chan : i32; calRefVolts : f64; gain : f64) : i32;`
BASIC Syntax	`FN Calibrate_1200(deviceNumber&, calOp&, saveNewCal&, EEPROMloc&, calRefChan&, groundRefChan&, DAQ0chan&, DAQ1chan&, calRefVolts#, gain#)`

Warning: ***Read the calibration chapter in your device user manual before using*** `Calibrate_1200`**.**

Description

calOp determines the operation to be performed.

1: Load calibration constants from **EEPROMloc**. If **EEPROMloc** is 0, the default load table is used and NI-DAQ will ensure that the constants loaded will be appropriate for the current polarity settings. If **EEPROMloc** is any other value you must ensure that the polarity of your device matches those of the calibration constants.

2: Calibrate the ADC using DC reference voltage **calRefVolts** connected to **calRefChan**. To calibrate the ADC, you must ground one input channel (**groundRefChan**) and connect a voltage reference between any other channel and AGND (pin 11). *Please remember that the ADC must be in referenced single-ended mode for successful calibration of the ADC.* After calibration, the calibration constants that were obtained during the process will remain in use by the ADC until the device is initialized again.

3: Calibrate the DACS. **DAC0chan** and **DAC1chan** are the analog input channels to which DAC0 and DAC1 are connected, respectively. To calibrate the DACs, you must wrap-back the DAC0 output (pin

10) and DAC1 out (pin 12) to any two analog input channels. Please remember that the ADC must be in referenced single-ended and bipolar mode and fully calibrated (using **calOp**=2) for successful calibration of the DACs. After calibration, the calibration constants that were obtained during the process will remain in use by the DACs until the device is initialized again.

4: invalid.

5: Edit the default load table so that the set of constants in the area identified by **EEPROMloc** (1-6, 9 or 10) become the default calibration constants for the ADC. NI-DAQ will change either the unipolar or bipolar pointer in the default load table depending on the polarity those constants are intended for. The factory default for the ADC unipolar pointer is **EEPROMloc=**9. The factory default for the ADC bipolar pointer is **EEPROMloc**=10. You can specify any user area in **EEPROMloc** after you have run a calibration on the ADC and saved the calibration constants to that user area. Or you can specify **EEPROMloc=**9 or 10 to reset the default load table to the factory calibration for unipolar and bipolar mode respectively.

6: Edit the default load table so that the set of constants in the area identified by **EEPROMloc** (1-6, 9 or 10) become the default calibration constants for the DACs. NI-DAQ's behavior for **calOp=**6 is identical to that for **calOp=**5. Just substitute DAC everywhere you see ADC.

saveNewCal is only valid when **calOp** is 2 or 3.

0: Do not save new calibration constants. Even though not permanently saved in the EEPROM, calibration constants created after a successful calibration will remain in use by your device until your device is initialized again.

1: Save new calibration constants in **EEPROMloc** (1-6).

EEPROMloc selects the storage location in the onboard EEPROM to be used. Different sets of calibration constants can be used to compensate for configuration or environmental changes.

0: Use the default load table (only valid if **calOp** = 1).
1: User calibration area 1.
2: User calibration area 2.
3: User calibration area 3.
4: User calibration area 4.
5: User calibration area 5.
6: User calibration area 6.
7: Invalid.
8: Invalid.
9: Factory calibration area for unipolar (write protected).
10: Factory calibration area for bipolar (write protected).

Notice that the user cannot write into **EEPROMloc** 9 and 10.

calRefChan is the analog input channel connected to the calibration voltage of **calRefVolts** when **calOp** is 2.
Range: 0 through 7.

groundRefChan is the analog input channel connected to ground when **calOp** is 2.
Range: 0 through 7.

DAC0chan is the analog input channel connected to DAC0 when **calOp** is 3.
Range: 0 through 7.

DAC1chan is the analog input channel connected to DAC1 when **calOp** is 3.
Range: 0 through 7.

calRefVolts is the value of the DC calibration voltage connected to **calRefChan** when **calOp** = 2. If you are calibrating at a gain other than 1, make sure you apply a voltage so that calRefVolts * gain is within the upper limits of the analog input range of the device.

gain is the device gain setting you want to calibrate at. When you perform an analog input operation, a calibration constant for that gain must be available. *When you run the* `Calibrate_1200` *function at a particular gain, the device can only be used to collect data accurately at that gain.* If you are creating a set of

calibration constants that you intend to use then you must be sure to calibrate at all gains that you intend to sample at.

Range: 1, 2, 5, 10, 50, or 100.

A calibration performed in bipolar mode is not valid for unipolar and vice versa. `Calibrate_1200` performs a bipolar or unipolar calibration, or loads the bipolar or unipolar constants (**calOp**=1, **EEPROMloc**=0), depending on the value of the polarity parameter in the last call to `AI_Configure` and `AO_Configure`. If analog input measurements are taken with the wrong set of calibration constants loaded, you may get erroneous data.

Calibrate for a particular gain if you plan to acquire at that gain. If you calibrate the device yourself make sure you calibrate at a gain that you are likely to use. Each gain has a different calibration constant. When you switch gains, NI-DAQ will automatically load the calibration constant for that particular gain. If you have not calibrated for that gain and saved the constant earlier, an incorrect value will be used.

How do you set up your own calibration constants in the user area for both unipolar and bipolar configuration? You want to create and store both unipolar and bipolar ADC calibration constants. And you want to modify the default load table so that NI-DAQ will automatically load your constants instead of the factory constants.

Change the polarity of your device to unipolar (you can use the `AI_Configure` call). Call `Calibrate_1200` to perform an ADC calibration (**calOp**=2) with **saveNewCal**=1 (save) and **EEPROMloc** set to any user area you prefer (say, 1). Next call the function with **calOp**=5 and **EEPROMloc**=1. NI-DAQ will automatically modify the ADC unipolar pointer in the default load table to point to user area 1.

Now, change the polarity of your device to bipolar. Call `Calibrate_1200` to perform another ADC calibration (**calOp**=2) with **saveNewCal**=1 (save) and **EEPROMloc** set to a different user area (say, 2). Next call the function with **calOp**=5 and **EEPROMloc**=2. NI-DAQ will automatically modify the ADC bipolar pointer in the default load table to point to user area 2. At this point, you have set up user area 1 to be your default load area when you operate the device in unipolar mode and user area 2 to be your default load area when you operate the device in bipolar mode. The loading of the appropriate constants will be handled automatically by NI-DAQ.

Failed calibrations leave your device in an incorrectly calibrated state. If you run this function with **calOp**=2 or 3 and receive an error, you must reload a valid set of calibration constants. If you have a valid set of user defined constants in one of the user areas you can load them. Otherwise you should reload the factory constants.

Calibrate_E_Series

Function

Use this function to calibrate your E Series device and to select a set of calibration constants to be used by NI-DAQ.

Warning: ***Read the calibration chapter in your device user manual before using*** `Calibrate_E_Series`**.**

Synopsis

C Syntax	`locus i32 Calibrate_E_Series(u32 deviceNumber, u32 calOp, u32 setOfCalConstants, f64 calRefVolts);`
Pascal Syntax	`function Calibrate_E_Series(deviceNumber : i32; calOp : i32; setOfCalConstants : i32; calRefVolts : f64) : i32;`
BASIC Syntax	`FN Calibrate_E_Series(deviceNumber&, calOp&, setOfCalConstants&, calRefVolts#)`

Description

The legal ranges for the **calOp** and **setOfCalConstants** parameters are given in terms of constants that are defined in a header file. The header file you should use depends on the language you are using:

- C programmers—`nidaqcns.h`
- Pascal programmers—`nidaq.p`
- BASIC programmers—`nidaq.bas`

calOp determines the operation to be performed.
Range:

`ND_SET_DEFAULT_LOAD_AREA:`	Make **setOfCalConstants** the default load area; do not perform calibration.
`ND_SELF_CALIBRATE:`	Perform self-calibration of the device.
`ND_EXTERNAL_CALIBRATE:`	Perform external calibration of the device.

setOfCalConstants selects the set of calibration constants to be used by NI-DAQ. These calibration constants reside in the onboard EEPROM or are maintained by NI-DAQ.
Range:

`ND_FACTORY_EEPROM_AREA`:	Factory calibration area of the EEPROM. You cannot modify this area, so you can set **setOfCalConstants** to `ND_FACTORY_EEPROM_AREA` only when **calOp** is set to `ND_SET_DEFAULT_LOAD_AREA`.
`ND_NI_DAQ_SW_AREA`:	NI-DAQ maintains calibration constants internally; no writing into the EEPROM occurs. You cannot use this setting when **calOp** is set to `ND_SET_DEFAULT_LOAD_AREA`. This setting is useful if you want to calibrate your device repeatedly during your program, and you do not want to store the calibration constants in the EEPROM.
`ND_USER_EEPROM_AREA`:	For the user calibration area of the EEPROM. If **calOp** is set to `ND_SELF_CALIBRATE` or `ND_EXTERNAL_CALIBRATE`, the new calibration constants will be written into this area, and this area will become the new default load area. You can use this setting if you want to run several NI-DAQ applications during one measurement session conducted at same temperature, and you do not want to recalibrate your device in each application.

calRefVolts is the value of the DC calibration voltage connected to analog input channel 0 when **calOp** is `ND_EXTERNAL_CALIBRATE`. This parameter is ignored when **calOp** is `ND_SET_DEFAULT_LOAD_AREA` or `ND_SELF_CALIBRATE`.
Range:
+6.0 to +9.999 V

Your device contains calibration D/A converters (calDACs) that are used for fine-tuning the analog circuitry. The calDACs must be programmed (loaded) with certain numbers called calibration constants. Those constants are stored in non-volatile memory (EEPROM) on your device or are maintained by NI-DAQ. To achieve specification accuracy, you should perform an internal calibration of your device just before a measurement session but after your computer and the device have been powered on and allowed to warm up for at least 15 minutes. Frequent calibration produces the most stable and repeatable measurement performance. The device is not harmed in any way if you recalibrate it as often as you like.

Two sets of calibration constants can reside in two *load areas* inside the EEPROM; one set is programmed at the factory, and the other is left for the user. One load area in the EEPROM corresponds to one set of constants. The load area NI-DAQ uses for loading calDACs with calibration constants is called the default load area. When you get the device from the factory, the default load area is the area that contains the calibration constants obtained by calibrating the device in the factory. NI-DAQ automatically loads the relevant calibration constants

stored in the load area the first time you call a function (an AI, AO, DAQ, SCAN, and WFM function) that requires them. NI-DAQ also automatically reloads calibration constants whenever appropriate; see the *Calibration Constant Loading by NI-DAQ* section later in this function for details. When you call the Calibrate_E_Series function with **setOfCalConstants** set to ND_NI_DAQ_SW_AREA, NI-DAQ uses a set of constants it maintains in a load area that does not reside inside the EEPROM.

Note: ***Calibration of your MIO device takes some time. Do not be alarmed if the*** Calibrate_E_Series ***function takes several seconds to execute.***

Note: ***After powering on your computer, you should wait for some time (typically 15 minutes) for the entire system to warm up before performing the calibration. You should allow the same warm-up time before any measurement session that will take advantage of the calibration constants determined by using the*** Calibrate_E_Series ***function.***

Warning: ***When you call the*** Calibrate_E_Series ***function with*** **calOp** ***set to*** ND_SELF_CALIBRATE ***or*** ND_EXTERNAL_CALIBRATE***, NI-DAQ will abort any ongoing operations the device is performing and set all configurations to defaults. Therefore we recommend that you call*** Calibrate_E_Series ***before calling other NI-DAQ functions (except*** USE ***functions) or when no other operations are going on.***

Explanations about using this function for different purposes (with different values of **calOp)** are given in the following sections.

Changing the Default Load Area

Set **calOp** to ND_SET_DEFAULT_LOAD_AREA if you want to change the area used for calibration constant loading. The storage location selected by **setOfCalConstants** becomes the new default load area.

Example:

You want to make the factory area of the EEPROM default load area. You should make the following call:

```
Calibrate_E_Series(deviceNumber, ND_SET_DEFAULT_LOAD_AREA,
ND_FACTORY_EEPROM_AREA, 0.0)
```

Performing Self-Calibration of the Board

Set **calOp** to ND_SELF_CALIBRATE if you want to perform self-calibration of your device. The storage location selected by **setOfCalConstants** becomes the new default load area.

Example:

You want to perform self-calibration of your device and you want to store the new set of calibration constants in the user area of the EEPROM. You should make the following call:

```
Calibrate_E_Series(deviceNumber, ND_SELF_CALIBRATE, ND_USER_EEPROM_AREA,
0.0)
```

The EEPROM user area will become the default load area.

Performing External Calibration of the Board

Set **calOp** to ND_EXTERNAL_CALIBRATE if you want to perform external calibration of your device. The storage location selected by **setOfCalConstants** becomes the new default load area.

Make the following connections before calling the Calibrate_E_Series function:

1. Connect the positive output of your reference voltage source to analog input channel 0.

2. Connect the negative output of your reference voltage source to analog input channel 8.

By performing these first two connections, you supply the reference voltage to analog input channel 0, which is configured for differential operation.

3. If your reference voltage source and your computer are floating with respect to each other, connect the negative output of your reference voltage source to the AIGND line as well as to analog input channel 8.

Example:

You want to perform an external calibration of your device using an external reference voltage source with a precise 7.0500 V reference, and you want NI-DAQ to maintain a new set of calibration constants without storing them in the EEPROM. You should make the following call:

```
Calibrate_E_Series (deviceNumber, ND_EXTERNAL_CALIBRATE, ND_NI_DAQ_SW_AREA,
7.0500)
```

The internal NI-DAQ area will become the default load area, and the calibration constants will be lost when your application ends.

Calibration Constant Loading by NI-DAQ

NI-DAQ automatically loads calibration constants into calDACs whenever you call functions that depend on them (`AI`, `AO`, `DAQ`, `SCAN`, and `WFM` functions). The following conditions apply:

- Calibration constants required by the E Series devices for unipolar analog input channels are different from those for bipolar analog input channels. If you are acquiring data from one channel, or if all of the channels you are acquiring data from are configured for the same polarity, NI-DAQ selects the appropriate set of calibration constants for you. If you are scanning several channels, and you mix channels configured for unipolar and bipolar mode in your scan list, NI-DAQ loads the calibration constants appropriate for the polarity that analog input channel 0 is configured for.

- Analog output channels on the PCI-MIO-16XE-50 can only be configured for bipolar operation. Therefore, NI-DAQ always uses the same constants for the analog output channels.

Get_DAQ_Device_Info

Function

Retrieves parameters pertaining to the device operation.

Synopsis

C Syntax	`locus i32 Get_DAQ_Device_Info(u32 deviceNumber, u32 infoType, u32 *infoValue);`
Pascal Syntax	`function Get_DAQ_Device_Info(deviceNumber : i32; infoType : i32; var infoValue : i32) : i32;`
BASIC Syntax	`FN Get_DAQ_Device_Info(deviceNumber&, infoType&, infoValue&)`

Description

The legal range for the **infoType** is given in terms of constants that are defined in a header file. The header file you should use depends on the language you are using:

- C programmers—`nidaqcns.h`
- Pascal programmers—`nidaq.p`
- BASIC programmers—`nidaq.bas`

Use **infoType** to let NI-DAQ know which parameter you want to retrieve. **infoValue** will reflect the value of the parameter. **infoValue** will be given either in terms of constants from the header file or as numbers, as appropriate.

infoType can be one of the following:

infoType	Description
ND_BASE_ADDRESS	Base address, in hexadecimal, of the device specified by **deviceNumber**.
ND_DATA_XFER_MODE_AI ND_DATA_XFER_MODE_AO_GR1 ND_DATA_XFER_MODE_AO_GR2 ND_DATA_XFER_MODE_GPCTR0 ND_DATA_XFER_MODE_GPCTR1 ND_DATA_XFER_MODE_DIO_GR1 ND_DATA_XFER_MODE_DIO_GR2	See the Set_DAQ_Device_Info function for details. ND_NOT_APPLICABLE if not relevant to the device.
ND_DEVICE_TYPE_CODE	Type of the device specified by **deviceNumber**. See Init_DA_Brds for a list of device type codes.
ND_DMA_A_LEVEL ND_DMA_B_LEVEL ND_DMA_C_LEVEL	Level of the DMA channel assigned to the device as channel A, B, and C. ND_NOT_APPLICABLE if not relevant or disabled.
ND_INTERRUPT_A_LEVEL ND_INTERRUPT_B_LEVEL	Level of the interrupt assigned to the device as interrupt A and B. ND_NOT_APPLICABLE if not relevant or disabled.
ND_COUNTER_1_SOURCE	See the Set_DAQ_Device_Info function for details. ND_NOT_APPLICABLE if not relevant to the device.

Note to C Programmers: **infoValue** *is a pass-by-reference parameter.*

Master_Slave_Config

Function

Configures one board as a master board and one or more other boards as slave boards. Currently used only by the NB-A2000 and the NB-A2150, this function ensures that, in a multiple frame acquisition, the slave boards are always re-enabled before the master board.

Synopsis

C Syntax	`locus i32 Master_Slave_Config(u32 deviceNumber, u32 slaveCount, u16 *slaveList);`
Pascal Syntax	`function Master_Slave_Config(deviceNumber : i32; slaveCount : i32; slaveList : pi16) : i32;`
BASIC Syntax	`FN Master_Slave_Config(deviceNumber&, slaveCount&, slaveList&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

slaveCount selects the number of slave boards to be configured and the number of elements in **slaveList**.

slaveList is an array that contains the device numbers of the slave boards. The values of the elements in **slaveList** can range from 1 to 14.

On the NB-A2000, a board is considered a master board if it is sending its TRIGGER*, START*, or CLOCKO signal to another board (see Chapter 9, *RTSI Bus Trigger Functions*, for more information on these signals). On the NB-A2150, a board is considered a master board if it is sending its RTSITRIG*, SWSTART*, or the A/D sampling clock signal to another board (see the description for `A2150_Config` earlier in this chapter and Chapter 9, *RTSI Bus Trigger Functions*, for more information on these signals). The board receiving these signals is considered a slave board because sampling is controlled by signals sent from the master board. In a multiple frame acquisition, for a slave board to always be able to respond to a master signal, the slave board must be enabled before the master board is enabled. If the master board is enabled first, it can send its signal to the slave boards before they are capable of responding. The initial start-up order is the responsibility of the application you are using. The master board should always be started last. The purpose of `Master_Slave_Config` is to ensure that the master is also started last for each subsequent frame acquired during a multiple frame acquisition.

MIO_16X_Config

Function

Configures the oscillator frequency for the ADC selected when using external timing sources on the NB-MIO-16X.

Synopsis

C Syntax	`locus i32  MIO_16X_Config(u32 deviceNumber, u32 adcType);`
Pascal Syntax	`function  MIO_16X_Config(deviceNumber : i32; adcType : i32) : i32;`
BASIC Syntax	`FN MIO_16X_Config(deviceNumber&, adcType&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

adcType indicates the minimum sample interval of the ADC. **adcType** has the following possible values:

0: 42 µs minimum sample interval (NB-MIO-16X-42).
1: 18 µs minimum sample interval (NB-MIO-16X-18).

The ADC used on the NB-MIO-16X has an oscillator frequency input in addition to the sampling frequency used for data acquisition. The ADC must use the correct oscillator frequency to operate correctly. You should use the value for **adcType** that corresponds with your version of the NB-MIO-16X.

`MIO_16X_Config` must be called when using an external timebase (the SOURCE5 input) or an external clock to control sampling. NI-DAQ for Macintosh defaults to an oscillator frequency that is compatible with both the NB-MIO-16X-42 and NB-MIO-16X-18, but sample intervals faster than 42 µs with the NB-MIO-16X-18 produces data that is not correct to 16 bits. If the internal sample clock is used, then NI-DAQ for Macintosh picks the appropriate oscillator input based on the sample interval supplied, and `MIO_16X_Config` does not have to be called.

MIO_Config

Function

Turns dithering (the addition of Gaussian noise to the analog input signal) on and off, for an E Series device (except the PCI-MIO-16XE-50), PCI-1200, and DAQCard-1200.

Synopsis

C Syntax	`locus i32 MIO_Config(u32 deviceNumber, u32 dither, u32 useAMUX);`
Pascal Syntax	`function MIO_Config(deviceNumber : i32; dither : i32; useAMUX : i32) : i32;`
BASIC Syntax	`FN MIO_Config(deviceNumber&, dither&, useAMUX&)`

Description

dither indicates whether to add approximately 0.5 LSB rms of white Gaussian noise to the input signal. This is useful for applications that involve averaging to increase the effective resolution of a device. For high-speed applications that do not involve averaging, dithering is not recommended and should be disabled.

0: Disable dithering.
1: Enable dithering.

This parameter is ignored for the PCI-MIO-16XE-50. Dithering is always enabled on this device.

useAMUX does not apply to any of these devices and is ignored.

SC_2040_Config

Function

Informs NI-DAQ that an SC-2040 Track-and-Hold accessory is attached to the device and communicates to NI-DAQ gain settings for one or all channels.

Synopsis

C Syntax	`locus i32 SC_2040_Config(u32 deviceNumber, u32 channel, u32 sc2040gain);`
Pascal Syntax	`function SC_2040_Config(deviceNumber : i32; channel : i32; sc2040gain : i32) : i32;`
BASIC Syntax	`FN SC_2040_Config(deviceNumber&, channel&, sc2040gain&)`

Description

channel allows you to specify an individual channel on the SC-2040 or all SC-2040 channels.
Range: -1 for all channels and 0 through 7 for individual channels.

sc2040gain allows you to indicate the gain you have selected with your SC-2040 jumpers.
Range: 1, 10, 100, 200, 300, 500, 600, 700, 800.

You must use this function before any analog input function that uses the SC-2040.

This function reserves the PFI 7 line on your E Series device for use by NI-DAQ and the SC-2040. This line is configured for output, and the output is a signal that indicates when a scan is in progress.

Warning: ***Do not attempt to drive the PFI 7 line after calling this function. If you do, you may damage your SC-2040, your E Series device, and your equipment.***

Example 1:

You have set the jumper for a gain of 100 for all your SC-2040 channels. You should call `SC_2040_Config` as follows:

```
SC_2040_Config(deviceNumber, -1, 100)
```

Example 2:

You have set the jumper for a gain of 100 for channels 0, 3, 4, 5, and 6 on your SC-2040, gain 200 for channels 1 and 2, and gain 500 for channel 7. You should call function `SC_2040_Config` several times as follows:

```
SC_2040_Config(deviceNumber, -1, 100)
SC_2040_Config(deviceNumber, 1, 200)
SC_2040_Config(deviceNumber, 2, 200)
SC_2040_Config(deviceNumber, 7, 500)
```

Select_Signal

Function

Chooses the source and polarity of a signal that the device uses (E Series devices only).

Synopsis

C Syntax	`locus i32 Select_Signal(u32 deviceNumber, u32 signal, u32 source, u32 sourceSpec);`
Pascal Syntax	`function Select_Signal(deviceNumber : i32; signal : i32; source : i32; sourceSpec : i32) : i32;`
BASIC Syntax	`FN Select_Signal(deviceNumber&, signal&, source&, sourceSpec&)`

Description

You can use the onboard DAQ-STC to select among many sources for various signals.

Use the **signal** parameter to specify the signal whose source you want to select. The following table shows the possible values for **signal**.

Group	signal	Description
Timing and Control Signals Used Internally by the Onboard DAQ-STC	ND_IN_START_TRIGGER	Start trigger for the DAQ and SCAN functions
	ND_IN_STOP_TRIGGER	Stop trigger for the DAQ and SCAN functions
	ND_IN_SCAN_CLOCK_TIMEBASE	Scan clock timebase for the SCAN functions
	ND_IN_CHANNEL_CLOCK_TIMEBASE	Channel clock timebase for the DAQ and SCAN functions
	ND_IN_CONVERT	Convert signal for the AI, DAQ and SCAN functions
	ND_IN_SCAN_START	Start scan signal for the SCAN functions
	ND_IN_EXTERNAL_GATE	External gate signal for the DAQ and SCAN functions
	ND_OUT_START_TRIGGER	Start trigger for the WFM functions
	ND_OUT_UPDATE	Update signal for the AO and WFM functions
	ND_OUT_UPDATE_CLOCK_TIMEBASE	Update clock timebase for the WFM functions
I/O Connector Pins	ND_PFI_0 through PFI_9	Signal present at the I/O connector pin PFI0 through PFI9.
	ND_GPCTR0_OUTPUT	Signal present at the I/O connector pin GPCTR0_OUTPUT
	ND_GPCTR1_OUTPUT	Signal present at the I/O connector pin GPCTR1_OUTPUT
	ND_FREQ_OUT	Signal present at the FREQ_OUT output pin on the I/O connector.
RTSI Bus Signals	ND_RTSI_0 through ND_RTSI_6	Signal present at the RTSI bus trigger line 0 through 7.
	ND_RTSI_CLOCK	Enable the device to drive the RTSI clock line or prevent it from doing it.
	ND_BOARD_CLOCK	Enable the device to receive the clock signal from the RTSI clock line or stop it from doing so.

Legal values for **source** and **sourceSpec** depend on the **signal** and are shown in the following tables:

signal = ND_IN_START_TRIGGER

source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_RTSI_0 through ND_RTSI_6	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_GPCTR0_OUTPUT	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_AUTOMATIC	ND_DONT_CARE

Use ND_IN_START_TRIGGER to initiate a data acquisition sequence. You can use an external signal or output of general-purpose counter 0 as a source for this signal, or you can specify that NI-DAQ generates it (corresponds to **source** = ND_AUTOMATIC).

If you do not call this function with **signal** = ND_IN_START_TRIGGER, NI-DAQ uses the default values, **source** = ND_AUTOMATIC and **sourceSpec** = ND_LOW_TO_HIGH.

If you call DAQ_Config with **startTrig** = 1, NI-DAQ calls Select_Signal with **signal** = ND_IN_START_TRIGGER, **source** = ND_PFI_0, and **sourceSpec** = ND_HIGH_TO_LOW.

If you call DAQ_Config with **startTrig** = 0, NI-DAQ calls Select_Signal with **signal** = ND_IN_START_TRIGGER, **source** = ND_AUTOMATIC, and **sourceSpec** = ND_DONT_CARE.

signal = ND_IN_STOP_TRIGGER

source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_RTSI_0 through ND_RTSI_6	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW

Use ND_IN_STOP_TRIGGER for data acquisition in the pretriggered mode. The selected transition on the **signal** line indicates to the device that it should acquire a specified number of scans after the trigger and stop.

If you do not call this function with **signal** = ND_IN_STOP_TRIGGER, NI-DAQ uses the default values, **source** = ND_PFI_1 and **sourceSpec** = ND_HIGH_TO_LOW. By default, ND_IN_STOP_TRIGGER is not used because the pretriggered mode is disabled.

If you call DAQ_StopTrigger_Config with **startTrig** = 1, NI-DAQ calls Select_Signal with **signal** = ND_IN_STOP_TRIGGER, **source** = ND_PFI_1, and **sourceSpec** = ND_HIGH_TO_LOW. Therefore, if you want to use different selection for ND_IN_STOP_TRIGGER, you need to call the Select_Signal function after DAQ_StopTrigger_Config.

signal = ND_IN_EXTERNAL_GATE

source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_PAUSE_ON_HIGH and ND_PAUSE_ON_LOW
ND_RTSI_0 through ND_RTSI_6	ND_PAUSE_ON_HIGH and ND_PAUSE_ON_LOW
ND_NONE	ND_DONT_CARE

Use ND_IN_EXTERNAL_GATE for gating the data acquisition. For example, if you call this function with **signal** = ND_IN_EXTERNAL_GATE, **source** = ND_PFI_9, and **sourceSpec** = PAUSE_ON_HIGH, the data acquisition will be paused whenever the PFI 9 is at the high level. The pausing is performed on a per scan basis, so no scans are split by the external gate.

If you do not call this function with **signal** = ND_IN_EXTERNAL_GATE, NI-DAQ uses the default values, **source** = ND_NONE and **sourceSpec** = ND_DONT_CARE; therefore, by default, the data acquisition is not gated.

signal = ND_IN_SCAN_START

source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_RTSI_0 through ND_RTSI_6	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_GPCTR0_OUTPUT	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_INTERNAL_TIMER	ND_LOW_TO_HIGH

Use this signal for scan timing. You can use a DAQ-STC timer for timing the scans, or you can use an external signal. You can also use the output of the general-purpose counter 0 for scan timing. This can be useful for applications such as Equivalent Time Sampling (ETS).

If you do not call this function with **signal** = ND_IN_SCAN_START, NI-DAQ uses the default values, **source** = ND_INTERNAL_TIMER and **sourceSpec** = ND_LOW_TO_HIGH.

If you call DAQ_Config with **extConv** = 2 or 3, NI-DAQ calls Select_Signal with **signal** = ND_IN_SCAN_START, **source** = ND_PFI_7, and **sourceSpec** = ND_HIGH_TO_LOW.

If you call DAQ_Config with **extConv** = 0 or 1, NI-DAQ calls Select_Signal with **signal** = ND_IN_SCAN_START, **source** = ND_INTERNAL_TIMER, and **sourceSpec** = ND_LOW_TO_HIGH.

signal = ND_IN_CONVERT

source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_RTSI_0 through ND_RTSI_6	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_GPCTR0_OUTPUT	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_INTERNAL_TIMER	ND_LOW_TO_HIGH

Use ND_IN_CONVERT for sample (channel interval) timing. This signal controls the onboard ADC. You can use a DAQ-STC timer for timing the samples, or you can use an external signal. You can also use output of the general-purpose counter 0 for sample timing.

If you do not call this function with **signal** = ND_IN_CONVERT, NI-DAQ uses the default values, **source** = ND_INTERNAL_TIMER and **sourceSpec** = ND_LOW_TO_HIGH.

If you call DAQ_Config with **extConv** = 1 or 3, NI-DAQ calls Select_Signal with **signal** = ND_IN_CONVERT, **source** = ND_PFI_2, and **sourceSpec** = ND_HIGH_TO_LOW.

If you call DAQ_Config with **extConv** = 0 or 2, NI-DAQ calls Select_Signal with **signal** = ND_IN_CONVERT, **source** = ND_INTERNAL_TIMER, and **sourceSpec** = ND_LOW_TO_HIGH.

signal = ND_IN_SCAN_CLOCK_TIMEBASE

source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_RTSI_0 through ND_RTSI_6	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_INTERNAL_20_MHZ	ND_LOW_TO_HIGH
ND_INTERNAL_100_KHZ	ND_LOW_TO_HIGH

Use ND_IN_SCAN_CLOCK_TIMEBASE as an input into the DAQ-STC scan timer. The scan timer generates timing by counting the signal at its input, and producing an IN_START_SCAN signal after the specified number of occurrences of the ND_IN_SCAN_CLOCK_TIMEBASE signal transitions.

If you do not call this function with **signal** = ND_IN_SCAN_CLOCK_TIMEBASE, NI-DAQ uses the default values, **source** = ND_INTERNAL_20_MHZ and **sourceSpec** = ND_LOW_TO_HIGH.

signal = ND_IN_CHANNEL_CLOCK_TIMEBASE

source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_RTSI_0 through ND_RTSI_6	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_INTERNAL_20_MHZ	ND_LOW_TO_HIGH
ND_INTERNAL_100_KHZ	ND_LOW_TO_HIGH

Use ND_IN_CHANNEL_CLOCK_TIMEBASE as an input into the DAQ-STC sample (channel interval) timer. The sample timer generates timing by counting the signal at its input, and producing an ND_IN_CONVERT signal after the specified number of occurrences of the ND_IN_CHANNEL_CLOCK_TIMEBASE signal transitions.

If you do not call this function with **signal** = ND_IN_SCAN_CLOCK_TIMEBASE, NI-DAQ uses the default values, **source** = ND_INTERNAL_20_MHZ and **sourceSpec** = ND_LOW_TO_HIGH.

signal = ND_OUT_START_TRIGGER

source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_RTSI_0 through ND_RTSI_6	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_IN_START_TRIGGER	ND_LOW_TO_HIGH
ND_AUTOMATIC	ND_LOW_TO_HIGH

Use ND_OUT_START_TRIGGER to initiate a waveform generation sequence. You can use an external signal or the signal used as the ND_IN_START_TRIGGER, or NI-DAQ can generate it. Setting source to ND_IN_START_TRIGGER is useful for synchronizing waveform generation with data acquisition.

If you do not call this function with **signal** = ND_OUT_START_TRIGGER, NI-DAQ uses the default values, **source** = ND_AUTOMATIC and **sourceSpec** = ND_LOW_TO_HIGH.

signal = ND_OUT_UPDATE

source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_RTSI_0 through ND_RTSI_6	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_GPCTR1_OUTPUT	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_INTERNAL_TIMER	ND_LOW_TO_HIGH

Use this signal for update timing. You can use a DAQ-STC timer for timing the updates, or you can use an external signal. You can also use output of the general-purpose counter 1 for update timing.

If you do not call this function with **signal** = ND_OUT_UPDATE, NI-DAQ uses the default values, **source** = ND_INTERNAL_TIMER and **sourceSpec** = ND_LOW_TO_HIGH.

signal = ND_OUT_UPDATE_CLOCK_TIMEBASE

source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_RTSI_0 through ND_RTSI_6	ND_LOW_TO_HIGH and ND_HIGH_TO_LOW
ND_INTERNAL_20_MHZ	ND_LOW_TO_HIGH
ND_INTERNAL_100_KHZ	ND_LOW_TO_HIGH

Use this signal as an input into the DAQ-STC update timer. The update timer generates timing by counting the signal at its input and producing an ND_OUT_UPDATE signal after the specified number of occurrences of the ND_OUT_UPDATE_CLOCK_TIMEBASE signal transitions.

If you do not call this function with **signal** = ND_OUT_UPDATE_CLOCK_TIMEBASE, NI-DAQ uses the default values, **source** = ND_INTERNAL_20_MHZ and **sourceSpec** = ND_LOW_TO_HIGH.

signal = ND_PFI_0 through ND_PFI_9

The following table summarizes all the signals and source for the I/O connector pins PFI0 through PFI9.

signal	source	sourceSpec
ND_PFI_0 through ND_PFI_9	ND_NONE	ND_DONT_CARE
ND_PFI_0	ND_IN_START_TRIGGER	ND_LOW_TO_HIGH
ND_PFI_1	ND_IN_STOP_TRIGGER	ND_LOW_TO_HIGH
ND_PFI_2	ND_IN_CONVERT	ND_HIGH_TO_LOW
ND_PFI_3	ND_GPCTR1_SOURCE	ND_LOW_TO_HIGH
ND_PFI_4	ND_GPCTR1_GATE	ND_POSITIVE
ND_PFI_5	ND_OUT_UPDATE	ND_HIGH_TO_LOW
ND_PFI_6	ND_OUT_START_TRIGGER	ND_LOW_TO_HIGH
ND_PFI_7	ND_IN_SCAN_START	ND_LOW_TO_HIGH
ND_PFI_7	ND_IN_SCAN_IN_PROG	ND_LOW_TO_HIGH
ND_PFI_8	ND_GPCTR0_SOURCE	ND_LOW_TO_HIGH
ND_PFI_9	ND_GPCTR0_GATE	ND_POSITIVE

Use ND_NONE to disable output on the pin.

signal = ND_GPCTR0_OUTPUT

source	sourceSpec
ND_NONE	ND_DONT_CARE
ND_GPCTR0_OUTPUT	ND_LOW_TO_HIGH
ND_RTSI_0 through ND_RTSI_6	ND_LOW_TO_HIGH

Use ND_NONE to disable output on the pin. When you disable output on this pin, you can use the pin as an input pin, and you can attach an external signal to it. This is useful because it enables you to communicate a signal from the I/O connector to the RTSI bus.

When you enable this pin for output, you can program it to output the signal present at any one of the RTSI bus trigger lines or the general-purpose counter 0 output. The RTSI selections are useful because they enable you to communicate a signal from the RTSI bus to the I/O connector.

signal = ND_GPCTR1_OUTPUT

source	sourceSpec
ND_NONE	ND_DONT_CARE
ND_GPCTR1_OUTPUT	ND_LOW_TO_HIGH
ND_RESERVED	ND_DONT_CARE

Use ND_NONE to disable the output on the pin, in other words, do place the pin in high impedance state.

NI-DAQ may use ND_RESERVED when you use this device with some of the SCXI modules. In this case, you can use general-purpose counter 1, but the output will not be available on the I/O connector because the pin is used for device-to-SCXI communication. Currently, there are no SCXI modules that require this.

signal = ND_FREQ_OUT

source	sourceSpec
ND_NONE	ND_DONT_CARE
ND_INTERNAL_10_MHZ	1 through 16
ND_INTERNAL_100_KHZ	1 through 16

Use ND_NONE to disable the output on the pin.

The signal present on the FREQ_OUT pin of the I/O connector is the divided-down version of one of the two internal timebases. Use **sourceSpec** to specify the divide-down factor.

signal = ND_RTSI_0 through ND_RTSI_6

source	sourceSpec
ND_NONE	ND_DONT_CARE
ND_IN_START_TRIGGER	ND_LOW_TO_HIGH
ND_IN_STOP_TRIGGER	ND_LOW_TO_HIGH
ND_IN_CONVERT	ND_HIGH_TO_LOW
ND_OUT_UPDATE	ND_HIGH_TO_LOW
ND_OUT_START_TRIGGER	ND_LOW_TO_HIGH
ND_GPCTR0_SOURCE	ND_LOW_TO_HIGH
ND_GPCTR0_GATE	ND_POSITIVE
ND_GPCTR0_OUTPUT	ND_DONT_CARE

Use ND_NONE to disable output on the RTSI line.

You can use the GPCTR0_OUTPUT pin on the I/O connector in two ways—as an output pin or an input pin. When you configure the pin as an output pin, you can program the pin to output a signal from a RTSI line or the general-purpose counter 0 output (see **signal** = ND_GPCTR0_OUTPUT in this function for details). When you configure the pin as an input pin, you can attach an external signal to the pin. When **signal** is one of the RTSI lines, and **source** = ND_GPCTR0_OUTPUT, the signal on the RTSI line will be the signal present at the GPCTR0_OUTPUT pin on the I/O connector, which is not always the output of the general-purpose counter 0.

signal = ND_RTSI_CLOCK

source	sourceSpec
ND_NONE	ND_DONT_CARE
ND_BOARD_CLOCK	ND_DONT_CARE

Use **source** = ND_NONE to stop the device from driving the RTSI clock line.

When **source** = ND_BOARD_CLOCK, this device drives the signal on the RTSI clock line.

signal = ND_BOARD_CLOCK

source	sourceSpec
ND_BOARD_CLOCK	ND_DONT_CARE
ND_RTSI_CLOCK	ND_DONT_CARE

Use **source** = ND_BOARD_CLOCK to stop the device from receiving the clock signal from the RTSI clock line.

Use **source** = ND_RTSI_CLOCK to program the device to receive the clock signal from the RTSI clock line.

If you have selected a signal that is not an I/O connector pin or a RTSI bus line, Select_Signal saves the parameters in the configuration tables for future operations. Functions which initiate data acquisition (DAQ_Start and SCAN_Start) and waveform generation operations use the configuration tables to set the device circuitry to the correct timing modes.

You do not need to call this function if you are satisfied with the default settings for the signals.

If you have selected a signal that is an I/O connector or a RTSI bus signal, Select_Signal performs signal routing and enables or disables output on a pin or a RTSI line.

Example: Sending a signal from your E Series device to the RTSI bus

To send a signal from your E Series device to the RTSI bus, set **signal** to the appropriate RTSI bus line and **source** to indicate the signal from your device. If you want to send the analog input start trigger on to RTSI line 3, use the following call:

```
Select_Signal(deviceNumber, ND_RTSI_3, ND_IN_START_TRIGGER, ND_LOW_TO_HIGH)
```

Example: Receiving a signal from the RTSI bus on your E Series device

To receive a signal from the RTSI bus and use it as a signal on your E Series device, set **signal** to indicate the appropriate E Series device signal and **source** to the appropriate RTSI line. If you want to use low-to-high transitions of the signal present on the RTSI line 4 as your scan clock, use the following call:

```
Select_Signal(deviceNumber, ND_IN_SCAN_START, ND_RTSI_4, ND_LOW_TO_HIGH)
```

Signal Name Equivalencies: For a variety of reasons, some timing signals are given different names in the hardware documentation and the software and its documentation. The following table lists the equivalencies between the two sets of signal names.

Table 2-1. E Series Signal Name Equivalencies

	Hardware Name	Software Name
AI-Related Signals	TRIG1	ND_IN_START_TRIGGER
	TRIG2	ND_IN_STOP_TRIGGER
	STARTSCAN	ND_IN_SCAN_START
	SISOURCE	ND_IN_SCAN_CLOCK_TIMEBASE
	CONVERT*	ND_IN_CONVERT
	AIGATE	ND_IN_EXTERNAL_GATE
	SI2SOURCE	ND_IN_CHANNEL_CLOCK_TIMEBASE
AO-Related Signals	WFTRIG	ND_OUT_START_TRIGGER
	UPDATE*	ND_OUT_UPDATE
	AOGATE	ND_OUT_EXTERNAL_GATE
	UISOURCE	ND_OUT_UPDATE_CLOCK_TIMEBASE
	AO2GATE	–
	UI2SOURCE	–

Set_DAQ_Device_Info

Function

This function can be used to change the data transfer mode (interrupts and DMA) for certain classes of data acquisition operations, some settings for an SC-2040 Track-and-Hold accessory and an SC-2043-SG strain-gauge accessory, as well as the source for the CLK1 signal on the DAQCard-700.

Synopsis

C Syntax	`locus i32 Set_DAQ_Device_Info(u32 deviceNumber, u32 infoType, u32 infoValue);`
Pascal Syntax	`function Set_DAQ_Device_Info(deviceNumber : i32; infoType : i32; infoValue : i32) : i32;`
BASIC Syntax	`FN Set_DAQ_Device_Info(deviceNumber&, infoType&, infoValue&)`

Description

Legal ranges for the **infoType** and **infoValue** are given in terms of constants that are defined in a header file. The header file you should use depends on the language you are using:

- C programmers—`nidaqcns.h`
- Pascal programmers—`nidaq.p`
- BASIC programmers—`nidaq.bas`

Use **infoType** to let NI-DAQ know which parameter you want to change. Use **infoValue** to specify the corresponding new value.

Values that **infoType** accepts depend on the device you are using. The legal range for **infoValue** depends on the device you are using and **infoType**.

infoType can be one of the following:

infoType	Description
ND_DATA_XFER_MODE_AI	Method NI-DAQ will use for data transfers when performing the DAQ, MDAQ, and SCAN operations.
ND_DATA_XFER_MODE_AO_GR1 ND_DATA_XFER_MODE_AO_GR2	Method NI-DAQ will use for data transfers when performing the waveform operations which require buffers from the computer memory.
ND_DATA_XFER_MODE_GPCTR0 ND_DATA_XFER_MODE_GPCTR1	Method NI-DAQ will use for buffered data transfers when using GPCTR operations with the general purpose counter.
ND_DATA_XFER_MODE_DIO_GR1 ND_DATA_XFER_MODE_DIO_GR2	Method NI-DAQ will use for data transfers for digital input and output operations.
ND_SC_2040_MODE	Used to enable or disable the track-and-hold circuitry on the SC-2040.
ND_SC_2043_MODE	Used to enable or disable the SC-2043-SG accessory.
ND_COUNTER_1_SOURCE	Used to select a source for counter 1 on the DAQCard-700.

infoValue can be one of the following:

infoValue	Description
ND_INTERRUPTS	NI-DAQ will use interrupts for data transfers.
ND_UP_TO_1_DMA_CHANNEL	NI-DAQ will use one DMA channel, if possible; if the DMA channel is not available, NI-DAQ will report an error and it will not perform the operation.
ND_NO_TRACK_AND_HOLD	Disables use of the track-and-hold circuitry on the SC-2040.[1]
ND_TRACK_AND_HOLD	Re-enables the track-and-hold circuitry on an SC-2040 if you have previously disabled it.[2]
ND_NONE	Cancels the effects of having accidentally called the SC_2040_Config function.
ND_STRAIN_GAUGE	Enables the SC-2043-SG accessory for strain-gauge measurements (no excitation on channel 0).
ND_STRAIN_GAUGE_EX0	Enables the SC-2043-SG accessory with excitation on channel 0.
ND_NO_STRAIN_GAUGE	Disables the SC-2043-SG accessory.
ND_INTERNAL_TIMER	Counter 1 will use the internal timer as the source for its CLK1 source.
ND_IO_CONNECTOR	Counter 1 will use the CLK1 signal from the I/O connector as the source for its CLK1 signal.

[1] You should use this setting if you want to use the SC-2040 only as a preamplifier, without using track and hold.

[2] with ND_NO_TRACK_AND_HOLD.

You can use this function to select the data transfer method for a given operation on a particular device. If you do not use this function, NI-DAQ will decide on the data transfer method that will typically take maximum advantage of available resources.

All possible data transfer methods for the devices supported by NI-DAQ are listed in the following table. If your device is not listed, none of the data transfer modes are applicable. An asterisk is placed next to the default data transfer mode for each device.

Device Type	infoType	infoValue
NB-A2150	ND_DATA_XFER_MODE_AI	ND_INTERRUPTS ND_UP_TO_1_DMA_CHANNEL*
NB-AO-6	ND_DATA_XFER_MODE_AO_GR1 ND_DATA_XFER_MODE_AO_GR2	ND_UP_TO_1_DMA_CHANNEL* ND_UP_TO_1_DMA_CHANNEL*
NB-DIO-32F	ND_DATA_XFER_MODE_DIO_GR1	ND_INTERRUPTS ND_UP_TO_1_DMA_CHANNEL*
	ND_DATA_XFER_MODE_DIO_GR2	ND_INTERRUPTS ND_UP_TO_1_DMA_CHANNEL*
NB-A2100	ND_DATA_XFER_MODE_AI	ND_INTERRUPTS ND_UP_TO_1_DMA_CHANNEL*
	ND_DATA_XFER_MODE_AO_GR1	ND_INTERRUPTS ND_UP_TO_1_DMA_CHANNEL*
NB-MIO-16	ND_DATA_XFER_MODE_AI	ND_INTERRUPTS ND_UP_TO_1_DMA_CHANNEL*
	ND_DATA_XFER_MODE_AO_GR1 ND_DATA_XFER_MODE_AO_GR2	ND_UP_TO_1_DMA_CHANNEL* ND_UP_TO_1_DMA_CHANNEL*
NB-MIO-16X	ND_DATA_XFER_MODE_AI	ND_INTERRUPTS ND_UP_TO_1_DMA_CHANNEL*
	ND_DATA_XFER_MODE_AO_GR1	ND_INTERRUPTS ND_UP_TO_1_DMA_CHANNEL*
PCI-MIO-16XE-50	ND_DATA_XFER_MODE_AI ND_DATA_XFER_MODE_AO_GR1 ND_DATA_XFER_MODE_GPCTR0 ND_DATA_XFER_MODE_GPCTR1	ND_INTERRUPTS* ND_INTERRUPTS* ND_INTERRUPTS* ND_INTERRUPTS*
DAQCard-500 DAQCard-700	ND_DATA_XFER_MODE_AI	ND_INTERRUPTS*
NB-A2000	ND_DATA_XFER_MODE_AI	ND_INTERRUPTS UP_TO_1_DMA_CHANNEL*
DAQCard-1200 Lab-LC Lab-NB PCI-1200	DATA_XFER_MODE_AI DATA_XFER_MODE_AO_GR1	ND_INTERRUPTS* ND_INTERRUPTS*

NI-DAQ uses interrupts and DMA channels for data transfers. The DMA data transfers are typically faster, so you may want to take advantage of them. Note that the data transfer mode ND_UP_TO_1_DMA_CHANNEL does not reserve the DMA channel or channels for a particular operation; it just authorizes NI-DAQ to use it, if it is available.

Chapter 3
Analog Input Functions

This chapter describes the functions for single A/D conversions. The chapter is divided into two sections to describe the Single-Channel Analog Input (`AI`) and Multiple-Channel Analog Input (`MAI`) functions used with the National Instruments boards for the Macintosh family of computers.

Single-Channel Analog Input functions cover single A/D conversions on one channel. The Multiple-Channel Analog Input functions cover single A/D conversions simultaneously sampled on a group of channels. See Appendix A, *NI-DAQ for Macintosh Function and Board Compatibility*, to determine which set works with your board.

Multiple A/D conversion functions are performed by the Data Acquisition functions (see Chapter 6, *Data Acquisition Functions*).

If you are using SCXI analog input modules, you need to program the SCXI hardware first using the SCXI functions in Chapter 7, *SCXI Functions*, before using the Analog Input functions.

Single-Channel Analog Input

NB-MIO-16 Analog Input

The NB-MIO-16 contains 16 single-ended analog input channels numbered 0 through 15. These inputs can also be configured as eight differential analog input channels, in which case the channels are numbered 0 through 7. The analog input channels are multiplexed into a single programmable gain stage and 12-bit ADC. The NB-MIO-16 has four gains. The NB-MIO-16L has gains of 1, 10, 100, and 500. The NB-MIO-16H has gains of 1, 2, 4, and 8.

Analog input on the NB-MIO-16 can be hardware jumpered for three different input ranges:

- 0 to +10 V (unipolar)
- -5 to +5 V (bipolar)
- -10 to +10 V (bipolar)

The NB-MIO-16 is shipped from the factory configured for an input range of -10 to +10 V.

A/D conversions can be initiated through software or by applying active low pulses to the EXTCONV* input on the NB-MIO-16 I/O connector. A 16-word-deep FIFO memory on the board stores up to 16 A/D conversion results.

NB-MIO-16X Analog Input

The NB-MIO-16X contains 16 single-ended analog input channels numbered 0 through 15. These inputs can also be configured as eight differential analog input channels, in which case the channels are numbered 0 through 7. The analog input channels are multiplexed into a single programmable gain stage and 16-bit ADC. Both versions of the NB-MIO-16X have four gains. The NB-MIO-16XL provides gains of 1, 10, 100, and 500. The NB-MIO-16XH provides gains of 1, 2, 4, and 8.

Analog input on the NB-MIO-16X can be hardware jumpered for four different input ranges:

- 0 to +10 V (unipolar)
- 0 to +5 V (unipolar)
- -5 to +5 V (bipolar)
- -10 to +10 V (bipolar)

The NB-MIO-16X is shipped from the factory jumpered for an input range of -10 to +10 V.

A/D conversions can be initiated through software or by applying active low pulses to the EXTCONV* input on the NB-MIO-16X I/O connector. A 16-word deep FIFO on the board stores up to 16 A/D conversion results.

Lab and 1200 Series Analog Input

The Lab and 1200 series contain eight single-ended analog input channels numbered 0 through 7. The analog input channels are multiplexed into a single programmable gain state and 12-bit ADC. Seven gains are provided—1, 2, 5, 10, 20, 50, and 100. The PCI-1200 and DAQCard-1200 also allow differential configuration of input channels 0, 1, 2, and 3.

Analog input on the Lab-NB and Lab-LC can be hardware jumper-configured for two different input ranges:

- 0 to +10 V (unipolar)
- -5 to +5 V (bipolar)

Table 3-1 gives the nominal input ranges for all combinations of polarity and gain. The Lab-NB and Lab-LC are shipped from the factory jumpered for the bipolar input range of -5 V to +5 V.

Table 3-1. Analog Input Ranges

Polarity	Gain	Input Range
Unipolar	1	0 to +10 V
	2	0 to +5 V
	5	0 to +2 V
	10	0 to +1 V
	20	0 to +0.5 V
	50	0 to +0.2 V
	100	0 to +0.1 V
Bipolar	1	-5 to +5 V
	2	-2.5 to +2.5 V
	5	-1 to +1 V
	10	-0.5 to +0.5 V
	20	-0.25 to +0.25 V
	50	-0.1 to +0.1 V
	100	-0.05 to +0.05 V

DAQCard-500 and DAQCard-700 Analog Input

The DAQCard-500 provides 8 single-ended analog input channels. The DAQCard-700 provides 16 single-ended or eight differential analog input channels. The analog input channels for both are driven into a 12-bit ADC. Neither device has gains on the analog input.

You can configure the DAQCard-700 analog input for three different bipolar input ranges:

- -2.5 to +2.5 V
- -5 to +5 V
- -10 to +10 V

You can configure the DAQCard-500 for only the -5 to +5 V range.

You can initiate A/D conversions through software or by applying active low pulses to the EXTCONV* input on the device I/O connector. A 512-word-deep FIFO memory on the DAQCard-700 stores up to 512 A/D conversion results. On the DAQCard-500, a 16-word-deep FIFO stores up to 16 A/D conversion results.

SCXI Analog Input

SCXI modules can be used as a data acquisition front end for the boards described above to provide signal conditioning for the input signals and channel multiplexing. The SCXI functions described in Chapter 7 set up the SCXI modules for analog input operations to be performed by the DAQCard-700, MIO devices, and Lab and 1200 series devices using the functions described as follows.

Single-Channel Analog Input Function Summary

The following functions are for analog input operations:

`AI_Configure`	Informs NI-DAQ of the input mode (single-ended or differential), input range, and input polarity selected for the device. Use this function if you have changed the jumpers affecting the analog input configuration from their factory settings. For the E Series devices, PCI-1200, DAQCard-1200, DAQCard-500, and DAQCard-700, which have no jumpers for analog input configuration, this function programs the device for the settings you want.
`AI_Read`	Reads the specified analog input channel (initiates an A/D conversion on an analog input channel and returns the result).
`AI_Read_Scan`	Returns readings for all analog input channels selected by `SCAN_Setup` (E Series devices only, with or without the SC-2040 accessory).
`AI_VScale`	Converts the binary result from an `AI_Read` call to the actual input voltage.

The following functions are for conversion operations triggered externally via the EXTCONV* input:

`AI_Check`	Returns the status of the analog input circuitry and an analog input reading, if available.
`AI_Clear`	Clears the analog input circuitry and A/D FIFO memory.
`AI_Setup`	Selects an analog input channel and gain setting for externally pulsed conversion operations.

The following function configures one or more external multiplexer boards:

`AI_Mux_Config` Configures the number of multiplexer (AMUX-64T) boards connected to an MIO board.

For most purposes, `AI_Read` is the only function required to perform single analog input readings. `AI_VScale` can then be used to convert the binary value returned to a voltage value, if desired.

If the jumper settings on the Lab-NB, Lab-LC, NB-MIO-16, or NB-MIO-16X analog input circuitry have been changed from the factory settings, you need to use `AI_Configure` to update the analog input configuration information for the drivers. This update needs to be made only once per system startup or board reset per board. Figure 3-1 shows the call sequence for performing single analog input readings. Read the `AI_Configure` description to double check your configuration.

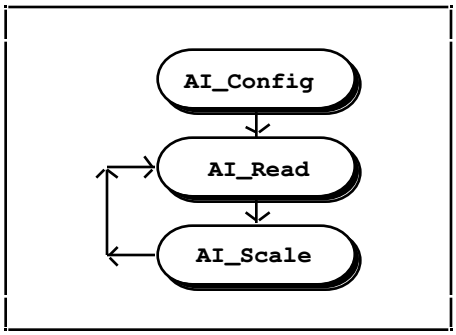

Figure 3-1. Flowchart for Analog Input Readings

`AI_Setup`, `AI_Check`, and `AI_Clear` are useful for externally triggered conversions as shown in Figure 3-2. See `AI_Check` for a description of this application.

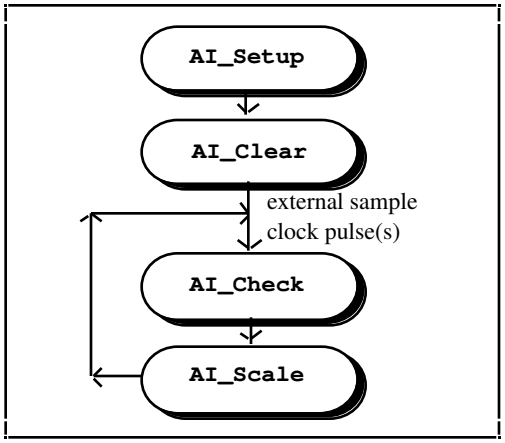

Figure 3-2. Flowchart for Externally Clocked Analog Input Readings

`AI_Mux_Config` is used to configure the number of multiplexer (AMUX-64T) boards connected to an MIO board to expand the number of signals up to 256 single-ended (128 differential). For more detailed information, see Appendix C, *Using an External Multiplexer*.

Note: ***Buffered analog input is implemented via the Data Acquisition functions presented in Chapter 6,*** *Data Acquisition Functions*.

AI_Check

Function

Returns the status of the analog input circuitry and an analog input reading, if available.

Synopsis

C Syntax	`locus i32 AI_Check(u32 deviceNumber, u16 *status, i16 *reading);`
Pascal Syntax	`function AI_Check(deviceNumber : i32; var status : i16; var reading : i16) : i32;`
BASIC Syntax	`FN AI_Check(deviceNumber&, status&, reading&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

status is the indicator in which the status of the analog input circuitry is returned. If **status** is 1, then an A/D conversion result is returned in **reading**. If **status** is 0, no A/D conversion result is available.

reading is the indicator in which the result of an A/D conversion is returned. The valid return values are listed in the Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*.

`AI_Check` checks the status of the analog input circuitry. If an A/D conversion has occurred, `AI_Check` returns **status** = 1 and the A/D conversion result. Otherwise, `AI_Check` returns **status** = 0.

`AI_Setup`, in conjunction with `AI_Check` and `AI_Clear`, is useful for externally timed A/D conversions. When `AI_Setup` is called, `AI_Clear` can be called to clear out the A/D FIFO of any previous conversion results. A conversion is then performed each time a pulse is received at the EXTCONV* input pin. `AI_Check` can be called to check for and return available conversion results.

AI_Clear

Function

Clears the analog input circuitry and A/D FIFO memory.

Synopsis

C Syntax	`locus i32 AI_Clear(u32 deviceNumber);`
Pascal Syntax	`function AI_Clear(deviceNumber : i32) : i32;`
BASIC Syntax	`FN AI_Clear(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`AI_Clear` clears the analog input circuitry and empties the analog input FIFO. `AI_Clear` also clears any analog input error conditions. `AI_Clear` should be called to clear out the A/D FIFO before any externally triggered conversion begins.

AI_Configure

Function

Informs NI-DAQ of the input mode (single-ended or differential), input range, and input polarity selected for the device. Use this function if you have changed the jumpers affecting the analog input configuration from their factory settings. For the E Series devices, PCI-1200, DAQCard-1200, DAQCard-500, and DAQCard-700, which have no jumpers for analog input configuration, this function programs the device for the settings you want.

For the E Series devices, you can configure the input mode and polarity on a per-channel basis.

Synopsis

C Syntax	`locus i32 AI_Configure(u32 deviceNumber, u32 channel, u32 inputMode, u32 inputRange, u32 inputPolarity, u32 driveAIS);`
Pascal Syntax	`function AI_Configure(deviceNumber : i32; channel : i32; inputMode : i32; inputRange : i32; inputPolarity : i32; driveAIS : i32) : i32;`
BASIC Syntax	`FN AI_Configure(deviceNumber&, channel&, inputMode&, inputRange&, inputPolarity&, driveAIS&)`

Description

channel is the analog input channel to be configured. Except for the E Series devices, you must set **channel** to -1 because the same analog input configuration applies to all of the channels. For the E Series devices, **channel** specifies the channel to be configured. If you want all of the channels to be configured identically, set **channel** to -1.

inputMode indicates whether the analog input channels are configured for single-ended or differential operation:

0: Differential (DIFF) configuration (default).
1: Referenced Single-Ended (RSE) configuration (used when the input signal does not have its own ground reference. The negative (-) input of the instrumentation amplifier is tied to the instrumentation amplifier signal ground to provide one).
2: Nonreferenced Single-Ended (NRSE) configuration (used when the input signal has its own ground reference. The input signal's ground reference is connected to AISENSE, which is tied to the negative (-) input of the instrumentation amplifier)

inputRange is the voltage range of the analog input channels.

inputPolarity indicates whether the ADC is configured for unipolar or bipolar operation:

0: Bipolar operation (default value).
1: Unipolar operation.

The following table shows all possible settings for **inputMode**, **inputRange**, and **inputPolarity**, with the default settings in italics. **inputMode** is independent of **inputRange** and **inputPolarity**.

Device	Possible Values for inputMode*	Analog Input Range			Software Configurable
		inputRange*	inputPolarity*	Resulting Analog Input Range	
PCI-MIO-16XE-50	*0*, 1, 2	ignored	unipolar	0 to +10 V	yes
		ignored	*bipolar*	-10 to +10V	
PCI-1200, DAQCard-1200	0, *1*	ignored	unipolar	0 to +10 V	yes
		ignored	*bipolar*	-5 to +5 V	
DAQCard-500	*1*	*10*	*bipolar*	-5 to 5 V	n/a
DAQCard-700	0, *1*	5	bipolar	-2.5 to +2.5 V	yes
		10	bipolar	-5 to +5 V	
		20	bipolar	-10 to +10 V	
* Italics indicates default settings.					

Note: ***If a device is software configurable, the* inputMode*, inputRange, and* inputPolarity *parameters are used to program the device for the configuration you want. If a device is not software configurable, this function uses these parameters to inform NI-DAQ of the device configuration, which you must set using hardware jumpers. If your device is software configurable and you have changed the analog input settings, you do not have to use* `AI_Configure`*, although it is good practice to do so in case you inadvertently change the configuration file.***

driveAIS indicates whether to drive AISENSE to onboard ground or not. This parameter is ignored for all devices on the Macintosh.

0: Do not drive AISENSE to ground.
1: Drive AISENSE to ground.

When you attach an SC-2040 or SC-2042-RTD to your DAQ device, you must configure channels 0 through 7 for differential mode. When you attach an SC-2043-SG to your DAQ device, you must configure these channels for nonreferenced single-ended mode.

See the `Calibrate_E_Series` function description for information about calibration constant loading on the E Series devices.

AI_Mux_Config

Function

Configures the number of multiplexer (AMUX-64T) boards connected to an MIO board.

Synopsis

C Syntax	`locus i32 AI_Mux_Config(u32 deviceNumber, u32 muxNumber);`
Pascal Syntax	`function AI_Mux_Config(deviceNumber : i32; muxNumber : i32) : i32;`
BASIC Syntax	`FN AI_Mux_Config(deviceNumber&, muxNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

muxNumber is the number of multiplexer (AMUX-64T) boards (0, 1, 2, or 4) connected to an MIO board. This input should be 0 if no external AMUX-64T boards are present. The default is 0.

An external multiplexer board (AMUX-64T) can be used to expand the number of analog input signals measured. The AMUX-64T has 16 separate four-to-one analog multiplexer circuits. One AMUX-64T board can multiplex up to 64 single-ended (32 differential) analog input signals. Four AMUX-64T boards can be cascaded to permit up to 256 single-ended (128 differential) signals to be multiplexed by one MIO board.

`AI_Mux_Config` configures the number of multiplexer boards connected to the MIO board. Input channels are then referenced in subsequent analog input calls (`AI_Read`, `AI_Setup`, and `DAQ_Start`) with respect to the external AMUX-64T analog input channel numbers rather than the MIO board onboard channel numbers. The call to `AI_Mux_Config` needs to be executed only once per board per system startup. See Appendix C, *Using an External Multiplexer*, for more information about using the AMUX-64T.

This function is not for use with SCXI.

AI_Read

Function

Reads the specified analog input channel (initiates an A/D conversion on an analog input channel and returns the result).

Synopsis

C Syntax	`locus i32 AI_Read(u32 deviceNumber, u32 channel, u32 gain, i16 *reading);`
Pascal Syntax	`function AI_Read(deviceNumber : i32; channel : i32; gain : i32; var reading : i16) : i32;`
BASIC Syntax	`FN AI_Read(deviceNumber&, channel&, gain&, reading&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog input channel number. If SCXI is being used, you must use the appropriate analog input channel on the DAQ board that corresponds to the desired SCXI channel. Please refer to Chapter 7, *SCXI Functions*, for more information on SCXI channel assignments.

Range: 0 through *n*-1, where *n* is the number of analog input channels available.

gain is the gain setting to be used for the selected channel. This gain setting applies only to the DAQ board; if SCXI is used, any gain desired at the SCXI module must be established either by setting jumpers on the module

or by calling `SCXI_Set_Gain`. See Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for the valid gain ranges for your hardware. The DAQCard-500 and DAQCard-700 ignore **gain**.

reading is the indicator in which the 12-bit (NB-MIO-16, Lab and 1200 series, DAQCard-500, or DAQCard-700) or 16-bit (NB-MIO-16X) result of the A/D conversion is returned. The valid return values are listed in Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*.

Pascal Note: ***If you are using an NB-MIO-16X in unipolar mode,* reading *is returned as a 16-bit unsigned integer. Because Pascal does not support unsigned representation, the values in the range 32,768 through 65,535 are treated as negative numbers in Pascal. You can use the*** `UToL` ***conversion function to convert* reading *to a Pascal long integer. Notice that* reading *should be passed to* `AI_Scale` ***without conversion. (See Chapter 11,*** *NI-DAQ for Macintosh Examples****, for a complete description of the*** `UToL` ***function.)***

`AI_Read` addresses the specified analog input channel, changes the input gain to the specified gain setting, and initiates an A/D conversion. `AI_Read` waits for the conversion to complete and returns the result.

AI_Read_Scan

Function

Returns readings for all analog input channels selected by `SCAN_Setup` (E Series devices only, with or without the SC-2040 accessory).

Synopsis

C Syntax	`locus i32 AI_Read_Scan(u32 deviceNumber, i16 *reading);`
Pascal Syntax	`function AI_Read_Scan(deviceNumber : i32; var reading : i16) : i32;`
BASIC Syntax	`FN AI_Read_Scan(deviceNumber&, reading&)`

Description

reading is an array of readings from each sampled analog input channel. The length of the **reading** array is equal to the number of channels selected in the `SCAN_Setup` **numChans** parameter. Range of elements in **reading** depends on your device A/D converter resolution and the unipolar/bipolar selection you make for a given channel.

`AI_Read_Scan` samples the analog input channels selected by `SCAN_Setup` at half the maximum rate permitted by your hardware.

AI_Setup

Function

Selects an analog input channel and gain setting for externally pulsed conversion operations.

Synopsis

C Syntax	`locus i32 AI_Setup(u32 deviceNumber, u32 channel, u32 gain);`
Pascal Syntax	`function AI_Setup(deviceNumber : i32; channel : i32; gain : i32) : i32;`
BASIC Syntax	`FN AI_Setup(deviceNumber&, channel&, gain&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog input channel number. If SCXI is being used, you must use the appropriate analog input channel on the DAQ board that corresponds to the desired SCXI channel. Please refer to Chapter 7, *SCXI Functions*, for more information on SCXI channel assignments.

Range: 0 through *n*-1, where *n* is the number of analog input channels available.

gain is the gain setting to be used for the selected channel. This gain setting applies only to the DAQ board; if SCXI is used, any gain desired at the SCXI module must be established either by setting jumpers on the module or by calling `SCXI_Set_Gain`. See Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for the valid gain settings for your hardware. The DAQCard-500 and DAQCard-700 ignore **gain**.

`AI_Setup` addresses the specified analog input channel and changes the input gain to the specified gain setting. `AI_Setup`, in conjunction with `AI_Check` and `AI_Clear`, is useful for externally timed A/D conversions.

If your application calls `AI_Read` after calling `AI_Setup` and either of the channel or gain parameters in the `AI_Read` call differ from those in the `AI_Setup` call, then `AI_Setup` must be called again if `AI_Check` is to work properly. On an E Series board, if your application calls `AI_Read`, after calling `AI_Setup` , your application must call `AI_Setup` again for `AI_Check` to work properly.

AI_VScale

Function

Converts the binary result from an `AI_Read` call to the actual input voltage.

Synopsis

C Syntax	`locus i32 AI_VScale(u32 deviceNumber, u32 channel, u32 gain, f64 gainAdjust, f64 offset, i32 reading, f64 *voltage);`
Pascal Syntax	`function AI_VScale(deviceNumber : i32; channel : i32; gain : i32; gainAdjust : f64; offset : f64; reading : i32; var voltage : f64) : i32;`
BASIC Syntax	`FN AI_VScale(deviceNumber&, channel&, gain&, gainAdjust#, offset#, reading&, voltage&)`

Description

channel is the onboard channel or AMUX channel on which NI-DAQ took the binary reading using `AI_Read`. For devices other than the E Series devices, this parameter is ignored because the scaling calculation is the same for all of the channels. However, you are encouraged to pass the correct channel number.

gain is the gain setting that you used to take the analog input reading. If you used SCXI to take the reading, this gain parameter should be the product of the gain on the SCXI module channel and the gain that the DAQ device used. Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for valid gain settings. Use of invalid gain settings causes NI-DAQ to return an error unless you are using SCXI. If you call `AI_VScale` for the DAQCard-500, NI-DAQ always ignores the gain; if you call `AI_VScale` for the DAQCard-700, NI-DAQ ignores the gain unless you are using SCXI.

gainAdjust is the multiplying factor to adjust the gain. Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for the procedure for determining **gainAdjust**. If you do not want to do any gain adjustment—for example, use the ideal gain as specified by the **gain** parameter—set **gainAdjust** to 1.

offset is the binary offset that needs to be subtracted from the **reading**. Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for the procedure for determining offset. If you do not want to do any offset compensation, set **offset** to 0.

reading is the result of the A/D conversion returned by `AI_Read`.

voltage is the variable in which NI-DAQ returns the input voltage converted from **reading**.

Note to C Programmers—voltage *is a pass-by-reference parameter.*

Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for the formula `AI_VScale` uses to calculate **voltage** from **reading**.

If your device polarity and range settings differ from the default settings shown in the `Init_DA_Brds` function, be sure to call `AI_Configure` to inform the driver of the correct polarity and range before using this function.

You must use the `SCAN_Setup` function prior to invoking this function.

You cannot use external signals to control A/D conversion timing and use this function at the same time.

Multiple-Channel Analog Input (MAI)

The remainder of this chapter describes the Multiple-Channel Analog Input functions used with the NB-A2000, NB-A2100, and NB-A2150 boards for Macintosh computers. The Multiple-Channel Analog Input functions cover single A/D conversions simultaneously sampled on a group of channels.

NB-A2000 Analog Input

The NB-A2000 contains four simultaneously sampled, single-ended analog input channels numbered 0 through 3. These input channels are multiplexed into a single unity gain stage followed by a 1-µs conversion time, 12-bit resolution, ADC.

The signal range of each input channel is ±5 V when DC coupling is selected and ±5 V peak AC with ± 25 VDC offset when AC coupling is selected.

A/D conversions can be initiated through software or by applying active-low pulses to the SAMPCLK* input on the NB-A2000 I/O connector or active high pulses to the CLOCKI RTSI bus input. The 1024-word deep FIFO memory on the board stores up to 1024 A/D conversion results.

NB-A2100 Analog Input

The NB-A2100 contains two simultaneously sampled analog input channels numbered 0 and 1. These 16-bit resolution A/D channels have 64-times oversampling delta-sigma modulating ADCs and digital anti-aliasing filters

for extremely high-accuracy data acquisition. The input also has a software-programmed switch for AC or DC coupling of the input signals.

The signal range for each input channel is ±2.828 V (2 Vrms) with a maximum input voltage rating of ±10 V powered on or off.

The ADCs can be run at 22.05 kHz, 24 kHz, 32 kHz, 44.1 kHz, or 48 kHz conversion rates. A 32-bits wide, 16 words deep FIFO memory on the board stores up to 32 A/D conversion results if one channel is being sampled, or 16 A/D conversion results for each channel if both analog input channels are being sampled.

The A/D conversion data can be sent serially over the RTSI bus to other National Instruments boards, such as the NB-DSP2300 digital signal processing board.

NB-A2150 Analog Input

The NB-A2150 contains four simultaneously sampled analog input channels numbered 0 through 3. These 16-bit resolution A/D channels have 64-times oversampling delta-sigma modulating ADCs and digital anti-aliasing filters for extremely high-accuracy data acquisition. The input also has a software-programmed switch for AC or DC coupling of the input signals.

The signal range for each input channel is ±2.828 V (2 Vrms) with a maximum input voltage rating of ±10 V powered on or off.

The ADCs can be run at four timebases and each of these timebases is divided by 1, 2, 4, or 8 to produce 16 sample rates from which to choose. The timebase values are as follows:

NB-A2150F: 51.2 kHz, 48 kHz, 32 kHz, and 30.72 kHz
NB-A2150C: 48 kHz, 44.1 kHz, 32 kHz, and F_u
NB-A2150S: 24 kHz, 20 kHz, 16 kHz, and F_u

F_u is a user-defined sample rate and is obtained by dividing the custom installed crystal frequency by 384.

Multiple-Channel Analog Input Function Summary

Use the following functions for multiple-channel analog input operations on the NB-A2000, NB-A2100, and the NB-A2150:

`MAI_Arm` Enables/disables the NB-A2000 to take a sample of selected input channels whenever an external pulse on the sample clock input is received. If external pulses are used, data is then stored in the board's A/D FIFO for later retrieval by `MAI_Read` (NB-A2000 only).

`MAI_Clear` Clears the A/D FIFO and related analog input circuitry (NB-A2000 only).

`MAI_Coupling` Selects coupling for all channels with programmable coupling.

`MAI_Read` Returns a reading for all of the selected analog input channels. If an external sample clock is being used and `MAI_Arm` has been called, samples generated by previous sample clock pulses are returned; otherwise, the inputs are read when the call is made.

`MAI_Scale` Given an array of acquired data, converts the values in the array to the actual voltage values measured.

`MAI_Setup` Selects the analog input channels read, sets the gain per channel, and sets the multiplexing rate between channels for all analog input operations–affects single read multiple-channel analog input (MAI functions) and multiple-channel data acquisition operations (MDAQ functions).

Multiple-Channel Analog Input Application Hints

For most operations, `MAI_Read` is the only function required to perform a single scan of all the analog input channels. The NB-A2000 reads all four analog input channels by default. The NB-A2100 reads both analog input channels by default. The NB-A2150 reads all four analog input channels by default. `MAI_Scale` can subsequently be used to convert the binary values to voltage values. If you want to change the analog input channels monitored, use `MAI_Setup`. `MAI_Coupling` is used to select AC or DC coupling on the NB-A2000, NB-A2100, or NB-A2150 analog input channels. The NB-A2000 is configured for AC coupling by default, and the NB-A2100 and NB-A2150 are configured for DC coupling by default.

The default settings for NB-A2000 analog input are as follows:

- AC coupling on all input channels
- Four analog input channels (channels 0 through 3) selected
- Internal, onboard sample clock used

The default settings for NB-A2100 analog input are as follows:

- DC coupling on both input channels
- Both analog input channels (channels 0 and 1) selected

The default settings for NB-A2150 analog input are as follows:

- DC coupling on all four input channels
- All analog input channels (channels 0 through 3) selected

Note: ***The defaults shown are the default values after system startup or a*** `Board_Reset` ***call.***

Typical Multiple-Channel Analog Input Function Usage

Figure 3-3 shows the typical order for using the multiple-channel Analog Input functions, with an optional scale step, to take multiple readings. The boxes represent steps that are optional or only necessary if the current settings need to be changed.

Note: ***The defaults shown are the default values after power up or a*** `Board_Reset` ***call.***

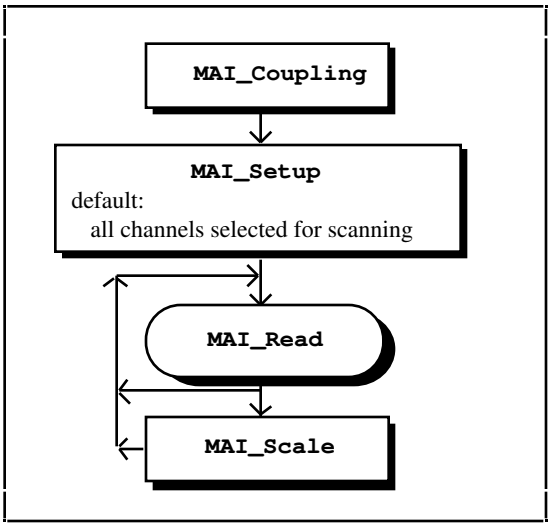

Figure 3-3. Flowchart for Multiple-Channel Analog Input Readings

After system startup or a board reset, the `MAI_Read` function returns a reading from all four of the NB-A2000 or NB-A2150 channels or both of the NB-A2100 channels. The `MAI_Setup` step is only necessary if an application needs to scan less than the default number of channels. The `MAI_Scale` step is shown as optional, although many applications perform this step for every `MAI_Read` done to convert the reading to the actual voltage values measured.

Buffered Analog Input

Buffered, multiple-channel analog input is implemented by the multiple-channel Data Acquisition functions presented in Chapter 6, *Data Acquisition Functions*. `MAI_Coupling`, `MAI_Setup`, and `MAI_Scale` are also used with the multiple-channel Data Acquisition functions.

Externally Clocked Analog Input (NB-A2000)

`MAI_Arm` and `MAI_Clear` are only used for externally clocked sampling. Use `A2000_Config` to select external sample clock, and use `MAI_Arm` to enable the NB-A2000 to sample its inputs and save the readings in the A/D FIFO whenever a sample clock edge is received. Call `MAI_Read` to retrieve the readings. `MAI_Read` returns the earliest sample in the A/D FIFO for the channels selected, or an error if no readings are present. `MAI_Clear` can be used at any time to clear the A/D FIFO or error conditions. `MAI_Arm` can be used again to disable externally clocked analog input.

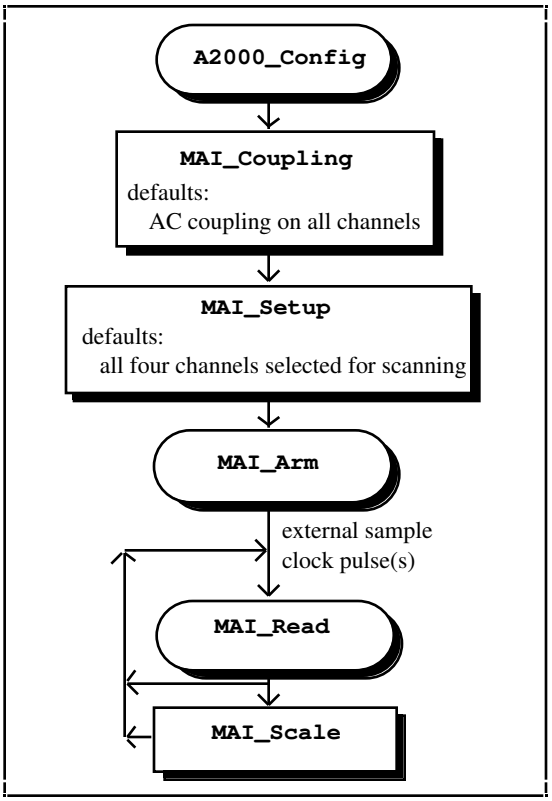

Figure 3-4. Flowchart for Externally Clocked Multiple-Channel Analog Input

---

MAI_Arm

Function

Enables/disables the NB-A2000 to take a sample of selected input channels whenever an external pulse on the sample clock input is received. If external pulses are used, data is then stored in the board's A/D FIFO for later retrieval by `MAI_Read` (NB-A2000 only).

Synopsis

C Syntax	`locus i32 MAI_Arm(u32 deviceNumber, u32 mode);`
Pascal Syntax	`function MAI_Arm(deviceNumber : i32; mode : i32) : i32;`
BASIC Syntax	`FN MAI_Arm(deviceNumber&, mode&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

mode indicates whether external pulses are used.

0: arm the board to convert on external conversion pulses.
1: disarm the board to convert on external conversion pulses.

When MAI_Arm is called, the A/D FIFO is cleared and the input signals are sampled whenever a rising edge is received on the SAMPCLK* input on the NB-A2000 I/O connector or a falling edge is received on the CLOCKI RTSI bus input. To retrieve these values, call MAI_Read.

A2000_Config must be called before MAI_Arm to select external sample clock for external conversions. A RTSI_Conn call must be made if the CLOCKI RTSI bus input is used (see Chapter 9, *RTSI Bus Trigger Functions*). After calling MAI_Arm to disarm, be sure to call A2000_Config to reset to the internal sample clock.

MAI_Clear

Function

Clears the A/D FIFO and related analog input circuitry (NB-A2000 only).

Synopsis

C Syntax	`locus i32 MAI_Clear(u32 deviceNumber);`
Pascal Syntax	`function MAI_Clear(deviceNumber : i32) : i32;`
BASIC Syntax	`FN MAI_Clear(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

MAI_Clear clears any analog input error conditions and unwanted samples from the A/D FIFO on the NB-A2000.

MAI_Coupling

Function

Selects coupling for all channels with programmable coupling.

Synopsis

C Syntax	`locus i32 MAI_Coupling(u32 deviceNumber, u32 channelCount, u16 *coupling);`
Pascal Syntax	`function MAI_Coupling(deviceNumber : i32; channelCount : i32; coupling : pi16) : i32;`
BASIC Syntax	`FN MAI_Coupling(deviceNumber&, channelCount&, coupling&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channelCount is the number of channels on the board that have programmable coupling settings.
5 for the NB-A2000 channels: ACH0, ACH1, ACH2, ACH3, and ATRIG.
2 for the NB-A2100 channels: ACH0, ACH1.
4 for the NB-A2150 channels: ACH0, ACH1, ACH2, and ACH3.

coupling is an array of length **channelCount** that selects AC or DC coupling for each analog input channel. Each value in the **coupling** array selects the setting for each channel as follows:
coupling[*i*] = 0: AC coupling.
coupling[*i*] = 1: DC coupling.
coupling[*i*] = 2: GND coupling. This selection grounds the input channel.

For the NB-A2000, the elements in **coupling** are interpreted as follows:
coupling[0]: coupling setting for ACH0.
coupling[1]: coupling setting for ACH1.
coupling[2]: coupling setting for ACH2.
coupling[3]: coupling setting for ACH3.
coupling[4]: coupling setting for ATRIG.
The GND coupling option is not available on the NB-A2000.

For the NB-A2100, the elements in **coupling** are interpreted as follows:
coupling[0]: coupling setting for ACH0.
coupling[1]: coupling setting for ACH1.
The coupling setting on both ACH0 and ACH1 must be the same on the NB-A2100.

For the NB-A2150, the elements in **coupling** are interpreted as follows:
coupling[0]: coupling setting for ACH0.
coupling[1]: coupling setting for ACH1.
coupling[2]: coupling setting for ACH2.
coupling[3]: coupling setting for ACH3.
The coupling setting on ACH0 must be the same as the coupling setting on ACH1, and the coupling setting on ACH2 must be the same as the coupling setting on ACH3 on the NB-A2150.

Note: ***All programmable channels must have a setting in the* coupling *array when* `MAI_Coupling` *is called.***

`MAI_Coupling` sets each NB-A2000, NB-A2100, or NB-A2150 analog input to the selected coupling. After system startup or a `Board_Reset` function call, the coupling of all programmable channels defaults to AC coupling on the NB-A2000 and DC coupling on the NB-A2100 and NB-A2150.

MAI_Read

Function

Returns a reading for all of the selected analog input channels. If an external sample clock is being used on the NB-A2000 and `MAI_Arm` has been called, samples generated by previous sample clock pulses are returned; otherwise, the inputs are read when the call is made.

Synopsis

C Syntax	`locus i32 MAI_Read(u32 deviceNumber, i16 *reading);`
Pascal Syntax	`function MAI_Read(deviceNumber : i32; var reading : i16) : i32;`
BASIC Syntax	`FN MAI_Read(deviceNumber&, reading&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

reading is an array of readings from each sampled analog input channel. The length of the **reading** array is equal to the number of channels selected in the `MAI_Setup` **channelCount** parameter. On the NB-A2000, the

elements of reading are 12-bit sign extended integers which range from -2,048 to +2,047. On the NB-A2100 and the NB-A2150, the elements of reading are 16-bit integers which range from -32,768 to +32,767.

`MAI_Read` samples the selected analog input channels selected by `MAI_Setup`. By default, all four NB-A2000 and NB-A2150 analog input channels and both NB-A2100 analog input channels are sampled. `MAI_Read` samples all selected input channels when the call is made unless the external sample clock is used.

If the external sample clock is used on the NB-A2000 and `MAI_Arm` has been called, `MAI_Read` returns the earliest reading stored in the A/D FIFO for the selected channels. An error is returned if no readings are stored. `MAI_Clear` can be used at any time to clear the A/D FIFO or overflow error conditions. If the NB-A2000 is configured to sample a single analog input channel (**channelCount** = 1 in `MAI_Setup`) with the external sample clock, two clock pulses must be received, because the NB-A2000 only stores samples in pairs. On return from `MAI_Read` **reading** contains two values with **reading**[0] holding the earlier sample.

MAI_Scale

Function

Given an array of acquired data, converts the values in the array to the actual voltage values measured.

Synopsis

C Syntax	`locus i32 MAI_Scale(u32 deviceNumber, u32 count, i16 *readings, f32 *voltages);`
Pascal Syntax	`function MAI_Scale(deviceNumber : i32; count : i32; readings : pi16; voltages : pf32) : i32;`
BASIC Syntax	`FN MAI_Scale(deviceNumber&, count&, readings&, voltages&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

count is the total number of scans in the values array.

readings is an array of acquired 12-bit or 16-bit binary data.

voltages is an array of 32-bit floating point values that is returned corresponding to the actual input voltages measured.

`MAI_Scale` calculates **voltages** from **readings** as follows:

For the NB-A2000: **voltages**[*i*] = **readings**[*i*] ∗ 5 / 2,048.

For the NB-A2100 and NB-A2150: **voltages**[*i*] = **readings**[*i*] ∗ 2.828/ 32,768.

Note: `MAI_Scale` ***assumes that all the settings that were in effect during the data acquisition are the same settings that are in effect when the data is being scaled. If data is logged to disk and then read later for scaling, the*** `MAI_Setup` ***function needs to be called before scaling if the settings have been changed.***

MAI_Setup

Function

Selects the analog input channels read, sets the gain per channel, and sets the multiplexing rate between channels for all analog input operations–affects single read multiple-channel analog input (MAI functions) and multiple-channel data acquisition operations (MDAQ functions).

Synopsis

C Syntax	`locus i32 MAI_Setup(u32 deviceNumber, u32 channelCount, u16 *channels, u16 *gains, u32 muxInterval, u32 timebase, u32 muxMode);`
Pascal Syntax	`function MAI_Setup(deviceNumber : i32; channelCount : i32; channels : pi16; gains : pi16; muxInterval : i32; timebase : i32; muxMode : i32) : i32;`
BASIC Syntax	`FN MAI_Setup(deviceNumber&, channelCount&, channels&, gains&, muxInterval&, timebase&, muxMode&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channelCount is the number of onboard channels to be scanned when sampling the analog input.
Valid values for the NB-A2000 and the NB-A2150:

1, 2, 4.

The following values of **channelCount** are valid for the NB-A2100:

1, 2.

channels is an integer array of length **channelCount** that contains a listing of the analog input channels to be scanned. Tables 3-3 and 3-4 outline the valid combinations and ordering of channels.

Table 3-2. Valid **channelCount** and **channels** Settings for the NB-A2000 and the NB-A2150

channelCount	channels entries	
1	**channels**[0] = 0 or 1 or 2 or 3	samples a single analog input
2	**channels**[0] = 0, **channels**[1] = 1, or **channels**[0] = 2, **channels**[1] = 3	samples two analog inputs
4	**channels**[0] = 0, **channels**[1] = 1, **channels**[2] = 2, **channels**[3] = 3	samples all analog inputs (default setting)

Table 3-3. Valid **channelCount** and **channels** Settings for the NB-A2100

channelCount	channels entries	
1	**channels**[0] = 0 or 1	samples a single analog input
2	**channels**[0] = 0, **channels**[1] = 1, or **channels**[0] = 1, **channels**[1] = 0	samples both analog inputs (default setting)

gains is an integer array of length **channelCount** that contains a gain setting for each channel selected in **channels**. The NB-A2000, NB-A2100, and NB-A2150 have a fixed gain of 1; therefore, the **gains** value is ignored for these boards.

muxInterval is the input multiplexer switching time interval, that is, the time lapse between when each successive channel in **channels** is sampled. The multiplexer switching time interval is a function of the timebase resolution selected by timebase. The actual interval in seconds is determined by the following formula:

muxInterval ∗ (timebase resolution)

For the NB-A2000, NB-A2100, and NB-A2150, set **muxInterval** to 0 because these boards simultaneously sample their input channels.

timebase is the resolution to use for the multiplexer switching interval. Because **muxInterval** is always set to 0 for the NB-A2000, NB-A2100, and NB-A2150, **timebase** is ignored.

muxMode indicates the number of external multiplexer boards connected to the board. For the NB-A2000, NB-A2100, and NB-A2150, set **muxMode** to 0 since the boards do not use an external multiplexer board.

The NB-A2000 and NB-A2150 are initially configured at system startup to sample all four input channels as shown in the last row of Table 3-2. The NB-A2100 is initially configured at system startup to sample both input channels as shown in the last row of Table 3-3. `MAI_Setup` is only needed if you want to sample fewer channels or the configured input channels have been changed.

Chapter 4
Analog Output Functions

This chapter describes the functions for single D/A conversions.

The Analog Output functions cover single D/A conversions. Multiple D/A operations functions can be performed with the Waveform Generation functions (see Chapter 10, *Waveform Generation Functions*). See Appendix A to determine what functions your board supports.

If you are using the SCXI-1124 analog output module, you need to use the SCXI functions described in Chapter 7, *SCXI Functions*, instead of the Analog Output Functions.

Analog Output

Table 4-1 summarizes the analog output characteristics of the data acquisition boards.

Table 4-1. Analog Output Characteristics Summary

Board	Analog Output Channels	DAC	Hardware Configuration for Analog Output Channels	Onboard Voltage References	Can Be Driven by External Reference Voltage Signal
NB-AO-6[1]	0–5	Double-buffered 12-bit	unipolar voltage bipolar voltage current output	+10 or +2.5 V drives the analog output channels	yes
NB-MIO-16 NB-MIO-16X	0–1	12-bit	unipolar voltage bipolar voltage	+10 V drives the analog output channels	yes
Lab and 1200 Series	0–1	12-bit	unipolar voltage (0 to +10 V) bipolar voltage (-5 to +5 V)	—	no
DAQCard-AO-2DC	0–1	12-bit	unipolar current (0 to 20 mA) unipolar voltage (0 to +10 V) bipolar voltage (-5 to +5 V)	—	no
PCI-MIO-16XE-50	0–1	12-bit	bipolar (-10 to +10 V)	+10 V drives the analog output channels	no

[1]On the NB-AO-6, each analog output channel can be immediately updated when written to, or all channels on the NB-AO-6 can be simultaneously updated at a later time by either an external update pulse or a software command (see `AO_Update`).

NB-A2100 Analog Output

The NB-A2100 contains two simultaneously updated analog output channels numbered 0 and 1. These 16-bit resolution D/A channels use 8-times oversampling digital anti-imaging filters for extremely high fidelity data output. Each channel also has a jumper to select AC or DC coupling.

The output range for each channel is ±3 V (or about 2.12 Vrms).

The DACs can be run at 16, 22.05, 24, 32, 44.1, or 48 kHz conversion rates. A 32-bit-wide, 16-word-deep FIFO memory on the board serves as a buffer to the DAC and can store 32 conversion values if one channel is being output or 16 conversion values for each channel if both channels are being output.

The D/A conversion data can be received serially over the RTSI bus from other National Instruments boards such as the NB-DSP2300 digital signal processing board.

Analog Output Function Summary

The following functions are used for analog output:

`AO_Change_Parameter`	Selects parameter settings for analog output.
`AO_Setup`	Configures each analog output channel.
`AO_Update`	Updates all analog output channels on the board to new voltage/current values.
`AO_VScale`	Converts voltage into a binary value to use with the `AO_Write` function to generate that voltage.
`AO_Write`	Writes a binary value to the analog output channel to change output current or voltage.

Analog Output Application Hints

For most purposes, `AO_Write` is the only function required to generate single analog output voltages. If needed, you can use `AO_VScale` to convert a voltage or current value to the binary value to be output.

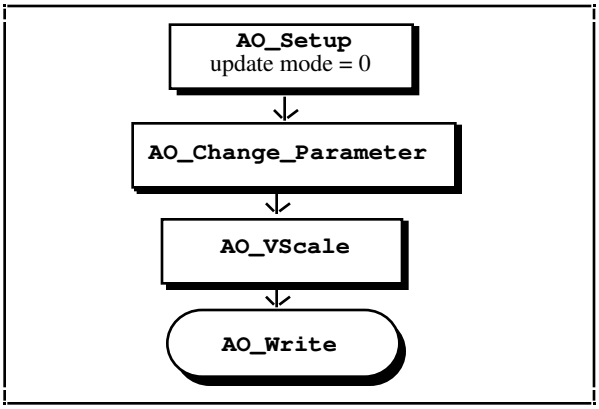

Figure 4-1. Immediate Update Analog Output Flowchart

The NB-AO-6 also supplies current outputs for each channel. For the current outputs, 0 V corresponds to 4 mA and 10 V corresponds to 20 mA when using the onboard 10 V reference.

If the jumper settings on the Lab-NB, Lab-LC, NB-MIO-16, NB-MIO-16X, or NB-AO-6 analog output circuitry have been changed from the factory settings, you need to use `AO_Setup` to update the analog output configuration information for the drivers. This update needs to be made only once per system startup. Read the `AO_Setup` description to double-check your configuration.

With `AO_Setup` you can also select whether to use delayed update on the Lab and 1200 series, NB-AO-6, PCI-MIO-16XE-50, or NB-MIO-16X (this feature is not available on the NB-MIO-16). With delayed update, you can use `AO_Write` to write to one or more analog output channels without changing the state of the analog outputs. You can then simultaneously change the state of all analog outputs at a later time by executing `AO_Update` or by applying an external update pulse.

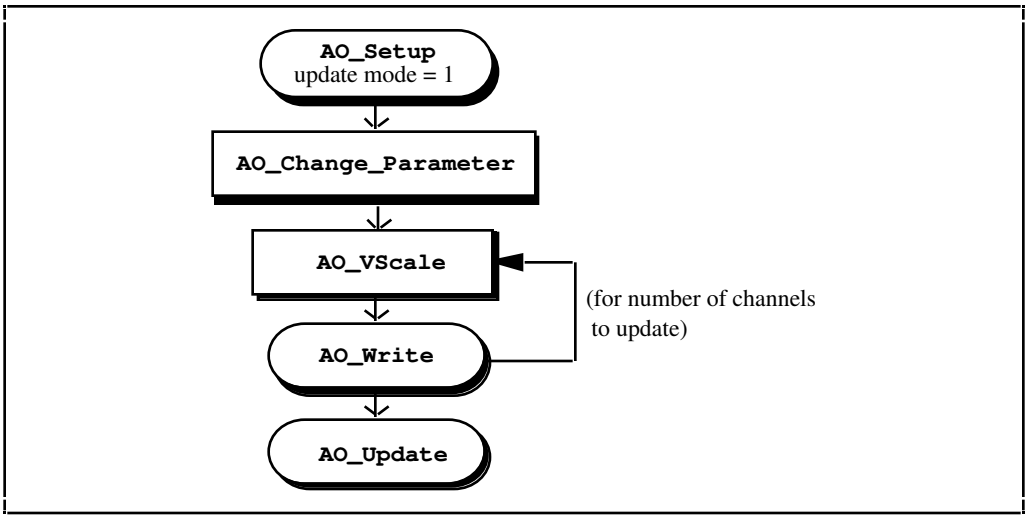

Figure 4-2. Delayed Update Analog Output Flowchart

The NB-A2100 always immediately updates the selected analog output channels with a bipolar value when `AO_Write` is called. So, the functions `AO_Setup` and `AO_Update` are not supported on the NB-A2100.

AO_Change_Parameter

Function

Selects a specific parameter setting for the analog output section of the device or an analog output channel. You can select parameters related to analog output not listed here through the `AO_Setup` function.

Synopsis

C Syntax	`locus i32 AO_Change_Parameter(u32 deviceNumber, u32 channel, u32 paramID, u32 paramValue);`
Pascal Syntax	`function AO_Change_Parameter(deviceNumber : i32; channel : i32; paramID : i32; paramValue : i32) : i32;`
BASIC Syntax	`FN AO_Change_Parameter(deviceNumber&, channel&, paramID&, paramValue&)`

Description

Legal ranges for **paramID** and **paramValue** are given in terms of constants defined in a header file. The header file you should use depends on the language you are using:

- C programmers—`NIDAQCNS.H`
- Pascal programmers—`NIDAQCNS.PAS`

Legal values for **channel** depend on the type of device you are using; analog output channels are labeled 0 through *n*-1, where *n* is the number of analog output channels on your device. You can set **channel** to -1 to indicate that you want the same parameter selection for all channels. You must set **channel** to -1 if you want to change a parameter you cannot change on per-channel basis.

Legal values for **paramValue** depend on **paramID**. The following paragraphs list features you can configure along with legal values for **paramID** with explanations and corresponding legal values for **paramValue**.

Voltage or Current Output
Some devices require separate calibration constants for voltage and current outputs. Setting the output type to voltage or current for these devices causes the driver to use the correct calibration constants and to interpret the input data correctly in `AO_VScale.` To change the output type, set **paramID** to `ND_OUTPUT_TYPE.`

Device Type	Per-Channel Selection Possible	Legal Range for paramValue	Default Setting for paramValue
DAQCard-AO-2DC	Yes	`ND_CURRENT_OUTPUT` and `ND_VOLTAGE_OUTPUT`	`ND_VOLTAGE_OUTPUT`

This function lets you customize the behavior of the analog output section of your device. You should call this function before calling NI-DAQ functions that cause output on the analog output channels. You can call this function as often as you need.

AO_Setup

Function

Configures the specified analog output channel.

Synopsis

C Syntax	`locus i32 AO_Setup(u32 deviceNumber, u32 channel, u32 outputMode, f64 outputRange, u32 updateMode, u32 updateSignal, u32 updateEdge);`
Pascal Syntax	`function AO_Setup(deviceNumber : i32; channel : i32; outputMode : i32; outputRange : f64; updateMode : i32; updateSignal : i32; updateEdge : i32) : i32;`
BASIC Syntax	`FN AO_Setup(deviceNumber&, channel&, outputMode&, outputRange#, updateMode&, updateSignal&, updateEdge&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog output channel number.
Range: 0 through *n*-1, where *n* is the number of analog output channels on the board.

outputMode selects whether the analog output channel is configured for unipolar or bipolar operation:
1: unipolar operation.
0: bipolar operation.

outputRange is the analog output channel voltage reference value. This parameter is ignored for the DAQCard-AO-2DC, Lab, and 1200 series boards since the **outputMode** determines their output range.

updateMode indicates whether a double-buffered analog output channel is updated when written to:
0: immediate update.
1: not updated when written to.

updateSignal indicates which signal is used to update the double-buffered analog output channel:
0: internal (default).
1: external.

updateEdge indicates whether the falling edge or rising edge is used (NB-AO-6 only) to update the double-buffered analog output channels:
0: falling.
1: rising.

Note: ***On the NB-AO-6, the double-buffered analog output channels are updated by the specified edge of the EXT.UPD signal; for group operations, the last channel configured specifies the appropriate setting for*** **updateEdge**.

Only the following combinations of **updateMode**, **updateSignal**, and **updateEdge** are valid on the NB-MIO-16X, Lab and 1200 series, and MIO E Series devices. An *x* indicates the value is ignored for that combination.

updateMode	updateSignal	updateEdge	Description
0	x	x	Analog output channels are updated as soon as `AO_Write` is executed.
1	1	x	For the NB-MIO-16X and all Lab and 1200 series devices except the Lab-LC , the analog output channels are updated when a low level is detected on EXTUPDATE*. For the Lab-LC, the analog output channels are updated when a high-to-low edge is detected on the EXTUPDATE* pin. For MIO E Series devices, the analog output channels are updated on an active low pulse applied to the PFI5 pin. To alter the pin and polarity, you can call the `Select_Signal` function.
1	0	x	Analog output channels are updated when `AO_Update` is executed.

`AO_Setup` stores information about the specified analog output channel on the specified board in the configuration table for that analog channel. After system startup, the analog output channel configuration tables default to the following:
outputMode = 0: bipolar (unipolar on the DAQCard-AO-2DC).
outputRange = 10 V (-5 to +5 V on the Lab and 1200 series).
updateMode = 0: immediate update.

If the physical configuration of the analog output channels on your board differs from these defaults, you must call `AO_Setup` with the actual configuration information in order for the remaining Analog Output functions to operate properly.

With `AO_Setup` you can also select whether to use delayed update on the NB-AO-6, Lab and 1200 series, MIO E Series, or NB-MIO-16X (this feature is not available on the NB-MIO-16). Delayed update is configured by setting **updateMode** to 1. You can use delayed update to use `AO_Write` to write to one or more analog output channels without changing the state of the analog outputs and then to simultaneously change the state of all analog outputs at a later time by executing `AO_Update` or by applying an external update pulse.

Note: `AO_Setup` ***replaces the*** `AO_Config` ***function used in previous versions of NI-DAQ for Macintosh.***

AO_Update

Function

Updates all analog output channels on the specified board to new voltage/current values.

Synopsis

C Syntax	`locus i32 AO_Update(u32 deviceNumber);`
Pascal Syntax	`function AO_Update(deviceNumber : i32) : i32;`
BASIC Syntax	`FN AO_Update(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`AO_Update` issues an update pulse to all analog output channels on the specified board. All analog output channel voltages/currents are then simultaneously changed to the last value written. This type of delayed update is provided for the NB-AO-6, Lab and 1200 series, MIO E Series, or NB-MIO-16X only.

AO_VScale

Function

Converts a floating point number to the appropriate binary value to use with the `AO_Write` function to generate that voltage or current.

Synopsis

C Syntax	`locus i32 AO_VScale(u32 deviceNumber, u32 channel, f64 voltage, i16 *value);`
Pascal Syntax	`function AO_VScale(deviceNumber : i32; channel : i32; voltage : f64; var value : i16) : i32;`
BASIC Syntax	`FN AO_VScale(deviceNumber&, channel&, voltage#, value&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog output channel number.
Range: 0 through *n*-1, where *n* is the number of analog output channels on the board.

voltage is the voltage, in volts, or current, in amps, to be converted to a binary value.

value is the converted binary value returned.

Using the following formula, `AO_VScale` calculates the binary value to be written to the specified analog output channel to generate an output voltage or current corresponding to **voltage**.

value = (voltage/outputRange) ∗ maxBinVal

For voltages the values of **outputRange** and maxBinVal are listed in the following table:

Device	Unipolar		Bipolar	
	outputRange	**maxBinVal**	**outputRange**	**maxBinVal**
Most devices	∗	4,096	∗	2,048
Lab and 1200 series, AO-2DC	10.0	4,096	5.0	2,048
NB-A2100	—	—	3.0	32,768
Note: ∗ ***indicates that you specify the value of*** **outputRange** ***in the*** `AO_Configure` ***function call.***				

outputRange is specified in a call to `AO_Setup`.

If you set the output type to current by calling `AO_Change_Parameter`, **voltage** indicates the current to output in amps. The values of **outputRange** and maxBinVal are listed in the following table:

Device	Unipolar	
	outputRange	**maxBinVal**
DAQCard-AO-2DC	0.002	4,096

AO_Write

Function

Writes a binary value to the analog output channel to change output current or voltage.

Synopsis

C Syntax	`locus i32 AO_Write(u32 deviceNumber, u32 channel, i32 value);`
Pascal Syntax	`function AO_Write(deviceNumber : i32; channel : i32; value : i32) : i32;`
BASIC Syntax	`FN AO_Write(deviceNumber&, channel&, value&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog output channel number.
Range: 0 through *n*-1, where *n* is the number of analog output channels on the board.

value is the digital value to be written to the analog output channel.
Range for unipolar: 0 to 4,095.
Range for bipolar: -2,048 to 2,047. (-32,768 to 32,767 on the NB-A2100)

`AO_Write` writes **value** to the DAC in the analog output channel. If the analog output channel is configured for immediate update, the output voltage or current changes immediately. Otherwise, delayed update is used and the output voltage or current changes when an update command or pulse is issued. This type of delayed update is available only on the NB-AO-6, Lab and 1200 series, MIO E Series, and NB-MIO-16X.

Chapter 5
Digital I/O Functions

This chapter describes the functions used to read from and write to digital ports, which can be addressed as a single entity or as individual digital lines. The following National Instruments boards for the Macintosh have digital I/O hardware:

- All MIO boards
- All Lab and 1200 series boards
- All DIO boards
- NB-TIO-10
- DAQCard-AO-2DC
- DAQCard-500 and DAQCard-700

Digital I/O ports on the National Instruments boards can have up to eight digital lines in width. Some of the digital I/O ports have less than eight digital lines. The name *port*, in fact, refers to a set of digital lines. In many instances, the set of digital lines is controlled as a group both for reading and writing purposes and for configuration purposes. For example, the port can be configured as either an input port or as an output port, which means that the set of digital lines making up the port all become input lines or output lines.

The digital ports are usually assigned a letter, and the digital lines making up the port are assigned numbers beginning with 0. For example, the NB-DIO-24 contains three ports of eight digital lines each. These ports are labeled PA, PB, and PC on the NB-DIO-24 I/O connector drawing, as shown in the *NB-DIO-24 User Manual*. The eight digital lines making up Port PA are labeled PA7 through PA0.

In some cases digital I/O ports can be further combined into a larger entity called a *group*. On the NB-DIO-32F, for example, any of its Ports DIOA through DIOD can be assigned to one of two groups. These groups control the digital lines of the ports assigned.

The Digital I/O functions can write to and read from both an entire port and single digital lines within the port. To write to an entire port, the digital output data is written to the port as an 8-bit value (range 0 to 255). To read from a port, the digital input data is returned as an 8-bit value. The mapping of the 8 bits to the digital I/O lines is as follows:

Bit Number	Digital I/O Line Number
7	7 Most significant bit
6	6
5	5
4	4
3	3
2	2
1	1
0	0 Least significant bit

For example, a value of 255 corresponds to all lines at a logic high level. A value of 32 corresponds to digital I/O line number 5 at a logic high level and to the remaining lines at a logic low level. In the cases where a digital I/O port has less than eight lines, the most significant bits in the 8-bit value are ignored.

Most of the digital I/O ports on the boards can be configured as either input ports or output ports. Some digital I/O ports are permanently fixed as either input ports or output ports. If a port is configured as an input port, reading that port returns the value of the digital lines. The state of the digital lines, in this case, is determined by external devices connected to those lines and driving them. If no external device is driving the lines, the lines float to some indeterminate state and can be read as either in state 0 (digital logic low) or state 1 (digital logic high). If a port is configured as an output port, writing to the port sets each digital line in the port to a digital logic high or low, depending on the pattern of the data written. In this case, these digital lines can be used to drive an external device. Many of the digital I/O ports have read-back capability; that is, if the port is configured as an output port, reading the port returns the output state of that port.

The digital I/O ports and groups on some of the boards support handshaking modes. Ports and groups can be configured for handshaking or no-handshaking. For the remainder of this chapter, no-handshaking mode is synonymous with nonlatched mode and handshaking mode is synonymous with latched mode. These two modes are described as follows:

- No-handshaking (nonlatched) mode: This mode simply changes the digital value at an output port when written to and returns a digital value from a digital input port when read. No handshaking signals are generated.

- Handshaking (latched) mode: In this mode, a digital input port latches the data present at the input port when it receives a handshake signal, and generates a handshake pulse when a digital output port is written to by the computer. The status of a port or of a group of ports can be read to determine whether an external device has accepted data written to an output port or has latched data into an input port.

The no-handshaking mode is often used for process control applications, such as controlling or monitoring relays. The handshaking mode is often used for communications applications, such as transferring data between two computers.

NB-DIO-24, DAQCard-DIO-24, NB-PRL, and Lab and 1200 Series Digital I/O

The NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series boards contain 24 bits of digital I/O. These bits are divided into a set of three digital I/O ports of eight bits each. Digital I/O on these boards is controlled by an 8255 or 82C55 programmable peripheral interface (PPI) chip. The digital I/O ports are labeled as Ports PA, PB, and PC on the I/O connector, as shown in the user manual for each board. All three ports can be configured either as input ports or output ports. These ports are referred to as Ports 0, 1, and 2 for the Digital I/O functions, where:

Port PA = Port 0
Port PB = Port 1
Port PC = Port 2

Ports 0 and 1 can be used with both latched (handshaking) mode and nonlatched (no-handshaking) mode. Port 2 can be used with nonlatched mode only. The digital lines making up Port 2 (PC) are used as handshaking lines for both Ports 0 and 1 whenever either is configured for latched mode; therefore, Port 2 is not available for Digital I/O functions whenever either Port 0 or 1 is configured for latched mode.

The NB-PRL has the same functionality as the NB-DIO-24, except that the NB-PRL has a 25-pin DSUB I/O connector so that the board can be used as a NuBus parallel interface for Centronics printers. All references to the NB-DIO-24 throughout the manual also apply to the NB-PRL.

Note: ***Using an SCXI chassis with a Lab or 1200 series board or DAQCard-DIO-24 will cause NI-DAQ to reserve some digital I/O lines. Refer to Chapter 7,*** *SCXI Functions,* ***for more information.***

NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 Series Groups

Any combination of ports 0 and 1 on the NB-DIO-24, DAQCard-DIO-24, NB-PRL, and Lab and 1200 series can be grouped together to make up larger ports. See *Digital I/O Application Hints* and `DIG_Scan_Setup` for more details.

NB-DIO-32F Digital I/O

The NB-DIO-32F contains 38 bits of digital I/O. These bits are divided into a set of four digital I/O ports of 8 bits each, a 3-bit digital input port, and a 3-bit digital output port. The 8-bit digital I/O ports are labeled as Ports DIOA, DIOB, DIOC, and DIOD on the I/O connector, as shown in the *NB-DIO-32F User Manual*. The 3-bit digital input port is labeled IN and the 3-bit digital output port is labeled OUT on the I/O connector. These ports are referred to as Ports 0 through 4 by the Digital I/O functions, where:

Port DIOA = Port 0
Port DIOB = Port 1
Port DIOC = Port 2
Port DIOD = Port 3
Ports OUT and IN = Port 4

You can configure Ports 0 through 3 as either input ports or output ports. When any of these ports is configured as an output port, it has read-back capability; that is, by reading the port, you can determine what digital value the output port is currently asserting. Port 4 is always configured for both input and output. However, because the input and output pins of Port 4 are physically separate, writing to and then reading from Port 4 does not return the value written (unless OUT1 is wired to IN1 and OUT2 to IN2 at the I/O connector).

You can also configure Ports 0 through 3 for both latched mode and nonlatched mode. If you configure the ports for latched mode, you must assign the ports to one of two handshake groups. The NB-DIO-32F I/O connector includes handshake lines for each of the two groups. These handshake lines are labeled REQ for request and ACK for acknowledge. Signals received or generated on these handshake lines affect only the ports assigned to the group.

Port 4 is always configured as an I/O port. Writing to Port 4 affects the output lines labeled OUT1, OUT2, and OUT3 on the I/O connector. Reading from Port 4 returns the digital value of the input lines labeled IN1, IN2, and IN3 on the I/O connector. These lines are mapped to the bits of the data pattern written to and read from Port 4 as follows:

Bit Number	Digital I/O Line Number
7 through 3	No significance
2	OUT3 IN3
1	OUT2 IN2
0	OUT1 IN1 Least significant bit

Port 4 cannot be configured for latched mode.

NB-DIO-32F Groups

You can assign any of the Ports 0 through 3 on the NB-DIO-32F to one of two groups for handshaking. These groups are referred to as Group 1 and Group 2. Group 1 uses handshake lines ACK1 and REQ1. Group 2 uses handshake lines ACK2 and REQ2. The ACK line is driven by the group, and the REQ line is sensed by the group. Refer to the *NB-DIO-32F User Manual* for more information.

Once ports are assigned to groups, the group acts as a single entity controlling 8, 16, or 32 digital lines simultaneously. The following assignments are valid group assignments.

Assigned Ports	Group Size
Port 0	8-bit group
Port 1	8-bit group
Port 2	8-bit group
Port 3	8-bit group
Ports 0 and 1	16-bit group
Ports 2 and 3	16-bit group
Ports 0, 1, 2, and 3	32-bit group

When you assign ports to a group, handshaking of that port is controlled by the group. These ports are then read from or written to simultaneously by writing or reading 8, 16, or 32 bits at one time from the group.

The groups can be configured for various handshake configurations. The configuration choices include level or edge-triggered handshaking, inverted or noninverted ACK and REQ lines, and a programmed transfer settling time.

NB-DIO-96 and PCI-DIO-96 Digital I/O

The NB-DIO-96 and PCI-DIO-96 boards contain 96 bits of digital I/O. These bits are divided into a set of 12 digital I/O ports of eight bits each. Digital I/O on this board is controlled by four 82C55A PPI chips. The digital I/O ports are labeled as Ports APA, APB, APC, BPA, BPB, BPC, CPA, CPB, CPC, DPA, DPB, and DPC on the I/O connector as shown in the *NB-DIO-96 User Manual* or *PCI-DIO-96 User Manual*. All 12 ports can be configured as either input ports or output ports.

These ports are referred to as Ports 0 through 11 for the Digital Input and Output functions where:

Port APA = Port 0
Port APB = Port 1
Port APC = Port 2
Port BPA = Port 3
Port BPB = Port 4
Port BPC = Port 5
Port CPA = Port 6
Port CPB = Port 7
Port CPC = Port 8
Port DPA = Port 9
Port DPB = Port 10
Port DPC = Port 11

Ports, 0, 1, 3, 4, 6, 7, 9, and 10 can be used for both latched (handshaking) and nonlatched (no-handshaking) modes. Ports 2, 5, 8, and 11 can be used only for nonlatched mode. The digital lines making up Port 2 (APC) are used as handshaking lines for Ports 0 and 1 whenever either is configured for latched mode; therefore, Port 2 is not available for Digital Input and Output functions whenever either Port 0 or Port 1 is configured for latched mode. The digital lines making up Port 5 (BPC) are used as handshaking lines for Ports 3 and 4 whenever either is configured for latched mode; therefore, Port 5 is not available for Digital Input and Output functions whenever either Port 3 or Port 4 is configured for latched mode. The digital lines making up Port 8 (CPC) are used as handshaking lines for Ports 6 and 7 whenever either is configured for latched mode; therefore, Port 8 is not available for Digital Input and Output functions whenever either Port 6 or Port 7 is configured for latched mode. The digital lines making up Port 11 (DPC) are used as handshaking lines for Ports 9 and 10 whenever either is configured for latched mode; therefore, Port 11 is not available for Digital Input and Output functions whenever either Port 9 or Port 10 is configured for latched mode.

NB-DIO-96 and PCI-DIO-96 Groups

Any combination of Ports 0, 1, 3, 4, 6, 7, 9, and 10 on the NB-DIO-96 and PCI-DIO-96 can be grouped together to make up larger ports. For example, Ports 0, 3, 9, and 10 can be programmed to make up a 32-bit handshaking port, or all eight ports can be programmed to make up a 64-bit handshaking port. See *Digital I/O Application Hints* and `DIG_Scan_Setup` for more details.

NB-MIO-16 and NB-MIO-16X Digital I/O

The NB-MIO-16 and NB-MIO-16X each contain 8 bits of digital I/O. These bits are divided into a set of two digital I/O ports of four bits each. The 4-bit digital I/O ports are labeled as Ports *DIOA* and *DIOB*. These ports are referred to as Ports 0 and 1 by the Digital I/O functions where:

Port DIOA = Port 0
Port DIOB = Port 1

You can configure Ports 0 and 1 as either input ports or output ports. Any port that you configure as an output port has read-back capability (that is, by reading the port, you can determine what digital value the output port is currently asserting).

The NB-MIO-16 and NB-MIO-16X digital I/O ports operate in nonlatched mode only.

Note: ***Using an SCXI chassis with an MIO board causes NI-DAQ to reserve some digital I/O lines. Refer to Chapter 7,*** *SCXI Functions,* ***for more information.***

PCI-MIO-16XE-50 Digital I/O

The E Series devices contain one 8-bit digital I/O port supplied by the DAQ-STC chip. This port is referred to as port 0 by the Digital I/O functions.

You can configure the entire digital port as either an input or an output port, or you can configure individual lines for either input or output. The port has read-back capability (that is, by reading the port, you can determine what digital value the output port is currently asserting). This port operates in nonlatched mode only.

Note: ***Connecting one or more AMUX-64T devices or an SCXI chassis to an E Series device renders various lines of the digital I/O port unavailable:***

One AMUX-64T device—Lines 0 and 1 are unavailable.
Two AMUX-64T devices—Lines 0, 1, and 2 are unavailable.
Four AMUX-64T devices—Lines 0, 1, 2, and 3 are unavailable.
SCXI—Lines 0, 1, 2, and 4 are unavailable.

The remaining lines of the digital I/O port are available for input or output. You should use `DIG_Line_Config` ***to configure these remaining lines.***

NB-TIO-10 Digital I/O

The NB-TIO-10 contains 16 bits of digital I/O. These bits are divided into a set of two digital I/O ports of eight bits each. Digital I/O on these ports is controlled by the Motorola MC6821 PIA chip. The 8-bit digital I/O ports are labeled as Port A and Port B. These ports are referred to as Ports 0 and 1 by the Digital I/O functions, where:

Port A = Port 0
Port B = Port 1

You can configure Ports 0 or 1 as either input ports or output ports. Either of these ports configured as an output port has read-back capability; that is, by reading the port, you can determine what digital value the output port is currently asserting. You can also configure each line on a port for direction, input or output. The NB-TIO-10 digital I/O ports operate in nonlatched mode only.

DAQCard-AO-2DC Digital I/O

The DAQCard-AO-2DC contains 16 bits of digital I/O. These bits are divided into a set of two digital I/O ports of eight bits each. The 8-bit digital I/O ports are labeled as Port A and Port B. These ports are referred to as Ports 0 and 1 by the Digital I/O functions, where:

Port A = Port 0
Port B = Port 1

You can configure Ports 0 or 1 as either input ports or output ports. Either of these ports configured as an output port has read-back capability; that is, by reading the port, you can determine what digital value the output port is currently asserting.

DAQCard-500 and DAQCard-700 Digital I/O

The DAQCard-500 and DAQCard-700 have one output port (Port 0) and one input port (Port 1) each; the DAQCard-500 ports are 4-bit ports, and the DAQCard-700 ports are 8-bit. The digital I/O ports are labeled DIN and DOUT on the I/O connector, as shown in the appropriate device user manual. The ports are referred to as ports 0 and 1 for the Digital I/O functions, in which:

- DOUT = port 0
- DIN = port 1

You can program ports 0 and 1 for nonlatched (no-handshaking) mode only. You can use port 0 for nonlatched digital output mode. You can use port 1 for nonlatched digital input mode.

Note: ***Using an SCXI chassis with the DAQCard-700 renders digital lines 4, 5, 6, and 7 of port 0 and line 6 of port 1 unavailable. Refer to Chapter 7,*** *SCXI Functions****, for more information.***

SCXI Signal Conditioning Hardware

You can use the following digital SCXI modules with the MIO boards, DIO boards, and Lab boards:

- SCXI-1160 16-channel electromechanical SPDT relay module
- SCXI-1161 8-channel electromechanical SPDT relay module
- SCXI-1162 32-channel optically isolated digital input module
- SCXI-1162HV 32-channel high-voltage optically isolated digital input module
- SCXI-1163 32-channel optically isolated digital output module
- SCXI-1163R 32-channel optically isolated digital solid-state relay module

These modules do not work with the NB-TIO-10.

If your SCXI modules are configured for Multiplexed (or Serial) mode, you must use the SCXI functions in Chapter 7, *SCXI Functions*, to read and write digital patterns. If your SCXI modules are configured for Parallel

mode, you can use either the SCXI functions in Chapter 7 or the `DIG_In_Port` and `DIG_Out_Port` functions in this chapter. Please refer to the *SCXI Modules and Compatible Data Acquisition Boards* section in Chapter 7 for more information on which ports drive the modules in Parallel mode.

Digital I/O Function Summary

Use the following functions for digital I/O operations on the NB-DIO-96, PCI-DIO-96, DAQCard-AO-2DC, NB-DIO-24, DAQCard-DIO-24, DAQCard-500, DAQCard-700, NB-PRL, Lab and 1200 series, NB-DIO-32F, NB-MIO-16, NB-MIO-16X, and NB-TIO-10:

`DIG_In_Line`	Returns the digital logic state of the specified digital input line in the specified port.
`DIG_In_Port`	Reads digital input data from the specified digital I/O port.
`DIG_Line_Config`	Configures the specified line in the specified port for the direction (input or output) selected.
`DIG_Out_Line`	Sets or clears the specified digital output line in the specified digital port.
`DIG_Out_Port`	Writes digital output data to the specified digital port.
`DIG_Prt_Config`	Configures the specified port for direction (input or output) and handshake mode.
`DIG_Prt_Status`	Returns a status word indicating the handshake status of the specified port (NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series, and NB-PRL only).

Use the following functions for digital I/O group operations on the NB-DIO-32F:

`DIG_Grp_Config`	Configures the specified group for port assignment, direction (input or output), and size.
`DIG_Grp_Mode`	Configures the group handshake signal modes.
`DIG_Grp_Status`	Returns a status word indicating the handshake status of the specified group.
`DIG_In_Group`	Reads digital input data from the specified digital group.
`DIG_Out_Group`	Writes digital output data to the specified digital group.

Use the following functions for digital I/O group operations on the NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series, and NB-DIO-32F:

`DIG_Blk_Check`	Checks to see if the current buffered digital input or output operation is complete.
`DIG_Blk_Clear`	Clears the current buffered digital input or output operation for the specified group.
`DIG_Blk_Start`	Reads/writes a specified number of digital data patterns to/from a digital I/O group.
`DIG_Scan_Setup`	Configures the specified group for port assignment, direction (input or output), and size (NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series only).

Digital I/O Application Hints

Nonlatched Digital I/O

All boards that support digital I/O can be used for nonlatched digital I/O. NI-DAQ for Macintosh expects nonlatched digital I/O to be used by default. For this case, `DIG_Prt_Config` can be used to configure port direction (input by default).

`DIG_In_Port`, `DIG_In_Line`, `DIG_Out_Port`, and `DIG_Out_Line` can be used to read from and write to ports and individual lines.

Latched Digital I/O with the NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, and Lab and 1200 Series

To use handshaking with the NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series, you must call `DIG_Prt_Config` to configure a port for handshaking. For output handshaking, you may execute `DIG_Prt_Status` to see that if port is ready for output before `DIG_Out_Port` is executed. For input handshaking, you must execute `DIG_Prt_Status` to see if the port has data to be read before `DIG_In_Port` is executed. For handshaking, `DIG_In_Line` and `DIG_Out_Line` should not be used.

Latched Digital I/O with the NB-DIO-32F

To use handshaking with the NB-DIO-32F, use only the group functions. These functions can perform handshaking of 8, 16, or 32 bits at a time. You must use `DIG_Grp_Config` to enable handshaking and to configure handshaking group size and direction. You may execute `DIG_Grp_Mode` to specify handshaking modes other than the default handshaking mode. For output handshaking, you must execute `DIG_Grp_Status` and indicate that the group is ready for output before `DIG_Out_Group` is executed. For input handshaking, you must call `DIG_Grp_Status` and indicate that the port has data to be read before `DIG_In_Group` is executed.

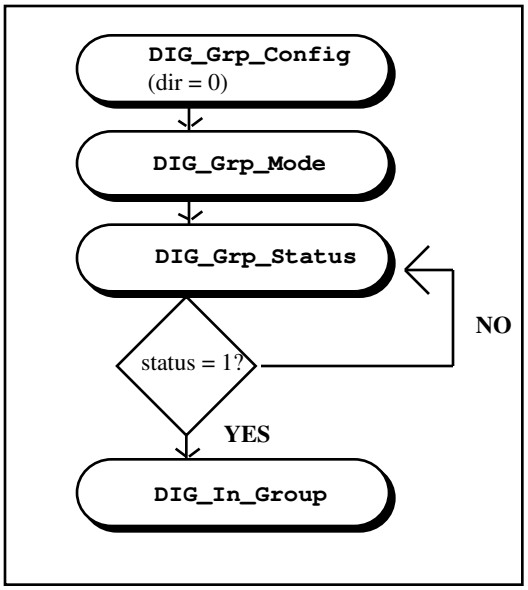

Figure 5-1. Flowchart for Latched Digital Group Input

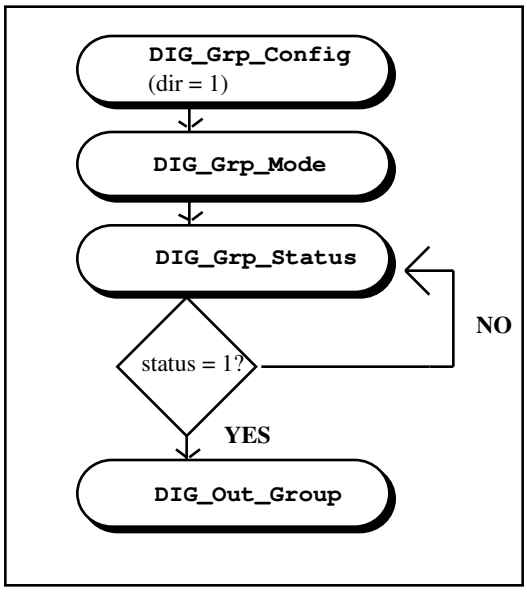

Figure 5-2. Flowchart for Latched Digital Group Output

Buffered Digital I/O with the NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, and Lab and 1200 Series

DIG_Blk_Start and DIG_Blk_Check perform buffered digital I/O operations to transfer a block of digital values between a group of digital ports and a user buffer. DIG_Scan_Setup must be executed to assign ports to a group and to configure the direction of the group. DIG_Blk_Start initiates the buffered digital I/O operation, and DIG_Blk_Check returns the completion status of the buffered digital I/O process. DIG_Blk_Start performs both input and output operations. If the group is configured for output, a user-defined buffer of data is passed to DIG_Blk_Start to output. If the group is configured for input, a user-defined buffer is passed to DIG_Blk_Start to be filled with input data. DIG_Blk_Start returns immediately after initiating the buffered digital transfer. DIG_Blk_Check returns the completion status of the operation. If a digital output transfer is initiated by DIG_Blk_Start, then the transfer is complete when DIG_Blk_Check returns status = 1. If a digital input transfer is initiated by DIG_Blk_Start, then the data is available in the user-defined buffer specified in DIG_Blk_Start when DIG_Blk_Check returns with status = 1. You should execute DIG_Blk_Clear when the buffered digital I/O operation is complete. When you use block function calls on these digital boards, you must use external handshaking signals for any buffered digital I/O operations.

Buffered Digital I/O with the NB-DIO-32F

DIG_Blk_Start and DIG_Blk_Check perform buffered digital I/O operations to transfer a block of digital values between a group of digital ports and a user buffer. DIG_Grp_Config must be executed to assign ports to a group and to configure the direction of the group. DIG_Blk_Start initiates the buffered digital I/O operation, and DIG_Blk_Check returns the completion status of the buffered digital I/O process. DIG_Blk_Start performs both input and output operations. If the group is configured for output, a user-defined buffer of data is passed to DIG_Blk_Start to output. If the group is configured for input, a user-defined buffer is passed to DIG_Blk_Start to be filled with input data. DIG_Blk_Start returns immediately after initiating the buffered digital transfer. DIG_Blk_Check returns the completion status of the operation. If a digital output transfer is initiated by DIG_Blk_Start, then the transfer is complete when DIG_Blk_Check returns status = 1. If a digital input transfer is initiated by DIG_Blk_Start, then the data is available in the user-defined buffer specified in DIG_Blk_Start when DIG_Blk_Check returns with status = 1. You should execute DIG_Blk_Clear when the buffered digital I/O operation is complete.

Normally, buffered digital I/O uses external handshaking signals to control the rate of input or output. If a DMA is present in the system, timed buffered digital I/O can be implemented by supplying timebase and interval values when executing `DIG_Blk_Start`. This interval specifies the amount of time to elapse between subsequent reads/writes for a digital group. External handshaking signals should not be used with timed buffered digital I/O operations. Only the NB-DIO-32F supports timed buffered digital I/O.

Example applications that perform buffered digital I/O are included on your NI-DAQ for Macintosh diskettes (see Chapter 11, *NI-DAQ for Macintosh Examples*).

DIG_Blk_Check

Function

Checks to see if the current buffered digital input or output operation is complete (NB-DIO-96, PCI-DIO-96, NB-DIO-32F, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series only).

Synopsis

C Syntax	`locus i32 DIG_Blk_Check(u32 deviceNumber, u32 group, u16 *status);`
Pascal Syntax	`function DIG_Blk_Check(deviceNumber : i32; group : i32; var status : i16) : i32;`
BASIC Syntax	`FN DIG_Blk_Check(deviceNumber&, group&, status&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is the handshake group number.

Range: 0 through *n*-1, where *n* is the number of groups on the board.

status indicates whether the current buffered digital input or output for this group is complete.

1: digital input or output is complete.
0: digital input or output is not yet complete.

If a digital output transfer is initiated by `DIG_Blk_Start`, then the output transfer is complete when `DIG_Blk_Check` returns **status** = 1.

If a digital input transfer is initiated by `DIG_Blk_Start`, then the data is available in the buffer specified in `DIG_Blk_Start` when `DIG_Blk_Check` returns with **status** = 1.

DIG_Blk_Clear

Function

Clears the current buffered digital input or output operation for the specified group (NB-DIO-96, PCI-DIO-96, NB-DIO-32F, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series only).

Synopsis

C Syntax	`locus i32 DIG_Blk_Clear(u32 deviceNumber, u32 group);`
Pascal Syntax	`function DIG_Blk_Clear(deviceNumber : i32; group : i32) : i32;`
BASIC Syntax	`FN DIG_Blk_Clear(deviceNumber&, group&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is the handshake group number.
Range: 0 through *n*-1, where *n* is the number of groups on the board.

DIG_Blk_Start

Function

Reads/writes a specified number of digital data patterns to/from a digital I/O group (NB-DIO-96, PCI-DIO-96, NB-DIO-32F, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series only).

Synopsis

C Syntax	`locus i32 DIG_Blk_Start(u32 deviceNumber, u32 group, u32 direction, u8 *buffer, u32 count, u32 interval, u32 timebase);`
Pascal Syntax	`function DIG_Blk_Start(deviceNumber : i32; group : i32; direction : i32; buffer : pi16; count : i32; interval : i32; timebase : i32) : i32;`
BASIC Syntax	`FN DIG_Blk_Start(deviceNumber&, group&, direction&, buffer&, count&, interval&, timebase&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is the handshake group number.
Range: 0 through *n*-1, where *n* is the number of groups on the board.

direction selects the direction of the digital transfer.
0: data is to be input.
1: data is to be output.

buffer is an array to be used for the digital transfer. If the group is configured for input, then buffer is an array to be filled with digital data patterns. If the group is configured for output, then **buffer** is an array of digital data patterns to be output. Each element of **buffer** corresponds to an 8-bit value, 16-bit value (NB-DIO-32F only), or 32-bit value (NB-DIO-32F only)–depending on the group size–and is mapped to the digital lines.

For the NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series, the elements of **buffer** are sequentially mapped to the digital ports making up the group. For example, if the **portList** specified in the `DIG_Scan_Setup` call uses 0, 4, 1, the first data value maps to Port 0, the second data value maps to Port 4, the third data value maps to Port 1, and so on.

For the NB-DIO-32F, the elements of **buffer** are sequentially mapped to the digital ports making up the group in the following way:

- If the group contains one port, the low-order eight bits of the pattern are written to or read from that port. For DMA input or output, such as required for timed buffered digital I/O (**interval** > 0), only Port 0 can be

contained in the group if an NB-DMA-8-G is used. If an NB-DMA2800 is used, either Port 0 or Port 2 can be contained in the group.

- If the group contains two ports, the low-order 16 bits of the pattern are written to or read from the ports. If the group contains Ports 0 and 1, the low-order eight bits map to Port 1, and the next eight bits map to Port 0. If the group contains Ports 2 and 3, the low-order eight bits map to Port 3, and the next eight bits map to Port 2. The two ports are written to simultaneously. For DMA input or output, such as required for timed buffered digital I/O (**interval** > 0), only Ports 0 and 1 can be contained in the group if an NB-DMA-8-G is used. If an NB-DMA2800 is used, either Ports 0 and 1 or Ports 2 and 3 can be contained in the group.

- If the group contains four ports, all 32 bits of the pattern are written to the ports. The least significant eight bits map to Port 3, the next 8 bits map to Port 2, the next eight bits map to Port 1, and the most significant bits map to Port 0. The four ports are written to or read from simultaneously.

Pascal Note: ***If the group size is 8-bit or 16-bit,*** **buffer** ***must be a pointer to an integer array (***`buffer : ^integer`***). For 8-bit groups, each 8-bit pattern occupies 16 bits in the array. If the group size is 32 bits, then*** **buffer** ***must be a pointer to a*** **longint** ***(***`buffer : ^longint`***).*** `DIG_Blk_Check` ***unpacks and aligns the data returned by NI-DAQ for Macintosh.***

count is the number of bytes to be written to or read from **buffer**.
Range: 2^{32}-1 for interrupts.
2^{24}-1 for DMA.

interval is the amount of time to elapse between each digital transfer, thereby permitting timed input or output. Timed interval updating is used only with the NB-DIO-32F.
Range: 0 for external handshaking; 2 through 65,535 for timed input or output.

To perform timed digital I/O, set **interval** to a non-zero value. You also must have an NB-DMA-8-G or NB-DMA2800 in the system to supply counter/timers to control timed digital I/O. External handshaking **interval** =0 does not require an NB-DMA-8-G or NB-DMA2800.

The **interval** is a function of the timebase resolution. The actual interval is calculated as follows:

interval ∗ (timebase resolution)

where the timebase resolution for each value of **timebase** is given in the **timebase** discussion that follows. For example, if **interval** = 25 and **timebase** = 2, then the output interval is 25 ∗ 10 µs = 250 µs.

timebase selects resolution to be used for the interval counter. The **timebase** parameter has the following possible values:
1: 1-MHz clock used as timebase (1-µs resolution).
2: 100-kHz clock used as timebase (10-µs resolution).
3: 10-kHz clock used as timebase (100-µs resolution).
4: 1-kHz clock used as timebase (1-ms resolution).
5: 100-Hz clock used as timebase (10-ms resolution).
6: SOURCE1 used as timebase.
7: SOURCE2 used as timebase.
8: SOURCE3 used as timebase.
9: SOURCE4 used as timebase.
10: SOURCE5 used as timebase.

SOURCE1 through SOURCE5 are timing signals available on the NB-DMA-8-G and NB-DMA2800. See the description of NB-DMA-8-G and NB-DMA2800 counters and timers in Chapter 8, *Counter/Timer Functions*, for more information about these signals. If the interval is to be externally controlled by the handshaking signals of the group (**interval** = 0), then the **timebase** parameter is ignored and can be any value.

`DIG_Blk_Start` initiates reading or writing of digital group data patterns sequentially from/to the group on the specified board and returns immediately. `DIG_Blk_Check` should be called to determine when the transfer completes. If a digital output transfer is initiated by `DIG_Blk_Start`, then the transfer is complete when `DIG_Blk_Check` returns **status** = 1. If a digital input transfer is initiated by `DIG_Blk_Start`, then the data is available in **buffer** when `DIG_Blk_Check` returns with **status** = 1. All ports in this group are updated/read simultaneously for each pattern in the array. If the specified group has not been configured for the

appropriate direction of transfer, the operation is not performed and an error is returned. If no ports have been assigned to the specified group, the operation is not performed and an error is returned. For the NB-DIO-32F, `DIG_Grp_Config` must be executed to assign ports to a group and to configure the group for the appropriate direction of transfer. For the NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series, `DIG_Scan_Setup` must be executed to assign ports to a group and to configure the group for the appropriate direction of transfer.

Note: ***You cannot use boards equipped with the 8255 PPI (NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series, and NB-DIO-96) in non-latched or no-handshaking mode for block transfers.***

DIG_Grp_Config

Function

Configures the specified group for port assignment, direction (input or output), and size (NB-DIO-32F only).

Synopsis

C Syntax	`locus i32 DIG_Grp_Config(u32 deviceNumber, u32 group, u32 groupSize, u32 port, u32 direction);`
Pascal Syntax	`function DIG_Grp_Config(deviceNumber : i32; group : i32; groupSize : i32; port : i32; direction : i32) : i32;`
BASIC Syntax	`FN DIG_Grp_Config(deviceNumber&, group&, groupSize&, port&, direction&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is the group to be configured.
Range: 0 through *n*-1, where *n* is the number of groups supported by the board.

groupSize selects the size of the group. The following values are permitted for **groupSize**:
0: 0 ports assigned – unassign any previously assigned ports.
1: 1 port assigned (8-bit group).
2: 2 ports assigned (16-bit group).
4: 4 ports assigned (32-bit group).

port selects the digital I/O ports assigned to the group. The value of **port** depends on the value of **groupSize**:
If **groupSize** = 1 **port** = 0 assigns Port 0 to the group.
port = 1 assigns Port 1 to the group.
port = 2 assigns Port 2 to the group.
port = 3 assigns Port 3 to the group.

If **groupSize** = 2 **port** = 0 assigns Ports 0 and 1 to the group.
port = 2 assigns Ports 2 and 3 to the group.
If **groupSize** = 4 **port** = 0 assigns Ports 0, 1, 2 and 3 to the group.

direction selects the direction, input or output, for which the group is to be configured.
0: **group** is configured as an input group (default).
1: **group** is configured as an output group.

`DIG_Grp_Config` configures the specified group according to the specified port assignment and direction. If **groupSize** is 0, any ports assigned to the group are released from the group and the group handshake circuitry is cleared. If **groupSize** is 1, 2, or 4, then the specified ports are assigned to the group and are configured for the

specified direction. Ports assigned to a group are subsequently written to or read from as a group using the `DIG_In_Group` and `DIG_Out_Group` functions. Any ports assigned to a group can no longer be accessed through any of the non-group calls listed previously in this description.

After system startup, no ports are assigned to groups.

See the *NB-DIO-32F User Manual* for group handshake timing.

DIG_Grp_Mode

Function

Configures the group handshake signal modes (NB-DIO-32F only).

Synopsis

C Syntax	`locus i32 DIG_Grp_Mode(u32 deviceNumber, u32 group, u32 pulseOrLevel, u32 edge, u32 requestPolarity, u32 acknowledgePolarity, u32 settlingTime);`
Pascal Syntax	`function DIG_Grp_Mode(deviceNumber : i32; group : i32; pulseOrLevel : i32; edge : i32; requestPolarity : i32; acknowledgePolarity : i32; settlingTime : i32) : i32;`
BASIC Syntax	`FN DIG_Grp_Mode(deviceNumber&, group&, pulseOrLevel&, edge&, requestPolarity&, acknowledgePolarity&, settlingTime&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is the group to be configured for particular handshake modes.
Range: 0 through *n*-1, where *n* is the number of groups on the board.

pulseOrLevel indicates whether the group is to be configured for level or pulsed (edge-triggered) handshake signals.
0: group is configured for level handshake signals.
1: group is configured for pulsed handshake signals.

edge indicates whether the group is to be configured for rising edge or falling edge pulsed signals. **edge** is valid only if **pulseOrLevel** is 1.
0: group is configured for rising (low-to-high) edge pulsed handshake signals.
1: group is configured for falling (high-to-low) edge pulsed handshake signals.

requestPolarityindicates whether the group request signal is to be active high or active low.
0: group is configured for active high (noninverted) request handshake signal polarity.
1: group is configured for active low (inverted) request handshake signal polarity.

acknowledgePolarity indicates whether the group acknowledge handshake signal is to be active high or active low.
0: group is configured for active high (noninverted) acknowledge handshake signal polarity.
1: group is configured for active low (inverted) acknowledge handshake signal polarity.

settlingTime selects the data settling time for the group. The value of **settlingTime** is the number of 100-ns intervals allowed for data settling.
Range: 0 through 7.
0: no settling time.
7: 700-ns settling time.

DIG_Grp_Mode configures the group handshake signals according to the specified port assignment and direction. After system startup, the default handshake mode for each group is as follows:

pulseOrLevel = 0: level handshake signals.
edge = 0: edge parameter not valid because **pulseOrLevel** = 0.
requestPolarity = 0: request handshake signal is not inverted (active high).
acknowledgePolarity = 0: acknowledge handshake signal is not inverted (active high).
settlingTime = 0: no settling time.

DIG_Grp_Mode needs to be called only if different handshake modes are required. Refer to the *NB-DIO-32F User Manual* for handshake timing and mode information.

DIG_Grp_Status

Function

Returns a status word indicating the handshake state of the specified group (NB-DIO-32F only).

Synopsis

C Syntax	`locus i32 DIG_Grp_Status(u32 deviceNumber, u32 group, u16 *status);`
Pascal Syntax	`function DIG_Grp_Status(deviceNumber : i32; group : i32; var status : i16) : i32;`
BASIC Syntax	`FN DIG_Grp_Status(deviceNumber&, group&, status&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is the handshake group number.
Range: 0 through *n*-1, where *n* is the number of groups on the board.

status returns the handshake status of the group. The significance of **status** depends on the configuration of the group. If the group is configured as an input group, **status** = 1 indicates that data has been latched into the ports making up that group. If the group is configured as an output group, **status** = 1 indicates that an external device has latched the output of the ports making up the group and that new data can be written to the group.

DIG_Grp_Status reads the handshake status of the specified group, and returns an indication of the group status in **status**. DIG_Grp_Status, along with DIG_Out_Group and DIG_In_Group, facilitates handshaking of digital data between systems. If the specified group is configured as an input group and DIG_Grp_Status returns **status** = 1, then DIG_In_Group can executed to retrieve the data an external device has latched in. If the specified group is configured as an output group and DIG_Grp_Status returns **status** = 1, then DIG_Out_Group can be called to write the next piece of data to the external device. If the specified group is not assigned any ports, then an error code and **status** = 0 are returned.

DIG_Grp_Config must be called to assign ports to a group and to configure a group for data direction. Group configuration is discussed in the DIG_Grp_Config description earlier in this chapter.

The state of **status** corresponds to the NB-DIO-32F DRDY bit and signal. Refer to the *NB-DIO-32F User Manual* for handshake timing details.

DIG_In_Group

Function

Reads digital input data from the specified digital group (NB-DIO-32F only).

Synopsis

C Syntax	`locus i32 DIG_In_Group(u32 deviceNumber, u32 group, u32 *pattern);`
Pascal Syntax	`function DIG_In_Group(deviceNumber : i32; group : i32; var pattern : i32) : i32;`
BASIC Syntax	`FN DIG_In_Group(deviceNumber&, group&, pattern&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is the handshake group number.

Range: 0 through *n*-1, where *n* is the number of groups on the board.

pattern returns the digital data read from the ports in the specified group.

Range: 0 through 255 for group size = 1.
0 through 65,535 for group size = 2.
0 through 2^{32}-1 for group size = 4.

pattern is an 8-bit, 16-bit, or 32-bit value, depending on the group size, and is mapped to the digital input lines. The **pattern** is mapped as follows to the digital input ports that make up the group:

- If the group contains one port, the eight bits read from that port are returned in the low-order eight bits of **pattern**.
- If the group contains two ports, the 16 bits read from the ports are returned in the low-order 16 bits of **pattern**. If the group contains Ports 0 and 1, the value read from Port 1 is returned in the low-order eight bits and the value read from Port 0 is returned in the next eight bits. If the group contains Ports 2 and 3, the value read from Port 3 is returned in the low-order eight bits and the value read from Port 2 is returned in the next eight bits. The two ports are read from simultaneously.
- If the group contains all four ports, all 32 bits read from the ports are returned in **pattern**. The least significant eight bits are read from Port 3, the next eight bits are read from Port 2, the next eight bits are read from Port 1, and the most significant eight bits are read from Port 0. The four ports are read from simultaneously.

Note: ***The MOST significant bits are read from the LOWEST numbered port, and the LEAST significant bits are read from the HIGHEST numbered port.***

`DIG_In_Group` returns the digital data from the group on the specified board. All ports making up the group are read simultaneously. If the group is configured as an input group, reading that group returns the digital logic state of the lines of the ports in the group as some external device is driving them. If the group is configured as an output group and has read-back capability, reading the group returns the output state of that group. If no ports have been assigned to the group, the operation is not performed and an error code is returned. `DIG_Grp_Config` must be called to assign ports to a group and to configure the group as an input or output group.

DIG_In_Line

Function

Returns the digital logic state of the specified digital input line in the specified port.

Synopsis

C Syntax	`locus i32 DIG_In_Line(u32 deviceNumber, u32 port, u32 line, u16 *state);`
Pascal Syntax	`function DIG_In_Line(deviceNumber : i32; port : i32; line : i32; var state : i16) : i32;`
BASIC Syntax	`FN DIG_In_Line(deviceNumber&, port&, line&, state&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

port is the digital I/O port number.
Range: 0 through *n*-1, where *n* is the number of digital I/O ports on the board.

line is the digital line to be read.
Range: 0 through *k*-1, where *k* is the number of digital I/O lines making up the port.

state returns the digital logic state of the specified line.
1: digital line is at a digital logic low.
0: digital line is at a digital logic high.

`DIG_In_Line` returns the digital logic state of the specified digital line in the specified port. If the specified port is configured as an input port, the state of the specified line is determined by how some external device is driving it. If the port is configured as an output port and the port has read-back capability, the state of the line is determined by how that port itself is driving it.

Note: `DIG_Prt_Config` ***must be called to configure a digital I/O port as an input or output port.***

DIG_In_Port

Function

Reads digital input data from the specified digital port.

Synopsis

C Syntax	`locus i32 DIG_In_Port(u32 deviceNumber, u32 port, u8 *pattern);`
Pascal Syntax	`function DIG_In_Port(deviceNumber : i32; port : i32; var pattern : i16) : i32;`
BASIC Syntax	`FN DIG_In_Port(deviceNumber&, port&, pattern&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

port is the digital I/O port number.
Range: 0 through *n*-1, where *n* is the number of digital I/O ports on the board.

pattern is the digital data to be written to the specified port.
Range: 0 through 255.

pattern returns the 8-bit value that is mapped from the digital output lines making up the port such that bit 0, the least significant bit, corresponds to digital input line 0. If the port is less than eight bits wide, only the high-order bits in **pattern** are set to 0. For example, since ports 0 and 1 on the NB-MIO-16 and NB-MIO-16X are four bits wide, only bits 0 through 3 of **pattern** reflect the digital state of these ports.

`DIG_In_Port` reads the digital data from the port on the specified board. If the port is configured as an input port, reading that port returns the digital logic state of the lines as some external device is driving them. If the port is configured as an output port and has read-back capability, reading the port returns the output state of that port.

Note: `DIG_Prt_Config` ***must be called to configure a digital I/O port as an input or output port.***

DIG_Line_Config

Function

Configures the specified line in the specified port for the direction (input or output) selected.

Synopsis

C Syntax	`locus i32 DIG_Line_Config(u32 deviceNumber, u32 port, u32 line, u32 direction);`
Pascal Syntax	`function DIG_Line_Config(deviceNumber : i32; port : i32; line : i32; direction : i32) : i32;`
BASIC Syntax	`FN DIG_Line_Config(deviceNumber&, port&, line&, direction&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

port is the digital I/O port number.
Range: 0 through *n*-1, where *n* is the number of digital I/O ports on the board.

line is the digital line to be configured.
Range: 0 through *k*-1, where *k* is the number of digital I/O lines making up the port.

direction indicates the direction, input or output, to configure the line.
0: line is an input line (default).
1: line is an output line.

With `DIG_Line_Config`, a port can have any combination of input and output lines. Use `DIG_Prt_Config` to set all lines on the port to be either all input or all output lines.

DIG_Out_Group

Function

Writes digital output data to the specified digital group (NB-DIO-32F only).

Synopsis

C Syntax	`locus i32 DIG_Out_Group(u32 deviceNumber, u32 group, u32 pattern);`
Pascal Syntax	`function DIG_Out_Group(deviceNumber : i32; group : i32; pattern : i32) : i32;`
BASIC Syntax	`FN DIG_Out_Group(deviceNumber&, group&, pattern&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is the handshake group number.

Range: 0 through *n*-1, where *n* is the number of groups on the board.

pattern is the digital data to be written to the specified port.

Range: 0 through 255 for group size = 1.
0 through 65,535 for group size = 2.
0 through 2^{32}-1 for group size = 4.

pattern is an 8-bit, 16-bit, or 32-bit value, depending on the group size, and is mapped to the digital output lines. The **pattern** is mapped as follows to the digital output ports that make up the group:

- If the group contains one port, the low-order eight bits of **pattern** are written to that port.
- If the group contains two ports, the low-order 16 bits of **pattern** are written to the ports. If the group contains Ports 0 and 1, the low-order eight bits are written to Port 1 and the next eight bits are written to Port 0. If the group contains Ports 2 and 3, the low-order eight bits are written to Port 3 and the next eight bits are written to Port 2. The two ports are written to simultaneously.
- If the group contains all four ports, all 32 bits of **pattern** are written to the ports. The least significant eight bits are written to Port 3, the next eight bits are written to Port 2, the next eight bits are written to Port 1, and the most significant eight bits are written to Port 0. The four ports are written to simultaneously.

Note: ***The MOST significant bits are written to the LOWEST numbered port, and the LEAST significant bits are written to the HIGHEST numbered port.***

`DIG_Out_Group` writes the specified digital group data to the group on the specified board. All ports in the group are updated simultaneously. If the specified group has not been configured as an output group, the operation is not performed and an error is returned. If no ports have been assigned to the specified group, the operation is not performed and an error is returned. `DIG_Grp_Config` must be called to configure a group.

DIG_Out_Line

Function

Sets or clears the specified digital output line in the specified digital port.

Synopsis

C Syntax	`locus i32 DIG_Out_Line(u32 deviceNumber, u32 port, u32 line, u32 state);`
Pascal Syntax	`function DIG_Out_Line(deviceNumber : i32; port : i32; line : i32; state : i32) : i32;`
BASIC Syntax	`FN DIG_Out_Line(deviceNumber&, port&, line&, state&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

port is the digital I/O port number.

Range: 0 through *n*-1, where *n* is the number of digital I/O ports on the board.

line is the digital output line to be written to.

Range: 0 through *k*-1, where *k* is the number of digital I/O lines making up the port.

state is the digital state to set the line to.

0: digital line is set to a digital logic low.
1: digital line is set to a digital logic high.

`DIG_Out_Line` sets the digital line in the specified port to the specified state. The remaining digital output lines making up the port are not affected by this call. If the port has not been configured as an output port, the operation is not performed and an error is returned. `DIG_Prt_Config` must be called to configure a digital I/O port as an output port.

DIG_Out_Port

Function

Writes digital output data to the specified digital port.

Synopsis

C Syntax	`locus i32 DIG_Out_Port(u32 deviceNumber, u32 port, u32 pattern);`
Pascal Syntax	`function DIG_Out_Port(deviceNumber : i32; port : i32; pattern : i32) : i32;`
BASIC Syntax	`FN DIG_Out_Port(deviceNumber&, port&, pattern&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

port is the digital I/O port number.
Range: 0 through *n*-1, where *n* is the number of digital I/O ports on the board.

pattern is the digital data to be written to the specified port.
Range: 0 through 255.

pattern is the 8-bit value that is mapped to the digital output lines making up the port such that bit 0, the least significant bit, corresponds to digital output line 0. If the port is less than eight bits wide, only the low-order bits in **pattern** affect the port. For example, since Ports 0 and 1 on the NB-MIO-16 and NB-MIO-16X are four bits wide, only bits 0 through 3 of **pattern** affect the digital output state of these ports.

`DIG_Out_Port` writes the specified digital data to the port on the specified board. If the specified port has not been configured as an output port, the operation is not performed and an error is returned. `DIG_Prt_Config` must be called to configure a digital I/O port as an output port.

DIG_Prt_Config

Function

Configures the specified port for direction (input or output) and handshake mode.

Synopsis

C Syntax	`locus i32 DIG_Prt_Config(u32 deviceNumber, u32 port, u32 direction, u32 mode);`
Pascal Syntax	`function DIG_Prt_Config(deviceNumber : i32; port : i32; direction : i32; mode : i32) : i32;`
BASIC Syntax	`FN DIG_Prt_Config(deviceNumber&, port&, direction&, mode&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

port is the digital I/O port number.
Range: 0 through *n*-1, where *n* is the number of digital I/O ports on the board.

direction selects the direction, input or output, for which the port is to be configured.
0: port is configured as an input port (default).
1: port is configured as an output port.

mode selects the handshake mode that the port is to be configured to use.
0: port is configured for no-handshaking (nonlatched) mode.
1: port is configured for handshaking (latched) mode. **mode** = 1 is valid only for Ports 0, 1, 3, 4, 6, 7, 9, and 10 of the NB-DIO-96 and PCI-DIO-96, or for Ports 0 and 1 of the NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series. **mode** = 0 must be used for all other ports and boards that do not permit handshaking. The NB-DIO-32F utilizes handshaking, but only through the group calls (see `DIG_Grp_Config`).

`DIG_Prt_Config` configures the specified port according to the specified direction and handshake mode. Any configuration that is invalid for the specified port returns an error, and the port configuration is not changed. Information about the valid configuration of any digital I/O port is given at the beginning of this chapter.

`DIG_Prt_Config` returns an error if the specified port has been assigned to a group.

After system startup, all digital I/O ports are configured as follows:

direction = 0: input port
mode = 0: no-handshaking mode

Additionally, ports on the NB-DIO-32F are not assigned to any group. If this is not the digital I/O configuration you want, you must call `DIG_Prt_Config` to change the port configuration. You must call `DIG_Grp_Config` to use handshaking modes on the NB-DIO-32F.

DIG_Prt_Status

Function

Returns the handshake state of the port (NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series only).

Synopsis

C Syntax	`locus i32 DIG_Prt_Status(u32 deviceNumber, u32 port, u16 *status);`
Pascal Syntax	`function DIG_Prt_Status(deviceNumber : i32; port : i32; var status : i16) : i32;`
BASIC Syntax	`FN DIG_Prt_Status(deviceNumber&, port&, status&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

port is the digital I/O port number.
Range: 0 through *n*-1, where *n* is the number of digital I/O ports on the board.

status returns the handshake status of the port. **status** is either 1 or 0. The significance of **status** depends on the port configuration. If the port is configured to be an input port, **status** = 1 indicates that data has been latched into the port. If the port is configured to be an output port, **status** = 1 indicates that an external device has latched the port output and new data can be written to the port.

`DIG_Prt_Status` reads the handshake status of the specified port and returns the port status in **status**. `DIG_Prt_Status`, along with `DIG_Out_Port` and `DIG_In_Port`, facilitates handshaking of digital data between systems. If the specified port is configured as an input port, `DIG_Prt_Status` indicates when to call `DIG_In_Port` to fetch the data that an external device has latched in. If the specified port is configured as an output port, `DIG_Prt_Status` indicates when to call `DIG_Out_Port` to write the next piece of data to the external device. If the specified port is not configured for handshaking, an error code and **status** = 0 are returned.

Refer to the user manual for each board for handshake timing information. If the port is configured for input handshaking, **status** corresponds to the state of the IBF bit. If the port is configured for output handshaking, **status** corresponds to the state of the OBF* bit. These bits and how they correspond to handshaking events are covered in the user manual for each board.

Note: `DIG_Prt_Config` ***must be called to configure a port for data direction and handshaking operation.***

DIG_Scan_Setup

Function

Configures the specified group for port assignment, direction (input or output), and size (NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series only).

Synopsis

C Syntax	`locus i32 DIG_Scan_Setup(u32 deviceNumber, u32 group, u32 groupSize, u16 *portList, u32 direction);`
Pascal Syntax	`function DIG_Scan_Setup(deviceNumber : i32; group : i32; groupSize : i32; portList : pi16; direction : i32) : i32;`
BASIC Syntax	`FN DIG_Scan_Setup(deviceNumber&, group&, groupSize&, portList&, direction&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is the group to be configured.

Range: 0 through 1 for the NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series.
0 through 7 for the NB-DIO-96 and PCI-DIO-96.

groupSize selects the number of 8-bit ports in the group.

Range: 0 through 2 for the NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series.
0 through 8 for the NB-DIO-96 and PCI-DIO-96.
Note: ***0 is to unassign any ports previously assigned to*** **group**.

portList is the list of ports in **group**. The order of the ports in the list determines how data is interleaved in the user's buffer when `DIG_Blk_Start` is called. The last port in the list determines the port whose handshaking signal lines are used to communicate with the external device and to generate hardware interrupts.

Range: 0 or 1 for the NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series.
0, 1, 3, 4, 6, 7, 9, or 10 for the NB-DIO-96 and PCI-DIO-96.

direction selects the direction, input or output, to which the **group** is to be configured.

0: group is configured as an input group (default).
1: group is configured as an output group.

`DIG_Scan_Setup` configures the specified group to contain the specified ports with the specified assignment and direction. If **groupSize** is 0, any ports previously assigned to **group** are released. Any configurations not supported by or invalid for the specified group return an error, and the group configuration is not changed. Ports assigned to a group are subsequently written to or read from as a group using `DIG_Blk_Start`. Any ports assigned to a group can no longer be accessed through any of the non-group calls listed previously.

Because each port on the NB-DIO-96, PCI-DIO-96, NB-DIO-24, DAQCard-DIO-24, NB-PRL, Lab and 1200 series has its own handshaking circuitry, extra wiring may be necessary to make data transfer of a group with more than one port reliable. If the group has only one port, no extra wiring is needed.

Each input port has a different Strobe Input (STB*) control signal.

PC4 on the I/O connector is for Port 0.
PC2 on the I/O connector is for Port 1.

Each input port also has a different Input Buffer Full (IBF) control signal.

PC5 on the I/O connector is for Port 0.
PC1 on the I/O connector is for Port 1.

Each output port has a different Output Buffer Full (OBF*) control signal.
PC7 on the I/O connector is for Port 0.
PC1 on the I/O connector is for Port 1.

Each output port also has a different Acknowledge Input (ACK*) control signal.
PC6 on the I/O connector is for Port 0.
PC2 on the I/O connector is for Port 1.

On the NB-DIO-96 and PCI-DIO-96 I/O connector, 4 different sets of PC pins can be found. They are APC, BPC, CPC, and DPC. APC pins correspond to Port 0 and Port 1, BPC pins correspond to Port 3 and Port 4, CPC pins correspond to Port 6 and Port 7, and DPC pins correspond to Port 9 and Port 10. For example, CPC7 is the Output Buffer Full (OBF) control signal for Port 6 and CPC1 is the OBF signal for Port 7 if both ports are configured as handshaking output ports.

If a group of ports is configured for input, you need to connect all the corresponding Strobe Input (STB) lines together and connect them to the appropriate handshaking signal of the external device. Only the Input Buffer Full (IBF) of the last port in **portList** should be connected to the external device. No connection is needed for the IBF of the other port(s) in **portList**.

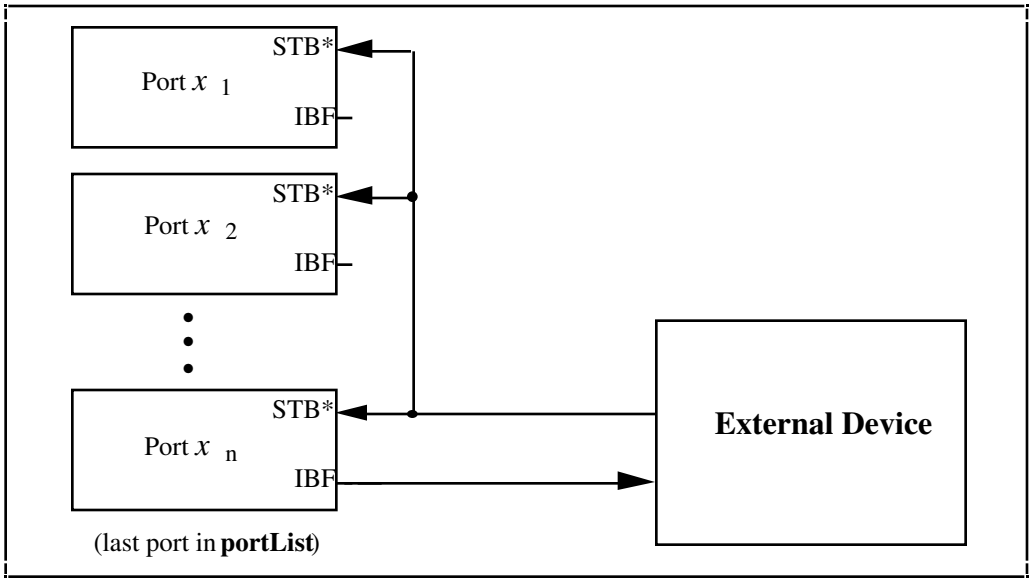

Figure 5-3. Digital Scanning Input Group Handshaking Connections

If a group of ports is configured as output, you should not make any connection on the control signals except those for the last port in **portList**. You should make the connection with the external device as if only the last port in **portList** is in the group. No connection is needed for any other port in the list.

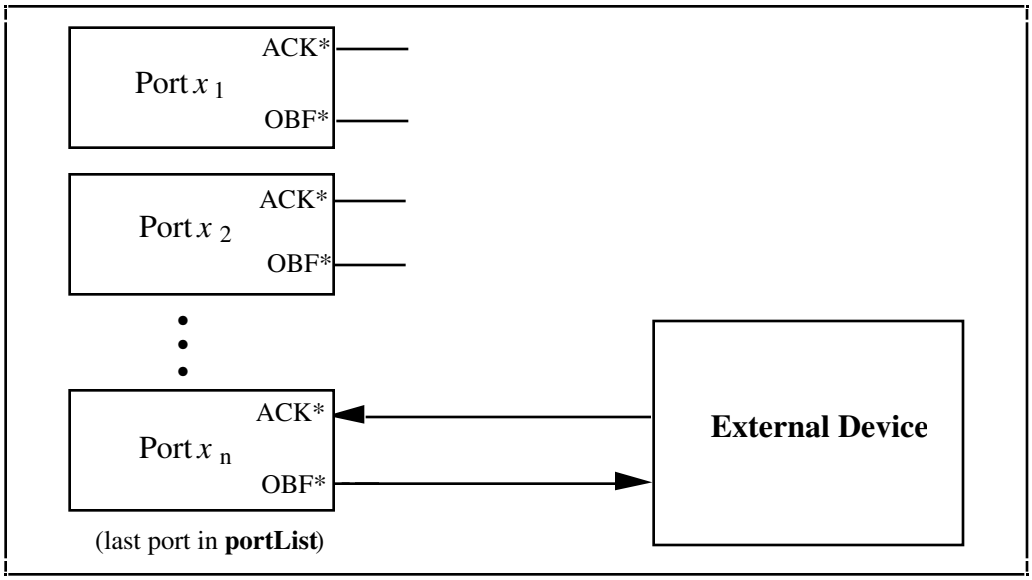

Figure 5-4. Digital Scanning Output Group Handshaking Connections

For NB-DIO-24, DAQCard-DIO-24, and NB-PRL users, the correct W1 jumper setting is required to allow `DIG_Blk_Start` to function properly. As long as Port 0 is not configured as a handshaking output port, the jumper should be set to PC6; otherwise, the jumper should be set to PC4.

Chapter 6
Data Acquisition Functions

This chapter explains the functions used for performing data acquisition operations. Single-channel acquisition, multiple-channel scan acquisition, interval scanning, pretrigger mode, posttrigger mode, double-buffered mode, and AMUX-64T multiplexer mode are all documented. These Data Acquisition functions are used with the National Instruments boards for the Macintosh. There are three sets of functions described: the Single-Buffered Data Acquisition functions (`DAQ` and `SCAN`), the Double-Buffered Data Acquisition functions (`DAQ2`), and the Multiple-Channel Data Acquisition functions (`MDAQ`).

The Single-Buffered Data Acquisition functions (`DAQ`, `Lab_ISCAN`, and `SCAN`) acquire a specified number of samples from one or more channels and return the data when the acquisition is complete.

The Double-Buffered Data Acquisition functions (`DAQ2`) can acquire samples from one or more channels into a circular buffer. With double-buffering, data can be retrieved from an acquisition in progress without interrupting the acquisition. Data can be collected continuously using a fixed amount of memory.

The Multiple-Channel Data Acquisition functions (`MDAQ`) retrieve multiple frames of data from one or more channels. Each frame is associated with a trigger and can contain both pre-trigger and post-trigger information. With the `MDAQ` functions, data can be retrieved from an acquisition in progress without interrupting the acquisition.

See Appendix A to determine which function set works with your board.

Note: ***If you are using analog input SCXI modules, you need to program the SCXI hardware first using the SCXI functions in Chapter 7,*** *SCXI Functions****, before using the Data Acquisition functions in this chapter.***

Data Acquisition Hardware

Table 6-1 shows the data acquisition hardware characteristics.

Table 6-1. Hardware Characteristics

Device	Analog Input Channels	Bits	Gains	Input Ranges
NB-MIO-16	0–15 (single-ended) 0–7 (differential)	12-bit	1, 10, 100, 500 (16L) 1, 2, 4, 8 (16H)	0 to 10 V (unipolar) -5 to +5 V (bipolar) -10 to +10 V (bipolar)
NB-MIO-16X	0–15 (single-ended) 0–7 (differential)	16-bit	1, 10, 100, 500 (16XL) 1, 2, 4, 8 (16XH)	0 to 10 V (unipolar) 0 to 5 V (unipolar) -5 to +5 V (bipolar) -10 to +10 V (bipolar)
PCI-MIO-16XE-50	0–15 (single-ended) 0–7 (differential)	16-bit	1, 2, 10, 100	0 to 10 V (unipolar) -10 to +10 V (bipolar)
DAQCard-500	0–7 (single-ended)	12-bit	1	No Hardware Jumpers
DAQCard-700	0–15 (single-ended) 0–7 (differential)	12-bit	1	No Hardware Jumpers
PCI-1200 DAQCard-1200	0–7 (single-ended) 0, 2, 4, 6 (differential)	12-bit	1, 2, 5, 10, 20, 50, 100	0 to 10 V (unipolar) -10 to +10 V (bipolar)
Lab-NB, Lab-LC	0–7 (single-ended)	12-bit	1, 2, 5, 10, 20, 50, 100	0 to 10 V (unipolar) -5 to +5 V (bipolar)

NB-MIO-16 and NB-MIO-16X Data Acquisition

The NB-MIO-16 and NB-MIO-16X analog input channels are multiplexed into a single software programmable gain stage and 12-bit (NB-MIO-16) or 16-bit (NB-MIO-16X) ADC.

Data acquisition with the NB-MIO-16 or NB-MIO-16X is performed in two modes: single-channel data acquisition and multiple-channel scan data acquisition. Single-channel data acquisition involves selecting a single analog input multiplexer and gain setting. Multiple-channel scan data acquisition can be used to scan a set of analog input channels, each with its own gain setting, in a round-robin mode. In this mode, a scan sequence is stored with a specified analog channel and gain setting for each step in Mux-Gain Memory on the NB-MIO-16 or NB-MIO-16X. The length of this scan sequence on the NB-MIO-16 can be 2, 4, 8, or 16. On the NB-MIO-16X, the length of the scan sequence can be 1 through 16.

During scanning, the analog input circuitry is set to the next entry in the scan sequence and an A/D conversion is performed once every sample interval. For maximum performance, this operation is pipelined so that the next channel is switched to while the current A/D conversion is performed.

On the NB-MIO-16, only one sampling interval is used for multiple-channel scanning acquisitions. This interval is the amount of time to elapse between each A/D conversion. When the end of the scan sequence is reached on the NB-MIO-16, the sequence starts over again until the required number of samples have been acquired.

The NB-MIO-16X has additional timing capabilities for multiple-channel scanning acquisitions. Besides choosing a sample interval (the time between samples from two different channels), you can also select a scan interval (the time between samples on any one channel). During scanning on the NB-MIO-16X, an A/D conversion is performed once every sample interval. When the end of the scan sequence is reached, the NB-MIO-16X waits for the remainder of the scan interval before scanning the channels again. The channels are scanned repeatedly at the beginning of each scan interval until the required number of samples have been acquired. Interval scanning has the advantage of simulating *simultaneous sampling* of a group of channels once every scan interval. A comparison of the scan interval and the sample interval is shown in Figure 6-1.

Interval scanning is also supported for the NB-MIO-16 in the following special case: If an `SCXI_SCAN_Setup` call has been made to set up an SCXI scan that includes an SCXI-1140 module, then the `SCAN_IntStart` call is able to implement interval scanning on the NB-MIO-16 as well as the NB-MIO-16X. In this special case, the sample timebase and the scan timebase specified must be the same. In all other cases, however, interval scanning is only available on the NB-MIO-16X.

SCXI modules can be used as a data acquisition front end for the NB-MIO-16 or the NB-MIO-16X to provide signal conditioning for the input signals and channel multiplexing. The SCXI functions described in Chapter 7 set up the SCXI modules for data acquisition operations to be performed by the NB-MIO-16 or NB-MIO-16X.

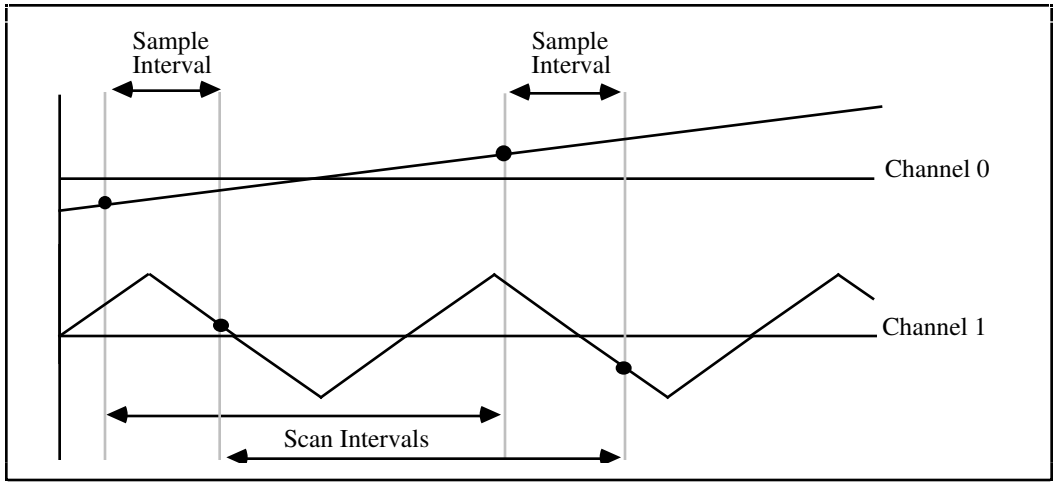

Figure 6-1. NB-MIO-16X Interval Scanning

NB-MIO-16 and NB-MIO-16X Data Acquisition Timing

Timing for data acquisition can be performed by the onboard counter/timers or can be performed externally. Data acquisition timing involves the following timing signals:

Signal Name	Description
Trigger	An edge-triggered signal that initiates a data acquisition sequence. A trigger can be supplied either externally through the I/O connector EXTTRIG* input on the NB-MIO-16, the I/O connector STARTTRIG* input on the NB-MIO-16X or under software control.
Conversion pulses	Generate a pulse once every sample interval, causing an A/D conversion to be initiated. This signal can be generated by the onboard programmable sample-interval clock supplied by the counter/timer or can be supplied externally through the I/O connector EXTCONV* input. **Note**: ***In most cases, external conversion pulses should not be used in scanning operations when SCXI is being used in multiplexed mode. The MIO-16 has no way of masking conversions before the data acquisition begins, so any conversion pulses that occur before the acquisition is triggered will advance the SCXI channels, causing the data for the channels to be shifted in the buffer.***
Sample counter	Counts the number of A/D conversions (samples) when conversion pulses are generated by the onboard sample-interval counter, and shuts down the data acquisition timing circuitry when the desired number of samples have been acquired.
Gate	A level-triggered signal that, when low, holds off data acquisition timing. This signal can be supplied externally through the I/O connector EXTGATE input on the NB-MIO-16.
Timebase clock	Supplies the timebase for the sample interval counter. Onboard selections of 1 MHz, 100 kHz, 10 kHz, 1 kHz, or 100 Hz are available. An external timebase clock can be supplied through the I/O connector at the SOURCE5 input on the NB-MIO-16 and NB-MIO-16X.

See the *NB-MIO-16 User Manual* or the *NB-MIO-16X User Manual* for more information regarding these signals.

NB-MIO-16 Data Acquisition Rates

The maximum recommended data acquisition rates for both single-channel and multiple-channel data acquisition are given below. These rates represent the fastest data conversion times that the board can achieve and still maintain accuracy. The data acquisition rates given later in this chapter allow for settling to a 10 V input signal change between conversions. It is possible to operate at faster speeds; however, accuracy may be compromised. Data acquisition errors occur if the following sample rates are exceeded by a large amount.

Maximum recommended data acquisition rates on a single channel (any gain setting) for the NB-MIO-16 are given in Table 6-2.

Table 6-2. Maximum Data Acquisition Rates for Single Channels on the NB-MIO-16

Board	Typical	Worst Case
NB-MIO-16(H/L)-25	45 kS/s	37 kS/s
NB-MIO-16(H/L)-15	71 kS/s	59 kS/s
NB-MIO-16(H/L)-9	100 kS/s	91 kS/s

With multiple-channel scan data acquisition, extra time is required by the data acquisition circuitry for gain/multiplexer settling due to channel switching. This required settling time depends on the gain setting used for each channel. Also, this settling time limits data acquisition rates. The recommended settling time versus gain for the NB-MIO-16 is given in Table 6-3.

Table 6-3. Recommended Settling Time Versus Gain for the NB-MIO-16

Gain Setting	Recommended Settling Time
1, 2, 4, 8	10 µs
10	20 µs
100	40 µs
500	80 µs

The maximum recommended data acquisition rates using multiple-channel scanning (gain dependent) for the NB-MIO-16 are shown in Table 6-4.

Table 6-4. Maximum Data Acquisition Rates for Multiple Channels on the NB-MIO-16

Board	Gain	Typical	Worst Case
NB-MIO-16(H/L)-25	1, 2, 4, 8, 10	45 kS/s	37 kS/s
	100	25 kS/s	
	500	12.5 kS/s	
NB-MIO-16(H/L)-15	1, 2, 4, 8	71 kS/s	59 kS/s
	10	50 kS/s	
	100	25 kS/s	
	500	12.5 kS/s	
NB-MIO-16(H/L)-9	1, 2, 4, 8	100 kS/s	91 kS/s
	10	50 kS/s	
	100	25 kS/s	
	500	12.5 kS/s	

If you are using SCXI with your DAQ board, refer to the *SCXI Data Acquisition Rates* section for the effect of SCXI module settling time on your DAQ board rates.

NB-MIO-16X Data Acquisition Rates

The maximum recommended data acquisition rates for both single-channel and multiple-channel data acquisition represent the fastest data throughput that the board is able to achieve and still maintain accuracy. The data acquisition rates given below allow for settling to a 10 V input signal change between conversions. It may be possible to operate at faster speeds; however, accuracy may be compromised. Data acquisition errors occur if the sample rates are exceeded by a large amount.

Maximum recommended data acquisition rates on a single channel (any gain setting) for the NB-MIO-16X are given in Table 6-5.

Table 6-5. Maximum Data Acquisition Rates for Single Channels on the NB-MIO-16X

Board	Acquisition Rate
NB-MIO-16X(H/L)-42	23.8 kS/s
NB-MIO-16X(H/L)-18	55.6 kS/s

With multiple-channel scan data acquisition, extra time is required by the data acquisition circuitry to allow for gain/multiplexer settling due to channel switching. This required settling time depends on the gain setting used for each channel. Also, this settling time limits data acquisition rates.

The recommended settling time versus gain for the NB-MIO-16X is given in Table 6-6.

Table 6-6. Recommended Settling Time Versus Gain for the NB-MIO-16X

Gain Setting	Accuracy		
	0.01%	0.005%	0.5 LSB
1, 2, 4, 8, 10	30 µs	50 µs	200 µs
100	50 µs	50 µs	500 µs
500	50 µs	100 µs	500 µs

The maximum data acquisition rates for multiple channel scanning (gain dependent) for the NB-MIO-16X are given in Table 6-7.

Table 6-7. Maximum Data Acquisition Rates for Multiple Channels on the NB-MIO-16X

Board	Gain	Data Acquisition Rate		
		0.01% Settling	0.005% Settling	0.5 LSB Settling
NB-MIO-16X(H/L)-42	1, 2, 4, 8, 10	23.8 kS/s	20 kS/s	5 kS/s
	100	20 kS/s	20 kS/s	2 kS/s
	500	20 kS/s	10 kS/s	2 kS/s
NB-MIO-16X(H/L)-18	1, 2, 4, 8, 10	33.3 kS/s	20 kS/s	5 kS/s
	100	20 kS/s	20 kS/s	2 kS/s
	500	20 kS/s	10 kS/s	2 kS/s

If you are using SCXI with your DAQ board, refer to the *SCXI Data Acquisition Rates* section for the effect of SCXI module settling time on your DAQ board rates.

Lab and 1200 Series Data Acquisition

The Lab and 1200 series contain eight single-ended analog input channels numbered 0 through 7. The analog input channels are multiplexed into a single software-programmable gain state and 12-bit ADC. Seven gains are provided on the Lab and 1200 series: 1, 2, 5, 10, 20, 50, and 100.

Analog input on the Lab-NB can be configured for two different *nominal* input ranges:

- 0 to +10 V (unipolar)
- -5 to +5 V (bipolar)

Data acquisition with the Lab and 1200 series uses one of two modes: single-channel data acquisition or multiple-channel scanned data acquisition. Single-channel data acquisition involves selecting a single analog input multiplexer and gain setting. In multiple-channel scanned data acquisition, a set of analog input channels is scanned with a single gain setting in a round-robin mode. This scanning is performed by specifying the number of channels to be scanned and the gain setting to be used for the scanning operation. During scanning, the analog input circuitry is set to the next channel in the scan sequence, and an A/D conversion is performed. When the end of the scan sequence is reached, the sequence is started over again until the required number of samples have been acquired.

The PCI-1200 and DAQCard-1200 boards support interval scanning. A scan interval is the time that elapses between two channel-scanning cycles.

SCXI modules can be used as a data acquisition front end for the Lab and 1200 series to provide signal conditioning for the input signals. All the modes described above can be used in conjunction with SCXI for single channel acquisitions; however, multiple-channel scanned acquisitions are only supported when using the SCXI-1120 or SCXI-1121 modules in parallel mode. The SCXI functions described in Chapter 7 set up the SCXI modules for data acquisition operations to be performed by the Lab and 1200 series.

Lab and 1200 Series Data Acquisition Timing

Timing for data acquisition is provided by the onboard counter/timers or can be performed externally. Data acquisition timing involves the following timing signals:

Signal Name	Description
Trigger	An edge-triggered signal that initiates a data acquisition sequence. A trigger can be supplied externally through the I/O connector EXTTRIG input.
Conversion pulses	Generate a pulse once every sample interval, causing an A/D conversion to be initiated. This signal can be generated by the onboard programmable sample interval clock supplied by a Counter/Timer or can be supplied externally through the I/O connector EXTCONV* input.
Sample counter	Counts the number of A/D conversions (samples) when conversion pulses are generated by the onboard sample interval counter, and shuts down the data acquisition timing circuitry when the desired number of samples have been acquired.
Timebase clock	Provides the timebase for the sample interval counter. Onboard selections of 1 MHz, 100 kHz, 10 kHz, 1 kHz, or 100 Hz are available.

See the specific board user manuals for more information regarding these signals.

Lab and 1200 Series Counter/Timer Signals

The onboard Counter A0 is used to produce the total sample interval for data acquisition. However, if the total sample interval is greater than 65,535 µs, Counter B0 is used to generate the clock for a slower timebase, which is used by Counter A0 to provide the total sample interval. Counter B0 then cannot be used by the `ICTR_Setup` and `ICTR_Reset` functions for the duration of the data acquisition operation. Counter B0 also cannot be used by the Waveform Generations functions if the total update interval for waveform generation is also greater than 65,535 µs and Counter B0 is required to produce a timebase for waveform generation different from the timebase being produced by Counter B0 for data acquisition. Counter B0 is available for data acquisition under the following conditions:

- If waveform generation is not in progress and no `ICTR_Setup` call has been made on Counter B0 since startup.
- If waveform generation is not in progress and an `ICTR_Reset` call has been made on Counter B0.

- If waveform generation is in progress and is using Counter B0 to obtain the timebase required to produce the total update interval, this timebase is the same as required by the Data Acquisition functions to produce the total sample interval. In this case, Counter B0 is used to provide the same timebase for both data acquisition and waveform generation.

Lab and 1200 Series Data Acquisition Rates

The maximum recommended data acquisition rates for both single-channel and multiple-channel data acquisition represent the fastest data throughput that the board is able to achieve and still maintain accuracy. The data acquisition rates given below allow for settling to a 10 V input signal change between conversions. It may be possible to operate at faster speeds; however, accuracy may be compromised. Data acquisition errors occur if the sample rates are exceeded by a large amount.

With data acquisition, extra time is required by the data acquisition circuitry for gain/multiplexer settling due to channel switching. This required settling time depends on the gain setting used for each channel and limits data acquisition rates. The recommended settling time versus gain for the Lab and 1200 series is given in Table 6-8.

Table 6-8. Recommended Settling Time Versus Gain for the Lab and 1200 Series

Gain Setting	Recommended Settling Time
1	16 µs
2, 5	20 µs
10, 20	30 µs
50, 100	100 µs

The maximum recommended data acquisition rates for single and multiple channels for the Lab and 1200 series are shown in Table 6-9.

Table 6-9. Maximum Data Acquisition Rates for Multiple Channels on the Lab and 1200 Series

Board	Gain	Data Acquisition Rate	
		Typical	Worst Case
Lab-LC Lab-NB	1 2, 5 10, 20 50, 100	62.5 kS/s 50 kS/s 33.3 kS/s 10 kS/s	
PCI-1200 DAQCard-1200	1 2, 5, 10 20 50 100	90.9 kS/s 76.9 kS/s 66.7 kS/s 37.0 kS/s 16.7 kS/s	71.4 kS/s 62.5 kS/s 52.6 kS/s 29.4 kS/s 12.5 kS/s

If you are using SCXI with your DAQ board, refer to the *SCXI Data Acquisition Rates* section for the effect of SCXI module settling time on your DAQ board rates.

DAQCard-500 and DAQCard-700 Data Acquisition

The DAQCard-500 and DAQCard-700 can perform single-channel data acquisition and multiple-channel scanned data acquisition. For single-channel data acquisition, you select a single analog input channel. The device performs a single A/D conversion on that channel every sample interval.

For multiple-channel scanned data acquisition, the device scans a sequence of analog input channels. A sample interval indicates the time to elapse between A/D conversions on each channel in the sequence. You need only to select a single starting channel to select the sequence of channels to scan. The device then scans the channels in consecutive order until channel 0 is reached and the scan begins anew with the starting channel. If the starting channel is channel 3, for example, the scan sequence is as follows:

channel 3, channel 2, channel 1, channel 0, channel 3, and so on

You can use both the single-channel and multiple-channel acquisitions with the double-buffered mode. Double-buffered mode fills the user-specified buffer continuously.

You can use SCXI modules as a data acquisition front end for the DAQCard-700 to signal condition the input signals and multiplex the channels. You can use all the modes just described in conjunction with SCXI for single-channel acquisitions; however, multiple-channel acquisitions are only supported when using the SCXI-1120 or SCXI-1121 modules in Parallel mode.

You cannot use the DAQCard-500 with SCXI.

DAQCard-500 and DAQCard-700 Data Acquisition Timing

Timing for data acquisition can be performed by the onboard MSM82C53 Counter/Timer or externally. The MSM82C53 Counter/Timer has three independent 16-bit counters/timers, which are assigned as follows:

- Counter 0 is a sample-interval counter for data acquisition that is available if no data acquisition is in progress.
- Counter 1 is available for general-purpose counting functions.
- Counter 2 is available for general-purpose counting functions.

Data acquisition timing involves the following timing signals:

A conversion pulse is a signal that generates a pulse once every sample interval, causing the device to initiate an A/D conversion. This signal can be generated by the onboard, programmable sample-interval clock supplied by the MSM82C53 Counter/Timer, or can be supplied externally through the I/O connector EXTCONV* input. You can select external conversion pulses by calling `DAQ_Config`. If you do not want to use external conversion pulses, you should disconnect the EXTCONV* pin on the I/O connector to prevent extra conversions.

A timebase clock is a clock signal that is the timebase for the sample-interval counter. Counter 0 of the MSM82C53 uses a 1 MHz clock as its timebase.

See your device user manual for more information regarding these signals.

If you are using SCXI with your DAQCard-700, refer to the *SCXI Data Acquisition Rates* section later in this chapter for the effect of SCXI module settling time on your DAQ device rates.

DAQCard-500 and DAQCard-700 Counter/Timers

The DAQCard-500 and DAQCard-700 contain an onboard MSM82C53 Programmable Interval Timer chip that has three independent 16-bit counter/timers. Counter 0 is used for data acquisition operations.

E Series Data Acquisition

The E Series devices can perform single-channel data acquisitions and multiple-channel scanned data acquisitions. For single-channel data acquisition, select a single analog input channel and gain setting. The device performs a single A/D conversion on that channel every sample interval.

For multiple-channel scanned data acquisition, the device scans a set of analog input channels, each with its own gain setting. In this method, a scan sequence indicates which analog channels to scan and the gain settings for each channel. The length of this scan sequence can be 1 to 512 channel/gain pairs. During scanning, the analog input circuitry performs an A/D conversion on the next entry in the scan sequence. The device performs an A/D conversion once every sample interval. For maximum performance, this operation is pipelined so that the device switches to the next channel while the current A/D conversion is performed. The device waits for a specified scan interval before scanning the channels again. The channels are scanned repeatedly at the beginning of each scan interval until the required number of samples has been acquired. For example, you can scan a sequence of four channels once every 10 s. The device could sample the channels at the beginning of the 10 s interval, within 20 µs, with a 5 µs sample interval between channels. If you use `SCAN_Start`, the scan sequence starts over again immediately at the end of each scan sequence without waiting for a scan interval. This causes the device to scan the channels repeatedly as fast as possible.

You can combine both single-channel and multiple-channel acquisition with any of the following additional modes:

- Posttrigger mode
- Pretrigger mode
- Double-buffered mode
- AMUX-64T mode
- SCXI mode

Posttrigger mode collects a specified number of samples after the device receives a trigger. Refer to the *start trigger* discussion in the appropriate data acquisition timing section for your device later in this chapter for details. After the user-specified buffer is full, the data acquisition stops.

Pretrigger mode collects data both before and after the device receives a trigger in posttrigger mode, either through software or by applying a hardware signal. The device collects samples and fills the user-specified buffer without stopping until the device receives the *stop trigger* signal. Refer to the *stop trigger* discussion in the appropriate data acquisition timing section for your device later in this chapter for details. The device then collects a specified number of samples and stops the acquisition. The buffer is treated as a circular buffer—when the entire buffer has been written to, data is stored at the beginning again, overwriting the old data. When data acquisition stops, the buffer has samples from before and after the stop trigger occurred. The number of samples saved depends on the length of the user-specified buffer and on the number of samples specified to be acquired after receipt of the trigger. Double-buffered mode, like pretrigger mode, also fills the user-specified buffer continuously.

In the AMUX-64T mode, you use one or more external AMUX-64T devices to extend the number of analog input channels available. You connect the external signals to the pins of the AMUX-64T devices, instead of directly to the pins of the DAQ device.

You can use SCXI modules as a data acquisition front end for the device to condition the input signals and multiplex the channels. You can use all the modes just described in conjunction with SCXI.

Note: ***Refer to the*** `Set_DAQ_Device_Info` ***function in Chapter 2,*** *Board-Specific Functions****, for information on data acquisition modes.***

MIO E Series Data Acquisition Timing

The following DAQ-STC counters are used for data acquisition timing and control:

- The *scan counter* is used to control the number of scans you will acquire. If you want to perform pretriggered acquisition, this counter ensures that you acquire selected number of scans before the stop trigger is recognized.

- The *scan timer* is a counter that you can use for *start scan* timing.
- The *sample interval timer* is a counter that you can use for *conversion* timing.

Data acquisition timing involves the following timing signals:

- A *start trigger* is a signal that initiates a data acquisition sequence. You can supply this signal externally through a selected I/O connector pin, through a RTSI bus trigger line, or by software.
- A *start scan* signal initiates individual scans. This signal can be supplied from the on-board programmable scan timer, externally through a selected I/O connector pin, through a RTSI bus trigger line, or by software.
- A *conversion* signal initiates individual analog-to-digital (A/D) conversions. This signal can be supplied from the on-board programmable sample timer, externally through a selected I/O connector pin, through a RTSI bus trigger line, or by software.
- A *stop trigger* is a signal used for pretriggered data acquisition to notify your device to stop acquiring data after a specified number of scans. Data acquisition operation is continuously performed until the device receives this signal. This signal can be supplied externally through a selected I/O connector pin, through a RTSI bus trigger line, or by software.
- *Gate* is a signal used for gating the data acquisition. When you enable gating, the data acquisition will proceed only on selected level of the gate signal. This signal can be supplied externally through a selected I/O connector pin, through a RTSI bus trigger line.
- *Scan timer timebase* is a signal used by the scan timer for scan interval timing. This signal is used only when the scan timer is used. This signal can be supplied from one of the on-board timebase sources, externally through a selected I/O connector pin, or through a RTSI bus trigger line.
- *Sample interval timer timebase* is a signal used by the sample interval timer for conversion timing. This signal is used only when the sample interval timer is used. This signal can be supplied from one of the on-board timebase sources, externally through a selected I/O connector pin, or through a RTSI bus trigger line.

See your DAQ device user manual for more information regarding these signals.

DAQ devices with the DAQ-STC use two counters, the scan interval counter and the sample interval counter. The E Series devices support both internal and external timebases. The internal timebases available on the DAQ-STC are 20 MHz (50 ns) and 100 kHz (10 µs). The scan interval counter is a 24-bit counter, and the sample interval counter is a 16-bit counter.

While the scan interval counter has the freedom to work with both internal and external timebases, the sample interval counter can use either the 20 MHz timebase or the timebase used by the scan interval counter.

MIO E Series Data Acquisition Rates

Refer to the appropriate user manual for single-channel and multiple-channel DAQ rates and settling accuracy.

If you are using SCXI with your DAQ device, refer to the following *SCXI Data Acquisition Rates* section for the effect of SCXI module settling time on your DAQ device rates.

SCXI Data Acquisition Rates

The settling time of the SCXI modules may affect the maximum data acquisition rates that can be achieved by your DAQ board. The settling times and maximum rates of the different SCXI modules at each gain setting are listed in Table 6-10. If the maximum rate listed here for your SCXI module is *slower* than the applicable maximum rate of your DAQ board, then you will have to use the maximum rate listed here in Table 6-10.

Table 6-10. Maximum SCXI Data Acquisition Rates

SCXI Module	Gain	Maximum Acquisition Rate	Settling Time
SCXI-1100	1-100 200 500 1000, 2000	143 kS/s 100 kS/s 62.5 kS/s 20 kS/s	7 µs 10 µs 16 µs 50 µs
SCXI-1120	1-2000	143 kS/s	7 µs
SCXI-1121	1-2000	143 kS/s	7 µs
SCXI-1140	1-500	143 kS/s	7 µs

If you are using the SCXI-1122, please refer to the *SCXI-1122* section in Chapter 7, *SCXI Functions*.

Single-Buffered Data Acquisition Function Summary

The single-buffered data acquisition functions (`DAQ` and `SCAN`) acquire a specified number of samples from one or more channels and return the data when the acquisition is complete. Additionally, the configuration, scaling and analog triggering (that is, waiting until the incoming analog data crosses a specified level) functions can be performed with both single-buffered and double-buffered data acquisition.

The following functions can be used for single-buffered data acquisition on the Lab and 1200 devices, DAQCard-500, and DAQCard-700 and MIO boards:

`DAQ_Config` Stores configuration information for subsequent data acquisition operations.

`DAQ_PreTrig` Stores pretrigger information for stopping single-buffered data acquisition when a specified number of samples have been acquired *after* the occurrence of an external trigger. Pretriggering is not available on the NB-MIO-16.

`DAQ_Trigger` Stores analog trigger configuration information for subsequent single-buffered data acquisition operations.

`DAQ_Check` Checks to see whether the current data acquisition operation is complete and returns its status.

`DAQ_Clear` Cancels the current data acquisition operation and reinitializes the data acquisition circuitry.

`DAQ_Start` Initiates a single-channel data acquisition operation and stores the results in an array.

`DAQ_VScale` Converts the values of an array of acquired binary data and the gain setting for that data to actual input voltages measured.

`Lab_ISCAN_Check`	Checks to see whether the current scanned data acquisition operation is complete and then returns the status (DAQCard-500, DAQCard-700, and Lab and 1200 series only).
`Lab_ISCAN_Start`	Initiates a multiple-channel scanned data acquisition operation and stores the results in an array (DAQCard-500, DAQCard-700, and Lab and 1200 series only).
`SCAN_Check`	Checks to see whether the current scanned data acquisition operation is complete, and returns its status.
`SCAN_Demux`	Demultiplexes data acquired by a SCAN operation into separate arrays for each channel (C and Pascal only).
`SCAN_IntStart`	Initiates a multiple-channel scanned data acquisition operation, and can also define a scan interval for the NB-MIO-16X and PCI-MIO-16XE-50.
`SCAN_Setup`	Initializes the circuitry on the MIO devices for a scanned data acquisition operation. Initialization includes storing a table of the channel sequence and gain setting for each channel.
`SCAN_Start`	Initiates a multiple-channel scanned data acquisition operation on an MIO board.

Single-Buffered Data Acquisition Application Hints

Single-Channel Data Acquisition

`DAQ_Start` and `DAQ_Check` perform data acquisition operations from a single analog input channel into a user buffer. `DAQ_Start` initiates the data acquisition process, and `DAQ_Check` returns the completion status of the data acquisition process. The data acquired is in binary format and can be scaled to voltage values by using `DAQ_Scale`. Data acquisition can be terminated by executing `DAQ_Clear`. Data acquisition is normally controlled by the onboard counters and is started as soon as `DAQ_Start` is called. Other timing modes can be used when `DAQ_Config` is called to set them up. If any jumpers on the analog input circuitry have been changed, use `AI_Configure` to update the analog input configuration information for NI-DAQ for Macintosh.

`DAQ_Trigger` can be executed to enable analog triggering and to set up trigger information on single-buffered data acquisitions (single-channel and multiple-channel scan). Subsequent single-buffered calls to `DAQ_Start` (while triggering is enabled) cause `DAQ_Start` to wait for an analog trigger to occur before collecting the array of sample data. Analog triggering can also be used in single-channel acquisitions using SCXI.

`DAQ_PreTrig` can be executed to acquire a specified number of samples *after* an external stop trigger has occurred. Subsequent calls to `DAQ_Start` (while pretriggering is enabled) cause `DAQ_Start` to stop single-buffered data acquisition when a specified number of samples have been acquired *after* the occurrence of an external trigger. Pretriggering is possible only on the NB-MIO-16X, PCI-MIO-16XE-50, and Lab and 1200 series boards.

Example applications that perform data acquisition operations are included on your NI-DAQ for Macintosh diskettes. (See Chapter 11, *NI-DAQ for Macintosh Examples*.)

Multiple-Channel (Scanned) Data Acquisition

`SCAN_Start`, `SCAN_IntStart`, `Lab_ISCAN_Start`, `SCAN_Check`, and `Lab_ISCAN_Check` perform data acquisition operations while scanning multiple analog input channels. `SCAN_Start`, `SCAN_IntStart`, and `Lab_ISCAN_Start` initiate the acquisition process and store acquired binary values into a one-dimensional array with the data from each channel interleaved in time. `SCAN_Check` or `Lab_ISCAN_Check` returns the completion status of the scanned data acquisition process. After `SCAN_Check` or `Lab_ISCAN_Check` signifies that acquisition is complete, the one-dimensional data can be converted into a two-dimensional array in which one dimension corresponds to each channel. The other dimension corresponds to the data for a single channel by passing

the array to SCAN_Demux. This data can be scaled to voltage values by using DAQ_Scale on one channel of data at a time.

SCAN_Start and SCAN_IntStart must be preceded by SCAN_Setup, which indicates the number of channels to be scanned and loads a channel-gain scan sequence. SCAN_Setup needs to be executed only once if the same scan setup is valid for all executions of SCAN_Start or SCAN_IntStart. SCAN_Setup is not used for the Lab and 1200 series.

SCAN_IntStart can be used instead of SCAN_Start on the NB-MIO-16X and PCI-MIO-16XE-50 and Lab_ISCAN_Start is used on the PCI-1200 and DAQCard-1200 to perform a multiple-channel acquisition. These can define a scan interval between channel sequences in addition to the sampling interval between channels. Interval scanning has the advantage of simulating *simultaneous sampling* of a group of channels once every scanning interval. Interval scanning can also be achieved by using a SCXI-1140 module. The DAQCard-500 and DAQCard-700 also use Lab_ISCAN_Start, but only for continuous scanning, not interval scanning.

To perform multiple-channel scanned acquisitions using the SCXI-1140 module, interval scanning must be used. If an SCXI_SCAN_Setup call has been made to set up an SCXI scan that includes an SCXI-1140 module, then the SCAN_IntStart call is able to implement interval scanning on the NB-MIO-16 as well as the NB-MIO-16X. In this special case, the sample timebase and the scan timebase specified must be the same. In all other cases, however, interval scanning is only available on the NB-MIO-16X.

Scanned data acquisition operations can be terminated by executing DAQ_Clear. Scanned data acquisition is normally controlled by the onboard counters and is started as soon as SCAN_Start, Lab_ISCAN_Start, or SCAN_IntStart is executed. Other timing modes can be used when DAQ_Config is called to set them up. If any jumpers on the analog input circuitry have been changed, use AI_Config to update the analog input configuration information for NI-DAQ for Macintosh.

DAQ_Trigger can be executed to enable triggering on analog input values and to set up trigger information on single-buffered data acquisitions (single-channel and multiple-channel scan). Subsequent single-buffered calls to SCAN_Start, Lab_ISCAN_Start, or SCAN_IntStart (while triggering is enabled) cause these functions to wait for trigger conditions to be met before collecting the array of sample data. Analog triggering can be used with SCXI in multiple-channel scanned acquisitions only when the SCXI modules are operated in Parallel mode.

DAQ_PreTrig can be executed to enable acquisition of a specified number of samples *after* an external trigger has occurred. Subsequent calls to SCAN_Start, Lab_ISCAN_Start, or SCAN_IntStart (while pretriggering is enabled) cause these functions to stop single-buffered data acquisition when a specified number of samples have been acquired *after* the occurrence of an external trigger. Pretriggering is possible only on the NB-MIO-16X, PCI-MIO-16XE-50, and Lab and 1200 series boards.

Example applications that perform multiple-channel data acquisition operations are included on your NI-DAQ for Macintosh diskettes. (See Chapter 11, *NI-DAQ for Macintosh Examples*.)

Using the NB-MIO-16X in Unipolar Mode with Pascal

If you are using an NB-MIO-16X or PCI-MIO-16XE-50 in unipolar mode, conversion values are returned as 16-bit unsigned integers. Because Pascal does not support unsigned representation, the values in the range 32,768 through 65,535 are treated as negative numbers in Pascal. You can use the UArrToLArr conversion function to convert values to Pascal long integers.

Note: ***These values should be passed to DAQ_VScale without conversion. (See Chapter 11,*** *NI-DAQ for Macintosh Examples*, ***for a complete description of the*** UArrToLArr **function.)**

DAQ_Check

Function

Checks to see whether the current data acquisition operation is complete and returns its status.

Synopsis

C Syntax	`locus i32 DAQ_Check(u32 deviceNumber, u16 *status, u32 *retrieved);`
Pascal Syntax	`function DAQ_Check(deviceNumber : i32; var status : i16; var retrieved : i32) : i32;`
BASIC Syntax	`FN DAQ_Check(deviceNumber&, status&, retrieved&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

status returns an indication of whether the data acquisition has completed.

1: the data acquisition operation is complete.
0: the data acquisition operation is not yet complete.

`DAQ_Check` checks the current background data acquisition operation to determine whether it has completed. If the operation is complete, `DAQ_Check` sets **status** to 1. Otherwise, **status** is set to 0. If the acquisition is single-buffered, then the data is available in the buffer when **status** is 1. If the acquisition is double-buffered in continuous mode, **status** will always return 0. If the acquisition is double-buffered in noncontinuous mode, **status** will return 1 only when the entire acquisition is complete.

If `DAQ_Check` returns an **overFlowError** or an **overRunError**, the data acquisition operation may never complete because of lost A/D conversions due to samples being acquired too rapidly (sample interval was too small). An **overFlowError** indicates that the A/D FIFO overflowed because the data acquisition servicing operation could not keep up with the sample rate. An **overRunError** indicates that the data acquisition circuitry could not keep up with the sample rate. If one of these errors occurs, then `DAQ_Check` executes `DAQ_Clear` to terminate the operation and to clear all error flags.

If NI-DAQ for Macintosh is configured for double-buffered mode, an **overWriteErr** can occur. An **overWriteErr** indicates that the large circular acquisition buffer used for double-buffered acquisitions overwrote acquired data before it was retrieved by `DAQ2Get`, `DAQ2TGet`, `DAQ2Tap`, or `DAQ2TTap`. An overwrite error can be corrected by increasing the size of the large acquisition buffer, retrieving more data each time, retrieving data more often, decreasing the size of the smaller dividing blocks, or reducing the sampling rate. The large acquisition buffer and smaller dividing block sizes are configured in `DAQ2Config`.

An error occurs if analog triggering has been enabled for a single-buffered acquisition (see `DAQ_Trigger`) and analog trigger conditions are not met before the specified timeout value expires.

In pretrigger mode, `DAQ_Check` automatically rearranges the array upon completion of the acquisition so that the oldest data point is at the beginning of the array.

DAQ_Clear

Function

Cancels the current data acquisition operation and reinitializes the data acquisition circuitry.

Synopsis

C Syntax	`locus i32 DAQ_Clear(u32 deviceNumber);`
Pascal Syntax	`function DAQ_Clear(deviceNumber : i32) : i32;`
BASIC Syntax	`FN DAQ_Clear(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`DAQ_Clear` cancels the current data acquisition operation. You should execute `DAQ_Clear` whenever you want to halt data acquisition operation. Any acquired data is lost.

If NI-DAQ for Macintosh is configured for double-buffered data acquisition (see `DAQ2Config`), then either `DAQ_Clear` or `DAQ2Clear` can be used to stop the current double-buffered acquisition. `DAQ_Clear` stops the current double-buffered acquisition but leaves NI-DAQ for Macintosh configured for double-buffered mode. `DAQ2Clear` stops the current acquisition and disables double-buffered mode.

DAQ_Config

Function

Stores configuration information for subsequent data acquisition operations.

Synopsis

C Syntax	`locus i32 DAQ_Config(u32 deviceNumber, u32 externalTrigger, u32 externalGate, u32 externalConvert);`
Pascal Syntax	`function DAQ_Config(deviceNumber : i32; externalTrigger : i32; externalGate : i32; externalConvert : i32) : i32;`
BASIC Syntax	`FN DAQ_Config(deviceNumber&, externalTrigger&, externalGate&, externalConvert&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

externalTrigger indicates whether the trigger to initiate data acquisition sequences is externally generated.

0: generate software trigger to start data acquisition sequence.
1: wait for external trigger to initiate data acquisition sequence (not supported on the DAQCard-500 and DAQCard-700).

externalGate indicates whether to enable external gating of data acquisition. **externalGate** is only used on the NB-MIO-16. See `Select_Signal` for enabling external gating on E Series boards.

0: disable external gating.
1: enable external gating.

externalConvert indicates whether timing of A/D conversions during the data acquisition sequence is controlled externally or internally with the sample-interval clock.

0: use onboard sample-interval clock to control data acquisition A/D conversions.
1: allow external clock to control data acquisition A/D conversions.
2: allow external clock to control scan-interval timing.
3: allow external control of sample-interval and scan-interval timing.

Note: ***When using an external clock to control A/D conversions on an NB-MIO-16X, be sure to call*** `MIO_16X_Config` ***before starting the acquisition***.

NB-MIO-16 or NB-MIO-16X Configuration

Only the following combinations of **externalTrigger**, **externalGate**, and **externalConvert** are valid on the NB-MIO-16 and
NB-MIO-16X:

externalTrigger	externalGate	externalConvert
0	0	0
1	0	0
0	1	0
0	0	1

The first setting causes A/D conversions to begin as soon as `DAQ_Start`, `SCAN_Start`, or `SCAN_IntStart` is called. The second setting causes A/D conversions to begin after a pulse is applied on the I/O connector once `DAQ_Start`, `SCAN_Start`, or `SCAN_IntStart` is called. The pulse is applied to the EXTTRIG* input on the NB-MIO-16. On the NB-MIO-16X, the pulse is applied to the STARTTRIG* input. The third setting causes A/D conversions to occur while the EXTGATE signal is high after `DAQ_Start` or `SCAN_Start` is called. External gating is available only on the NB-MIO-16. In these three cases, the sample interval is timed by the onboard counter/timer. In the fourth case, individual A/D conversions are caused by pulses applied to the EXTCONV* line. In this case, the sample interval is determined by the period of the pulse applied. More information about data acquisition timing on the NB-MIO-16 or NB-MIO-16X is given at the beginning of this chapter and under the descriptions of `DAQ_Start`, `SCAN_Start`, and `SCAN_IntStart`.

Note: ***In most cases, external conversion pulses should not be used in scanning operations when SCXI is being used in multiplexed mode. The NB-MIO-16 and NB-MIO-16X have no way of masking conversions before the data acquisition begins, so any conversion pulses that occur before the acquisition is triggered will advance the SCXI channels.***

Lab and 1200 Series Configuration

Only the following combinations of **externalTrigger**, **externalGate**, and **externalConvert** are valid on the Lab and 1200 series boards. An X signifies external gating is not used on the Lab and 1200 series boards and therefore **externalGate** can be any value:

externalTrigger	externalGate	externalConvert
0	X	0
1	X	0
0	X	1
1	X	1

The first setting causes A/D conversion to begin as soon as `DAQ_Start` or `Lab_ISCAN_Start` is called. The second setting causes A/D conversions to begin after a pulse is applied on the I/O connector once `DAQ_Start` or `Lab_ISCAN_Start` is called. The pulse is applied to the EXTTRIG input. In these two

cases, the sample interval is timed by an onboard counter/timer. In the third case, individual A/D conversions are caused by pulses applied to the EXTCONV* line. In this case, the sample interval is determined by the period of the pulse applied. In the fourth case, the signal applied to the EXTCONV* line is ignored until a low-to-high pulse is applied at the EXTTRIG input. After this trigger has been received, individual A/D conversions are caused by pulses applied to the EXTCONV* line. The first falling edge, following a rising edge, generates the first A/D conversion in the case when external conversion pulses are used. More information about data acquisition timing on the Lab and 1200 series is given at the beginning of this chapter and under the descriptions of `DAQ_Start` and `Lab_ISCAN_Start`.

All Devices

`DAQ_Config` saves the parameter values in the configuration table for data acquisition. The configuration table is used by `DAQ_Start`, `SCAN_Start`, `SCAN_IntStart`, and `Lab_ISCAN_Start` to set up the data acquisition circuitry to the correct timing modes.

The default settings for data acquisition modes after system startup are as follows:

externalTrigger = 0: data acquisition sequences are initiated through software.
externalGate = 0: external gating of data acquisition is disabled.
externalConvert = 0: onboard sample-interval clock is used to time A/D conversions.

If you want a data acquisition timing configuration that is different from the default setting, then you must call `DAQ_Config` with the desired configuration before any data acquisition sequences are initiated. `DAQ_Config` needs to be called only when the data acquisition configuration is changed.

The configuration information for the analog input circuitry is controlled by the `AI_Config` call. This configuration information also affects data acquisition. After system startup, the analog input configuration table defaults to the following values:

For the MIO boards:
input_mode = 0: differential.
input_range = 10 V.
polarity = 0: bipolar.

For the Lab and 1200 series boards:
input_mode = 1: single-ended.
polarity = 0: bipolar (-5 to +5 V).

For the DAQCard-500:
input_mode = 1: single-ended.
input_range = 10 V.
polarity = 0: bipolar (-5 to +5 V).

For the DAQCard-700:
input_mode = 1: single-ended.
input_range = 10 V.
polarity = 0: bipolar (-5 to +5 V).

where **input_mode**, **input_range**, and **polarity** are specified in the `AI_Config` call.

If the physical configuration of the analog input circuitry on your board differs from these settings, you must call `AI_Config` with the correct configuration information in order for the remaining Data Acquisition functions to operate properly.

DAQ_PreTrig

Function

Stores pretrigger information for stopping single-buffered data acquisition when a specified number of samples have been acquired *after* the occurrence of an external trigger. Pretriggering is not available on the NB-MIO-16.

Synopsis

C Syntax	`locus i32 DAQ_PreTrig(u32 deviceNumber, u32 alternateTrigger, u32 count);`
Pascal Syntax	`function DAQ_PreTrig(deviceNumber : i32; alternateTrigger : i32; count : i32) : i32;`
BASIC Syntax	`FN DAQ_PreTrig(deviceNumber&, alternateTrigger&, count&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

alternateTrigger enables or disables pretriggering. If pretriggering is enabled, subsequent single-buffered data acquisition operations acquire data continuously until the external stop trigger occurs and the specified number of samples after the trigger have been acquired.

0: disable pretriggering.
1: enable pretriggering.

count indicates how many samples to acquire *after* the external stop trigger has occurred before actually stopping data acquisition.

Range: 2 through 2^{24} for the E Series.
3 to 2^{31}-1 for the NB-MIO-16X.
3 to 65,535 for the Lab and 1200 series.

Note: `DAQ_PreTrig` ***enables pretriggering for subsequent single-buffered (***`DAQ_Start`**,** `SCAN_Start`**,** `SCAN_IntStart`**, and** `Lab_ISCAN_Start`***) data acquisition operations. See*** `DAQ2TGet` ***and*** `DAQ2TTap` ***for enabling pretriggering with double-buffered (***`DAQ2`***) data acquisitions.***

Lab-NB and Lab-LC Note: ***If the on-board sample interval clock is used to time A/D conversions, you may get one extra sample after the trigger. So, if you configured with* count = 10*, you will get 10 samples after the trigger if the trigger occurred when the conversion signal is low. You will get 11 samples after the trigger if the trigger occurred when the conversion signal was high.***

If an external clock is used to time A/D conversions, you may get one or two extra samples after the trigger. So, if you configured with* count = 10*, you will get either 10, 11, or 12 samples after the trigger, depending on when the trigger occurred.

A pretriggered acquisition can be implemented by first calling `DAQ_PreTrig` with **alternateTrigger** set to 1 and with **count** set to the number of samples to acquire after an external trigger occurs. `DAQ_Start`, `SCAN_Start`, `SCAN_IntStart`, or `Lab_ISCAN_Start` can be used to indicate the total number of samples to acquire and to start the acquisition. A trigger applied at the external trigger iput o the connector triggers NI-DAQ to acquire the number of samples specified by **count** and then stop the acquisition. For example, if `DAQ_PreTrig` is called to enable pretriggering with **count** set to 20 and `DAQ_Start` is called to acquire 100 samples, then 100 samples are returned after a trigger is applied and `DAQ_Check` indicates that the acquisition is complete. The first 80 samples in the returned array occurred immediately before the trigger; the last 20 were sampled after the trigger.

DAQ_Start

Function

Initiates a single-channel data acquisition operation and stores the results in an array.

Synopsis

C Syntax	`locus i32 DAQ_Start(u32 deviceNumber, u32 channel, u32 gain, i16 *buffer, u32 count, u32 timebase, u32 sampleInterval);`
Pascal Syntax	`function DAQ_Start(deviceNumber : i32; channel : i32; gain : i32; buffer : pi16; count : i32; timebase : i32; sampleInterval : i32) : i32;`
BASIC Syntax	`FN DAQ_Start(deviceNumber&, channel&, gain&, buffer&, count&, timebase&, sampleInterval&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog input channel number. If SCXI is being used, you must use the appropriate analog input channel on the DAQ board that corresponds to the desired SCXI channel. Please refer to Chapter 7, *SCXI Functions*, for more information on SCXI channel assignments.

Range: 0 through *n*-1, where *n* is the number of analog input channels available.

gain is the gain setting to be used for the specified channel. Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for valid gain settings. If you use invalid gain settings, NI-DAQ returns an error.

This gain setting applies only to the DAQ board; if SCXI is used, any gain desired at the SCXI module must be established either by setting jumpers on the module or by calling `SCXI_Set_Gain`.

buffer is a buffer of length **count.** When `DAQ_Check` returns **status** = 1, then **buffer** contains the acquired data. The elements of **buffer** are the results of each A/D conversion in the data acquisition operation. The elements of **buffer** are integers (16-bit values).

Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for the range of each element for **buffer**.

Note: ***If NI-DAQ for Macintosh has been configured for double-buffered mode (see*** `DAQ2Config`***), then this*** **buffer** ***parameter is not used and must be 0. The larger buffer allocated by*** `DAQ2Config` ***is used as the acquisition buffer for double-buffered acquisitions.*** `DAQ2Get`**,** `DAQ2TGet`**,** `DAQ2Tap`**,** ***and*** `DAQ2TTap` ***can be used to acquire blocks of data from the double-buffered acquisition in progress. (See*** *Starting a Double-Buffered Acquisition with* `DAQ_Start`***.)***

count is the number of samples to be acquired (that is, the number of A/D conversions to be performed).

Range: 3 through 2^{31}-1. (With DMA on an MIO board, the range is limited to 3 through 2^{23}.)

Note: ***If NI-DAQ for Macintosh is configured for*** *continuous* ***double-buffered acquisition (see*** `DAQ2Config`***), then the*** **count** ***parameter is ignored and should be 0. In continuous mode, the total number of samples to acquire is not indicated and the data acquisition runs continuously until you stop the process by executing*** `DAQ_Clear` ***or*** `DAQ2Clear`***. (See*** *Starting a Double-Buffered Acquisition with* `DAQ_Start`***.)***

timebase is the resolution to be used for the sample-interval counter. **timebase** has the following possible values:

Most devices:

0: External clock used as timebase (SOURCE5 input) (NB-MIO-16 or NB-MIO-16X).
1: 1-MHz clock used as timebase (1-µs resolution).
2: 100-kHz clock used as timebase (10-µs resolution).
3: 10-kHz clock used as timebase (100-µs resolution).
4: 1-kHz clock used as timebase (1-ms resolution).
5: 100-Hz clock used as timebase (10-ms resolution).

E Series:

-3: 20-MHz clock used as timebase (50-ns resolution).
0: If you use this function with the timebase set at 0, you must call `Select_Signal` with **signal** set to `ND_IN_SCAN_CLOCK_TIMEBASE` (not `ND_IN_CHANNEL_CLOCK_TIMEBASE`), and **source** set to a value other than `ND_INTERNAL_20_MHZ` and `ND_INTERNAL_100_KHZ` before calling `DAQ_Start` with **timebase** set to 0; otherwise, `DAQ_Start` will select low-to-high transitions on the PFI 8 I/O connector pin as your external timebase.
2: 100 kHz clock used as timebase (10 µs resolution).

If sample-interval timing is to be externally controlled, the **timebase** parameter is ignored and can be any value. When using external timing sources with the NB-MIO-16X, be sure to call `MIO_16X_Config` before starting the acquisition.

sampleInterval indicates the length of the sample interval (that is, the amount of time to elapse between each A/D conversion).

Range: 2 through 65,536.
2 through 2^{24} (E Series)

The sample interval is a function of the timebase resolution. The actual sample interval in seconds is determined by the following formula:

sampleInterval ∗ (timebase resolution)

where the timebase resolution for each value of **timebase** is specified above. For example, if **sampleInterval** = 25 and **timebase** = 2, the sample interval is 25 ∗ 10 µs = 250 µs. If the sample interval is to be externally controlled, the **sampleInterval** parameter is ignored and can be any value.

Note: ***(Lab-NB and Lab-LC only) Do not drive GATB0 input low on the Lab-NB or Lab-LC I/O connector if the actual sample interval is greater than 65,535 µs. In this case, Counter B0 is used to give the actual sample interval. Refer to*** *Lab and 1200 Series Counter/Timer Signals* ***earlier in this chapter.***

Using This Function

`DAQ_Start` initiates a single-channel data acquisition either in single-buffered or double-buffered mode. For both modes, `DAQ_Start` configures the analog input multiplexer and gain circuitry as indicated by **channel** and **gain**. If external sample-interval timing has not been selected by a call to `DAQ_Config`, the sample-interval counter is set to the specified **sampleInterval** and **timebase** parameters. If external sample-interval timing has been selected, then the data acquisition circuitry relies on pulses received on the EXTCONV* input to initiate individual A/D conversions, and the **sampleInterval** and **timebase** parameters are ignored.

If external gating of the data acquisition operation has been selected, a signal at the EXTGATE I/O connector input on the NB-MIO-16 controls the sample-interval counter. When the EXTGATE signal is high, the sample-interval counter is enabled, causing A/D conversions to occur. When the EXTGATE signal is low, the sample-interval counter is suspended and no A/D conversions occur. External gating is available only on the NB-MIO-16.

Starting a Single-Buffered Acquisition with DAQ_Start

In a single-buffered acquisition, the DAQ_Start call specifies the number of samples to acquire (**count**) and an integer array to store the acquired data (**buffer**). After DAQ_Start has returned, the background process stores up to **count** A/D conversions into the **buffer** and ignores any subsequent conversions. The acquired samples are available when the DAQ_Check call returns **status** = 1. A second call to DAQ_Start cannot be made without terminating this background process. If a call to DAQ_Check returns **status** = 1, the samples are available and the process is terminated. A call to DAQ_Clear also terminates a background data acquisition process.

Starting a Double-Buffered Acquisition with DAQ_Start

In a double-buffered acquisition, data can be returned from an acquisition in progress without interrupting the acquisition. NI-DAQ for Macintosh can be configured for double-buffered mode by executing DAQ2Config before DAQ_Start is called. DAQ2Config allocates a large internal circular buffer for the data storage and configures subsequent data acquisitions for double-buffered mode.

In double-buffered mode, DAQ_Start ignores the **buffer** parameter. Once DAQ_Start completes with **error** = 0, NI-DAQ for Macintosh acquires and stores the A/D conversions into the large buffer allocated by DAQ2Config. This buffer is treated as a circular buffer and is continually filled with data until **count** samples are acquired. If *continuous* double-buffered mode has been specified in DAQ2Config, then the total number of samples is not specified and the **count** value in DAQ_Start is ignored. The data acquisition continues to run until the process is stopped by executing DAQ_Clear or DAQ2Clear.

Smaller blocks of data can be retrieved from the large internal buffer without interrupting the acquisition by repeatedly executing the DAQ2Get, DAQ2TGet, DAQ2Tap, and DAQ2TTap retrieval functions. An integer array to store the acquired data and the number of samples to retrieve are passed to the retrieval functions. The array is returned with a copy of a block of data from the internal circular buffer. (See the descriptions of DAQ2Get, DAQ2TGet, DAQ2Tap, and DAQ2TTap for more information on retrieving double-buffered data.)

Using DAQ_Start to Start a Trigger Acquisition Using Single-Buffered Mode

NI-DAQ for Macintosh can be configured for external hardware triggering using DAQ_Config. If you select external triggering for the data acquisition operation, a high-to-low edge at the EXTTRIG* I/O connector input on the NB-MIO-16 or at the STARTTRIG* input on the NB-MIO-16X, or a low-to-high edge at the EXTTRIG I/O connector input on the Lab and 1200 series initiates the data acquisition operation after the DAQ_Start call is complete. Otherwise, DAQ_Start issues a software trigger to initiate the data acquisition operation before returning.

Data acquisition also can be triggered on the slope and level of the analog input values. DAQ_Trigger stores trigger information and enables triggering on analog input values for subsequent single-buffered acquisitions. When executing DAQ_Trigger, you indicate a trigger channel, slope, and level. A trigger occurs when the analog input values of the trigger channel are within the specified slope and level.

If a single-buffered acquisition is started with triggering enabled, DAQ_Start waits for triggering conditions to be met before collecting the array of sample data.

Pretriggering can be implemented for single-buffered acquisitions. With pretriggering, data acquisition is stopped when a specified number of samples have been acquired after the occurrence of an external trigger. See the description of DAQ_PreTrig for more information on pretriggering.

Using DAQ_Start to Start a Trigger Acquisition Using Double-Buffered Mode

NI-DAQ for Macintosh can be configured for external hardware triggering for double-buffered mode also by using DAQ_Config. External triggering with double-buffering operates differently on the NB-MIO-16, Lab-NB, and Lab-LC from the way it works on the NB-MIO-16X.

If you select external triggering for the data acquisition operation (DAQ_Config) on the NB-MIO-16, Lab and 1200 series, or Lab-LC, a high-to-low edge at the EXTTRIG* I/O connector input initiates the data acquisition operation after DAQ_Start begins execution. Otherwise, DAQ_Start issues a software trigger to initiate the data acquisition operation before returning.

If external triggering is disabled (in DAQ_Config) on the NB-MIO-16X, a software trigger is issued to initiate *each block* of the data acquisition operation. Otherwise, if you select external triggering for the data acquisition operation, a high-to-low edge at the STARTTRIG* I/O connector input on the NB-MIO-16X, initiates *each block* of the data acquisition operation after DAQ_Start begins execution. When a high-to-low edge is received, the number of samples in a block (specified in DAQ2Config) are then acquired. NI-DAQ for Macintosh then waits for another high-to-low edge before the next block of data is acquired.

Data acquisition also can be triggered on the slope and level of the analog input values. For double-buffered acquisition, triggering conditions can be specified for each retrieved block of data. A trigger channel, slope, and level can be specified in the DAQ2Get, DAQ2TGet, DAQ2Tap, and DAQ2TTap functions to implement condition triggering for double-buffered acquisitions. A trigger occurs when the analog input values of the trigger channel are within the specified slope and level.

Using the AMUX-64T with DAQ_Start

For details on using the AMUX-64T with Data Acquisition functions on an NB-MIO-16, NB-MIO-16X, or E Series device, see Appendix C, *Using an External Multiplexer.*

NuBus DMA

If an NB-DMA-8-G or NB-DMA2800 is not detected in the system, then the NB-MIO-16 and NB-MIO-16X use interrupts to acquired the data. Double-buffered acquisitions that use interrupts require a sampling interval of at least 120 µs. An **overFlowError** is returned if no DMA is used and the sampling interval is less than 120 µs for double-buffered acquisitions. (See *Starting a Single-Buffered Acquisition with DAQ_Start* if faster sampling rates are needed for non-DMA acquisitions.)

DAQ_Start initializes a background process to handle storage of A/D conversion samples as they occur. If an NB-DMA-8-G or NB-DMA2800 board is detected in the system, a DMA process is initialized to automatically handle data acquisition on an NB-MIO-16 and NB-MIO-16X. In either case, the background process handles incoming data after DAQ_Start has returned.

DAQ_Trigger

Function

Stores analog trigger configuration information for subsequent single-buffered data acquisition operations.

Synopsis

C Syntax	`locus i32 DAQ_Trigger(u32 deviceNumber, u32 triggerChannel, u32 triggerSlope, i32 triggerLevel, u32 timeout);`
Pascal Syntax	`function DAQ_Trigger(deviceNumber : i32; triggerChannel : i32; triggerSlope : i32; triggerLevel : i32; timeout : i32) : i32;`
BASIC Syntax	`FN DAQ_Trigger(deviceNumber&, triggerChannel&, triggerSlope&, triggerLevel&, timeout&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

triggerChannelis the analog input channel number to trigger on in subsequent data acquisitions. If **triggerSlope** is 0, no triggering is performed and this **triggerChannel**value is ignored. If SCXI is used, this parameter should be the onboard channel number.

Range: 0 through *n*-1, where *n* is the number of analog input channels available.

triggerSlope is the slope to trigger on. Triggering is disabled by setting **triggerSlope** to 0.

0: no triggering is performed.
1: negative slope.
2: positive slope.

triggerLevel is the analog input value to trigger on. If **triggerSlope** is 0, no triggering is performed and **triggerLevel** is ignored.

Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for the range of values for **triggerLevel**

timeout is the number of ticks (60ths of a second) to wait for the occurrence of an analog trigger after data acquisition starts. `DAQ_Check`, `SCAN_Check`, and `Lab_ISCAN_Check` return an error if analog input trigger conditions are not met before the specified number of ticks expire. If **timeout** is 0, no time limit for triggering is imposed, in which case data is not collected until trigger conditions are met.

If SCXI is being used, analog triggering is possible during single-channel acquisitions and during multiple-channel scanning acquisitions *if the SCXI modules are operated in Parallel mode*. Analog triggering is not possible during multiple-channel scanning if the SCXI modules are operated in Multiplexed mode. When analog triggering is used with SCXI, the **triggerChannel**parameter specified refers to the DAQ board channel number. Refer to Chapter 7, *SCXI Functions*, for more information on SCXI operating modes and channel assignments.

Analog triggering is enabled for single-buffered data acquisitions (single-channel and multiple-channel scan) by executing `DAQ_Trigger` with **triggerSlope** set to 2 (positive slope) or **triggerSlope** set to 1 (negative slope). All subsequent calls to `DAQ_Start`, `SCAN_Start`, `SCAN_IntStart`, or `Lab_ISCAN_Start` that acquire data from **triggerChannel**wait for analog trigger conditions to be met before collecting the requested number of samples. An analog trigger occurs when the analog input values are within the specified **triggerSlope** and **triggerLevel**. If analog triggering is enabled and a data acquisition operation is performed on a channel other than **triggerChannel**, then no triggering is performed before the samples are returned. Executing `DAQ_Trigger` with **triggerSlope** set to 0 disables the analog triggering feature.

Analog triggering is implemented by software on the NB-MIO-16X, NB-MIO-16, PCI-MIO-16XE-50, Lab and 1200 series, DAQCard-500, and DAQCard-700. NI-DAQ for Macintosh inspects each sampled data point for the trigger condition. When the trigger condition is met, the acquisition sequence specified by `DAQ_Start`, `SCAN_Start`, `SCAN_IntStart`, or `Lab_ISCAN_Start` begins. The acquisition of data before the trigger condition is met *does not* use any available DMA processor. Therefore, attempts to use analog triggering at very high sampling rates may result in an overflow of the FIFO.

Note: `DAQ_Trigger` ***permits analog triggering for subsequent SINGLE-BUFFERED (***`DAQ_Start`***, ***`SCAN_Start`***, and*** `SCAN_IntStart`***) data acquisition operations. See*** `DAQ2Get`***,*** `DAQ2TGet`***,*** `DAQ2Tap`***, and*** `DAQ2TTap` ***for enabling analog triggering with double-buffered (***`DAQ2`***) data acquisition on an NB-MIO-16 or NB-MIO-16X.***

DAQ_VScale

Function

Converts the values of an array of acquired binary data and the gain setting for that data to actual input voltages measured.

Synopsis

C Syntax	`locus i32 DAQ_VScale(u32 deviceNumber, u32 channel, u32 gain, f64 gainAdjust, f64 offset, u32 count, i16 *readings, f64 *voltages);`
Pascal Syntax	`function DAQ_VScale(deviceNumber : i32; channel : i32; gain : i32; gainAdjust : f64; offset : f64; count : i32; readings : pi16; voltages : pf64) : i32;`
BASIC Syntax	`FN DAQ_VScale(deviceNumber&, channel&, gain&, gainAdjust#, offset#, count&, readings&, voltages&)`

Description

channel is the onboard channel or AMUX channel on which the binary data was acquired. For devices other than E Series devices, this parameter is ignored because the scaling calculation is the same for all of the channels. However, you are encouraged to pass the correct channel number.

gain is the gain setting at which NI-DAQ acquired the data in **binArray**. If you used SCXI to take the reading, this gain parameter should be the product of the gain on the SCXI module channel and the gain used by the DAQ device.

gainAdjust is the multiplying factor to adjust the gain. Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for the procedure for determining **gainAdjust**. If you do not want to do any gain adjustment, (for example, the ideal gain as specified by the parameter **gain**) you must set **gainAdjust** to 1.

offset is the binary offset that needs to be subtracted from **reading**. Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for the procedure for determining offset. If you do not want to do any offset compensation, **offset** must be set to zero. The data type is double to allow for offset fractional LSBs. For example, you could use `DAQ_Op` to acquire many samples from a grounded input channel and average them to obtain the offset.

count is the length of **binArray** and **voltArray**.

binArray is an array of acquired binary data.

voltArray is an array of double-precision values returned by `DAQ_VScale` and is the voltage representation of **binArray**.

Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for the formula used by `DAQ_VScale` to calculate voltages from binary readings.

Lab_ISCAN_Check

Function

Checks whether the current multiple-channel scanned data acquisition begun by the `Lab_ISCAN_Start` function is complete and returns the status, the number of samples acquired to that point, and the scanning order of the channels in the data array (DAQCard-500, DAQCard-700, and Lab and 1200 series boards only).

Synopsis

C Syntax	`locus i32 Lab_ISCAN_Check(u32 deviceNumber, u16 *status, u32 *retrieved, u16 *scanOrder);`
Pascal Syntax	`function Lab_ISCAN_Check(deviceNumber : i32; var status : i16; var retrieved : i32; scanOrder : pi16) : i32;`
BASIC Syntax	`FN Lab_ISCAN_Check(deviceNumber&, status&, retrieved&, scanOrder&)`

Description

daqStopped returns an indication of whether the data acquisition has completed.

1: The data acquisition operation has stopped. Either NI-DAQ has acquired all the samples or an error has occurred.

0: The data acquisition operation is not yet complete.

retrieved indicates the progress of an acquisition. The meaning of **retrieved** depends on whether you have enabled pretrigger mode (see `DAQ_Pretrig`).

If pretrigger mode is disabled, **retrieved** returns the number of samples collected by the acquisition at the time of the call to `Lab_ISCAN_Check`. The value of **retrieved** increases until it equals the total number of samples to be acquired, at which time the acquisition terminates.

However, if pretrigger mode is enabled, **retrieved** returns the offset of the position in your buffer where NI-DAQ places the next data point when the function acquires. After the value of **retrieved** reaches **count** - 1 and rolls over to 0, the acquisition begins to overwrite old data with new data. When you apply a signal to the stop trigger input, the acquisition collects an additional number of samples specified by **ptsAfterStoptrig** in the call to `DAQ_Pretrig` and then terminates. When `Lab_ISCAN_Check` returns a status of 1, **retrieved** contains the offset of the oldest data point in the array (assuming that the acquisition has written to the entire buffer at least once). In pretrigger mode, `Lab_ISCAN_Check` automatically rearranges the array upon completion of the acquisition so that the oldest data point is at the beginning of the array. Thus, **retrieved** always equals 0 upon completion of a pretrigger mode acquisition. Since the stop trigger can occur in the middle of a scan sequence, the acquisition can end in the middle of a scan sequence. So, when the function rearranges the data in the buffer, the first sample may not belong to the first channel in the scan sequence. You can examine the **finalScanOrder** array to find out the way the data is arranged in the buffer.

finalScanOrder is an array that indicates the scan channel order of the data in the buffer passed to `Lab_ISCAN_Start`. The size of **finalScanOrder** must be at least equal to the number of channels scanned. This parameter is valid only when NI-DAQ returns **daqStopped** as 1 and is useful only when you enable pretrigger mode.

If you do not use pretrigger mode, the values contained in **finalScanOrder** are, in single-ended mode, *n*-1, *n*-2, ...1, 0 to 0, in that order, and in differential mode, 2*(*n*-1), 2*(*n*-2), ..., 2, 0, in that order, where *n* is the number of channels scanned. For example, if you scanned three channels in single-ended mode, the **finalScanOrder** returns:

finalScanOrder[0] = 2.
finalScanOrder[1] = 1.
finalScanOrder[2] = 0.

So the first sample in the buffer belongs to channel 2, the second sample belongs to channel 1, the third sample belong to channel 0, the fourth sample belongs to channel 2, and so on. This is the scan order expected from the device and **finalScanOrder** is not useful in this case.

If you use pretrigger mode, the order of the channel numbers in **finalScanOrder** depends on where in the scan sequence the acquisition ended. This can vary because the stop trigger can occur in the middle of a scan sequence, which would cause the acquisition to end in the middle of a scan sequence so that the oldest data point in the buffer can belong to any channel in the scan sequence. `Lab_ISCAN_Check` rearranges the buffer so that the oldest data point is at index 0 in the buffer. This rearrangement causes the scanning order to change. This new scanning order is returned by **finalScanOrder**. For example, if you scanned three channels, the original scan order is channel 2, channel 1, channel 0, channel 2, channel 1, channel 0, and so on. However, after the stop trigger, if the acquisition ends after taking a sample from channel 1, the oldest data point belongs to channel 0. So **finalScanOrder** returns:

finalScanOrder[0] = 0.
finalScanOrder[1] = 2.
finalScanOrder[2] = 1.

So the first sample in the buffer belongs to channel 0, the second sample belongs to channel 2, the third sample belongs to channel 1, the fourth sample belongs to channel 0, and so on.

`Lab_ISCAN_Check` checks the current background data acquisition operation to determine whether it has completed and returns the number of samples acquired at the time that you called `Lab_ISCAN_Check`. If the operation is complete, `Lab_ISCAN_Check` sets **daqStopped** = 1. Otherwise, **daqStopped** is set to 0. Before `Lab_ISCAN_Check` returns **daqStopped** = 1, it calls `DAQ_Clear`, allowing another Start call to execute immediately.

If `Lab_ISCAN_Check` returns an **overFlowError** or an **overRunError**, NI-DAQ has terminated the data acquisition operation because of lost A/D conversions due to a sample rate that is too high (sample interval was too small). An **overFlowError** indicates that the A/D FIFO memory overflowed because the data acquisition servicing operation was not able to keep up with sample rate. An **overRunError** indicates that the data acquisition circuitry was not able to keep up with the sample rate. Before returning either of these error codes, `Lab_ISCAN_Check` calls `DAQ_Clear` to terminate the operation and reinitialize the data acquisition circuitry.

Lab_ISCAN_Start

Function

Initiates a multiple-channel scanned data acquisition operation and stores its input in an array (DAQCard-500, DAQCard-700, and Lab and 1200 series only).

Synopsis

C Syntax	`locus i32 Lab_ISCAN_Start(u32 deviceNumber, u32 channelCount, u32 gain, i16 *buffer, u32 count, u32 timebase, u32 sampleInterval, u32 scanInterval);`
Pascal Syntax	`function Lab_ISCAN_Start(deviceNumber : i32; channelCount : i32; gain : i32; buffer : pi16; count : i32; timebase : i32; sampleInterval : i32; scanInterval : i32) : i32;`
BASIC Syntax	`FN Lab_ISCAN_Start(deviceNumber&, channelCount&, gain&, buffer&, count&, timebase&, sampleInterval&, scanInterval&)`

Description

channelCount is the number of channels to be scanned in a single scan sequence. The value of this parameter also determines which channels NI-DAQ scans because these supported devices have a fixed scanning order. The scanned channels range from **channelCount** - 1 to channel 0. If you are using SCXI modules with additional multiplexers, you must scan the appropriate analog input channels on the DAQ device that corresponds to the SCXI channels you want. You should select the SCXI scan list using `SCXI_SCAN_Setup` before you call this function.

Range:

Lab and 1200 Series:

1 through 4 in differential mode (except on the Lab-NB and Lab-LC)

1 through 8 in single-ended mode.

DAQCard-500

0 through 7 (single-ended)

DAQCard-700

0 through 15 (single-ended)

0 through 7 (differential)

gain is the gain setting to be used for the scanning operation. NI-DAQ applies the same gain to all the channels scanned. This gain setting applies only to the DAQ device; if you are using SCXI modules with additional gain selection, you must establish any gain you want at the SCXI module either by setting jumpers on the module or by calling `SCXI_Set_Module_Gain`. The following gain settings are valid: 1, 2, 5, 10, 20, 50, 100. If you use an invalid gain setting, NI-DAQ returns an error.

buffer is an integer array**.** **buffer** must have a length equal to or greater than **count**.

count is the total number of samples to be acquired (that is, the number of A/D conversions to be performed). For double-buffered acquisitions, **count** must be even.

Range: 3 through 2^{32} - 1

timebase is the timebase, or resolution, to be used for the sample-interval counter. The sample-interval counter controls the time that elapses between acquisition of samples within a scan sequence.

timebase has the following possible values:

1: 1 MHz clock used as timebase (1 µs resolution).
2: 100 kHz clock used as timebase (10 µs resolution).
3: 10 kHz clock used as timebase (100 µs resolution).
4: 1 kHz clock used as timebase (1 ms resolution).
5: 100 Hz clock used as timebase (10 ms resolution).

If sample-interval timing is to be externally controlled, NI-DAQ ignores **timebase** and the parameter can be any value.

sampleInterval indicates the length of the sample interval (that is, the amount of time to elapse between each A/D conversion within a scan sequence).

Range: 2 through 65,535.

The sample interval is a function of the timebase resolution. NI-DAQ determines the actual sample interval in seconds by the following formula:

sampleInterval ∗ (sample timebase resolution)

where the sample timebase resolution is equal to one of the values of **timebase** as specified above. For example, if **sampleInterval** = 25 and **timebase** = 2, the actual sample interval is 25 ∗ 10 µs = 250 µs. The total sample interval (the time to complete one scan sequence) in seconds is the actual sample interval * number of channels scanned. If the sample interval is to be externally controlled by conversion pulses applied to the EXTCONV* input, NI-DAQ ignores the **sampleInterval** and the parameter can be any value.

scanInterval indicates the length of the scan interval. This is the amount of time to elapse between scans. The function performs a scan each time NI-DAQ samples all channels in the scan sequence. Therefore, **scanInterval** must be greater than or equal to **sampleInterval** * **channelCount** +5 µs. This value must be 0 for the Lab-LC and Lab-NB because **scanInterval** is not supported for these boards.

Range: 0 and 2 through 65,535.
A value of 0 disables interval scanning.

If you did not specify external sample-interval timing by the `DAQ_Config` call, NI-DAQ sets the sample-interval counter to the specified **sampleInterval** and **timebase**, and sets the sample counter up to count the number of samples acquired and to stop the data acquisition process when the number of samples acquired equals **count**. If you have specified external sample-interval timing, the data acquisition circuitry relies on pulses received on the EXTCONV* input to initiate individual A/D conversions.

`Lab_ISCAN_Start` initializes a background data acquisition process to handle storing of A/D conversion samples into the buffer as NI-DAQ acquires them. When you use posttrigger mode (with pretrigger mode disabled), the process stores up to **count** A/D conversion samples into the buffer and ignores any subsequent conversions. The order of the scan is from channel *n-1* to channel 0, where *n* is the number of channels being scanned. For example, if **channelCount** is 3 (that is, you are scanning three channels), NI-DAQ stores the data in the buffer in the following order:

First sample from channel 2, first sample from channel 1, first sample from channel 0, second sample from channel 2, and so on.

You cannot make the second call to `Lab_ISCAN_Start` without terminating this background data acquisition process. If a call to `Lab_ISCAN_Check` returns **daqStopped** = 1, the samples are available and NI-DAQ terminates the process. In addition, a call to `DAQ_Clear` terminates the background data acquisition process. Notice that if a call to `Lab_ISCAN_Check` returns **overFlowError** or **overRunError**, or **daqStopped** = 1, the process is automatically terminated and there is no need to call `DAQ_Clear`.

If you enable pretrigger mode, `Lab_ISCAN_Start` initiates a cyclical acquisition that continually fills the buffer with data, wrapping around to the start of the buffer once NI-DAQ has written to the entire buffer. When you apply the signal at the stop trigger input, `Lab_ISCAN_Start` acquires an additional number of samples specified by the **ptsAfterStoptrig** parameter in `DAQ_StopTrigger_Config` and then terminates.

Since the trigger can occur at any point in the scan sequence, the scanning operation can end in the middle of a scan sequence. See the description for `Lab_ISCAN_Check` to determine how NI-DAQ rearranges the buffer after the acquisition ends. When you enable pretrigger mode, the length of the buffer, which is greater than or equal to **count**, should be an integral multiple of **channelCount**.

If you have selected external start triggering of the data acquisition operation, a low-to-high edge at the EXTTRIG I/O connector input initiates the data acquisition operation after the `Lab_ISCAN_Start` call is complete. Otherwise, `Lab_ISCAN_Start` issues a software trigger to initiate the data acquisition operation before returning.

SCAN_Check

Function

Checks to see whether the current scanned data acquisition operation is complete, and returns its status.

Synopsis

C Syntax	`locus i32  SCAN_Check(u32 deviceNumber, u16 *status);`
Pascal Syntax	`function  SCAN_Check(deviceNumber : i32; var status : i16) : i32;`
BASIC Syntax	`FN SCAN_Check(deviceNumber&, status&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

status returns an indication of whether the data acquisition operation has completed.

1: the data acquisition operation is complete.
0: the scanned data acquisition operation is not yet complete.

`SCAN_Check` checks the current background scanned data acquisition operation to determine whether it has completed. If the operation is complete, `SCAN_Check` sets **status** to 1. Otherwise, **status** is set to 0. If the acquisition is single-buffered, then the data is available in the buffer when **status** is 1. If the acquisition is double-buffered in continuous mode, **status** will always return 0. If the acquisition is double-buffered in noncontinuous mode, **status** will return 1 only when the entire acquisition is complete.

If `SCAN_Check` returns an **overFlowError** or an **overRunError**, the data acquisition operation may never complete because of lost A/D conversions due to samples being acquired too rapidly (sample interval was too small). An **overFlowError** indicates that the A/D FIFO overflowed because the data acquisition servicing operation was not able to keep up with the sample rate. An **overRunError** indicates that the data acquisition circuitry was not able to keep up with the sample rate. If one of these errors occurs, then `SCAN_Check` executes `DAQ_Clear` to terminate the operation and to clear all error flags.

If NI-DAQ for Macintosh is configured for double-buffered mode, an **overWriteErr** can occur. An **overWriteErr** indicates that the large circular acquisition buffer used for double-buffered acquisitions overwrote acquired data before it was retrieved by `DAQ2Get`, `DAQ2TGet`, `DAQ2Tap`, or `DAQ2TTap`. An overwrite error can be corrected by increasing the size of the large acquisition buffer, retrieving more data each time, retrieving data more often, decreasing the size of the smaller dividing blocks, or reducing the sampling rate. The large acquisition buffer and smaller dividing block sizes are configured in `DAQ2Config`.

An error occurs if analog triggering has been enabled (see `DAQ_Trigger`) and analog trigger conditions are not met before the specified timeout value expires.

SCAN_Demux

Function

Demultiplexes data acquired by a SCAN operation into separate arrays for each channel (C and Pascal only).

Synopsis

C Syntax	`locus i32 SCAN_Demux(u32 deviceNumber, i16 *inputBuffer, u32 count, i16 **outputBufferArray);`
Pascal Syntax	`function SCAN_Demux(deviceNumber : i32; inputBuffer : pi16; count : i32; outputBufferArray : ppi16) : i32;`
BASIC Syntax	`FN SCAN_Demux(deviceNumber&, inputBuffer&, count&, outputBufferArray&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

inputBuffer is a pointer to an array returned by a SCAN operation.

count is the number of samples contained in **inputBuffer**.

outputBufferArray is an array of pointers to integer buffers. The length of **outputBufferArray** is the total number of scanned channels as specified in the latest `Lab_ISCAN_Start` or `SCAN_Setup` call. The length of each integer array is expected to be **count** divided by the total number of scanned channels**.**

Note: ***The* channelCount *and* channel*[i] values specified in* `SCAN_Setup` *refer to the NB-MIO-16 or NB-MIO-16X onboard channel numbers (from 0 through 15). If one or more external boards (AMUX-64Ts) are used, then the total number of scanned channels equals (four-to-one multiplexer) * (the number of onboard channels scanned) * (the number of external multiplexer boards), or the total number of scanned channels equals (4) * (*channelCount*) * (*muxMode*). If SCXI was used to acquire the data in Multiplexed mode, the total number of channels scanned was determined by the*** `channelCount` ***array in the*** `SCXI_SCAN_Setup` ***call.***

`SCAN_Demux` demultiplexes the buffer returned by a SCAN operation by copying the A/D conversions taken from each channel in the scan sequence into a separate buffer for each channel. Figures 6-2 and 6-3 show how `SCAN_Demux` copies from **inputBuffer** to the separate buffers pointed to by **outputBufferArray**.

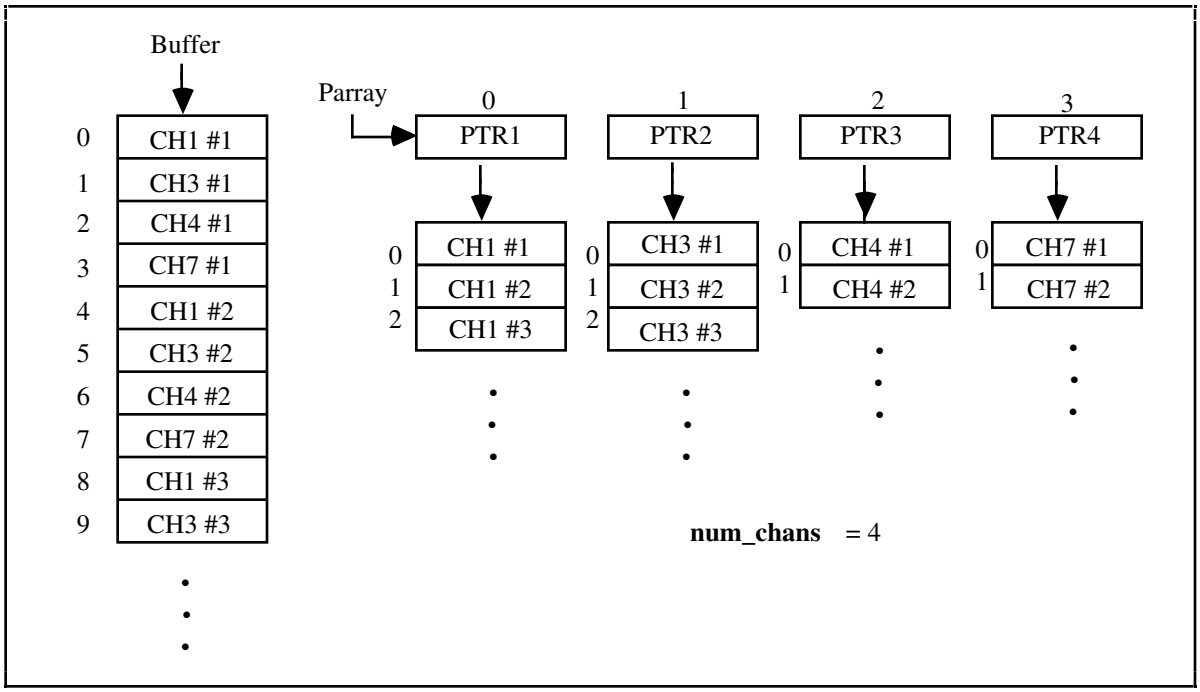

Figure 6-2. SCAN_Demux Buffer Translation for the NB-MIO-16 and NB-MIO-16X

Figure 6-2 shows the case for **channelCount** = 4 and for a **channel** = {1, 3, 4, 7} using an NB-MIO-16 or NB-MIO-16X. CH1 #1 represents the first sample for channel 1, while CH3 #2 represents the second sample for channel 3, and so on. **outputBufferArray** is supplied by the user and contains four elements labeled PTR1, PTR2, PTR3, and PTR4. PTR1 points to the integer buffer for the first channel in the scan sequence (in this case channel 1). PTR2 points to the second channel in the scan sequence, and so on.

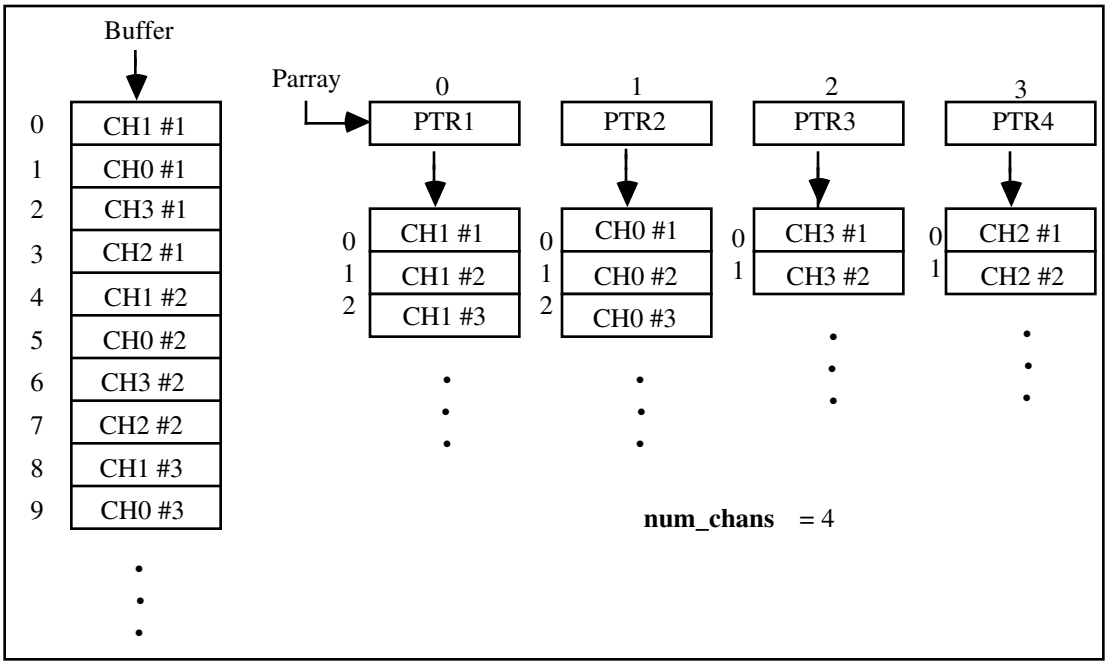

Figure 6-3. `SCAN_Demux` Buffer Translation for the Lab and 1200 Series

Figure 6-3 shows the case for **channelCount** = 4 and **scan_order** = {1, 0, 3, 2} returned by `Lab_ISCAN_Check` using a Lab and 1200 series board. CH3 #1 represents the first sample for channel 3, while CH0 #2 represents the second sample for channel 0, and so on. **outputBufferArray** is supplied by the user and contains four elements labeled PTR1, PTR2, PTR3, and PTR4. PTR1 points to the integer buffer for the first channel in the final scan order (in this case, channel 1. PTR2 points to the second channel in the final scan order, and so on.

Once `SCAN_Demux` is used to demultiplex the buffer returned by the SCAN operation, `DAQ_Scale` can be used to scale each channel buffer.

SCAN_IntStart

Function

Initiates a multiple-channel scanned data acquisition operation with internal scanning. This function can also be used to define a scan internal for the NB-MIO-16 under certain conditions that are defined in the section *Interval Scanning with the NB-MIO-16*.

Synopsis

C Syntax	`locus i32 SCAN_IntStart(u32 deviceNumber, i16 *buffer, u32 count, u32 sampleTimebase, u32 sampleInterval, u32 scanTimebase, u32 scanInterval);`
Pascal Syntax	`function SCAN_IntStart(deviceNumber : i32; buffer : pi16; count : i32; sampleTimebase : i32; sampleInterval : i32; scanTimebase : i32; scanInterval : i32) : i32;`
BASIC Syntax	`FN SCAN_IntStart(deviceNumber&, buffer&, count&, sampleTimebase&, sampleInterval&, scanTimebase&, scanInterval&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

buffer is a buffer of length **count**. When `SCAN_Check` returns **status** = 1, then **buffer** contains the acquired data. The elements of **buffer** are the results of each A/D conversion in the scanned data acquisition operation.

Note: ***If NI-DAQ for Macintosh has been configured for double-buffered mode (see*** `DAQ2Config`***), then this*** **buffer** ***parameter is not used and must be 0. The larger buffer allocated by*** `DAQ2Config` ***is used as the acquisition buffer for double-buffered acquisitions.*** `DAQ2Get`**,** `DAQ2TGet`**,** `DAQ2Tap`**, and** `DAQ2TTap` ***can then be used to acquire blocks of data from the double-buffered acquisition in progress. (See*** *Starting a Double-Buffered Acquisition with SCAN_IntStart.)*

count is the number of samples to be acquired (that is, the number of A/D conversions to be performed).
Range: 3 through 2^{31}-1. (With DMA the range is limited to 3 through 2^{23}.) (except the E Series)
2 through 2^{24} * (total number of channels scanned) or 2^{32}-1, whichever is less (E Series)

count must be an integer multiple of the total number of channels scanned. **count** refers to the *total* number of A/D conversions to be performed.

The **channelCount** and **channel**[*i*] values specified in `SCAN_Setup` refer to the MIO onboard channel numbers (from 0 through 15). If one or more external boards (AMUX-64Ts) are used, then the total number of scanned channels equals (four-to-one multiplexer) * (the number of onboard channels scanned) * (the number of external multiplexer boards), or the total number of scanned channels equals (4) * (**channelCount**) * (**muxMode**). If SCXI was used to acquire the data in Multiplexed mode, the total number of channels scanned was determined by the `channelCount` array in the `SCXI_SCAN_Setup` call.

Note: ***If NI-DAQ for Macintosh is configured for CONTINUOUS double-buffered mode (see*** `DAQ2Config`***), then this*** **count** ***parameter is ignored and should be 0. In continuous mode, the total number of samples to acquire is not specified and the data acquisition runs continuously until you stop the process by executing*** `DAQ_Clear` ***or*** `DAQ2Clear`***. (See*** *Starting a Double-Buffered Acquisition with SCAN_IntStart.)*

sampleTimebase is the resolution to use for the sample-interval counter. If sample-interval timing is to be externally controlled, the **sampleTimebase** parameter is ignored and can be any value.

scanTimebase is the resolution to use for the scan-interval counter.

sampleTimebase and **scanTimebase** have the following possible values:

Most devices:
0: External clock used as timebase (SOURCE5 input).
1: 1-MHz clock used as timebase (1-µs resolution).
2: 100-kHz clock used as timebase (10-µs resolution).
3: 10-kHz clock used as timebase (100-µs resolution).
4: 1-kHz clock used as timebase (1-ms resolution).
5: 100-Hz clock used as timebase (10-ms resolution).

E Series:
-3: 20-MHz clock used as timebase (50-ns resolution).
0: If you use this function with the timebase set at 0, you must call `Select_Signal` with **signal** set to `ND_IN_SCAN_CLOCK_TIMEBASE` (not `ND_IN_CHANNEL_CLOCK_TIMEBASE`), and **source** set to a value other than `ND_INTERNAL_20_MHZ` and `ND_INTERNAL_100_KHZ` before calling `DAQ_Start` with **timebase** set to 0; otherwise, `DAQ_Start` will select low-to-high transitions on the PFI 8 I/O connector pin as your external timebase.
2: 100 kHz clock used as timebase (10 µs resolution).

sampleInterval indicates the length of the sample interval (that is, the amount of time to elapse between each A/D conversion).
Range: 2 through 65,536.

The sample interval is a function of the timebase resolution. The actual sample interval in seconds is determined by the following formula:

sampleInterval ∗ (timebase resolution)

where the timebase resolution for each value of **timebase** is as indicated earlier in this function description; that is, if **sampleInterval** = 25 and **timebase** = 2, then the sample interval is 25 ∗ 10 µs = 250 µs. If the sample interval is to be externally controlled, the **sampleInterval** parameter is ignored and can be any value.

scanInterval indicates the length of the scan interval (that is, the amount of time to elapse between samples on any one channel).
Range: 2 through 65,536.
2 through 2^{24} (E Series)

The scan interval is a function of the timebase resolution. The actual scan interval in seconds is determined by the following formula:

scanInterval ∗ (scan timebase resolution)

where the scan timebase resolution for each value of **timebase** is as specified earlier in this function description. That is, if **scanInterval** = 25 and **scanTimebase** = 2, then the scan interval is 25 ∗ 10 µs = 250 µs. If the samples interval is to be externally controlled, the **scanInterval** parameter is ignored and can be any value.

Interval scanning has the advantage of simulating *simultaneous sampling* of a group of channels once every scan interval. A comparison of the scan interval and the sample interval is shown in Figure 6-4.

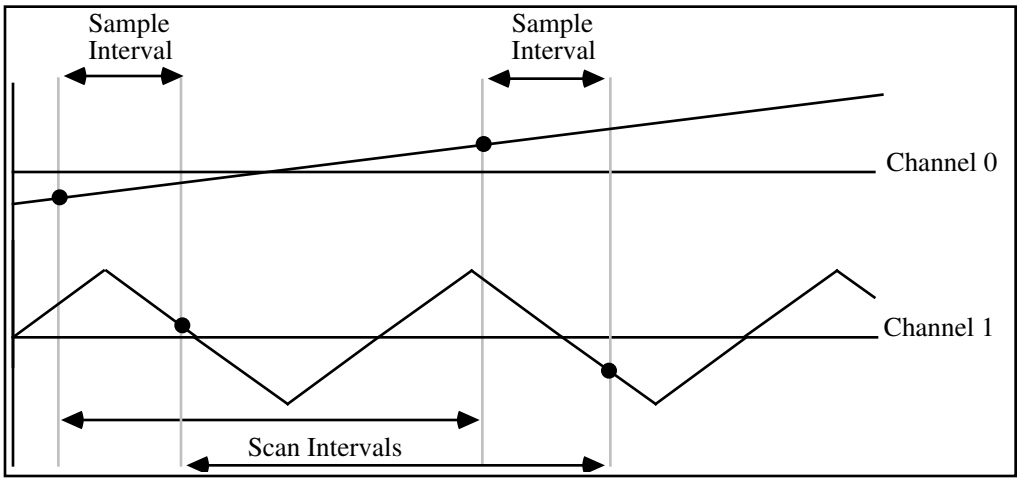

Figure 6-4. Scan and Sample Intervals

If a DMA is not detected in the system, then interrupts are used to acquire the data. Double-buffered acquisitions using interrupts require a sampling interval of at least 120 µs. An **overFlowError** is returned if no DMA is used and the sampling interval is less than 120 µs for double-buffered acquisitions. (See *Starting a Single-Buffered Acquisition with* `SCAN_IntStart` if faster sampling rates are needed for non-DMA acquisitions.)

Note: ***For scanned data acquisition, the sample interval still refers to the period of time between each A/D conversion. The sample interval per channel is equal to (sample interval) ∗ (*channelCount*); that is, each entry in the scan sequence is sampled once every (sample interval) ∗ (*channelCount*) seconds.***

`SCAN_IntStart` initiates a multiple-channel data acquisition either in single-buffered or double-buffered mode. For both modes, `SCAN_IntStart` initializes the Mux-Gain Memory Table to point to the start of the scan sequence as specified by `SCAN_Setup`. If external sample interval timing is not selected in the `DAQ_Config` call, the sample-interval counter is set to the specified **sampleInterval** and **timebase** parameters. If external sample-interval timing has been selected, the data acquisition circuitry relies on pulses received on the EXTCONV* input to initiate individual A/D conversions, and the **sampleInterval** and **timebase** parameters are ignored.

`SCAN_IntStart` initializes a background process to handle storage of A/D conversions as they occur. If a DMA board is detected in the system, a DMA process is initialized to handle data acquisition. If no DMA board is detected, an interrupt routine is initialized to handle data acquisition. In either case, the background process handles incoming data after `SCAN_IntStart` has returned. The acquired samples are stored into the buffer with the channel scan sequence data interleaved; that is, the first sample is the conversion from the first channel, the second sample is the conversion from the second channel, and so on.

If external gating has been selected for the data acquisition operation, a signal at the EXTGATE I/O connector input controls the sample-interval counter. When the EXTGATE signal is high, the sample-interval counter is enabled, causing A/D conversions to occur. When the EXTGATE signal is low, the sample-interval counter is suspended, and no A/D conversions occur. External gating is available only on the NB-MIO-16.

Starting a Single-Buffered Acquisition with SCAN_IntStart

In a single-buffered acquisition, the SCAN_IntStart call specifies the number of samples to acquire (**count**) and an integer array to store the acquired data (**buffer**). After SCAN_IntStart has returned, the background process stores up to **count** A/D conversions in the **buffer** and ignores any subsequent conversions. The acquired samples are available when the SCAN_Check call returns **status** = 1. A second call to SCAN_IntStart cannot be made without terminating this background process. If a call to SCAN_Check returns **status** = 1, the samples are available and the process is terminated. A call to DAQ_Clear also terminates a background data acquisition process.

Starting a Double-Buffered Acquisition with SCAN_IntStart

In a double-buffered acquisition, data can be returned from an acquisition in progress without interrupting the acquisition. NI-DAQ for Macintosh can be configured for double-buffered mode by executing DAQ2Config before SCAN_IntStart is called. DAQ2Config allocates a large internal circular buffer for the data storage and configures subsequent data acquisitions for double-buffered mode.

In double-buffered mode, SCAN_IntStart ignores the **buffer** parameter. Once SCAN_IntStart completes with **error** = 0, NI-DAQ for Macintosh acquires and stores the A/D conversions in the large buffer allocated by DAQ2Config. This buffer is treated as a circular buffer and is continually filled with data until **count** samples are acquired. If *continuous* double-buffered mode has been specified in DAQ2Config, then the total number of samples is not specified and the **count** value in SCAN_IntStart is ignored. The data acquisition runs continuously until the process is stopped by executing DAQ_Clear or DAQ2Clear.

Smaller blocks of data, ranging in size from 1 sample to the number of samples in the buffer, can be retrieved from the large internal buffer without interrupting the acquisition by repeatedly executing the DAQ2Get, DAQ2TGet, DAQ2Tap, and DAQ2TTap retrieval functions. An integer array to store the acquired data and the number of samples to retrieve are passed to the retrieval functions. The array is returned with a copy of a block of data from the internal circular buffer. (See the descriptions of DAQ2Get, DAQ2TGet, DAQ2Tap, and DAQ2TTap for more information on retrieving double-buffered data.)

Using SCAN_IntStart to Start a Trigger Acquisition Using Single-Buffered Mode

NI-DAQ for Macintosh can be configured for external hardware triggering using DAQ_Config. If you select external triggering for the data acquisition operation, a high-to-low edge at the STARTTRIG* input on the NB-MIO-16X initiates the data acquisition operation after the SCAN_IntStart call is complete. Otherwise, SCAN_IntStart issues a software trigger to initiate the data acquisition operation before returning.

Data acquisition also can be triggered on the slope and level of the analog input values. DAQ_Trigger stores trigger information and enables triggering on analog input values for subsequent single-buffered acquisitions. When executing DAQ_Trigger, you indicate a trigger channel, slope, and level. A trigger occurs when the analog input values of the trigger channel are within the specified slope and level.

If a single-buffered acquisition is started with triggering enabled, SCAN_IntStart waits for trigger conditions to be met before collecting the array of sample data.

Pretriggering can be implemented on the NB-MIO-16X for single-buffered acquisitions. With pretriggering, a data acquisition is stopped when a specified number of samples have been acquired after the occurrence of an external stop trigger. See the description of DAQ_PreTrig for more information on pretriggering with the NB-MIO-16X.

Using SCAN_IntStart to Start a Trigger Acquisition Using Double-Buffered Mode

NI-DAQ for Macintosh can be configured for external hardware triggering for double-buffered mode by using `DAQ_Config`.

If external triggering is disabled (in `DAQ_Config`) on the NB-MIO-16X, a software trigger is issued to initiate *each block* of the data acquisition operation. Otherwise, if you select external triggering for the data acquisition operation, a high-to-low edge at the STARTTRIG* I/O connector input on the NB-MIO-16X, initiates *each block* of the data acquisition operation after `SCAN_IntStart` begins execution. When a high-to-low edge is received, the number of samples in a block (specified in `DAQ2Config`) are then acquired. NI-DAQ for Macintosh then waits for another high-to-low edge before the next block of data is acquired.

Data acquisition also can be triggered on the slope and level of the analog input values. For double-buffered acquisition, triggering conditions can be specified for each retrieved block of data. A trigger channel, slope, and level can be specified in the `DAQ2Get`, `DAQ2TGet`, `DAQ2Tap`, and `DAQ2TTap` functions to implement condition triggering for double-buffered acquisitions. A trigger occurs when the analog input values of the trigger channel are within the specified slope and level.

Once the multiple-channel buffer of data is acquired, `DAQ_Scale` can be used to scale the 16-bit values in the buffer array to the actual voltages measured. `DAQ_Scale` must be passed a one-dimensional array of data acquired from one channel.

Interval Scanning with the NB-MIO-16

To perform multiple-channel scanned acquisitions using the SCXI-1140 module, interval scanning must be used. If an `SCXI_SCAN_Setup` call has been made to set up an SCXI scan that includes an SCXI-1140 module, then the `SCAN_IntStart` call is able to implement interval scanning on the NB-MIO-16. In this special case, the sample timebase and the scan timebase specified must be the same.

SCAN_Setup

Function

Initializes the circuitry on the NB-MIO-16, E Series or NB-MIO-16X for a scanned data acquisition operation. Initialization includes storing a table of the channel sequence and gain setting for each channel.

Synopsis

C Syntax	`locus i32 SCAN_Setup(u32 deviceNumber, u32 channelCount, u16 *channels, u16 *gains, u32 muxMode);`
Pascal Syntax	`function SCAN_Setup(deviceNumber : i32; channelCount : i32; channels : pi16; gains : pi16; muxMode : i32) : i32;`
BASIC Syntax	`FN SCAN_Setup(deviceNumber&, channelCount&, channels&, gains&, muxMode&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channelCount indicates the number of *onboard* channels to be scanned during the data acquisition scan operation.

Range:

2, 4, 8, 16 (NB-MIO-16)
1 through 16 (NB-MIO-16X)
1 through 512 (E Series)

channel is an integer array of length **channelCount** that contains the channel scan sequence to be followed. **channel** can contain any onboard analog input channel number from 0 through 15 in any order. For example, if **channelCount** is 4 and if the second element in the **channel** is 7, then the second channel to be scanned is analog input channel 7 and four analog input channels are scanned.

Note: **channelCount** ***and*** **channel*****[*****i*****] values refer to the onboard channel numbers (range: 0 through 15).***

If one or more external boards (AMUX-64T) are used, then the total number of scanned channels equals (four-to-one multiplexer) ∗ (the number of onboard channels scanned) ∗ (the number of external multiplexer boards), or the total number of scanned channels equals (4) ∗ (**channelCount*****) ∗ (*****muxMode*****). For example, if one external board (AMUX-64T) is used and eight onboard channels are scanned, then the total number of channels is equal to (4) ∗ (8) ∗ (1) = 32.***

If SCXI is being used, you must scan the appropriate analog input channels on the DAQ board that correspond to the desired SCXI channels. Please refer to Chapter 7, *SCXI Functions****, for more information on SCXI channel assignments.***

gain is an integer array of length **channelCount** that contains the gain setting to be used for each channel specified in **channel**. This gain setting applies only to the DAQ board; if SCXI is used, any gain desired at the SCXI module must be established either by setting jumpers on the module or by calling `SCXI_Set_Gain`.

Refer to Appendix E, *Analog Input Channel and Gain Settings and Voltage Calculation*, for valid gain settings.

For example, if **channelCount** is 8 and the sixth element in the **gain** is 10 (assuming an NB-MIO-16L or NB-MIO-16XL board), then when the sixth channel is scanned, the gain circuitry is set to a gain of 10 and eight analog input channels are scanned. Notice also that **gain**[*i*] corresponds to **channel**[*i*].

Another example (using C) is, if **gain**[2] = 100 and **channel**[2] = 3, then the third channel to be scanned is analog input channel 3 and its gain is set to 100.

muxMode indicates the number of external multiplexer boards connected to the MIO board. An external multiplexer board (AMUX-64T) can be used to expand the number of analog input signals that can be measured with the MIO board. (See the *AMUX-64T User Manual* for more information on the external multiplexer board.) This parameter is not used when SCXI is used.

Valid values:

0, 1, 2, and 4.

The default value is 0.

Note: ***The default value of*** **muxMode** ***was changed in the Version 2.0 of NI-DAQ for Macintosh. If no AMUX-64T boards are being used, then*** **muxMode** ***should be 0.***

`SCAN_Setup` stores the **channelCount**, **channel**, and **gain** in the Mux-Gain Memory Table on the MIO board. This memory table is used during scanning operations to automatically sequence through an arbitrary set of analog input channels and to automatically change gains during scanning.

`SCAN_Setup` needs to be called initially to set up a scan sequence for scanned operations and needs to be subsequently called only when you want a different scan sequence. If `DAQ_Start` or `AI_Read` is called, the Mux-Gain Memory Table on the MIO board is modified; therefore, `SCAN_Setup` should be used again after these calls to reinitialize the scan sequence.

SCAN_Start

Function

Initiates a multiple-channel scanned data acquisition operation on an MIO board.

Synopsis

C Syntax	`locus i32 SCAN_Start(u32 deviceNumber, i16 *buffer, u32 count, u32 timebase, u32 sampleInterval);`
Pascal Syntax	`function SCAN_Start(deviceNumber : i32; buffer : pi16; count : i32; timebase : i32; sampleInterval : i32) : i32;`
BASIC Syntax	`FN SCAN_Start(deviceNumber&, buffer&, count&, timebase&, sampleInterval&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

buffer is a buffer of length **count**. When `SCAN_Check` returns **status** = 1, then **buffer** contains the acquired data. The elements of **buffer** are the results of each A/D conversion in the scanned data acquisition operation.

Note: ***If NI-DAQ for Macintosh has been configured for double-buffered mode (see*** `DAQ2Config`***), then this*** **buffer** ***parameter is not used and must be 0. The larger buffer allocated by*** `DAQ2Config` ***is used as the acquisition buffer for double-buffered acquisitions.*** `DAQ2Get`**,** `DAQ2TGet`**,** `DAQ2Tap`**,** ***and*** `DAQ2TTap` ***can then be used to acquire blocks of data from the double-buffered acquisition in progress. (See*** *Starting a Double-Buffered Acquisition with SCAN_Start.)*

count is the number of samples to be acquired (that is, the number of A/D conversions to be performed).
Range: 3 through 2^{31}-1. (With DMA, the range is limited to 3 through 2^{23}.) (Except E Series)
2 through 2^{24} * (total number of channels scanned) or 2^{32}-1, whichever is less (E Series)

count must be an integer multiple of the total number of channels scanned (**channelCount**). **count** refers to the *total* number of A/D conversions to be performed; therefore, the number of samples acquired from each channel is equal to **count** divided by **channelCount**.

Note: ***If NI-DAQ for Macintosh is configured for*** *continuous* ***double-buffered mode (see*** *DAQ2Config)****, then the*** **count** ***parameter is ignored and should be 0. In continuous mode, the total number of samples to acquire is not specified and the data acquisition runs continuously until you stop the process by executing*** `DAQ_Clear` ***or*** `DAQ2Clear`**.** ***(See*** *Starting a Double-Buffered Acquisition with SCAN_Start.)*

timebase is the resolution to use for the sample-interval counter. **timebase** has the following possible values:

Most devices:
0: External clock used as timebase (SOURCE5 input).
1: 1-MHz clock used as timebase (1-µs resolution).
2: 100-kHz clock used as timebase (10-µs resolution).
3: 10-kHz clock used as timebase (100-µs resolution).
4: 1-kHz clock used as timebase (1-ms resolution).
5: 100-Hz clock used as timebase (10-ms resolution).

E Series:
-3: 20-MHz clock used as timebase (50-ns resolution).
0: If you use this function with the timebase set at 0, you must call `Select_Signal` with **signal** set to `ND_IN_SCAN_CLOCK_TIMEBASE` (not `ND_IN_CHANNEL_CLOCK_TIMEBASE`), and **source** set to a value other than `ND_INTERNAL_20_MHZ` and `ND_INTERNAL_100_KHZ` before calling `DAQ_Start` with **timebase** set to 0; otherwise, `DAQ_Start` will select low-to-high transitions on the PFI 8 I/O connector pin as your external timebase.
2: 100 kHz clock used as timebase (10 µs resolution).

If sample-interval timing is to be externally controlled, the **timebase** parameter is ignored and can be any value. When using external timing sources with the NB-MIO-16X, be sure to call `MIO_16X_Config` before starting the acquisition.

sampleInterval is the length of the sample interval (that is, the amount of time to elapse between each A/D conversion).
Range: 2 through 65,536.
2 through 2^{24} (E Series)

The sample interval is a function of the timebase resolution. The actual sample interval in seconds is determined by the following formula:

sampleInterval ∗ (timebase resolution)

where the timebase resolution for each value of **timebase** is as indicated above. That is, if **sampleInterval** = 25 and **timebase** = 2, then the sample interval is 25 ∗ 10 µs = 250 µs. If the sample interval is to be externally controlled, the **sampleInterval** parameter is ignored and can be any value.

If an NB-DMA-8-G or NB-DMA2800 is not detected in the system, then interrupts are used to acquire the data. Double-buffered acquisitions using interrupts require a sampling interval of at least 120 µs. An **overFlowError** is returned if no DMA is used and the sampling interval is less than 120 µs for double-buffered acquisitions. (See *Starting a Single-Buffered Acquisition with SCAN_Start* if faster sampling rates are needed for non-DMA acquisitions.)

Note: ***For scanned data acquisition, the sample interval still refers to the period of time between each A/D conversion. The sample interval per channel is equal to (sample interval) ∗ (*****channelCount****); that is, each entry in the scan sequence is sampled once every (sample interval) ∗ (*****channelCount****) seconds.***

`SCAN_Start` initiates a multiple-channel data acquisition operation either in single-buffered or double-buffered mode. For both modes, `SCAN_Start` initializes the Mux-Gain Memory Table to point to the start of the scan sequence as specified by `SCAN_Setup`. If external sample-interval timing is not specified in the `DAQ_Config` call, the sample-interval counter is set to the specified **sampleInterval** and **timebase** parameters. If external sample-interval timing has been selected, the data acquisition circuitry relies on pulses received on the EXTCONV* input to initiate individual A/D conversions, and the **sampleInterval** and **timebase** parameters are ignored.

`SCAN_Start` initializes a background process to handle storage of A/D conversions as they occur. If a DMA board is detected in the system, a DMA process is initialized to handle data acquisition. If no DMA board is detected, an interrupt routine is initialized to handle data acquisition. In either case, the background process handles incoming data after `SCAN_Start` has returned. The acquired samples are stored into the buffer with the channel scan sequence data interleaved–that is, the first sample is the conversion from the first channel, the second sample is the conversion from the second channel, and so on.

If external gating of the data acquisition operation has been selected, a signal at the EXTGATE I/O connector input controls the sample-interval counter. When the EXTGATE signal is high, the sample-interval counter is enabled, causing A/D conversions to occur. When the EXTGATE signal is low, the sample-interval counter is suspended, and no A/D conversions occur. External gating is available only on the NB-MIO-16.

Starting a Single-Buffered Acquisition with SCAN_Start

In a single-buffered acquisition, the SCAN_Start call specifies the number of samples to acquire (**count**) and an integer array to store the acquired data (**buffer**). After `SCAN_Start` has returned, the background process stores up to **count** A/D conversions in the **buffer** and ignores any subsequent conversions. The acquired samples are available when the `SCAN_Check` call returns **status** = 1. A second call to `SCAN_Start` cannot be made without terminating this background process. If a call to `SCAN_Check` returns **status** = 1, the samples are available and the process is terminated. A call to `DAQ_Clear` also terminates a background data acquisition process.

Starting a Double-Buffered Acquisition with SCAN_Start

In a double-buffered acquisition, data can be returned from an acquisition in progress without interrupting the acquisition. NI-DAQ for Macintosh can be configured for double-buffered mode by executing `DAQ2Config` before `SCAN_Start` is called. `DAQ2Config` allocates a large internal circular buffer for the data storage and configures subsequent data acquisitions for double-buffered mode.

In double-buffered mode, the **buffer** parameter in `SCAN_Start` is ignored but must be set to 0. Once `SCAN_Start` completes with **error** = 0, NI-DAQ for Macintosh acquires and stores the A/D conversions in the large buffer allocated by `DAQ2Config`. This buffer is treated as a circular buffer and is continually filled with data until **count** samples are acquired. If *continuous* double-buffered mode has been specified in `DAQ2Config`, then the total number of samples is not specified and the **count** value in `SCAN_Start` is ignored. The data acquisition runs continuously until the process is stopped by executing `DAQ_Clear` or `DAQ2Clear`.

Smaller blocks of data can be retrieved from the large internal buffer without interrupting the acquisition by repeatedly executing the `DAQ2Get`, `DAQ2TGet`, `DAQ2Tap`, and `DAQ2TTap` retrieval functions. An integer array to store the acquired data and the number of samples to retrieve are passed to the retrieval functions. The array is returned with a copy of a block of data from the internal circular buffer. (See the descriptions of `DAQ2Get`, `DAQ2TGet`, `DAQ2Tap`, and `DAQ2TTap` for more information on retrieving double-buffered data.)

Using SCAN_Start to Start a Trigger Acquisition Using Single-Buffered Mode

NI-DAQ for Macintosh can be configured for external hardware triggering using `DAQ_Config`. If you select external triggering for the data acquisition operation, a high-to-low edge at the EXTTRIG* I/O connector input on the NB-MIO-16, or at the STARTTRIG* input on the NB-MIO-16X initiates the data acquisition operation after the `SCAN_Start` call is complete. Otherwise, `SCAN_Start` issues a software trigger to initiate the data acquisition operation before returning.

Data acquisition also can be triggered on the slope and level of the analog input values. `DAQ_Trigger` stores trigger information and enables triggering on analog input values for subsequent single-buffered acquisitions. When executing `DAQ_Trigger`, you indicate a trigger channel, slope, and level. A trigger occurs when the analog input values of the trigger channel are within the specified slope and level.

If a single-buffered acquisition is started with triggering enabled, `SCAN_Start` waits for trigger conditions to be met before collecting the array of sample data.

Pretriggering can be implemented on the NB-MIO-16X for single-buffered acquisitions. With pretriggering, data acquisition is stopped when a specified number of samples have been acquired after the occurrence of an external stop trigger. See the description of `DAQ_PreTrig` for more information on pretriggering with the NB-MIO-16X.

Using SCAN_Start to Start a Trigger Acquisition Using Double-Buffered Mode

NI-DAQ for Macintosh can be configured for external hardware triggering for double-buffered mode by using `DAQ_Config`. External triggering with double-buffering operates differently on the NB-MIO-16 from the way it works on the NB-MIO-16X.

If you select external triggering for the data acquisition operation (`DAQ_Config`) on the NB-MIO-16, a high-to-low edge at the EXTTRIG* I/O connector input on the NB-MIO-16 initiates the data acquisition operation after `SCAN_Start` begins execution. Otherwise, `SCAN_Start` issues a software trigger to initiate the data acquisition operation before returning.

If external triggering is disabled (in `DAQ_Config`) on the NB-MIO-16X, a software trigger is issued to initiate *each block* of the data acquisition operation. Otherwise, if you select external triggering for the data acquisition operation, a high-to-low edge at the STARTTRIG* I/O connector input on the NB-MIO-16X initiates *each block* of the data acquisition operation after `SCAN_Start` begins execution. When a high-to-low edge is received, the number of samples in a block (specified in `DAQ2Config`) are then acquired. NI-DAQ for Macintosh waits for another high-to-low edge before the next block of data is acquired.

Data acquisition also can be triggered on the slope and level of the analog input values. For double-buffered acquisition, triggering conditions can be specified for each retrieved block of data. A trigger channel, slope, and level can be specified in the `DAQ2Get`, `DAQ2TGet`, `DAQ2Tap`, and `DAQ2TTap` functions to implement condition triggering for double-buffered acquisitions. A trigger occurs when the analog input values of the trigger channel are within the specified slope and level.

Once the multiple-channel buffer of data is acquired, `DAQ_Scale` can be used to scale the 12-bit (NB-MIO-16) or 16-bit (NB-MIO-16X) values in the buffer array to the actual voltages measured. `DAQ_Scale` must be passed a one-dimensional array of a channel's data.

Double-Buffered Data Acquisition Function Summary

The Double-Buffered Data Acquisition functions (`DAQ2`) can acquire samples from one or more channels into a circular buffer. With double-buffering, data can be retrieved from an acquisition in progress without interrupting the acquisition. Data can be collected continuously using a fixed amount of memory.

The following functions can be used for double-buffered data acquisition on the MIO boards, DAQCard-500, DAQCard-700, E Series, and Lab and 1200 series:

`DAQ2Clear`	Cancels any current double-buffered data acquisition operation, reinitializes the data acquisition circuitry, deallocates the acquisition buffer allocated by `DAQ2Config`, and disables double-buffering for subsequent data acquisition operations.
`DAQ2Config`	Configures subsequent data acquisition operations for double-buffered mode, allocates a large buffer for the background acquisition, and stores double-buffered mode configuration information.
`DAQ2Get`	Returns a block of data from a background double-buffered acquisition (both single-channel and multiple-channel scan). `DAQ2Get` can be executed repeatedly to return sequential blocks of data. `DAQ2Get` waits until the block of data is available before returning, unless the timeout expires. This function can also be used to define a trigger that determines when to begin acquiring a block of data (MIO boards).
`DAQ2MemConfig`	Configures NI-DAQ for Macintosh to use a memory expansion board for subsequent data acquisition in double-buffered mode. Memory for the large buffer allocated by `DAQ2Config` for the background acquisition is allocated from the memory space on the expansion board.

`DAQ2Tap`	Returns the most recently acquired block of data from the background double-buffered data acquisition (both single-channel and multiple-channel scan). This function can also be used to define a trigger that determines when to start acquiring the block of data (MIO boards).
`DAQ2TGet`	Returns a block of data from a background double-buffered acquisition (both single-channel and multiple-channel scan). `DAQ2TGet` can be executed repeatedly to return sequential blocks of data. `DAQ2TGet` waits until the block of data is available before returning, unless the timeout expires. This function can also be used to define a trigger that determines when to begin acquiring the block of data and where this trigger occurs within the block of data.
`DAQ2TTap`	Returns the most recently acquired block of data from the background double-buffered data acquisition (both single-channel and multiple-channel scan). This function can also be used to define a trigger that determines when to start acquiring the block of data and where this trigger occurs within the block of data.

Double-Buffered Data Acquisition Application Hints

The double-buffered (`DAQ2`) Data Acquisition functions can return data from an acquisition in progress without interrupting the acquisition on an NB-MIO-16, NB-MIO-16X, E Series, DAQCard-500/700, and Lab and 1200 series. These functions use a double or circular buffering scheme that retrieves and processes chunks of data as they become available. With a circular buffer, this scheme can be used to collect an unlimited amount of data without requiring an unlimited amount of memory. Double-buffered data acquisition is useful for applications such as streaming to disk and real-time display of data. NI-DAQ for Macintosh can use double-buffered data acquisition for both single-channel and multiple-channel scan data acquisition. Double-buffered acquisition is available only with the NB-MIO-16, NB-MIO-16X, E-Series, DAQCard-500/700, and Lab and 1200 series.

Examples of double-buffered data acquisition applications are included on your NI-DAQ for Macintosh diskettes. (See Chapter 11, *NI-DAQ for Macintosh Examples*.) The following paragraphs explain the operation of the double-buffered Data Acquisition functions.

Initializing Double-Buffered Data Acquisition

You can configure NI-DAQ for Macintosh for double-buffered acquisition operations by executing `DAQ2Config`. NI-DAQ for Macintosh remains configured for double-buffered mode until `DAQ2Clear` is called. If NI-DAQ for Macintosh is in double-buffered mode, all subsequent data acquisitions (single-channel and multiple-channel scan) are background acquisitions.

When executing `DAQ2Config` to configure NI-DAQ for Macintosh for double-buffering, you indicate the size of a large circular buffer allocated by NI-DAQ for Macintosh. NI-DAQ for Macintosh uses this buffer to store data in subsequent background data acquisitions. You also indicate the size of smaller *blocks* that divide up the larger buffer. The background acquisition can be thought of as actually being performed in continuous *chunks* that are the size of the specified smaller blocks. Figure 6-5 shows the blocks in the large circular buffer. The large circular buffer is continually filled with one block of data at a time. When the large buffer has been filled, data is stored at the beginning of the buffer again (writing over the previously stored data), and continues filling the buffer until the specified number of samples have been acquired or until the acquisition operation is cleared.

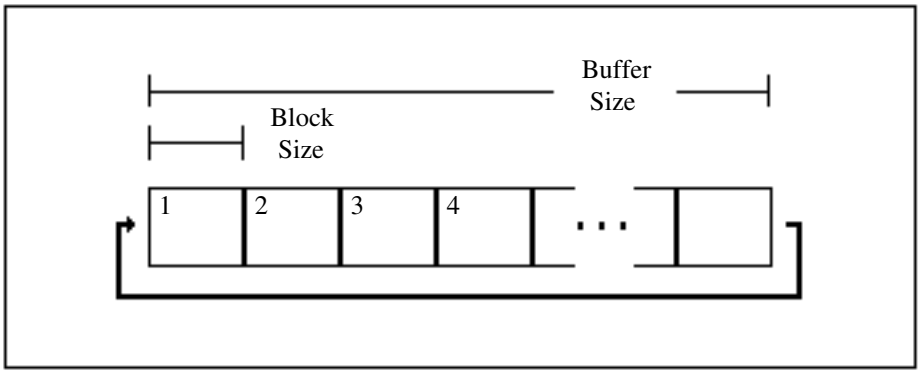

Figure 6-5. Double-Buffered Acquisition Buffer and Blocks

After configuring NI-DAQ for Macintosh for double-buffered mode, you can start a single-channel double-buffered data acquisition by executing DAQ_Start. After configuring NI-DAQ for Macintosh for double-buffered mode, you can start a multiple-channel double-buffered data acquisition by executing SCAN_Start or SCAN_IntStart. This action begins the A/D conversion operation and the storing of acquired data in the large acquisition buffer. DAQ2Get, DAQ2TGet, DAQ2Tap, or DAQ2TTap can then be executed to retrieve blocks of data from the background acquisition. You indicate the number of samples to retrieve from the large acquisition buffer, and the conversion data is returned by these functions.

Retrieving Acquired Data

Two mechanisms can be used for retrieving data during double-buffered data acquisition. The Get data mechanism retrieves blocks of data *in the order that they are acquired*. The Tap data mechanism retrieves the *most recently acquired block of data*. The size of a block when retrieving data does not have to be equal to the block/size specified when initializing a double-buffered data acquisition. DAQ2Get, DAQ2TGet, DAQ2Tap, or DAQ2TTap can be used to retrieve blocks of acquired data from a single-channel acquisition in progress or a multiple-channel scan acquisition in progress. The two functions DAQ2TGet and DAQ2TTap are identical to DAQ2Get and DAQ2Tap except that they also can define a trigger position, allowing data to be retrieved before and after the trigger occurrence. (The T in the function name stands for trigger.) Any combination of DAQ retrieval functions can be used together. These functions are described in more detail later.

The DAQ2Get and DAQ2TGet functions wait until the requested number of samples have been retrieved before returning. The data is copied from the large acquisition buffer into the sample array. A marker into the large acquisition buffer is updated to keep track of what has been retrieved by DAQ2Get and DAQ2TGet calls. Each time either DAQ2Get or DAQ2TGet is called, the marker is updated by the number of samples retrieved. Therefore, DAQ2Get or DAQ2TGet can be executed repeatedly to return sequential blocks of data. DAQ2Get and DAQ2TGet are useful for applications such as concurrent processing of acquired data or logging acquired data to disk.

The DAQ2Tap and DAQ2TTap functions return the most recently acquired block of data. The marker into the large acquisition buffer is not updated. Unlike DAQ2Get and DAQ2TGet, DAQ2Tap, and DAQ2TTap do not wait until the samples are available before returning. If the requested number of samples are not yet available, both DAQ2Tap and DAQ2TTap return with an error code. DAQ2Tap and DAQ2TTap are useful for applications such as displaying data in real time.

Figures 6-6 through 6-8 illustrate the difference in the DAQ2Get/DAQ2TGet and DAQ2Tap/DAQ2TTap retrieval functions. Figure 6-6 shows the buffer of an acquisition in progress. At this point in the acquisition, the first three blocks of the buffer have been filled with data. Figure 6-6 shows the results of executing DAQ2Get and DAQ2Tap at this point in the acquisition operation. (In this example, the number of samples requested in DAQ2Get and DAQ2Tap is equal to the block size configured in DAQ2Config.) DAQ2Get returns the first

block of acquired data and updates the marker to the second block. `DAQ2Tap` returns the most recent block of data, which is the third block.

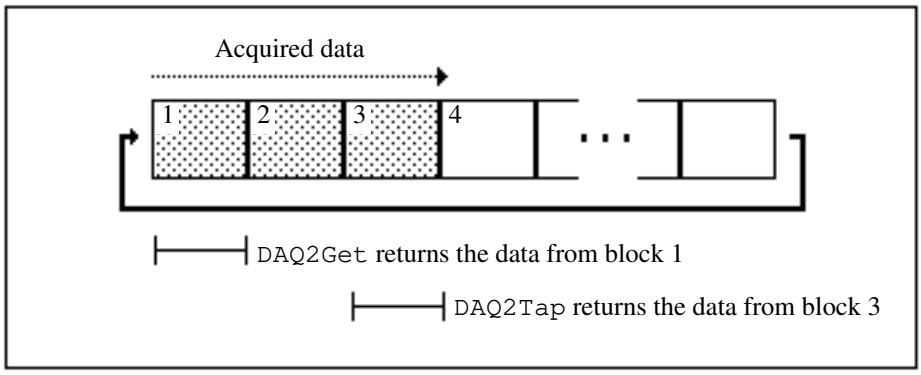

Figure 6-6. First Execution of DAQ2Get and DAQ2Tap

Figure 6-7 shows the results of executing `DAQ2Get` and `DAQ2Tap` later in the acquisition. `DAQ2Get` returns the second block of acquired data and updates the marker to the third block. `DAQ2Tap` returns the most recent block of data, which is the fourth block.

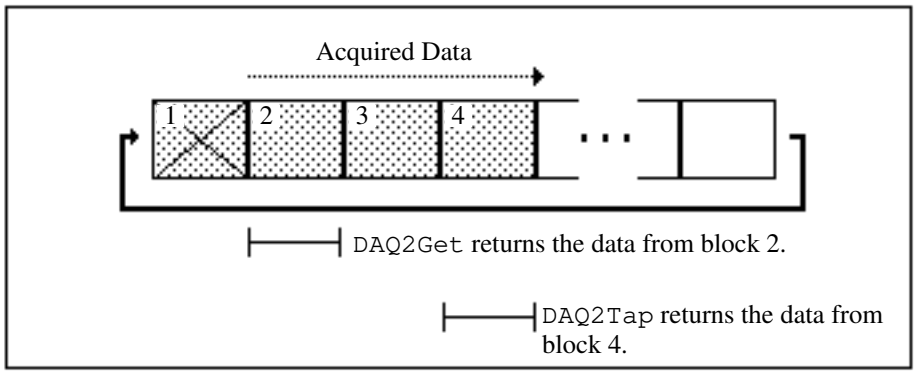

Figure 6-7. Second Execution of DAQ2Get and DAQ2Tap

When the large buffer has been filled, data is stored at the beginning of the buffer again, overwriting the previous data. If all the data is to be retrieved sequentially (for example, logging all data to disk), then `DAQ2Get` must retrieve the blocks of data from the large circular buffer before the data is overwritten. An **overWriteErr** is returned by `DAQ2Get` if unretrieved data has been overwritten in the large buffer. An **overWriteErr** is returned by `DAQ2Tap` if the most recent block of data is overwritten as it is being retrieved. Figure 6-8 shows the results of executing `DAQ2Get` and `DAQ2Tap` when data has been overwritten. In Figure 6-7, there was no **overWriteErr** because `DAQ2Get` returned a copy of block 1 before data was overwritten.

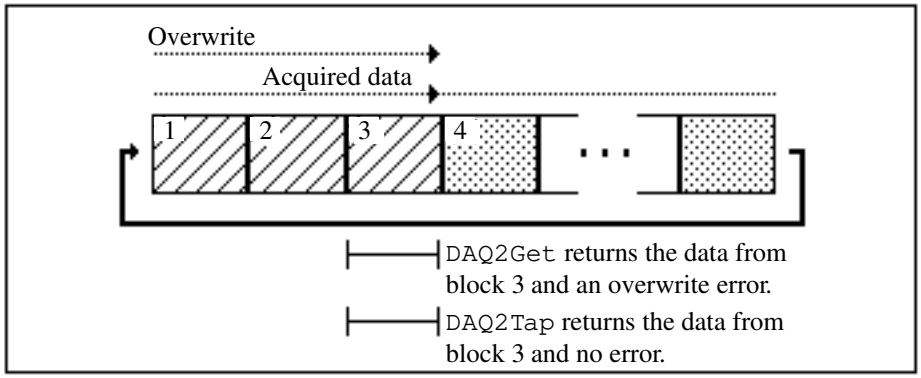

Figure 6-8. Executing DAQ2Get and DAQ2Tap when Overwrite Occurred

If an acquisition has completed (the specified number of conversions have been performed), then DAQ2Tap and DAQ2TTap return the last block acquired. DAQ2Get and DAQ2TGet continue returning sequential blocks of data until all of the data has been retrieved or until the buffer has been deallocated by executing DAQ2Clear.

DAQ2Config also gives you the option of acquiring data continuously. In continuous mode, you do not indicate the number of samples to be acquired in DAQ_Start, SCAN_Start, or SCAN_IntStart. The data acquisition continues until you stop the process by executing DAQ_Clear or DAQ2Clear. DAQ_Clear stops the current double-buffered acquisition but leaves NI-DAQ for Macintosh configured for double-buffered mode. DAQ2Clear stops the current acquisition and disables double-buffered mode. Once a double-buffered acquisition is in progress (single-channel or multiple-channel), you can use DAQ_Check or SCAN_Check to return the status.

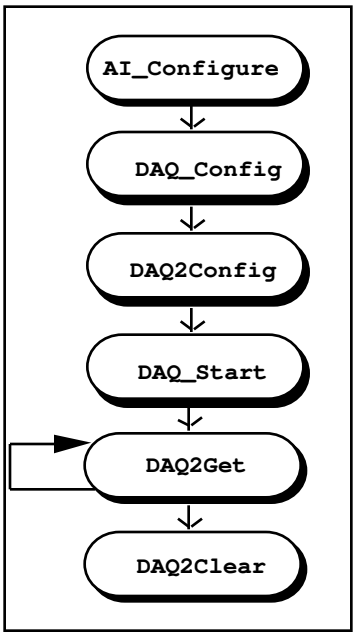

Figure 6-9. Single-Channel, Double-Buffered Acquisition

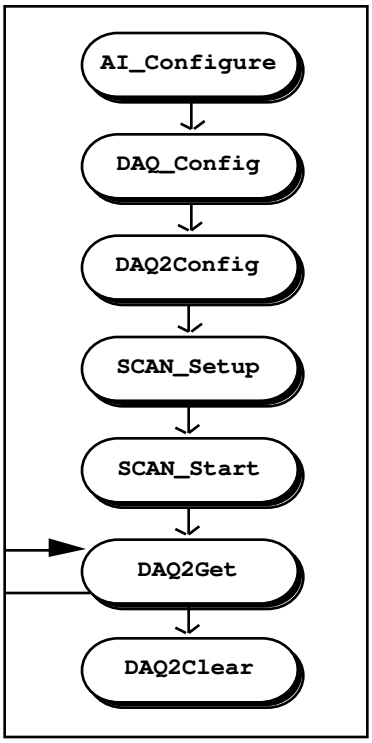

Figure 6-10. Multiple-Channel, Double-Buffered Acquisition (MIO Boards)

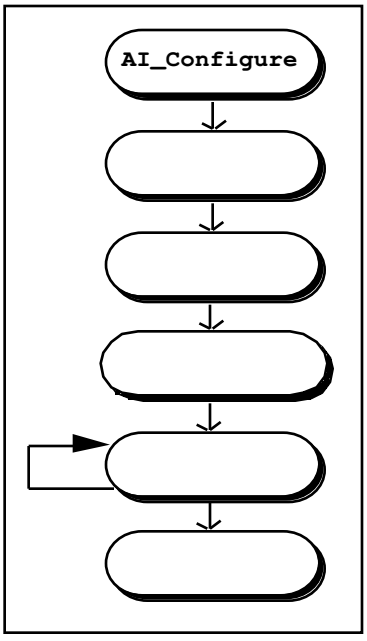

Figure 6-11. Multiple-Channel, Double-Buffered Acquisition (Lab and 1200 Series)

Using Double-Buffered Data Acquisition with Analog Triggering

Triggering on analog input values can be enabled for any of the double-buffered Data Acquisition functions. A trigger is enabled by specifying a level and slope for an active analog input channel. The functions `DAQ2Get` and `DAQ2Tap` return a block of data whose first data point matches the trigger conditions. The other double-buffered Data Acquisition functions, `DAQ2TGet` and `DAQ2TTap`, can select a trigger position to place the trigger anywhere within the block of data. With this scheme, the data acquisition operation can retrieve data before and after the trigger occurs.

If SCXI is used, analog triggering is possible during single-channel acquisitions and during multiple-channel scanning acquisitions *if the SCXI modules are operated in Parallel mode*. Analog triggering is not possible during multiple-channel scanning if the SCXI modules are operated in Multiplexed mode. When analog triggering is used with SCXI, the trigger channel specified should be the DAQ board channel number that corresponds to the desired SCXI channel. Refer to Chapter 7, *SCXI Functions*, for more information on SCXI operating modes and channel assignments.

DAQ2Clear

Function

Cancels any current double-buffered data acquisition operation, re-initializes the data acquisition circuitry, deallocates the acquisition buffer allocated by `DAQ2Config`, and disables double-buffering for subsequent data acquisition operations.

Synopsis

C Syntax	`locus i32 DAQ2Clear(u32 deviceNumber);`
Pascal Syntax	`function DAQ2Clear(deviceNumber : i32) : i32;`
BASIC Syntax	`FN DAQ2Clear(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`DAQ2Clear` cancels any current double-buffered data acquisition operation, reinitializes the data acquisition circuitry, deallocates the acquisition buffer allocated by `DAQ2Config`, and disables double-buffering for subsequent data acquisition operations.

When NI-DAQ for Macintosh is configured for double-buffered data acquisition (see `DAQ2Config`), then either `DAQ_Clear` or `DAQ2Clear` can be used to stop the current double-buffered acquisition. `DAQ_Clear` stops the current double-buffered acquisition but leaves NI-DAQ for Macintosh configured for double-buffering. `DAQ2Clear` stops the current acquisition and disables double-buffering.

DAQ2Config

Function

Configures subsequent data acquisition operations for double-buffered mode, allocates a large buffer for the background acquisition, and stores double-buffered mode configuration information.

Synopsis

C Syntax	`locus i32 DAQ2Config(u32 deviceNumber, u32 mode, u32 bufferSize, u32 blockCount);`
Pascal Syntax	`function DAQ2Config(deviceNumber : i32; mode : i32; bufferSize : i32; blockCount : i32) : i32;`
BASIC Syntax	`FN DAQ2Config(deviceNumber&, mode&, bufferSize&, blockCount&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

mode indicates whether subsequent data acquisitions should be in continuous mode.

0: Noncontinuous (disable continuous mode). Subsequent double-buffered data acquisitions acquire the number of samples specified in `DAQ_Start`, `SCAN_Start`, or `SCAN_IntStart`.

1: Continuous (enable continuous mode). Subsequent double-buffered data acquisitions acquire data continuously.

In continuous mode, the sample count value is ignored in subsequent `DAQ_Start`, `SCAN_Start`, and `SCAN_IntStart` calls. After you execute `DAQ_Start`, `SCAN_Start`, or `SCAN_IntStart`, data acquisition continues until you stop the process by executing `DAQ_Clear` or `DAQ2Clear`. Continuous mode can be disabled by executing `DAQ2Config` with **mode** set to 0.

bufferSize indicates the size of the large circular data acquisition buffer used to acquire data in a background double-buffered data acquisition. **bufferSize** is the number of samples in the large buffer. Once `DAQ2Config` is executed, the buffer is allocated and can be used repeatedly in subsequent double-buffered acquisitions. While double-buffering is enabled, the **buffer** value is ignored in subsequent `DAQ_Start`, `SCAN_Start`, and `SCAN_IntStart` calls. Executing `DAQ2Clear` disposes of the large circular buffer and disables double-buffering.

blockCount indicates the size of the smaller blocks that divide up the larger buffer. The **blockCount** is the number of samples in a block of data. The background acquisition can be thought of as actually being performed in smaller continuous *chunks*, each containing the number of samples specified by **blockCount**. To implement a true double-buffered scheme, **blockCount** should be half of **bufferSize**.

Increasing **bufferSize** beyond twice the **blockCount** creates some leeway for processing to catch up with the data acquisition operation. If a multiple-channel scan is to be performed, **blockCount** should be an integer multiple of the number of channels to be scanned. **blockCount** cannot be less than 200 samples in `DAQ2Config`. The `DAQ2Get`, `DAQ2TGet`, `DAQ2Tap`, and `DAQ2TTap` functions can be used to retrieve from 1 to **bufferSize** number of samples at a time from the larger buffer while a background acquisition is in progress.

If you use an NB-MIO-16X with external triggering to trigger each block, **blockCount** must not be greater than 65,535.

Note: ***DMA is used for data acquisition operations on an NB-MIO-16 or NB-MIO-16X when an NB-DMA-8-G or NB-DMA2800 board is present. If DMA is used, NI-DAQ for Macintosh truncates* bufferSize *so that the actual size of the large acquisition buffer is a multiple of* blockCount*. The* blockCount *parameter used in* `DAQ2Get`*,* `DAQ2TGet`*,* `DAQ2Tap`*, or* `DAQ2TTap` *should be less than or equal to the* blockCount *parameter specified in* `DAQ2Config` *for optimum performance when using DMA and double-buffering.***

DAQ2Get

Function

Returns a block of data from a background double-buffered acquisition (both single-channel and multiple-channel scan). `DAQ2Get` can be executed repeatedly to return sequential blocks of data. `DAQ2Get` waits until the block of data is available before returning, unless the timeout expires. This function can also be used to define a trigger that determines when to begin acquiring a block of data (MIO boards).

DAQ2TGet

Function

Same functionality as `DAQ2Get`. In addition, `DAQ2TGet` can indicate where the trigger occurs within the block of data.

DAQ2Get Synopsis

C Syntax	`locus i32 DAQ2Get(u32 deviceNumber, u32 triggerChannel, u32 triggerSlope, i32 triggerLevel, i16 *buffer, u32 blockCount, u32 timeout);`
Pascal Syntax	`function DAQ2Get(deviceNumber : i32; triggerChannel : i32; triggerSlope : i32; triggerLevel : i32; buffer : pi16; blockCount : i32; timeout : i32) : i32;`
BASIC Syntax	`FN DAQ2Get(deviceNumber&, triggerChannel&, triggerSlope&, triggerLevel&, buffer&, blockCount&, timeout&)`

DAQ2TGet Synopsis

C Syntax	`locus i32 DAQ2TGet(u32 deviceNumber, u32 triggerChannel, u32 triggerSlope, i32 triggerLevel, i16 *buffer, u32 triggerPosition, u32 blockCount, u32 *actualCount, u32 timeout);`
Pascal Syntax	`function DAQ2TGet(deviceNumber : i32; triggerChannel : i32; triggerSlope : i32; triggerLevel : i32; buffer : pi16; triggerPosition : i32; blockCount : i32; var actualCount : i32; timeout : i32) : i32;`
BASIC Syntax	`FN DAQ2TGet(deviceNumber&, triggerChannel&, triggerSlope&, triggerLevel&, buffer&, triggerPosition&, blockCount&, actualCount&, timeout&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

triggerChannel is the analog input channel number to trigger on. If **triggerSlope** is 0, no triggering is performed and this **triggerChannel** value is ignored. If SCXI is used, this parameter should be the onboard channel number. Analog triggering can be used with SCXI only for single-channel acquisitions and for multiple-channel scanned acquisitions in Parallel mode.

Range: 0 through *n*-1, where *n* is the number of analog input channels available.

triggerSlope is the slope to trigger on. Triggering is disabled by setting **triggerSlope** to 0.

0: no triggering is performed.

1: negative slope.
2: positive slope.

triggerLevel is the analog input value to trigger on. If **triggerSlope** is 0, no triggering is performed and this **triggerLevel** value is ignored.

buffer is a buffer of length **blockCount**. When `DAQ2Get` completes without error, then **buffer** contains the next block of acquired data. The elements of **buffer** are the 12-bit (NB-MIO-16, DAQCard-500/700, Lab-NB, and Lab-LC) or 16-bit (PCI-MIO-16XE-50, NB-MIO-16X) results of each A/D conversion in the data acquisition operation.

blockCount is the number of samples to be acquired from the large acquisition buffer while a double-buffered acquisition is in progress (that is, the number of A/D conversions). This value represents the size of the smaller blocks that divide up the larger acquisition buffer. For optimum performance, **blockCount** should be equal to or less than the **blockCount** value that was specified in `DAQ2Config`.

Range: 1 to 2^{31}-1.

If the current data acquisition is a multiple-channel scanning acquisition, then **blockCount** is equal to the number of samples per channel multiplied by the *total* number of channels being scanned.

The input values indicated in `SCAN_Setup` determine the total number of scanned channels. If no external multiplexer boards are used, then the total number of scanned channels is the number of onboard channels specified in the `SCAN_Setup` call. If one or more external multiplexer (AMUX-64T) boards are used, then the total number of channels equals (four-to-one multiplexer) ∗ (the number of onboard channels scanned) ∗ (the number of external multiplexer boards); that is, the total number of channels equals (4) ∗ (number of onboard channels) ∗ **muxMode**. For example, if one external board (AMUX-64T) is used and eight onboard channels are scanned, then the total number of scanned channels is equal to (4) ∗ (8) ∗ (1) = 32. If SCXI is used, the total number of channels scanned depends on the operating modes of the modules and the number of channels specified in the `SCXI_SCAN_Setup` call.

blockCount must be an integer multiple of the number of channels.

triggerPosition is the number of samples from the trigger channel to be retrieved before the trigger. The sample at this position in the retrieved data buffer matches the trigger slope and level criteria. (`DAQ2TGet` only)

actualCount is the actual number of samples left in the large acquisition buffer after `DAQ2TGet` is executed. If the number of samples remaining to be retrieved is less than the requested **blockCount** and the data acquisition operation is complete, then an error is returned. In this case, the **buffer** array contains the rest of the acquisition data, and **actualCount** indicates the number of valid samples returned. (`DAQ2TGet` only)

timeout is the number of ticks (60ths of a second) to wait for valid data before returning, if the retrieval was unsuccessful. `DAQ2Get` returns a **timeOutErr** if analog input trigger conditions are not met, or if not enough data points have been acquired before the specified number of ticks expire. If **timeout** is 0, no time limit is imposed, in which case `DAQ2Get` does not complete until the data can be returned successfully.

After a double-buffered data acquisition is started by executing `DAQ_Start`, `SCAN_Start`, or `SCAN_IntStart`, `DAQ2Get` can then be executed to retrieve blocks of data from the background acquisition. `DAQ2Get` waits until the requested number of samples (**blockCount**) are available before returning. An index into the large buffer is updated with each `DAQ2Get` call to keep track of what has been retrieved with `DAQ2Get` calls. Therefore, `DAQ2Get` can be executed repeatedly to return sequential blocks of data. `DAQ2Get` is useful for applications such as concurrent processing of acquired data or logging acquired data to disk while the acquisition is in progress.

Analog triggering can be enabled by setting **triggerSlope** to 2 (positive) or setting **triggerSlope** to 1 (negative) and by selecting a trigger channel (**triggerChannel**), **level**, and **timeout** value. Enabling analog triggering causes `DAQ2Get` to return a *triggered* block of data–a block whose first analog input value for the trigger channel is within the specified **triggerSlope** and **level**. If triggering is enabled, `DAQ2Get` scans the large

acquisition buffer and returns the next block of data acquired after trigger conditions are met. `DAQ2Get` waits until the requested number of samples (**sample count**) following the trigger value are available before returning.

If analog triggering is enabled and no double-buffered data acquisition operation is being performed on the selected trigger **channel**, then the trigger conditions are ignored and the next sequential block of data is returned. `DAQ2Get` returns a **timeOutErr** if the data cannot be retrieved successfully before the number of ticks specified in **timeout** expire. Executing `DAQ2Get` with **triggerSlope** set to 0 disables the analog triggering feature and returns the next sequential block of data after it is acquired.

If `DAQ2Get` returns an **overFlowError**, **overRunError**, or **overWriteErr**. A/D conversions may have been lost due to samples being acquired too rapidly as specified by `DAQ_Start`, `SCAN_Start`, or `SCAN_IntStart` (sample interval is too small). An **overFlowError** indicates that the A/D FIFO overflowed because the data acquisition servicing operation could not keep up with the sample rate. An **overRunError** indicates that the data acquisition circuitry could not keep up with the sample rate. An **overWriteErr** indicates that the large circular acquisition buffer overwrote acquired data before it was retrieved by `DAQ2Get` . An overwrite error can be corrected by increasing the size of the large acquisition buffer, retrieving more data each time, retrieving data more often, decreasing the size of the smaller dividing blocks, or reducing the sampling rate. The large acquisition buffer and smaller dividing block sizes are specified by `DAQ2Config`.

Once `DAQ2Get` completes with **error** = 0, `DAQ_VScale` can be used to scale the values in the **buffer** array to the actual voltages measured.

Pascal and C Note: ***If the double-buffered acquisition is a scanning acquisition, then*** `SCAN_Demux` ***must be called to demultiplex the*** **buffer** ***array by channel before the data is scaled.***

DAQ2MemConfig

Function

Configures NI-DAQ for Macintosh to use a memory expansion board for subsequent data acquisition in double-buffered mode. Memory for the large buffer allocated by `DAQ2Config` for the background acquisition is allocated from the memory space on the expansion board.

Synopsis

C Syntax	`locus i32 DAQ2MemConfig(u32 deviceNumber, u32 memorySlot, u32 mode);`
Pascal Syntax	`function DAQ2MemConfig(deviceNumber : i32; memorySlot : i32; mode : i32) : i32;`
BASIC Syntax	`FN DAQ2MemConfig(deviceNumber&, memorySlot&, mode&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

memorySlot is the board slot number of the memory expansion board to be used for double-buffered acquisition.

Range: 1 through 6.

Note: ***Because some memory expansion boards do not have configuration ROMs, NI-DAQ for Macintosh is unable to verify that a memory expansion board is actually in*** **memorySlot***. Make sure* **memorySlot** ***is valid or system errors can occur.***

mode indicates whether to enable or disable the use of the memory expansion board in **memorySlot** for subsequent double-buffered data acquisitions. Possible values of **mode** are as follows:

0: disable use of the memory expansion board for subsequent double-buffered acquisitions.
1: enable use of the memory expansion board for subsequent double-buffered acquisitions.

`DAQ2MemConfig` configures NI-DAQ for Macintosh to use a memory expansion board for subsequent double-buffered data acquisitions. When use of the memory board has been enabled, `DAQ2Config` allocates memory for the large circular buffer on the expansion memory board. Calls to `DAQ2Get`, `DAQ2TGet`, `DAQ2Tap`, and `DAQ2TTap` retrieve blocks from the large buffer on the memory board.

NI-DAQ for Macintosh initially disables the use of a memory expansion board. Once `DAQ2MemConfig` is called with **mode** set to 1, the memory expansion board is used for all subsequent double-buffered acquisitions until this feature is disabled by calling `DAQ2MemConfig` with **mode** set to 0.

DAQ2Tap

Function

Returns the most recently acquired block of data from the background double-buffered data acquisition (both single-channel and multiple-channel scan). This function can also be used to define a trigger that determines when to start acquiring the block of data (MIO boards).

DAQ2TTap

Function

The same functionality as `DAQ2Tap`. In addition, `DAQ2TTap` can indicate where the triggerPosition occurs within the block of data.

DAQ2Tap Synopsis

C Syntax	`locus i32 DAQ2Tap(u32 deviceNumber, u32 triggerChannel, u32 triggerSlope, i32 triggerLevel, i16 *buffer, u32 blockCount);`
Pascal Syntax	`function DAQ2Tap(deviceNumber : i32; triggerChannel : i32; triggerSlope : i32; triggerLevel : i32; buffer : pi16; blockCount : i32) : i32;`
BASIC Syntax	`FN DAQ2Tap(deviceNumber&, triggerChannel&, triggerSlope&, triggerLevel&, buffer&, blockCount&)`

DAQ2TTap Synopsis

C Syntax	`locus i32 DAQ2TTap(u32 deviceNumber, u32 triggerChannel, u32 triggerSlope, i32 triggerLevel, i16 *buffer, u32 triggerPosition, u32 blockCount);`
Pascal Syntax	`function DAQ2TTap(deviceNumber : i32; triggerChannel : i32; triggerSlope : i32; triggerLevel : i32; buffer : pi16; triggerPosition : i32; blockCount : i32) : i32;`
BASIC Syntax	`FN DAQ2TTap(deviceNumber&, triggerChannel&, triggerSlope&, triggerLevel&, buffer&, triggerPosition&, blockCount&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

triggerChannel is the analog input channel number to trigger on. This must be a channel that is currently being sampled. If **triggerSlope** is 0, no triggering is performed and this **triggerChannel** value is ignored. If SCXI is used, **triggerChannel** should be the onboard channel number. Analog triggering can be used with SCXI only for single-channel acquisitions and for multiple-channel scanned acquisitions in Parallel mode.

Range: 0 through *n*-1, where *n* is the number of analog input channels available.

triggerSlope is the slope to trigger on. Triggering is disabled by setting **triggerSlope** to 0.

triggerSlope = 0: no triggering is performed.
triggerSlope = 1: negative slope.
triggerSlope = 2: positive slope.

triggerLevel is the analog input value to trigger on. If **triggerSlope** is 0, no triggering is performed and this **triggerLevel** value is ignored.

blockCount is the number of samples to be acquired from the large acquisition buffer while a double-buffered acquisition is in progress (that is, the number of A/D conversions). This value represents the size of the smaller blocks that divide up the larger acquisition buffer. For optimum performance, **blockCount** should be equal to or less than the **blockCount** value that is specified in `DAQ2Config`.

Range: 1 to 2^{31}-1.

If the current data acquisition is a multiple-channel scanning acquisition, then **blockCount** is equal to the number of samples per channel multiplied by the *total* number of channels being scanned.

The input values specified in `SCAN_Setup` determine the total number of scanned channels. If no external multiplexer boards are used, then the total number of scanned channels is the number of onboard channels specified by the `SCAN_Setup` call. If one or more external multiplexer (AMUX-64T) boards are used, then the total number of channels equals (four-to-one multiplexer) ∗ (the number of onboard channels scanned) ∗ (the number of external multiplexer boards); that is, the total number of channels = (4) ∗ (number of onboard channels) ∗ **muxMode**. For example, if one external board (AMUX-64T) is used and eight onboard channels are scanned, then the total number of scanned channels is equal to (4) ∗ (8) ∗ (1) = 32. If SCXI is used, the number of channels scanned depends on the operating modes and the number of channels specified in the `SCXI_SCAN_Setup` call.

blockCount must be an integer multiple of the number of channels.

buffer is a buffer of length **blockCount**. When `DAQ2Tap` completes without error, then **buffer** contains the most recent block of acquired data. The elements of **buffer** are the 12-bit (NB-MIO-16, Lab and 1200 series) or 16-bit (NB-MIO-16X) results of each A/D conversion in the data acquisition operation.

triggerPosition is the number of samples from the trigger channel to be retrieved before the trigger. The sample at this position in the retrieved data buffer matches the trigger slope and level criteria. (`DAQ2TTap` only)

After a double-buffered data acquisition is started by executing `DAQ_Start`, `SCAN_Start`, or `SCAN_IntStart`, `DAQ2Tap` can then be executed to retrieve blocks of data from the background acquisition. Unlike `DAQ2Get`, `DAQ2Tap` does not update an index into the large buffer. Therefore, `DAQ2Tap` can be executed repeatedly to return the most recent blocks of data while the data is being acquired in the background. If the requested number of samples are not yet available, `DAQ2Tap` returns an error. Displaying data in a real-time mode is an example of an application using `DAQ2Tap`.

Analog triggering can be enabled by setting **triggerSlope** to 2 (positive) or setting **triggerSlope** to 1 (negative) and specifying trigger channel (**triggerChannel**), **level**, **triggerPosition**, and **timeout** values. Enabling analog triggering causes `DAQ2TTap` to return the most recent *triggered* block of data–a block in which the analog input value at the **triggerPosition** position for the channel **triggerChannel** meets the specified **triggerSlope** and **triggerLevel** conditions.

If triggering is enabled, `DAQ2TTap` scans the large acquisition buffer and returns the most recent block of data that meets the trigger conditions. If such a block is not yet available, `DAQ2TTap` returns an error. If analog triggering is enabled and no double-buffered data acquisition operation is being performed on the selected **triggerChannel**, then the trigger conditions are ignored and the most recently acquired block of data is returned. Executing `DAQ2TTap` with **triggerSlope** set to 0 disables the analog triggering feature and returns the most recently acquired block.

If `DAQ2Tap` returns an **overFlowError** or an **overRunError**. A/D conversions may have been lost due to samples being acquired too rapidly as specified by `DAQ_Start`, `SCAN_Start`, or `SCAN_IntStart` (sample interval was too small). An **overFlowError** indicates that the A/D FIFO overflowed because the data acquisition servicing operation could not keep up with sample rate. An **overRunError** indicates that the data acquisition circuitry could not keep up with the sample rate. `DAQ2Tap` returns an overwrite error if the most recent block of data is overwritten as it is being returned. An overwrite error can be corrected by increasing the

size of the large acquisition buffer, retrieving more data each time, retrieving data more often, decreasing the size of the smaller dividing blocks, or reducing the sampling rate. The large acquisition buffer and smaller dividing block sizes are specified by `DAQ2Config`.

Once `DAQ2Tap` completes with **error** = 0, `DAQ_VScale` can be used to scale the values in the **buffer** array to the actual voltages measured.

Pascal and C Note: ***If the double-buffered acquisition is a scanning acquisition, then*** `SCAN_Demux` ***must be called to demultiplex the*** **buffer** ***array by channel before the data is scaled.***

Multiple-Channel Data Acquisition (MDAQ)

The remainder of this chapter describes the Multiple-Channel Data Acquisition functions used with the NB-A2000, NB-A2100, and NB-A2150 boards for Macintosh computers. These boards do not support SCXI. The Multiple-Channel Data Acquisition functions retrieve multiple frames of data from one or more channels.

NB-A2000 Data Acquisition

The NB-A2000 contains four simultaneously-sampled, single-ended analog input channels numbered 0 through 3. Each analog input channel has a sample-and-hold circuit. The NB-A2000 samples one, two or four input channels simultaneously. These input channels are then multiplexed into a single unity gain stage followed by a 1-µs conversion time, 12-bit resolution, ADC which reads and converts each selected channel in turn. The channels that may be selected are as follows:

One channel: Channels 0, 1, 2 or 3
Two channels: Channels 0 and 1, or 2 and 3
Four channels: Channels 0 through 3

The NB-A2000 operates exactly the same way whether one or many channels are sampled.

The signal range of each input channel is ±5 V when DC coupling is selected and ±5 V peak AC with ±25 VDC offset when AC coupling is selected.

A/D conversions can be initiated through software or by applying active-low pulses to the SAMPCLK* input on the NB-A2000 I/O connector or active-high pulses to the CLOCKI RTSI bus input. A 1,024-word deep FIFO memory on the board stores up to 1,024 A/D conversion results.

The NB-A2000 operates in several trigger modes for data acquisition: pretrigger mode, posttrigger mode, or posttrigger mode with delay. In pretrigger mode, the NB-A2000 acquires a programmed number of samples both before and after a trigger is received. In posttrigger mode, a programmed number of samples is acquired after the trigger. In posttrigger mode with delay, the NB-A2000 waits to acquire samples until a programmed time interval has elapsed after the trigger is received.

The NB-A2000 has two main trigger sources: analog or digital. The analog trigger may be received from any one of the input channels or the ATRIG input on the I/O connector. Analog trigger circuitry causes a trigger when the selected input channels reach a pre-programmed slope and level. A rising or falling edge digital trigger may be received from the DTRIG I/O connector input. Alternatively, digital triggers may be received over the RTSI bus. In posttrigger mode, acquisition may be immediately started by software when the appropriate call is executed. Once configured to acquire samples, the NB-A2000 can trigger and acquire data each time a trigger is received without being stopped or reprogrammed. This is called multiple-frame data acquisition where a frame is the data acquired with each trigger.

NB-A2000 Data Acquisition Timing

Timing for data acquisition is performed by the onboard Am9513A Counter/Timer or by the external sample clock, SAMPCLK*. Data acquisition timing involves the timing signals and counters shown in the following table.

Signal Name	Description
Trigger	Signal generated from software or received from the ATRIG or DTRIG I/O connector input or from the RTSI bus. This signal determines when posttrigger sampling begins.
Sample Clock	Signal generated locally on the NB-A2000 or received from the I/O connector or from the RTSI bus that causes all selected inputs to be sampled simultaneously.
Sample Interval Counter	NB-A2000 counter that generates the onboard sample clock.
Sample Counter	NB-A2000 counter that counts posttrigger scans (multiple-channel samples) and stops acquisition when the programmed number has been acquired.
Delay Counter	NB-A2000 counter that counts the specified time delay after the trigger and then starts posttrigger acquisition when time expires.
Timebase	Onboard clock sources for the sample-interval and delay counters. Available timebases include 5 MHz, 1 MHz, 100 kHz, 10 kHz, 1 kHz, and 100 Hz. In addition, external timebase clocks can be supplied through the RTSI switch signals SOURCE2 and GATE2.

Additional timing signals can be received from the RTSI bus. See the *NB-A2000 User Manual* for more information regarding these signals.

NB-A2000 Data Acquisition Rates

The maximum data acquisition rate for the NB-A2000 is 1 µs/channel, in other words, 1 µs, 2 µs, or 4 µs for one, two, or four channels, respectively (see Table 6-11). Converting at a rate faster than the maximum data acquisition rate causes points to be missed, and the data returned will be an inaccurate representation of the signal being measured.

Table 6-11. Maximum NB-A2000 Data Acquisition Rates

Number of Channels	Maximum Data Acquisition Rate
one channel	1 µs (1 MS/s)
two channels	2 µs (500 kS/s)
four channels	4 µs (250 kS/s)

NB-A2100 Data Acquisition

The NB-A2100 contains two simultaneously-sampled analog input channels numbered 0 and 1. These 16-bit resolution A/D channels have 64-times oversampling delta-sigma modulating ADCs and digital anti-aliasing filters for extremely high-accuracy data acquisition. The input also has a software-programmed switch for AC or DC coupling of the input signals.

The signal range for each input channel is ±2.828 V (2 Vrms) with a maximum input voltage rating of ±10 V powered on or off.

The ADCs can be run at 22.05 kHz, 24 kHz, 32 kHz, 44.1 kHz, or 48 kHz conversion rates. Maximum data acquisition rate is 48 kHz for one or two channels. A 32-bit wide, 16 word deep FIFO memory on the board stores up to 32 A/D conversion results if one channel is being sampled, or 16 A/D conversion results for each channel if both analog input channels are being sampled.

The A/D conversion data can be sent serially over the RTSI bus to other National Instruments boards, such as the NB-DSP2300 digital signal processing board.

Data acquisition can be started by applying a software trigger or by applying a high-to-low edge on the external digital trigger. The digital trigger may be received through the EXTTRIG* pin on the I/O connector or over the RTSI bus.

A software analog trigger scheme is also implemented on the NB-A2100. The software analog trigger monitors the incoming data after the data acquisition has started for the specified trigger conditions to be met before storing data in the buffer.

NB-A2150 Data Acquisition

The NB-A2150 contains four simultaneously sampled analog input channels numbered 0 through 3. These 16-bit resolution A/D channels have 64-times oversampling delta-sigma modulating ADCs and digital anti-aliasing filters for extremely high-accuracy data acquisition. The input also has a software-programmed switch for AC or DC coupling of the input signals.

The signal range for each input channel is ±2.828 V (2 Vrms) with a maximum input voltage rating of ±10 V powered on or off.

The ADCs can be run at four timebases and each of these timebases is divided by 1, 2, 4, or 8 to produce 16 sample rates from which to choose. The timebase values are as follows:

NB-A2150F: 51.2 kHz, 48 kHz, 32 kHz, and 30.72 kHz
NB-A2150C: 48 kHz, 44.1 kHz, 32 kHz, and F_u
NB-A2150S: 24 kHz, 20 kHz, 16 kHz, and F_u

F_u is a user-defined sample rate and is obtained by dividing the custom installed crystal frequency by 384.

The NB-A2150 uses a 32-bit wide, 256 word deep FIFO to store up to 512 A/D conversion results.

You can start data acquisition by applying a software trigger, a high-to-low edge on an external digital trigger, or via a trigger generated from the internal level-and-slope detection trigger circuit. You can send the digital trigger from a signal applied at the EXTTRIG* pin on the I/O connector or from the RTSITRIG* or SWSTART* signals over the RTSI bus.

Multiple-Channel Data Acquisition Function Summary

Use the following functions for multiple-channel buffered data acquisition operations on the NB-A2000, the NB-A2100, and the NB-A2150:

`MDAQ_Check`	Reports whether the acquisition is complete, the current number of frames acquired, and optionally the current number of scans acquired.
`MDAQ_Clear`	Stops data acquisition but does not change the current configuration (for example, channel coupling, trigger mode and conditions, sampling rate, active input channels). Also clears the acquisition buffer and deallocates any resources used during the acquisition.
`MDAQ_Get`	Transfers acquired data from the acquisition buffer into the user's buffer while data acquisition is in progress or after data acquisition is complete. `MDAQ_Get` can retrieve data from anywhere in the acquisition buffer.
`MDAQ_ScanRate`	Selects the data acquisition scan rate, that is, the rate at which all selected input channels are sampled.

`MDAQ_Setup`	Selects how much data to buffer in memory and how much data to acquire for each trigger.
`MDAQ_Start`	Starts a multiple-channel data acquisition.
`MDAQ_Stop`	Stops the data acquisition but leaves all settings in effect and the acquisition buffer accessible.
`MDAQ_Trig_Config`	Selects trigger source and configures the analog and digital trigger conditions.
`MDAQ_Trig_Delay`	Selects the time to delay after a trigger is received before acquiring data. (Posttrigger mode only)

Multiple-Channel Data Acquisition Application Hints

The multiple-channel Data Acquisition functions perform both single-channel and multiple-channel data acquisition operations. The following terminology is used to describe the multiple-channel Data Acquisition functions:

frame	A set of samples acquired from all selected channels with each trigger. The number of samples per frame is equal to (**preScans** + **postScans**) $*$ number of channels selected.
scan	One sample from each of the selected analog input channels.
scanInterval	Time between the initiation of consecutive **scans**. Equivalent to the interval between samples on a given channel.
preScans	Number of **scans** to acquire before the trigger.
postScans	Number of **scans** to acquire after the trigger.

For both single read analog input and data acquisition, `MAI_Setup` selects the analog input channels to be sampled and `MAI_Coupling` selects AC or DC coupling for all inputs.

Frame-Oriented and Scan-Oriented Data Acquisition

Data acquisition with the NB-A2000, the NB-A2100, or the NB-A2150 can be performed in two modes: frame-oriented data acquisition or scan-oriented data acquisition. Both modes are double buffered, that is, data can be retrieved from the acquisition buffer while data is being acquired.

The frame-oriented data acquisition mode allows multiple frames to be acquired. A frame is acquired each time the board receives a trigger. Each frame can contain both pretrigger and posttrigger data. All frames are the same size and use the same trigger modes, number of channels, and acquisition rates. `MDAQ_Setup` configures the frame size and the number of pretrigger and posttrigger scans. `MAI_Setup` selects the analog input channels and `MDAQ_ScanRate` configures the acquisition rate. Using the `MDAQ_Start` function, either a specified number of frames can be acquired, or an unlimited number of frames can be acquired until acquisition is stopped by the user (`MDAQ_Stop`).

The scan-oriented data acquisition mode is a posttrigger, single-frame data acquisition case. After a trigger is received, either a specified number of scans is acquired before the acquisition is automatically stopped, or an unlimited number of scans is acquired until the acquisition is stopped by the user.

Configuring the Trigger Conditions

The `MDAQ_Setup`, `MDAQ_Trig_Config`, and `MDAQ_Trig_Delay` functions configure the acquisition's triggering conditions.

The NB-A2000, NB-A2100, and NB-A2150 have digital triggering capability in hardware. The NB-A2000 and the NB-A2150 also have analog triggering capability in hardware. The NB-A2100 implements analog triggering in software. You select and enable the triggers in `MDAQ_Trig_Config`, which arms both the analog and the digital trigger, as well as disables both triggers (thereby configuring a software triggered acquisition). When the acquisition is in posttrigger mode, you can set a delay between the trigger and the conversion of the first scan using the `MDAQ_Trig_Delay` function.

The NB-A2000 and the NB-A2150 support the following combinations of trigger modes:

- No trigger (software posttrigger acquisition).
- Hardware digital or analog posttrigger with or without delay.
- Hardware digital or analog pretrigger without delay.

For the NB-A2150, you can enable the digital trigger through the EXTTRIG* pin on the I/O connector using `MDAQ_Trig_Config`. To enable the digital trigger over the RTSI bus, you should call `RTSI_Conn` and make the RTSITRIG* or RTSISTART* line an input.

The NB-A2100 supports the following combinations of trigger modes:

- No trigger (software posttrigger acquisition).
- Hardware posttrigger with or without delay.
- Hardware pretrigger without delay.
- Software analog trigger without delay.
- Software analog trigger after a hardware posttrigger (with or without delay). In this case, after the hardware trigger starts the acquisition, the incoming data is monitored until the analog trigger conditions are met and the data is then stored in the buffer.

NB-A2100 and NB-A2150 Triggering

The NB-A2100 and NB-A2150 use analog and digital filters to implement anti-aliasing filters which reject signal components whose frequency exceeds one-half the sample rate. However, the implementation of the filters is such that after a data acquisition trigger (hardware or software) is applied, the first sample after the trigger does not appear in the A/D FIFO until 34 or 35 samples (depending on when the trigger occurred) from each channel being sampled have been acquired. This means that in the posttrigger mode, you have 34 or 35 samples of pretrigger data in the acquisition buffer. In the pretrigger mode, a software trigger started the acquisition. So, in this case the first 34 or 35 samples that are put in the buffer are samples taken before the software trigger was applied.

For the NB-A2100, in the analog trigger mode and after a hardware or software trigger starts the acquisition, the first 34 or 35 samples put in the A/D FIFO to be examined for analog trigger conditions are the samples taken before the hardware or software trigger occurred.

Stopping Data Acquisition

MDAQ_Stop can be called to halt data acquisition in progress. This is necessary when continuously scanning or when gathering an unlimited number of frames. After stopping, settings that do not affect the buffer or frame size can be changed before performing another MDAQ_Start without having to call MDAQ_Setup. This is useful in such applications as a digital oscilloscope where the scan rate often changes, but not the number of channels being scanned. The settings that can be changed are the scan rate (MDAQ_ScanRate), coupling (MAI_Coupling), trigger enabling (MDAQ_Trig_Config), and trigger delay (MDAQ_Trig_Delay)–if already in posttrigger mode.

Typical Multiple-Channel Data Acquisition Function Usage

In the function order sequence (see Figure 6-12 and Figure 6-13), the MDAQ_SCAN_Rate function is optional for the NB-A2100 and NB-A2150. The frequency selected by the last MDAQ_SCAN_Rate call is used for data acquisition if MDAQ_SCAN_Rate is not called. At startup or after a Board_Reset call, the 48-kHz frequency on the NB-A2100, the 51.2-kHz frequency on the NB-A2150F, the 48-kHz frequency on the NB-A2150C, and the 24-kHz frequency on the NB-A2150S is selected for A/D sampling.

A typical function order needed to start an acquisition is given as follows:

MAI_Coupling to select coupling on input channels.
MAI_Setup to select number of channels to monitor.
MDAQ_Setup to select frame size and number of pretrigger and posttrigger scans.
MDAQ_ScanRate to select acquisition rate. (optional for NB-A2100 and NB-A2150)
MDAQ_Trig_Config to select trigger type and conditions.
MDAQ_Trig_Delay to select posttrigger delay (if triggering is enabled and the posttrigger mode is used).
MDAQ_Start to select number of frames to acquire and to start acquisition.

While data is acquired, the following functions can be used:

MDAQ_Check to monitor the status of the acquisition.
MDAQ_Get to fetch data from anywhere in the acquisition buffer.
MDAQ_Stop to stop the data acquisition, after which MDAQ_Get can be used to fetch the data.
MDAQ_Clear to clear all resources allocated to the data acquisition operation, including the acquisition buffer.

Figures 6-12 and 6-13 illustrate the ways to use the multiple-channel Data Acquisition functions. Two examples of performing a data acquisition are shown, followed by a more detailed look at the possible combinations of functions used initially, during, and after the completion of an acquisition.

The functions in Figure 6-12 are the minimal set of functions necessary to acquire a given number of untriggered frames (that is, no hardware triggering enabled).

The example program MDAQ_Op calls a similar sequence of functions to acquire one frame of data. This program can be used as an additional high-level multiple-channel Data Acquisition function.

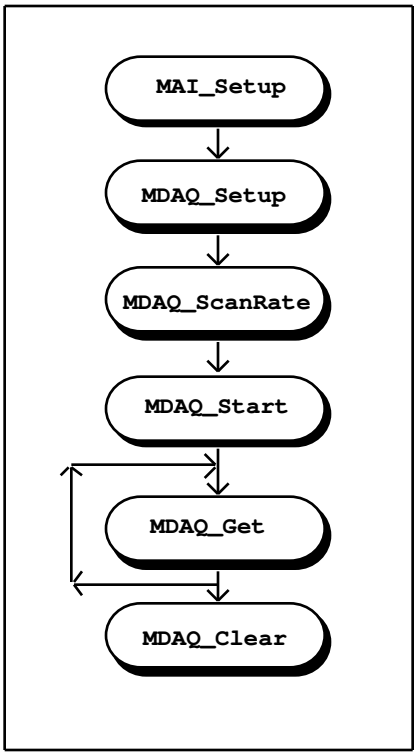

Figure 6-12. Minimum Function Flowchart for Multiple-Channel Data Acquisition

MAI_Setup configures the channels to scan during the acquisition. MDAQ_Setup configures and allocates the acquisition buffer. MDAQ_ScanRate configures the rate at which the input channels are scanned, and MDAQ_Start initiates the data acquisition operation. After starting, MDAQ_Get is called to retrieve the frames from the acquisition buffer, and finally MDAQ_Clear stops the acquisition and clears the acquisition buffer. MDAQ_Clear must always be called at the end of any data acquisition program and before exiting your application to release resources allocated by NI-DAQ for Macintosh for use during an acquisition.

Figure 6-13 gives a more general view of the multiple-channel Data Acquisition functions needed to set up an acquisition to acquire a given number of frames, configure the triggering circuitry, and then retrieve and scale the acquired frames.

Figure 6-13 adds a few functions to the sequence given in Figure 6-12. The boxed functions represent optional steps. The optional MAI_Coupling step is used to set AC or DC coupling for the analog input and analog trigger channels. At power-up and after an Board_Reset, the coupling for all channels defaults to AC on the NB-A2000 and DC on the NB-A2100 and NB-A2150. The optional MDAQ_Trig_Config step enables and configures the analog and/or digital triggering conditions that must be met to initiate the acquisition of each frame. The optional MDAQ_Trig_Delay step sets a delay of up to 10.9 minutes on the NB-A2000 and up to 65,535 sample intervals on the NB-A2100 and NB-A2150 after the trigger occurs before beginning to acquire data when in posttrigger mode.

If the data acquisition is configured to continuously scan or acquire an unlimited number of frames, the MDAQ_Stop function can be called to stop the acquisition. After stopping, MDAQ_Get can still be used to retrieve any data present on the acquisition buffer, and MDAQ_Check returns the progress of the acquisition when MDAQ_Stop was called. If used, MDAQ_Stop should be called after MAI_Scale and before MDAQ_Clear as in Figure 6-13.

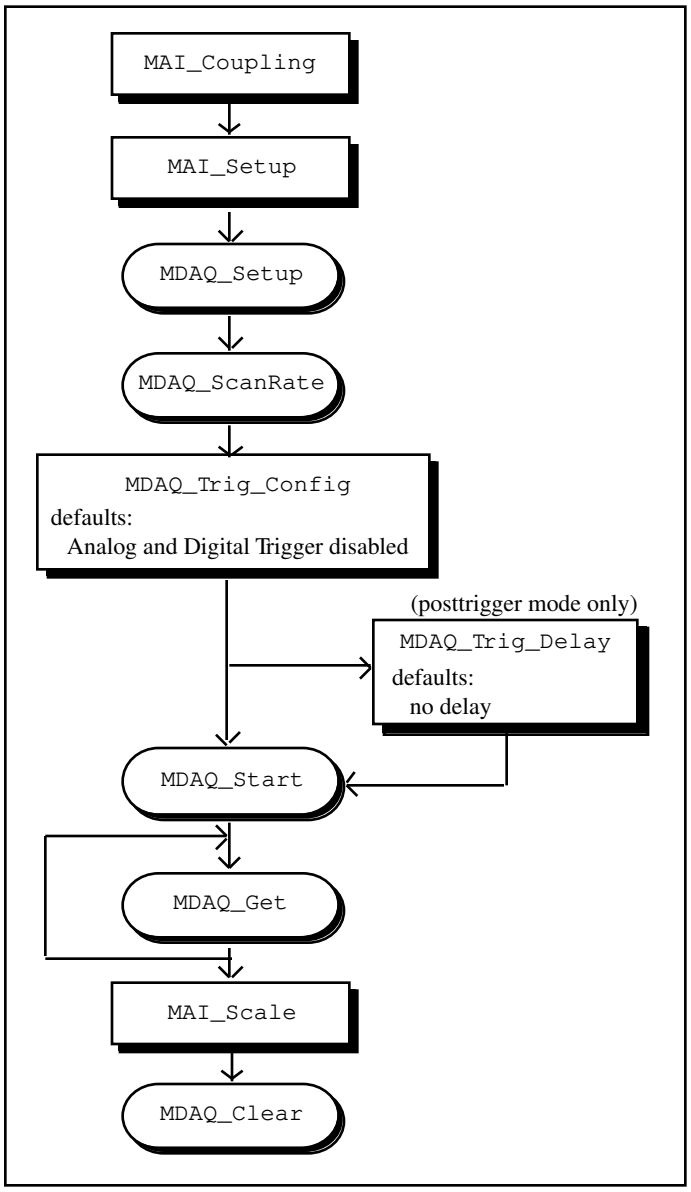

Figure 6-13. Multiple-Channel Data Acquisition with Optional Coupling and Triggering Configuration

MDAQ_Check

Function

Reports whether the acquisition is complete, the current number of frames acquired, and optionally the current number of scans acquired.

Synopsis

C Syntax	`locus i32 MDAQ_Check(u32 deviceNumber, u32 fullCheck, u16 *done, u32 *currentFrame, u32 *currentScan);`
Pascal Syntax	`function MDAQ_Check(deviceNumber : i32; fullCheck : i32; var done : i16; var currentFrame : i32; var currentScan : i32) : i32;`
BASIC Syntax	`FN MDAQ_Check(deviceNumber&, fullCheck&, done&, currentFrame&, currentScan&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

fullCheck indicates whether to return complete or partial status information. Partial information includes all parameters except **currentScan**.
0: return partial status information.
1: return complete status information.

Updating the **currentScan** parameter when DMA is used requires temporarily suspending servicing of the board being checked. If performing many `MDAQ_Check` operations with **fullCheck** = 1 during a high-speed acquisition or while operating many boards together, a board's FIFO can overflow and halt the data acquisition on that board. For these cases and when only the status information concerning the completion of the acquisition is desired, setting **fullCheck** to 0 is recommended.

done returns an indication of whether the data acquisition has completed.
0: the data acquisition is not yet complete.
1: the acquisition is complete.

If you have chosen either continuous acquisition of scans in the `MDAQ_Setup` function or unlimited acquisition of frames after the first trigger in the `MDAQ_Start` function, then the status remains 0 until the acquisition is stopped with `MDAQ_Stop` or an error (such as a FIFO overflow) has stopped the acquisition, after which **done** becomes 1.

currentScan returns the number of the last completed frame. In frame-oriented acquisitions, **currentScan** ranges from 0 (when the acquisition has not yet started–for example, when waiting for the first trigger in posttrigger mode) to the total number of frames to be acquired (as defined in `MDAQ_Start`). This number reaches 2,147,483,647 (the maximum long value) before rolling over to 0 and continuing to count. In scan-oriented acquisitions, **currentScan** is 0 until the acquisition is complete, at which time **currentScan** becomes 1.

currentScan returns the most recent scan number that has been acquired within the current frame. In frame-oriented acquisitions, **currentScan** ranges from 0 (when the frame has not yet started–for example, when waiting for a trigger in posttrigger mode) to the number of scans in a frame (as defined in `MDAQ_Setup`). Furthermore, if the frames include pretrigger scans, then **currentScan** reaches the number of scans in a frame then wrap around to 1 and continue counting. In scan-oriented acquisitions, **currentScan** ranges from 0 to the number of posttrigger scans to collect. If continuously scanning, then **currentScan** reaches 2,147,483,647 (the maximum signed long value) before rolling over to 1 and continuing to count. When **fullCheck** is 0, **currentScan** is always set to 0.

MDAQ_Clear

Function

Stops data acquisition but does not change the current configuration (for example, channel coupling, trigger mode and conditions, sampling rate, active input channels). Also clears the acquisition buffer and deallocates any resources used during the acquisition.

Synopsis

C Syntax	`locus i32 MDAQ_Clear(u32 deviceNumber);`
Pascal Syntax	`function MDAQ_Clear(deviceNumber : i32) : i32;`
BASIC Syntax	`FN MDAQ_Clear(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`MDAQ_Clear` does not affect any settings, but it does release the acquisition buffer and other resources used. To start another data acquisition, a `MDAQ_Setup` is required before executing `MDAQ_Start` again.

`MDAQ_Clear` *must* be called before exiting your application; otherwise, resources are not properly deallocated.

MDAQ_Get

Function

Transfers acquired data from the acquisition buffer into the user's buffer while data acquisition is in progress or after data acquisition is complete. `MDAQ_Get` can retrieve data from anywhere in the acquisition buffer.

Synopsis

C Syntax	`locus i32 MDAQ_Get(u32 deviceNumber, u32 scansOrFrames, u32 getOrTap, u32 count, u32 startFrame, u32 startScan, u32 timeout, i16 *buffer, u32 *actualCount, u32 *currentFrame, u32 *currentSample, u16 *done);`
Pascal Syntax	`function MDAQ_Get(deviceNumber : i32; scansOrFrames : i32; getOrTap : i32; count : i32; startFrame : i32; startScan : i32; timeout : i32; buffer : pi16; var actualCount : i32; var currentFrame : i32; var currentSample : i32; var done : i16) : i32;`
BASIC Syntax	`FN MDAQ_Get(deviceNumber&, scansOrFrames&, getOrTap&, count&, startFrame&, startScan&, timeout&, buffer&, actualCount&, currentFrame&, currentSample&, done&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

scansOrFrames indicates whether to retrieve scans or frames from the acquisition buffer.

0: **scansOrFrames**: get scans from the acquisition buffer.
1: **scansOrFrames**: get frames from the acquisition buffer.

getOrTap indicates the retrieval method used to retrieve data from the acquisition buffer.
0 (performing a GET): get oldest data sequentially.
1 (performing a TAP): get most recently acquired data.

In either case, a starting frame and scan can be specified. When performing a GET, NI-DAQ for Macintosh uses an internal sequential retrieval pointer to keep track of what has been retrieved to guarantee sequential access. No such pointer is maintained when performing a TAP.

count is the number of scans or frames to retrieve from the acquisition buffer. If **scansOrFrames** is 0 and the acquisition is frame-oriented, then **count** ranges from 1 to the number of scans in a frame (**pre_trig_scans + postScans**). Scans can not be retrieved across frame boundaries. If **scansOrFrames** is 1, then **count** ranges from 1 to the number of frames in the acquisition buffer.

startFrame is the frame number to begin copying from. If performing a GET (**getOrTap** = 0) and **startFrame** is not 0, then copying begins from **startFrame**, and after copying the internal sequential retrieval pointer is set to the frame after the last frame copied. If performing a GET and **startFrame** is 0, then the internal sequential retrieval pointer is used to determine where to begin copying. If performing a TAP (**getOrTap** = 1) and **startFrame** is not 0, then copying begins from **startFrame**. If **getOrTap** is 1 and **startFrame** is 0, then the most recently acquired data is copied. When performing a TAP, no internal retrieval pointers are used or modified.

startScan is the scan number to begin copying within **startFrame**. If **scansOrFrames** is 1, then **startScan** is ignored. If **scansOrFrames** is 0 and **startFrame** is 0, then **startScan** must also be 0.

timeout is the number of clock ticks (1/60 s) to wait for data that has not yet been acquired. There are two special cases for timeout:
-1: wait indefinitely.
0: return immediately if the data has not been acquired.

buffer is the address of a non-relocatable array or otherwise allocated memory to store the data retrieved from the acquisition buffer. The integer data is copied from the acquisition buffer to **buffer**. If more than one input channel is being sampled, the samples from the different channels is interleaved in the following order:

(Sample 1, channel 0), (Sample 1, channel 1), (Sample 1, channel 2), (Sample 1, channel 3), (Sample 2, channel 0), (Sample 2, channel 1), (Sample 2, channel 2), (Sample 2, channel 3), (Sample 3, channel 0), (Sample 3, channel 1), and so on.

actualCount returns the number of items copied from the acquisition buffer to **buffer**. When **scansOrFrames** is 1, **actualCount** indicates the number of complete frames that were transferred. When **scansOrFrames** is 0, **actualCount** indicates the number of scans that were transferred. When an error has not occurred, **actualCount** should equal **count**. If an error has occurred, then **actualCount** indicates how much of **buffer** is actually valid data.

currentFrame returns the number of the last frame from which data was copied. If **currentFrame** is 0, then no data was copied.

currentSample returns the last scan number within **currentFrame** that was copied from the acquisition buffer to **buffer**. If **currentSample** is 0, no data was copied.

done returns an indication of whether the data acquisition has completed.
0: the data acquisition is not yet complete.
1: the acquisition is complete.

If you have chosen either continuous acquisition of scans in the MDAQ_Setup function or unlimited acquisition of frames after the first trigger in the MDAQ_Start function, then the status is always 0 because data acquisition runs indefinitely until stopped by an MDAQ_Stop or MDAQ_Clear call.

Call `MDAQ_Get` to retrieve data from the acquisition buffer both while acquisition is in progress and after acquisition is complete. `MDAQ_Get` allows double-buffered data acquisition. Because the acquisition buffer is circular, you must retrieve data fast enough to keep pace with the acquisition or data can be overwritten. If you attempt to retrieve overwritten data, an **overWriteErr** is returned.

If performing a `GET` (**getOrTap** = 0) and a starting place that has been overwritten is selected, an **overWriteErr** is returned, as well as the oldest available data. If performing a `TAP` (**getOrTap** = 1) and a starting place that has been overwritten is selected, then an **overWriteErr** and no data is returned.

MDAQ_ScanRate

Function

Selects the data acquisition scan rate, that is, the rate at which all selected input channels are sampled.

Synopsis

C Syntax	`locus i32 MDAQ_ScanRate(u32 deviceNumber, u32 scanInterval, i32 timebase);`
Pascal Syntax	`function MDAQ_ScanRate(deviceNumber : i32; scanInterval : i32; timebase : i32) : i32;`
BASIC Syntax	`FN MDAQ_ScanRate(deviceNumber&, scanInterval&, timebase&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

scanInterval indicates the length of the scan interval (that is, the amount of time to elapse between samples on any one channel).

Range: 2 through 65,535 on the NB-A2000.
1, 2 on the NB-A2100.
1, 2, 4, 8 on the NB-A2150.

Because the NB-A2100 only samples at fixed timebase rates, the **scanInterval** must be 1 for the NB-A2100.

On the NB-A2000, the **scanInterval** is a function of the timebase resolution selected by **timebase**. The actual scan interval in seconds is determined by the following formula:

scanInterval * (timebase resolution)

On the NB-A2100 and NB-A2150, the actual sampling frequency in hertz is determined by the following formula:

$$\frac{\textbf{timebase frequency}}{\textbf{scanInterval}}$$

timebase is the resolution to use for the sample-interval counter on the NB-A2000 or the sampling frequency to use on the NB-A2100.

For the NB-A2000, **timebase** has the following possible values:

-1: 200 ns
0: reserved
1: 1 µs
2: 10 µs

3: 100 µs
4: 1 ms
5: 10 ms

For the NB-A2100, **timebase** has the following possible values:

1: 48 kHz
2: 44.1 kHz
3: 32 kHz

On the NB-A2100, although it is possible to select a 16 kHz conversion rate, data integrity is not guaranteed or specified and the ADC is likely to perform erratically. However, it is recommended that if you are not using the ADC but are using the DAC at the 16 kHz update rate, you should also set the ADC conversion rate at the 16-kHz rate for less noise interference between the A/D and D/A word clock signals.

For the NB-A2150, **timebase** has the following possible values:

timebase	NB-A2150F	NB-A2150C	NB-A2150S
0	30.72 kHz	F_u	F_u
1	51.2 kHz	48 kHz	24 kHz
2	48 kHz	44.1 kHz	20 kHz
3	32 kHz	32 kHz	16 kHz

If **timebase** 0 is selected on the NB-A2150C or NB-A2150S, you must have a custom crystal installed on the board that defines F_u. F_u is the user-defined sample rate and is obtained by dividing the custom-installed crystal frequency by 384. F_u should range from 8 kHz to 51.2 kHz.

Table 6-12 gives the minimum scan rate values allowed on the NB-A2000.

Table 6-12. Minimum Scan Rate Values on the NB-A2000

channelCount	timebase	scanInterval	actual interval
1	-1	5	1 µs
2	-1	10	2 µs
4	-1	20	4 µs

If external sample clock was selected in `A2000_Config`, the **scanInterval** and **timebase** values are ignored and an error is returned as a warning.

MDAQ_Setup

Function

Selects how much data to buffer in memory and how much data to acquire for each trigger.

Synopsis

C Syntax	`locus i32 MDAQ_Setup(u32 deviceNumber, u32 bufferSize, u32 scansOrFrames, u32 preScans, u32 postScans, i16 *buffer);`
Pascal Syntax	`function MDAQ_Setup(deviceNumber : i32; bufferSize : i32; scansOrFrames : i32; preScans : i32; postScans : i32; buffer : pi16) : i32;`
BASIC Syntax	`FN MDAQ_Setup(deviceNumber&, bufferSize&, scansOrFrames&, preScans&, postScans&, buffer&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

bufferSize indicates the size of the large circular buffer (the acquisition buffer) used during the data acquisition. **bufferSize** is either the number of scans or the number of frames in the acquisition buffer depending on the setting of **scansOrFrames** (see below). `MDAQ_Setup` returns an error if **bufferSize** is too large. How the size of the acquisition buffer in bytes is calculated is explained in **buffer**.

scansOrFrames indicates whether the acquisition is scan oriented or frame oriented.

0: scan-oriented acquisition, a single posttrigger frame is acquired, and **bufferSize** indicates the number of scans in the acquisition buffer.

1: frame-oriented acquisition, **bufferSize** indicates the number of frames in the acquisition buffer.

preScans is the number of scans before a trigger to collect in each frame.

postScans is the number of scans after a trigger to collect in each frame.

When **scansOrFrames** = 1 (frame-oriented), then the frame size is **preScans** + **postScans**. The acquisition buffer contains **bufferSize** number of frames of this size.

The valid combinations of settings for **scansOrFrames**, **preScans**, and **postScans** and their relationship to acquisition modes are shown in Table 6-13.

Table 6-13. Valid Combinations of MDAQ_Setup Parameters

Parameter Combination	Results
scansOrFrames = 1 **preScans** = 0 **postScans** > 0	frame-oriented data acquisition posttrigger mode **frame_size** = (**postScans**) ∗ **channelCount** **bufferSize** is in frames
scansOrFrames = 1 **preScans** = 0 **postScans** > 0	frame-oriented data acquisition pretrigger mode **preScans**) ∗ **channelCount** **bufferSize** is in frames
scansOrFrames = 0 **preScans** = 0 **postScans** > 0	scan-oriented data acquisition posttrigger mode (only valid mode) **bufferSize** is in scans **postScans** is total number of scans to acquire use **frameCount** = 1 in `MDAQ_Start` call
scansOrFrames = 0 **preScans** = 0 **postScans** > 0	scan-oriented data acquisition posttrigger mode (only valid mode) **bufferSize** is in scans. Use **frameCount** = 1 in `MDAQ_Start` call.
Note: When **scansOrFrames** = 0, setting **preScans** greater than 0 returns an error. When **scansOrFrames** = 1, setting **postScans** equal to 0 returns an error.	

buffer is a pointer to a non-relocatable memory space allocated by the user for the acquisition buffer. A zero pointer tells the driver to automatically allocate the acquisition buffer. We recommend that you set **buffer** to 0 to let the driver handle memory for you unless you need to use a NuBus memory expansion board. To use a memory expansion board, set **buffer** to the 32-bit mode address of the start of the buffer on the memory board. When you allocate a buffer and pass the pointer to the buffer in **buffer**, be sure to allocate the correct amount of storage. For example, if you want a buffer to hold 10 frames, where

channelCount (from `MAI_Setup`) = 2 (two channels)
preScans = 500
postScans = 2000

then you want to allocate bytes of storage (as explained in the following equation) and set **bufferSize** to 10.

(channelCount) ∗ (preScans = postScans) ∗ (number of frames) ∗ (16-bit integer)
= 2 ∗ (500 + 2000) ∗ 10 ∗ 2
= 100,000 bytes

Conversely, if you want a buffer to hold 20,000 scans for a four channel acquisition, be sure to allocate 160,000 bytes of storage (as explained in the following equation) and set **bufferSize** to 20,000.

(number of channels) ∗ (number of scans) ∗ (16-bit integer) = 4 ∗ 20,000 ∗ 2 = 160,000 bytes

Notice that if you allocate a buffer, it is your responsibility to deallocate the memory; NI-DAQ for Macintosh does not attempt to release memory that it does not allocate.

If the memory allocated for the buffer supports block-mode memory transfers and you are using an NB-A2000 board with an NB-DMA2800, you can force the driver to use high-speed block-mode transfers by setting the `A2000_Config` **mem_type** parameter to 1. When using block-mode DMA transfers, the buffer base address

must be aligned at a NuBus block boundary. The NuBus block size is 64 bytes (16 32-bit words); a NuBus block boundary is a multiple of NuBus block size. When acquiring frames, the frame size (sum of **preScans** and **postScans**) must be a multiple of NuBus block size. When acquiring scans, **bufferSize** and **postScans** (if **postScans** is not 0) must each be a multiple of NuBus block size. If any of the above conditions are not met, an error is returned.

If you are using an NB-A2000 with an NB-DMA2800 on a Macintosh Quadra 700 or Quadra 900, buffer size is a multiple of NuBus block size as described previously, and either **buffer** is 0 or **buffer** is at a NuBus block boundary, the driver will automatically use block-mode DMA transfers.

On the NB-A2000, **postScans** must always be at least 2, and when a frame-oriented acquisition is to acquire pretrigger data, **preScans** must be at least 3. Additionally, if `MAI_Setup` has been called to configure an acquisition for a single channel, **postScans** must be even for a scan-oriented acquisition, and the sum of **postScans** and **preScans** must be even for a frame-oriented acquisition. When an acquisition is configured for two or more channels, no such restrictions apply.

On the NB-A2100 or NB-A2150, if `MAI_Setup` has been called to configure and sample only one channel, the number of scans to acquire (**postScans**) in the scans-oriented acquisition, or the number of scans to acquire per frame (**postScans** and **preScans**) in the frames-oriented acquisition must be a multiple of 2. Also, if the number of scans to acquire in the scans-oriented acquisition or the number of scans to acquire per frame in the frames-oriented acquisition is greater than 131,072 if one channel is being scanned, greater than 65,536 if two channels are being scanned, or greater than 32,768 if four channels are being scanned, the number of scans should meet one of the following conditions:

- Be a power of 2 (that is, 65,536 (2^{16}), 131,072 (2^{17}), 262,144 (2^{18}), and so on).
- Be a multiple of 10 if less than 500,000.
- Be a multiple of 100 if greater than 500,000 and less than 5,000,000.
- Be a multiple of 1,000 if greater than 5,000,000 and less than 50,000,000.

MDAQ_Start

Function

Starts a multiple-channel data acquisition operation.

Synopsis

C Syntax	`locus i32 MDAQ_Start(u32 deviceNumber, u32 frameCount);`
Pascal Syntax	`function MDAQ_Start(deviceNumber : i32; frameCount : i32) : i32;`
BASIC Syntax	`FN MDAQ_Start(deviceNumber&, frameCount&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

frameCount is the number of triggers recognized and for which data is acquired. If **scansOrFrames** has been set to 1 (frames) in `MDAQ_Setup`, then **frameCount** indicates the number of frames to acquire. The acquisition runs until **frameCount** number of frames has been acquired, at which time the acquisition stops (as if `MDAQ_Stop` had been called). Setting **frameCount** to 0 means that the acquisition is in *unlimited* frame acquisition mode. Unlimited frame acquisition mode indicates that the driver recognizes and acquire frames of

data for triggers indefinitely. In this case, the acquisition does not end until either `MDAQ_Stop` or `MDAQ_Clear` is called.

If **scansOrFrames** has been set to 0 (scan-oriented data acquisition) in `MDAQ_Setup` , then **frameCount** *must* be set to 1 (only one posttrigger frame is acquired) and scans are acquired after the first trigger is received. Furthermore, if **postScans** has been set to 0 (**preScans** must be 0 when **scansOrFrames** is 1), then the NB-A2000, NB-A2100, or NB-A2150 acquires an unlimited number of scans until either `MDAQ_Stop` or `MDAQ_Clear` is called.

If you chose continuous acquisition of scans in `MDAQ_Setup` or unlimited acquisition of frames in `MDAQ_Start`, you get an error if the NB-A2100 board was unable to acquire a DMA channel for data acquisition. This error is designed to prevent your computer from locking up when continuous data acquisition is done in programmed I/O mode on the NB-A2100.

MDAQ_Stop

Function

Stops the data acquisition but leaves all settings in effect and the acquisition buffer accessible.

Synopsis

C Syntax	`locus i32 MDAQ_Stop(u32 deviceNumber);`
Pascal Syntax	`function MDAQ_Stop(deviceNumber : i32) : i32;`
BASIC Syntax	`FN MDAQ_Stop(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`MDAQ_Stop` stops the data acquisition process but leaves the acquisition buffer accessible for subsequent `MDAQ_Get` calls. Because all settings are left in effect, the data acquisition can be restarted (not resumed) by calling `MDAQ_Start` again.

MDAQ_Trig_Config

Function

Selects trigger source and configures the analog and digital trigger conditions.

Synopsis

C Syntax	`locus i32 MDAQ_Trig_Config(u32 deviceNumber, u32 digitalTrigger, u32 edge, u32 analogTrigger, u32 triggerCount, u32 source, u32 triggerSlope, i32 triggerLevel, u32 hysteresisWindow, u32 timeout);`
Pascal Syntax	`function MDAQ_Trig_Config(deviceNumber : i32; digitalTrigger : i32; edge : i32; analogTrigger : i32; triggerCount : i32; source : i32; triggerSlope : i32; triggerLevel : i32; hysteresisWindow : i32; timeout : i32) : i32;`
BASIC Syntax	`FN MDAQ_Trig_Config(deviceNumber&, digitalTrigger&, edge&, analogTrigger&, triggerCount&, source&, triggerSlope&, triggerLevel&, hysteresisWindow&, timeout&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

digitalTriggerenables or disables the digital trigger input (DTRIG on the NB-A2000 or EXTTRIG* on the NB-A2100 and NB-A2150) I/O connector.

0: digital trigger disabled.
1: digital trigger enabled.

edge selects which edge of the digital trigger input signal generates a trigger.

0: trigger on falling edge.
1: trigger on rising edge.

edge is ignored unless **digitalTrigger**is 1.

Because the NB-A2100 and NB-A2150 always trigger on the falling edge, **edge** must be 0 for these boards.

analogTrigger enables or disables the analog trigger circuitry on the NB-A2000 or NB-A2150 or the software trigger capability on the NB-A2100.

0: analog trigger disabled.
1: analog trigger enabled.

All of the remaining parameters, described as follows, are ignored if **analogTrigger** is 0.

triggerCount is the number of analog trigger occurrences to wait for before storing data.

Range: On the NB-A2000 and NB-A2150, the analog trigger is recognized through the hardware, and this value must be 1.
Range: On the NB-A2100, the analog trigger is recognized through the software, and this value must be greater than or equal to 1.

source selects the analog trigger source as follows:

For the NB-A2000:
0 = analog input channel 0.
1 = analog input channel 1.
2 = analog input channel 2.
3 = analog input channel 3.
4 = external analog trigger input ATRIG.

For the NB-A2100:
0 = analog input channel 0.

1 = analog input channel 1.
For the NB-A2150:
0 = analog input channel 0.
1 = analog input channel 1.
2 = analog input channel 2.
3 = analog input channel 3.

triggerSlope selects which slope condition at the selected analog trigger input generates a trigger.
0: trigger on negative slope.
1: trigger on positive slope.

triggerLevelis the code for the input value of the selected analog trigger signal that generates a trigger.
Range: -2,048 to 2,047 on the NB-A2000 which corresponds to a ±5.12 V analog trigger range in 2.5 mV steps. For example, **atrig_level** = 800 corresponds to a voltage trigger level of +2 V.
Range: -32,768 to 32,767 on the NB-A2100 and NB-A2150 which corresponds to a ±2.828 V analog trigger range.

hysteresisWindow is the number of digitizing levels below **triggerLevel**(for positive slope) or above **triggerLevel**(for negative slope) that the input signal must go before a valid trigger crossing at **triggerLevel**is recognized.
Range: The NB-A2000 does not provide a hysteresis analog trigger, and **hysteresisWindow** must be 0 on the NB-A2000.
Range: 0 to 65,535 on the NB-A2100 and NB-A2150. However, the following restrictions apply:

- If positive slope is selected through **triggerSlope**, **triggerLevel**- **hysteresisWindow** ≥ -32,768.
- If negative slope is selected through **triggerSlope**, **triggerLevel**+ **hysteresisWindow** ≤ 32,767.

timeout is the number of clock ticks (1/60 s) to wait for the specified number of analog triggers to occur after data acquisition starts. **timeout** is ignored for the NB-A2000 and NB-A2150 because these boards implement the analog trigger in hardware and wait indefinitely for the trigger to occur. There are two special cases for **timeout** on the NB-A2100:
-1: wait indefinitely.
0: stop immediately if analog trigger conditions have not been met.

`MDAQ_Trig_Config` *must* be called to configure trigger circuitry for hardware triggered acquisition. For pretrigger data acquisition, a hardware trigger *must* be selected. If neither analog nor digital trigger is enabled, the driver starts acquisition immediately by a software trigger when `MDAQ_Start` is called. Thus, only posttrigger data is acquired. On the NB-A2000, when analog and digital triggers are both enabled, the first trigger conditions met are recognized as the trigger and the other trigger conditions are ignored within each frame. See *Configuring the Trigger Conditions* earlier in this chapter for a description of the valid trigger modes on the NB-A2100 and NB-A2150. Triggering can also be controlled from the RTSI bus (see Chapter 9, *RTSI Bus Trigger Functions*).

The NB-A2100 and NB-A2150 provide hysteresis when looking at an analog trigger. Hysteresis acts like a *noise filter*, ensuring that small variations in the input signal are not mistaken for actual triggers. The amount of variation that is to be ignored is selected by **hysteresisWindow**. For example, if a positive slope has been selected, **triggerLevel**is 2,048, and **hysteresisWindow** is 256, the triggering scheme first looks for a sample that is at 1,792 or below–that is, 256 below **atrig_level**. After finding this sample, the triggering scheme starts looking for a sample that is at 2,048 (the **atrig_level**) or above, and when the triggering scheme finds this sample, it triggers. If a negative slope has been selected, the triggering scheme first looks for a sample that is **hysteresisWindow** digitizing levels above the **triggerLevel**and then for a sample that is at or below the **triggerLevel**.

On the NB-A2100, analog triggering is implemented in software and all 16 bits of the triggering levels are used for triggering. On the NB-A2150, analog triggering is implemented in hardware and only the upper eight bits of the hexadecimal value of **triggerLevel**and the upper eight bits of the hexadecimal value obtained by adding and subtracting **hysteresisWindow** to or from **triggerLevel**, are used for triggering. For example, values 2,048 (800 hex) and 2,303 (8ff hex) for **triggerLevel**will both be treated as 2,048 on the NB-A2150.

MDAQ_Trig_Delay

Function

Selects the time to delay after a trigger is received before acquiring data. (Posttrigger mode only)

Synopsis

C Syntax	`locus i32 MDAQ_Trig_Delay(u32 deviceNumber, u32 delayInterval, i32 timebase);`
Pascal Syntax	`function MDAQ_Trig_Delay(deviceNumber : i32; delayInterval : i32; timebase : i32) : i32;`
BASIC Syntax	`FN MDAQ_Trig_Delay(deviceNumber&, delayInterval&, timebase&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

delayInterval is the time to wait after the trigger before acquiring data in posttrigger mode. On the NB-A2000, setting **delayInterval** to 0 causes data to be acquired immediately after the trigger occurs. Setting **delayInterval** to 65,535 and **timebase** to 5 (10-ms timebase) causes the acquisition to start 655.35 s, or 10.9 min, after the trigger (this is the longest possible delay).

Range: 0, 3 through 65,535

timebase is the resolution to be used for the delay counter. For the NB-A2000, **timebase** has the following possible values:

-1: 200 ns.
0: reserved.
1: 1 µs.
2: 10 µs.
3: 100 µs.
4: 1 ms.
5: 10 ms.

On the NB-A2100 and NB-A2150, timebase must be set to 0. The sampling rate timebase set in `MDAQ_ScanRate` is also used as the delay counter timebase. So, if **delayInterval** is set to 64,000, the **timebase** frequency is set to 32 kHz, and scan interval is set to 1, then the acquisition would start

$$\frac{64{,}000}{(32{,}000/1)}$$

or 2 s after the trigger. At system startup or after a `Board_Reset`, the frequency corresponding to **timebase** 1 and **scanInterval** 1 is selected for the A/D sampling rate.

timebase is ignored if **delayInterval** is 0. The NB-A2000, NB-A2100, and NB-A2150 default to no posttrigger delay.

If NI-DAQ for Macintosh has been configured to acquire pretrigger data, then a delay after triggering before beginning to sample is meaningless. In this case, the delay is not accepted and a trigDelayIgnoredErr is returned as a warning.

If NI-DAQ for Macintosh has been configured to acquire only posttrigger data, then a delay can be set. If `MDAQ_Setup` is called to setup an acquisition to acquire pretrigger data as well, then the delay is reset to zero and an error is given.

Chapter 7
SCXI Functions

This chapter describes functions used to configure and communicate with SCXI modules and chassis.

SCXI hardware can condition analog input signals, isolate analog and digital I/O signals, and multiplex channels to increase the number of analog and digital signals that a plug-in DAQ board in your computer can process. An SCXI system consists of the following components, as shown in the Figure 7-1:

- Plug-in DAQ boards in your computer
- External SCXI chassis that house plug-in SCXI modules. You connect one or more modules to the plug-in DAQ boards using SCXI ribbon cable assemblies.
- SCXI terminal blocks that you use to connect signals to the SCXI modules

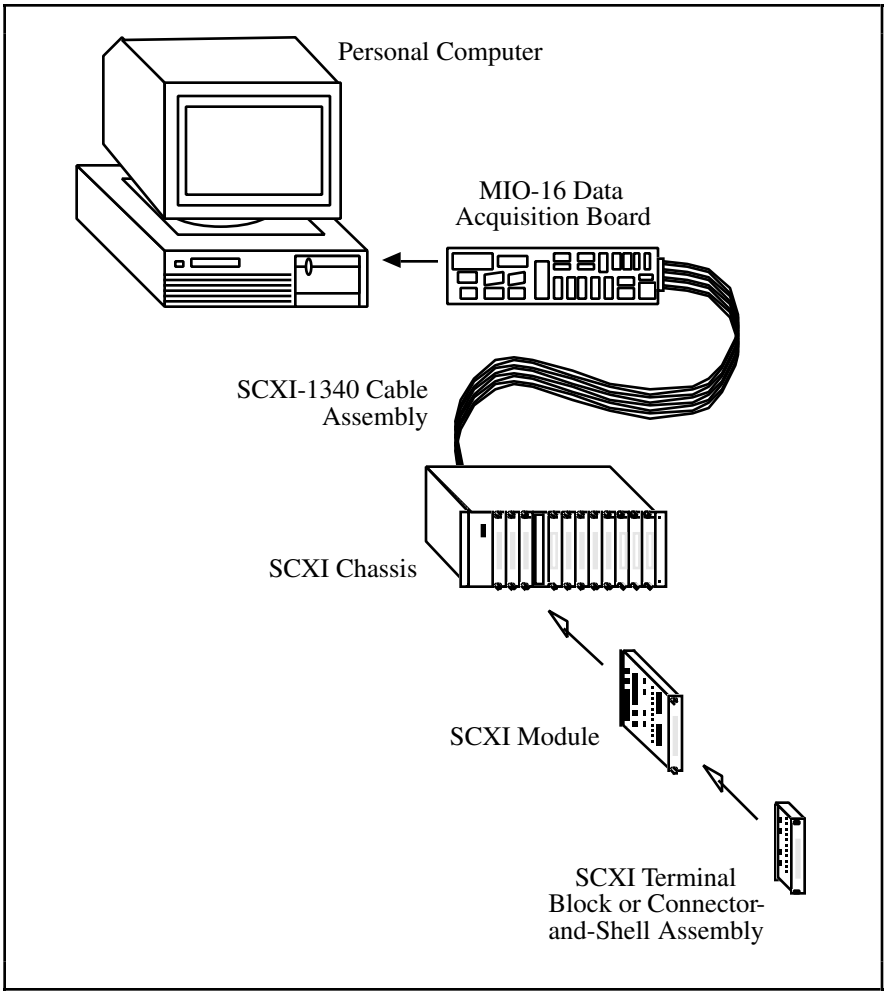

Figure 7-1. The SCXI System

NI-DAQ for Macintosh works with the following SCXI chassis and modules:

- SCXI-1000 4-slot chassis
- SCXI-1001 12-slot chassis
- SCXI-1100 32-channel multiplexer amplifier module
- SCXI-1102 32-channel multiplexer amplifier module for thermocouples
- SCXI-1120 8-channel isolation amplifier module
- SCXI-1121 4-channel isolation amplifier with excitation module
- SCXI-1122 16-channel transducer multiplexer module
- SCXI-1124 6-channel isolated analog output module
- SCXI-1140 8-channel simultaneously sampling differential amplifier module
- SCXI-1141 8-channel analog input module with programmable gains and filters
- SCXI-1160 16-channel electromechanical SPDT relay module
- SCXI-1161 8-channel electromechanical SPDT relay module
- SCXI-1162 32-channel optically isolated digital input module
- SCXI-1162HV 32-channel high-voltage optically isolated digital input module
- SCXI-1163 32-channel optically isolated digital output module
- SCXI-1163R 32-channel optically isolated solid-state relay module

You can use the following DAQ boards in combination with SCXI hardware and NI-DAQ for Macintosh:

- All DIO boards
- All MIO boards
- All Lab and 1200 series boards
- DAQCard-700

Please refer to the *SCXI Modules and Compatible Data Acquisition Boards* section later in this chapter for information about the functionality of each DAQ board with each type of SCXI module.

SCXI Installation and Configuration

To install your SCXI system, follow the instructions in the *Installing Your SCXI Hardware* section in Chapter 1, *Getting Started*. After you assemble your SCXI system, you must run the `NI-DAQ` Control Panel to enter your SCXI configuration; NI-DAQ needs the configuration information to program your SCXI system correctly. The *SCXI Configuration* section in Chapter 1 contains detailed instructions for entering your SCXI configuration using the `NI-DAQ` Control Panel.

Using SCXI Modules with the NI-DAQ Functions

For analog input operations, use the SCXI functions for configuration and setup. After you configure the SCXI system, you can use the Analog Input functions in Chapter 3, *Analog Input Functions*, and the Data Acquisition functions in Chapter 6, *Data Acquisition Functions*, to actually acquire the data on the DAQ board.

For digital I/O operations, you can use the SCXI functions for configuration, setup, and to perform the digital operations themselves. You can use the Digital I/O functions in Chapter 5, *Digital I/O Functions*, with those modules that operate in Parallel mode.

For SCXI analog output modules, you must use the SCXI functions to generate output voltages at the SCXI module.

The *SCXI Applications* section later in this chapter contains flowcharts and explanations of how to use SCXI modules with NI-DAQ.

SCXI Operating Modes

The way that DAQ boards have access to the signals from the modules depends on the operating modes of the modules. There are two basic operating modes for SCXI modules–Multiplexed and Parallel. The operating mode is a parameter that you enter in the configuration utility.

Multiplexed Mode for Analog Input Modules

When an analog input module operates in Multiplexed mode, all of its input channels are multiplexed to one module output. When you cable a DAQ board to a multiplexed analog input module, the DAQ board has access to that module's multiplexed output, as well as the outputs of all other multiplexed modules in the chassis via the SCXIbus. The SCXI functions route the multiplexed analog signals on the SCXIbus for you transparently. So, if you operate all modules in the chassis in Multiplexed mode, you only need to cable one of the modules directly to the DAQ board.

If you use an MIO DAQ board, a PCI-1200, or a DAQCard-1200 you can multiplex all the analog input channels in the SCXI chassis to one onboard channel *dynamically during a timed acquisition*. The SCXI functions program the chassis with a module scan list that dynamically controls which module sends its output to the SCXIbus during a scan. You can specify that the modules be scanned in any order and specify an arbitrary number of channels for each module; however, the channels on each module must be scanned in consecutive, ascending order.

Note: ***The DAQCard-700 and Lab series boards support only single-channel acquisitions in Multiplexed mode.***

By default, when you cable a DAQ board to a multiplexed module, the multiplexed output of the module (and all other multiplexed modules in the chassis) appears at analog input channel 0 of the DAQ board.

You can use more than one SCXI chassis with one MIO board if the modules operate in Multiplexed mode. You must use one SCXI-1350 multichassis adapter for each additional chassis (refer to your SCXI module user manuals). You should also enter a unique jumper-selected address in the configuration utility for each chassis. The multichassis adapter scheme sends the output of the module in the first chassis to analog input channel 0 of the DAQ board, the output of the module in the second chassis to analog input channel 1 of the DAQ board, and so on. When you want to acquire data from the additional chassis, you must specify the correct onboard DAQ board channel in the MIO channel scan list that is passed to the `SCAN` functions.

Multiplexed Mode for Digital and Relay Modules

Multiplexed mode is referred to as *Serial mode* in the digital and relay module hardware manuals. When you operate your digital or relay module in Multiplexed (or Serial) mode, NI-DAQ communicates the module channel states serially over the SCXIbus backplane. The SCXI-1162, SCXI-1162HV, SCXI-1163, and SCXI-1163R

modules have jumpers that you must set correctly for the module to operate in Multiplexed (or Serial) mode. Because NI-DAQ can communicate with the multiplexed modules over the SCXIbus backplane, you only need to cable one multiplexed module in each chassis directly to the DAQ board in the computer.

Multiplexed Mode for Analog Output Modules

The SCXI-1124 analog output module supports only Multiplexed mode (or Serial mode). This means that NI-DAQ sets the analog output channel states by communicating serially over the SCXIbus. Because NI-DAQ can communicate with the multiplexed modules over the SCXIbus backplane, you only need to cable one multiplexed module in each chassis directly to the DAQ board in the computer.

Parallel Mode for Analog Input Modules

When an analog input module operates in Parallel mode, it sends each of its input channels directly to a separate analog input channel of the DAQ board cabled to the module. You *cannot* multiplex parallel outputs of a module on the SCXIbus; only a DAQ board that you cable directly to a module in Parallel mode has access to its input channels. In this configuration, the total number of analog input channels is limited to the number of channels available on the DAQ board. In some cases, however, you can cable more than one DAQ board to modules in an SCXI chassis. For example, you can use two Lab-NB boards and cable each one to a separate SCXI-1120 module in the chassis operating in Parallel mode. You must be sure to enter the correct device numbers in the **Cabled Device** field of the configuration utility for each module you operate in Parallel mode.

By default, when a module operates in Parallel mode, the module sends its channel 0 output to analog input channel 0 of the DAQ board, the channel 1 output to analog input channel 1 of the DAQ board, and so on.

Parallel Mode for Digital Modules

When you operate a digital module in Parallel mode, the digital lines on your DAQ board directly drive the individual digital channels on your SCXI module. You must cable a DAQ board directly to every module that you operate in Parallel mode. The SCXI-1162, SCXI-1162HV, SCXI-1163, and SCXI-1163 R modules have jumpers that you must set correctly for the module to operate in Parallel mode.

You may wish to use Parallel mode instead of Multiplexed mode for faster updating or reading of the SCXI digital channels. For the fastest performance in Parallel mode, you can use the Digital I/O functions in Chapter 5 with the appropriate onboard port numbers instead of using the SCXI functions. Refer to the *SCXI Modules and Compatible Data Acquisition Boards* section later in this chapter for information about which digital ports on each DAQ board are actually used in Parallel mode.

Note: ***A DAQ board that is cabled to an SCXI-1162, SCXI-1162HV, SCXI-1163, or SCXI-1163R in Parallel mode cannot be the communication path in the configuration utility.***

If you are using a DIO-96, you can also operate a digital module in Parallel mode using the digital ports on the second half of the ribbon cable (pins 51 to 100). So, the DIO-96 can operate two digital modules in Parallel mode–one module using the first half of the ribbon cable (pins 1 to 50) and another module using the second half of the ribbon cable (pins 51 to 100). Set the **operating mode** in the configuration utility to **Parallel** (**secondary**) for the module that will be using the second half of the ribbon cable.

SCXI Modules and Compatible Data Acquisition Boards

The capabilities and limitations described in this section should help you determine how your hardware components can work together in your application, and help you determine the best SCXI configuration for your application. Please refer to your SCXI module and chassis user manuals and DAQ board user manuals for detailed information about the capabilities and limitations of your hardware.

The SCXI-1100

The SCXI-1100 module has 32 differential analog input channels. The input voltage range is -10 to +10 V at a gain of 1. The SCXI-1100 has a software-selectable gain with values of 1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, and 2,000. You use the `SCXI_Set_Gain` function to program the module gain. NI-DAQ does *not* use the gain menu in the `NI-DAQ` Control Panel; LabVIEW only uses that menu.

The SCXI-1100 also has a software-selectable calibration mode that you can use to determine the zero offset of the module (see the `SCXI_Calibrate_Setup` function description).

The SCXI-1300 and SCXI-1303 terminal blocks that can be used with the SCXI-1100 module each have an onboard temperature sensor that is jumper-configurable to be either multiplexed with the other input channels (MTEMP), or to be sent directly to a different DAQ board channel (DTEMP). In the MTEMP configuration, you can select to read the temperature sensor using the `SCXI_Single_Chan_Setup` function; in the DTEMP configuration, the temperature sensor output will appear on DAQ board channel 1. If you multiply the voltage read from the temperature sensor on the SCXI-1300 by 100, you will get the temperature in degrees Celsius. The temperature sensor on the SCXI-1303 is a thermistor; you must use the thermistor conversion routine described in Appendix D.

Please refer to the *SCXI-1100 User Manual* for more information on the hardware-selectable signal conditioning features on the module.

The SCXI-1100 supports only the Multiplexed operating mode; it does *not* support Parallel mode.

You can cable an MIO board directly to an SCXI-1100 module using the SCXI-1340 cable. You must use the SCXI-1341 cable to connect the SCXI-1100 to a Lab or 1200 series board. Use an SCXI-1342 cable with the DAQCard-700.

The SCXI-1102

The SCXI-1102 has 32 differential analog input channels and one cold-junction sensor channel (CJSTEMP) that is selectable through the `SCXI_Single_Chan_Setup` function. When you use the module with an SCXI-1300 or SCXI-1303 terminal block, the terminal block temperature sensor connects to CJSTEMP. The module can multiplex CJSTEMP with the other 32 input channels during a hardware-controlled scan. On each channel, including CJSTEMP, the SCXI-1102 has a 3-pole low-pass filter to reject 60 Hz noise. Each of the 32 differential analog input channels (but not CJSTEMP) also has an amplifier with a selectable gain of 1 or 100, selected through the `SCXI_Set_Gain` function. The amplification and filtering occur before multiplexing.

When you change the gain on a channel, the output will take several seconds to settle. The module contains a Status Register to indicate that the output is in the process of settling, and this information is available to applications through the `SCXI_Get_Status` function.

The SCXI-1102 supports only Multiplexed operating mode; it does not support Parallel mode.

The SCXI-1120 and the SCXI-1121

The SCXI-1120 and SCXI-1121 are 8-channel and 4-channel isolation modules, respectively. The input voltage range on both modules is -5 V to +5 V. The modules have a hardware-selectable gain on each input channel with values of 1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, and 2,000. NI-DAQ does *not* use the gain menu in the `NI-DAQ` Control Panel; LabVIEW only uses that menu. Refer to the `SCXI_Scale` function to compensate for SCXI-1120 and SCXI-1121 gain.

The SCXI-1121 also has four excitation channels that can be used for voltage or current excitation.

The SCXI-1320 and SCXI-1328 terminal blocks that you can use with the SCXI-1120 and SCXI-1121 modules each have an onboard temperature sensor that is jumper configurable to be either multiplexed along with the other

input channels in Multiplexed mode (MTEMP), or to be sent directly to another DAQ board channel (DTEMP). In the MTEMP configuration, you can select to read the temperature sensor using the `SCXI_Single_Chan_Setup` function; in the DTEMP configuration, the temperature sensor output will appear on DAQ board channel 15 for the SCXI-1120 and channel 4 for the SCXI-1121. Notice that the DAQ board must be in Pseudodifferential mode to read the temperature sensor in DTEMP mode with the SCXI-1120. If you multiply the voltage read from the temperature sensor on the SCXI-1320 by 100, you will get the temperature in degrees Celsius. The temperature sensor on the SCXI-1328 is a thermistor; you must use the thermistor conversion routine described in Appendix D.

The SCXI-1321 terminal block that can be used with the SCXI-1121 module has shunt resistors that can be enabled by using the `SCXI_Calibrate_Setup` function. The SCXI-1327 terminal block can divide the input signals applied to the SCXI-1120 or SCXI-1121 by 100. See the `SCXI_Scale` function to compensate for this attenuation.

Please refer to the SCXI-1120 and SCXI-1121 user manuals for information on the hardware-selectable signal conditioning features available on the modules.

The SCXI-1120 and the SCXI-1121 modules support both Multiplexed and Parallel operating modes.

You can cable an MIO board directly to an SCXI-1120 or SCXI-1121 module using the SCXI-1340 cable. You must use the SCXI-1341 cable to connect the SCXI-1120 or SCXI-1121 to a Lab or 1200 series board. Use an SCXI-1342 cable with the DAQCard-700.

The SCXI-1122

The SCXI-1122 has 16 differential analog input channels. The input voltage range is -5 to +5 V at a gain of 1. The SCXI-1122 has a software-selectable gain that applies to all channels on the module; use the `SCXI_Set_Gain` function to program the module gain. NI-DAQ does *not* use the gain menu in the `NI-DAQ` Control Panel; LabVIEW 3.0 only uses that menu. This module also has a programmable lowpass filter with cut-off frequencies of 4 Hz and 4 kHz. Use the `SCXI_Set_Filter` function to select the filter setting.

The SCXI-1122 supports Multiplexed mode only; it does not support Parallel mode.

The SCXI-1322 terminal block that you can use with the SCXI-1122 has an onboard thermistor that you can use to do cold-junction compensation for temperature readings. You can use the thermistor conversion routine described in Appendix D to convert the thermistor voltage to temperature.

You can configure the SCXI-1122 for four-wire scanning mode, which means that the module will switch the current excitation source to drive one of the channels 8 through 15 as an excitation output channel whenever the corresponding input channel 0 through 7 is selected. In this mode the module has eight analog input channels and eight corresponding current excitation channels. See the `SCXI_Set_Input_Mode` function description.

The SCXI-1122 uses relays to switch the input channels; these relays require 10 ms to switch. As a result, you cannot use a sampling rate greater than 100 Hz in a channel-scanning operation. In addition, the relays have a finite lifetime. If you plan to take many samples from each channel and average them to eliminate noise, you should use the single-channel or software scanning applications described later in the chapter in Figure 7-3. This means you should select one channel on the module, acquire many samples from that channel, then select the next channel, and so on. You should not use the channel-scanning method shown in Figure 7-5 if you want to take many samples from each channel and average them.

The SCXI-1122 has an onboard EEPROM which contains a set of factory calibration constants for the amplifier on the module. NI-DAQ automatically reads these constants and uses them in the `SCXI_Scale` function to compensate for amplifier gain and offset errors when scaling binary data to voltage. You can also perform your own module calibration by taking readings and using the `SCXI_Cal_Constants` function to store your own calibration constants in the EEPROM.

The SCXI-1122 has two software-selectable calibration modes that you use the `SCXI_Calibrate_Setup` function to select. You can ground the module amplifier inputs so that you can read the amplifier offset. You can

also switch a shunt resistor across your bridge circuit to test your circuit (refer to the *SCXI-1122 User Manual* for more information about the shunt resistor).

The SCXI-1124

The SCXI-1124 is a six-channel analog output module capable of generating voltages between -10 and +10 volts or currents between 0 and 20 mA. The SCXI-1124 has six independent 12-bit DACs. Each DAC channel has a software selectable voltage or current output range. Use the `SCXI_AO_Write` function to set the output range and write voltages, currents, or binary values to the DACs. The SCXI-1124 is designed for single-point output operations; only one channel can be written to at a time. The SCXI-1124 is not intended for use in waveform generation.

The SCXI-1124 has an onboard EEPROM that contains a set of factory-calibration constants for each DAC. NI-DAQ automatically loads these constants so that the `SCXI_AO_Write` function can compute the 12-bit binary pattern needed to produce your desired voltage at the output. You can also compute your own calibration constants by writing binary values to the DACs, measuring the output voltage with a voltmeter, and using the `SCXI_Cal_Constants` function to calculate and store the constants in the module EEPROM.

The SCXI-1124 supports Multiplexed mode only. You can cable an MIO board, DAQCard-700, or Lab or 1200 series board to the SCXI-1124, in which case you should set the jumpers on the module for MIO operation. You can cable a DIO board to the SCXI-1124, in which case you should set the jumpers on the module for DIO operation. If there is another module in the chassis cabled to a DAQ board in Multiplexed mode, you do not need to cable the SCXI-1124 to anything; NI-DAQ will communicate with the module using the SCXIbus backplane. In this case, the MIO/DIO jumpers on the module are irrelevant. If you plan to use analog input SCXI modules in addition to the SCXI-1124, you should cable one of the analog input modules to the DAQ board.

The SCXI-1140

The SCXI-1140 is an 8-channel simultaneously sampling differential amplifier module. The input voltage range of the module is -10 to +10 V. It has a hardware-selectable gain on each input channel with values of 1, 10, 100, 200, and 500. NI-DAQ does *not* use the gain menu in the `NI-DAQ` Control Panel; LabVIEW only uses that menu. Refer to the `SCXI_Scale` function to compensate for SCXI-1140 gain.

This module supports both Multiplexed mode and Parallel mode.

The SCXI-1140 will simultaneously sample all the input signals and hold those values while the DAQ board reads the desired channels one by one. When the module is holding the input channel values, it is in *Hold mode*; when it comes out of Hold mode so that it can sense the new values on the input channels, it is in *Track mode*. There is a control signal on the module that will determine when the module is in Track mode, and when the module will go into Hold mode. This signal is derived either from a counter/timer output on the DAQ board, from an external source connected to a pin on the front connector of the module, or from a trigger line on the SCXIbus.

The SCXI-1140 Track/Hold setup is software configurable for single-channel operations or for interval-scanning operations. During single-channel operations, an SCXI function call can put the module into Hold mode before `AI` functions are used to acquire the data, and put it back into Track mode to sense new input values. During interval-scanning operations, the scan interval timer will cause the module to go into Hold mode at the beginning of each scan and go back into Track mode at the end of each scan. Effectively, the input channels of the SCXI-1140 are *simultaneously sampled* at the beginning of each scan. The scan interval timer can either be a counter on the DAQ board or an external source connected to the front connector of the module. In addition, multiple SCXI-1140 modules can be synchronized by using the SCXIbus so that all SCXI-1140 modules will go into Hold mode at the same time. If multiple SCXI-1140 modules are being scanned in Multiplexed mode along with other types of SCXI modules, the module that is cabled to the DAQ board must be an SCXI-1140 module for the Track/Hold control signals to be properly routed and synchronized.

Note: ***Because the SCXI-1140 uses the scan interval timer of the DAQ board to control the state of the module during scanning, only DAQ boards that support interval scanning will support channel scanning on the***

SCXI-1140. The DAQCard-700 and Lab and 1200 series do not support interval scanning, and therefore do not support timed channel scanning on the SCXI-1140 regardless of the operating mode. The NB-MIO-16 can support interval scanning if used with an SCXI-1140 module. However, all of the DAQ boards support single-channel operations using the SCXI-1140. Please refer to** SCXI Applications **later in this chapter for more information on building applications with the SCXI-1140 module.

It is important to be aware of the Track/Hold timing requirements of the SCXI-1140. For accurate data, the module must be in Track mode for at least 7 µs before going into Hold mode. During an interval-scanning operation, this means that the scan interval should be at least 7 µs longer than the total sample interval. After the module is in Hold mode, the latched data at the input channels will droop at a rate of 10 mV/s, so you must be careful to sample all the desired channels relatively quickly after putting the module into Hold mode.

The SCXI-1141

The SCXI-1141 is an 8-channel analog input module with programmable gains and filters. The input range of the module is -5 to +5 V. It has programmable gains on each channel of 1, 2, 5, 10, 20, 50, and 100; use the `SCXI_Set_Gain` function to program the gain on a per-channel basis. The filters have a programmable cutoff frequency from 10 Hz to 25 kHz, or this frequency can be derived from an external clock; use the `SCXI_Configure_Filter` function to select the filter settings on a per-module basis or to bypass any of the filters on a per-channel basis. The SCXI-1141 supports both Multiplexed and Parallel modes.

The SCXI-1141 has a software selectable calibration mode that you can select with the `SCXI_Calibrate_Setup` function. You can ground each input of each amplifier so that you can read the amplifier offsets. The SCXI-1141 also has an onboard EEPROM that contains a set of factory gain adjustment calibration constants for each amplifier on the module. NI-DAQ automatically reads these constants and uses them in the `SCXI_Scale` function to compensate for amplifier gain errors when scaling binary data to voltage data. You can also perform your own amplifier calibration by taking readings and using the `SCXI_Cal_Constants` function to store your own calibration constants in the EEPROM.

The SCXI-1304 terminal block provides either AC or DC coupling of input signals. This terminal block also provides a ground reference for floating signals.

The SCXI-1160 and the SCXI-1161

The SCXI-1160 is a 16-channel electromechanical single-pole double-throw (SPDT), also referred to as Form C, relay module with 16 independent SPDT relays. The relays are latched–that is, the module powers up with the relays in the position in which they were left at power down. Each relay can be set or reset without affecting the other relays, or all relays can change state at the same time.

The SCXI-1161 is an 8-channel electromechanical SPDT relay module with eight independent SPDT relays. The relays are nonlatched, and the module powers up with the relays in the default closed position. Each relay can be set or reset without affecting the other relays, or all relays can change state at the same time.

Use the SCXI functions `SCXI_Get_State`, `SCXI_Set_State`, and `SCXI_Get_Status` to control the relay modules. Call the `SCXI_Reset` function to initialize all the relays to the default closed position. For more information on these functions, refer to the section entitled *SCXI Function Summary*.

The SCXI-1160 and SCXI-1161 modules only support Multiplexed (or Serial) mode. If you cable an MIO board to these modules using the SCXI-1340 cable, a Lab or 1200 series board using the SCXI-1341 cable, or a DAQCard-700 using the SCXI-1342 cable, you must set the module jumpers to the MIO position. If you cable a DIO board to the module, you must set the jumpers to the DIO position. If there is another module in the chassis cabled to a DAQ board in Multiplexed mode, you do not need to cable the SCXI-1160 or SCXI-1161 to anything; NI-DAQ will communicate with the module using the SCXIbus backplane. In this case, the MIO/DIO jumpers on the module are irrelevant. If you plan to use analog input SCXI modules in addition to the SCXI-1160 or SCXI-1161, you should cable one of the analog input modules to the DAQ board.

The SCXI-1162 and the SCXI-1162HV

The SCXI-1162 and SCXI-1162HV are 32-channel optically isolated digital input modules. They accept 32 input signals from external equipment and condition the signals for input to a DAQ board while maintaining optical isolation from the host computer. The SCXI-1162 accepts 0 to +5 V digital signals, and the SCXI-1162HV senses AC or DC signals up to 250 V.

You can call the `SCXI_Get_State` function to read the logical states of the digital input lines on the module.

The SCXI-1162 and SCXI-1162HV modules support both Multiplexed (or *Serial*) mode and Parallel mode. You must set jumpers on the modules correctly for Multiplexed or Parallel mode. If you cable an MIO board to these modules using the SCXI-1340 cable, a Lab or 1200 series board using the SCXI-1341 cable, or a DAQCard-700 using the SCXI-1342 cable, you must set the module jumpers to the MIO position. If you cable a DIO board to these modules, you must set the jumpers to the DIO position.

The SCXI-1163 and the SCXI-1163R

The SCXI-1163 and the SCXI-1163R are 32-channel optically isolated digital output modules. The SCXI-1163 makes available to external equipment up to 32 digital outputs from a DAQ board while maintaining optical isolation from the host computer and eliminating ground-loop problems. The SCXI-1163R is functionally equivalent to the SCXI-1163 but incorporates solid-state relays in place of the digital outputs. You can open or close each relay independently.

You can call the `SCXI_Set_State` function to control the digital output lines or relays of the modules. You can call the `SCXI_Get_State` function to obtain the current states of the modules. It is important to remember that `SCXI_Get_State` makes a hardware read only if the module is jumper-configured and operating in Parallel mode. When operated in Serial mode, the driver retains the states of the digital output lines in memory. Consequently, a hardware write must take place before you can obtain the states on the module.

The modules power up with digital output lines in a high state or relays open. Calling `SCXI_Reset` also sets all the digital output lines to a high state.

The SCXI-1163 and SCXI-1163R modules support both Multiplexed (or *Serial*) mode and Parallel mode. You must set jumpers on the module correctly for Multiplexed or Parallel mode. If you cable an MIO board to these modules using the SCXI-1340 cable, a Lab or 1200 series board using the SCXI-1341 cable, or a DAQCard-700 using the SCXI-1341 cable, you must set the module jumpers to the MIO position. If you cable a DIO board to these modules, you must set the jumpers to the DIO position.

The MIO Boards

The MIO DAQ boards support the following analog input functionality when using the SCXI analog input modules:

- Single analog input (using the `AI` class of functions described in Chapter 3, *Analog Input Functions*)
- Single-channel data acquisition (using the `DAQ` class of functions described in Chapter 6, *Data Acquisition Functions*)
- Multiple-channel and interval scanning (using the `SCAN` class of functions described in Chapter 6, *Data Acquisition Functions*)

The NB-MIO-16 does not support interval channel scanning *unless* it is being used in an SCXI scan that includes an SCXI-1140 module. If an `SCXI_SCAN_Setup` call has been made for an NB-MIO-16 with a module scan list that includes an SCXI-1140, then the `SCAN_IntStart` call can be used to start an interval-scanning operation on the NB-MIO-16. In this special case, the sample timebase and scan timebase specified must be the same. The double-buffered data acquisition functions (the `DAQ2` class of functions) can also be used.

You can also use the analog output modules and digital modules in Multiplexed mode. If you are using analog input modules with digital and/or analog output modules, you must cable the MIO board to an analog input module.

It is important to remember that when a DAQ board is cabled to an SCXI module, some of the DAQ board I/O connector pins and therefore some of the board resources will be reserved for SCXI use. The following MIO resources are reserved by SCXI:

- Digital I/O lines ADIO0 through ADIO2 are used as output lines to the SCXI; ADIO3 is available for general use as a digital output line. Digital line BDIO0 is used as an input line from the SCXI; the remaining lines of Digital Port B are available for general use as input.
- When an SCXI-1140 module is used, counter 2 is used to control the Track/Hold state of the module. When the module is set up for a single-channel or an interval-channel scanning operation, counter 2 is reserved. Refer to the `SCXI_Track_Hold_Setup` function for more information.
- The SCXI-1100, SCXI-1102, and SCXI-1122 modules will drive analog input channel 0; if the SCXI terminal block is used with the temperature sensor in the DTEMP configuration, analog input channel 1 will also be driven.
- The SCXI-1120 module will drive analog input channels 0 through 7, *even if the module is to be operated in Multiplexed mode*. In addition, if the temperature sensor on the terminal block is configured for DTEMP mode, analog input channel 15 will also be driven. Notice that the DAQ board must be operated in Pseudodifferential mode to read the temperature sensor in the DTEMP configuration.
- The SCXI-1121 module will drive analog input channels 0 through 3, *even if the module is to be operated in Multiplexed mode*. In addition, if the temperature sensor on the terminal block is in the DTEMP configuration, analog input channel 4 will also be driven.
- The SCXI-1140 module will drive analog input channels 0 through 7, *even if the module is to be operated in Multiplexed mode*.
- The SCXI-1141 module will drive analog input channels 0 through 7, *even if the module is to be operated in Multiplexed mode*.

The DIO-32F

The NB-DIO-32F digital I/O board can be cabled directly to an analog output or digital SCXI module. When a digital I/O board is cabled to an SCXI module configured for Multiplexed mode, some of the digital board I/O pins, and therefore some of the board resources, are reserved for SCXI use. SCXI reserves the following NB-DIO-32F resources when cabled to a digital or analog output module in Multiplexed mode:

- NB-DIO-32F digital I/O lines DIOB0 to DIOB3 are the output lines to the SCXI module. The remaining lines of this port are available for output only.
- NB-DIO-32F digital I/O line DIOA0 is the input line from the SCXI module. The remaining lines of this port are available for input only.

When you cable an NB-DIO-32F to an SCXI-1162, SCXI-1162HV, SCXI-1163, or SCXI-1163R in Parallel mode, the 32 digital lines are directly connected to the module's 32 digital channels. You can use the `DIG_In_Port` and `DIG_Out_Port` functions in Chapter 5 to access the SCXI channels in Parallel mode. You cannot cable an NB-DIO-32F to an analog input module.

The DIO-24 and the DIO-96

The NB-DIO-24, DAQCard-DIO-24, NB-DIO-96, and PCI-DIO-96 digital I/O boards can be cabled directly to analog output or digital modules. When a digital I/O board is cabled to an SCXI module configured for Multiplexed mode, some of the digital board I/O pins, and therefore some of the board resources, are reserved for SCXI use. The following NB-DIO-24, DAQCard-DIO-24, and NB-DIO-96 resources are reserved by SCXI when cabled to a digital or analog output module in Multiplexed mode:

- NB-DIO-24 and DAQCard-DIO-24 digital I/O lines PB0 to PB3 and the NB-DIO-96 and PCI-DIO-96 digital output lines APB0 to APB3 are the output lines to the SCXI module. The remaining lines of these ports are available for output only.

- NB-DIO-24 and DAQCard-DIO-24 digital I/O line PA0 and the NB-DIO-96 and PCI-DIO-96 digital I/O line APA0 are the input lines from the SCXI module. The remaining lines of these ports are available for input only.

When you cable a DIO-24 to an SCXI-1162, SCXI-1162HV, SCXI-1163, or SCXI-1163R in Parallel mode, the 24 digital lines are directly connected to the module's digital channels 0 to 23. When you cable a DIO-96 to an SCXI-1162, SCXI-1162HV, SCXI-1163, or SCXI-1163R in Parallel mode, the DIO-96 ports 0 to 3 are directly connected to the module's digital channels. When you cable a DIO-96 in Parallel (secondary) mode, DIO-96 ports 6 to 9 are directly connected to the module's digital channels. You can use the `DIG_In_Port` and `DIG_Out_Port` functions in Chapter 5, using the appropriate onboard ports to access the SCXI digital lines in Parallel mode. You cannot cable a DIO-24 or DIO-96 to an analog input module.

The DAQCard-700 and Lab and 1200 Series Boards

The DAQCard-700 and Lab and 1200 series boards support the following analog input functionality when using the SCXI analog input modules:

- Single analog input (using the `AI` class of functions described in Chapter 3)

- Single-channel data acquisition (using the `DAQ` class of functions described in Chapter 6)

- Continuous channel-scanning *on the SCXI-1120 and SCXI-1121 modules only, in Parallel mode only* (using the `Lab_ISCAN` class of functions described in Chapter 6)

The double-buffered data acquisition functions (the `DAQ2` class of functions) can also be used.

You can also use the analog output modules and digital modules in Multiplexed mode. If you are using analog input modules with digital and/or analog output modules, you must cable the Lab or 1200 series board to an analog input module.

It is important to remember that when a DAQ board is cabled to an SCXI module, some of the DAQ board I/O connector pins and therefore some of the board resources will be reserved for SCXI use. The following resources are reserved by SCXI:

- Lab and 1200 series digital I/O lines PB4 to PB7 are used as output communication lines to SCXI. DAQCard-700 digital output lines DOUT4 to DOUT7 are used as output communication lines to SCXI. The entire port is reserved by NI-DAQ.

- Lab and 1200 series digital I/O line PC1 is used as an input communication line to SCXI. DAQCard-700 digital input line DIN6 is used as an input communication line to SCXI. The remaining lines of these ports are available for input only.

- When you use an SCXI-1140 module, counter B1 of the Lab and 1200 series and counter 2 of the DAQCard-700 control the Track/Hold state of the module. When the module is not set up for an input operation, these counters are available for general use; otherwise, they are reserved. Refer to the `SCXI_Track_Hold_Setup` function description for more information.

- The SCXI-1100 module drives analog input channel 0 of the DAQ device; if you use the SCXI-1300 terminal block with the temperature sensor in the DTEMP configuration, the SCXI-1100 also drives analog input channel 1. The SCXI-1100 cannot read the temperature sensor in DTEMP mode.
- The SCXI-1102 module drives analog input channel 0 of the DAQ device.
- The SCXI-1120 module drives analog input channels 0 to 7, even if you are operating the module in Multiplexed mode. In addition, if the temperature sensor on the terminal block is in the DTEMP configuration, the SCXI-1120 also drives analog input channel 15.
- The SCXI-1121 module drives analog input channels 0 to 3, even if you are operating the module in Multiplexed mode. In addition, if the temperature sensor on the terminal block is in the DTEMP configuration, the SCXI-1121 also drives analog input channel 4.
- The SCXI-1122 module drives analog input channel 0 of the DAQ device; if you use the SCXI-1300 terminal block with the temperature sensor in the DTEMP configuration, the SCXI-1122 also drives analog input channel 1. The SCXI-1122 cannot read the temperature sensor in DTEMP mode.
- The SCXI-1140 module drives analog input channels 0 to 7, *even if you are operating the module in Multiplexed mode*.
- The SCXI-1141 module drives analog input channels 0 to 7, e*ven if you are operating the module in Multiplexed mode*.

SCXI Function Summary

`SCXI_AO_Write`	Sets the DAC channel on the SCXI-1124 module to the specified voltage or current output value. You can also use this function to write a binary value directly to the DAC channel, or to translate a voltage or current value to the corresponding binary value.
`SCXI_Cal_Constants`	Calculates calibration constants for the given channel and range or gain using measured voltage/binary pairs. You can use this function with any SCXI analog input or analog output module. The constants can be stored and retrieved from NI-DAQ memory or the module EEPROM (if your module has an EEPROM). The driver uses the calibration constants to more accurately scale analog input data when you use the `SCXI_Scale` function and output data when you use `SCXI_AO_Write`.
`SCXI_Calibrate_Setup`	Grounds the amplifier inputs of an SCXI-1100, SCXI-1122, or SCXI-1141 so that you can determine the amplifier offset. You can also use this function to switch a shunt resistor across your bridge circuit to test the circuit. Shunt calibration is supported for the SCXI-1122 and the SCXI-1121 with the SCXI-1321 terminal block.
`SCXI_Change_Chan`	Selects a new channel of a multiplexed module that has previously been set up for a single-channel analog input operation using the `SCXI_Single_Chan_Setup` function.
`SCXI_Configure_Filter`	Sets the specified channel to the given filter setting on any SCXI module that supports programmable filter settings. Currently, only the SCXI-1122 and SCXI-1141 has programmable filter settings; the other analog input modules have hardware-selectable filters.
`SCXI_Get_Chassis_Info`	Returns current chassis configuration information.
`SCXI_Get_Module_Info`	Returns current configuration information for a given chassis slot number.

`SCXI_Get_State`	Returns the state of a single channel or an entire port on any digital or relay module.
`SCXI_Get_Status`	Returns the data in the Status Register of the specified module. This function can be used with the SCXI-1122 or SCXI-1160 to determine if the relays have finished switching, with the SCXI-1124 module to determine if the DACs have settled, or with the SCXI-1102 to determine if the filters have settled after changing the gain.
`SCXI_Load_Config`	Loads the SCXI chassis configuration information that was established in the configuration utility.
`SCXI_MuxCtr_Setup`	Enables or disables counter 1 to be used as a mux counter during SCXI multiplexed channel scanning to synchronize the DAQ board scan list with the module scan list that has been downloaded to Slot 0 of the SCXI chassis.
`SCXI_Reset`	Resets a specified module to its default state, to reset the Slot 0 scanning circuitry, or to reset the entire chassis.
`SCXI_Scale`	Scales an array of binary data acquired from an SCXI channel to voltage. `SCXI_Scale` will use stored software calibration constants if applicable for the given module when it scales the data. The SCXI-1102, SCXI-1122 and SCXI-1141 have default software calibration constants loaded from the module EEPROM; all other analog input modules have no software calibration constants unless you follow the analog input calibration procedure outlined in the `SCXI_Cal_Constants` function description.
`SCXI_SCAN_Setup`	Sets up the SCXI chassis for a multiplexed scanning data acquisition operation to be performed by the given DAQ board. Modules may be scanned in any order; channels on each module must be scanned in consecutive order. A module scan list is downloaded to Slot 0 in the SCXI chassis that will determine the sequence of modules that will be scanned and how many channels on each module will be scanned. Each module is programmed with its given start channel.
`SCXI_Set_Config`	Allows you to change the configuration of the SCXI chassis that was established in the configuration utility.
`SCXI_Set_Gain`	Sets the specified channel to the given gain setting on any SCXI module that supports programmable gain settings. Currently, the SCXI-1100, SCXI-1102, SCXI-1122, and SCXI-1141 have programmable gains; the other analog input modules have hardware-selectable gains.
`SCXI_Set_Input_Mode`	Configures the SCXI-1122 channels for two-wire mode or four-wire mode.
`SCXI_Set_State`	Sets the state of a single channel or an entire port on any digital output or relay module.
`SCXI_Single_Chan_Setup`	Sets up a multiplexed module for a single channel analog input operation to be performed by the given DAQ board. Sets the module channel, enables the module output, and routes the module output on the SCXIbus if necessary. Resolves any contention on the SCXIbus by disabling the output of any module that was previously driving the SCXIbus. This function can also be used to read the temperature sensor on a terminal block connected to the front of a module.
`SCXI_Track_Hold_Control`	If an SCXI-1140 module has been configured for a single-channel operation, this function can be used to put the module into Track mode or Hold mode.

`SCXI_Track_Hold_Setup` Establishes the track/hold behavior of an SCXI-1140 module, and sets up the module for either a single-channel operation or an interval-scanning operation.

SCXI Applications

There are three categories of SCXI applications–analog input applications, analog output applications, and digital applications.

Figure 7-2 shows the basic structure of an SCXI application.

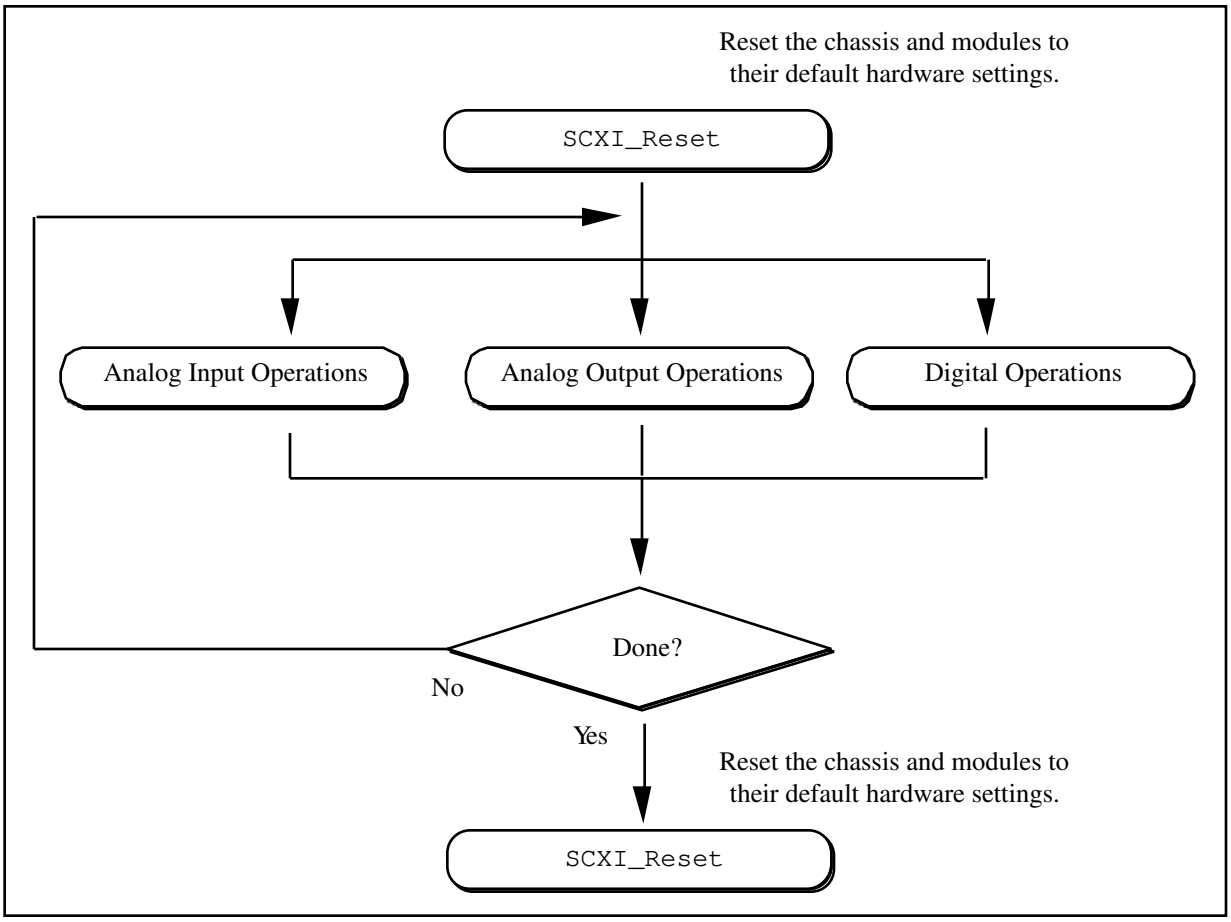

Figure 7-2. General SCXIbus Application

The figures in the following sections show the detailed call sequences for SCXI operations. In effect, each of the remaining flowcharts in the chapter is an enlargement of the Analog Input Operations, the Analog Output Operations, or the Digital Operations node in Figure 7-2. Please refer to the function descriptions later in the chapter for detailed information about each function used in the flowcharts.

Analog Input Applications

In this section, the SCXI applications have been divided into two types–single-channel applications and channel-scanning applications. The distinction between the two types is simple–single-channel applications do not involve automatic channel switching by the hardware during an analog input process; channel-scanning applications do.

Single-channel applications use the `AI` class of functions described in Chapter 3 or the `DAQ` class of functions described in Chapter 6 to acquire the input data after the SCXI has been set up. To acquire data from more than one channel, multiple `AI` or `DAQ` function calls are needed, and explicit SCXI function calls may be needed to change the SCXI channel that has been selected; this specific type of single-channel application is referred to as *software scanning*.

Channel-scanning applications will use the `SCAN` and `Lab_SCAN` classes of functions described in Chapter 6 to acquire the input data after the SCXI has been set up.

Building Analog Input Applications in Multiplexed Mode

Multiplexed applications require the use of `SCXI` functions to select the multiplexed channels, select the programmable module features, route signals on the SCXIbus, and program Slot 0. After the SCXI chassis and modules have been set up, the `AI`, `DAQ`, and `SCAN` functions can be used to acquire the data. The **channel** parameter that is passed to each of these functions will almost always be zero because, by default, the multiplexed output of a module is connected to analog input channel 0 of the DAQ board. When multiple chassis are used, the modules in each chassis are multiplexed to a separate analog input channel. In that case, the channel parameters of the `AI`, `DAQ`, and `SCAN` functions should be the DAQ board channel that corresponds to the desired chassis for the operation.

Figure 7-3 shows the function call sequence of a single-channel or software-scanning application using an SCXI-1100, SCXI-1102, SCXI-1120, SCXI-1121, SCXI-1122, or SCXI-1141 module operating in Multiplexed mode.

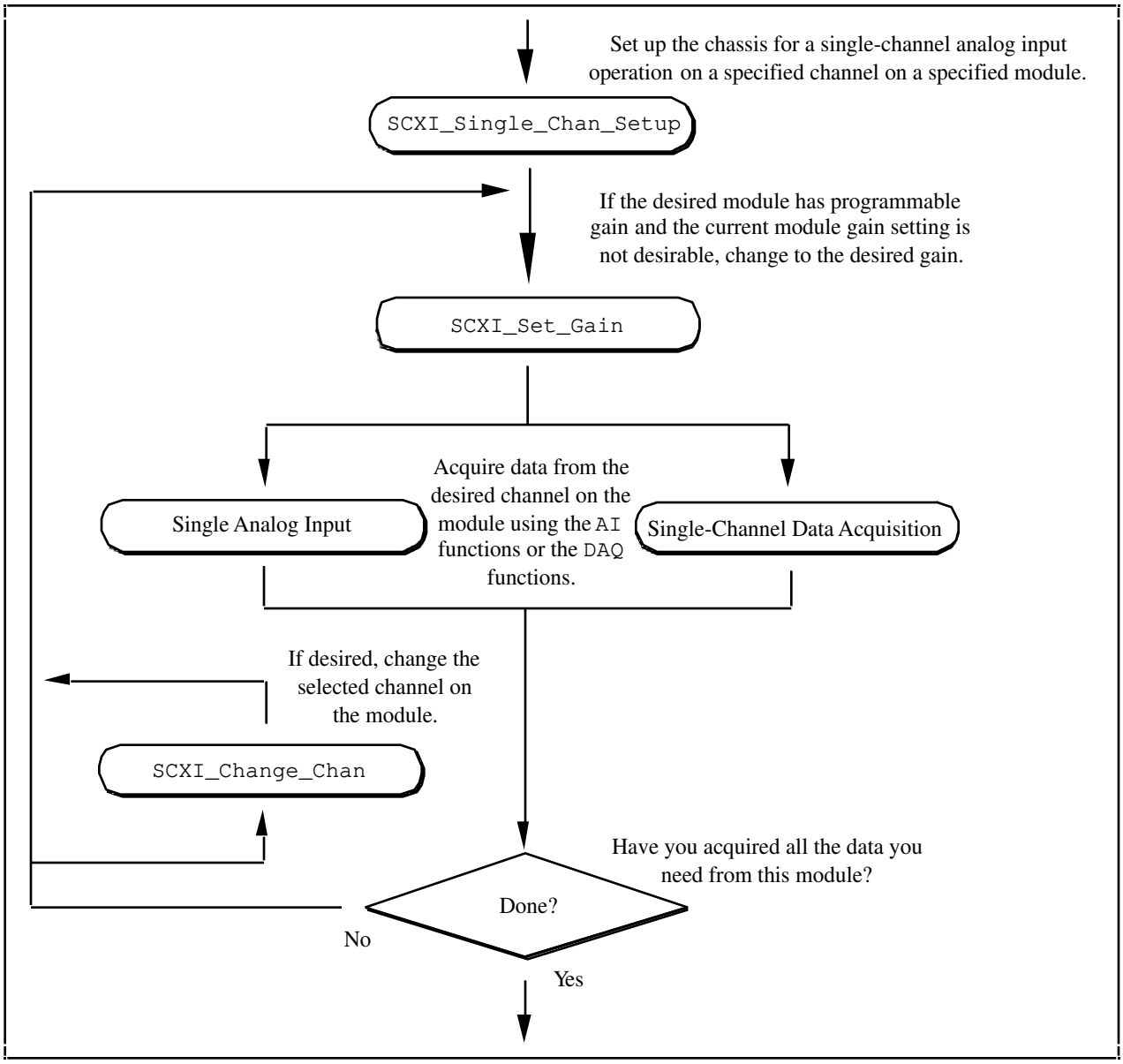

Figure 7-3. Single-Channel or Software-Scanning Operation Using the SCXI-1100, SCXI-1102, SCXI-1120, SCXI-1121, SCXI-1122, or SCXI-1141 in Multiplexed Mode

The `SCXI_Single_Chan_Setup` function will select the given channel to appear at the module output. If the given module is not directly cabled to the DAQ board, the function will send the module output on the SCXIbus; then it will configure the module that *is* cabled to the DAQ board to send the signal that is present on the SCXIbus to the DAQ board.

The `SCXI_Set_Gain` function is used to change the gain of an SCXI-1100, SCXI-1102, SCXI-1122, or SCXI-1141 module. The module will maintain this gain setting until the function is used again to change it. You can also do other module-specific programming at this point, such as `SCXI_Set_Filter` or `SCXI_Set_Input_Mode`.

To achieve software scanning, a different channel on the module can be selected using the `SCXI_Change_Chan` function after acquiring data from the desired channel with the `AI` or `DAQ` functions. If a channel on a different

module is desired, you will have to call the `SCXI_Single_Chan_Setup` function again to enable the appropriate module outputs and manage the SCXIbus signal routing.

Figure 7-4 shows the function call sequence of a single channel or software scanning application using an SCXI-1140 in Multiplexed mode.

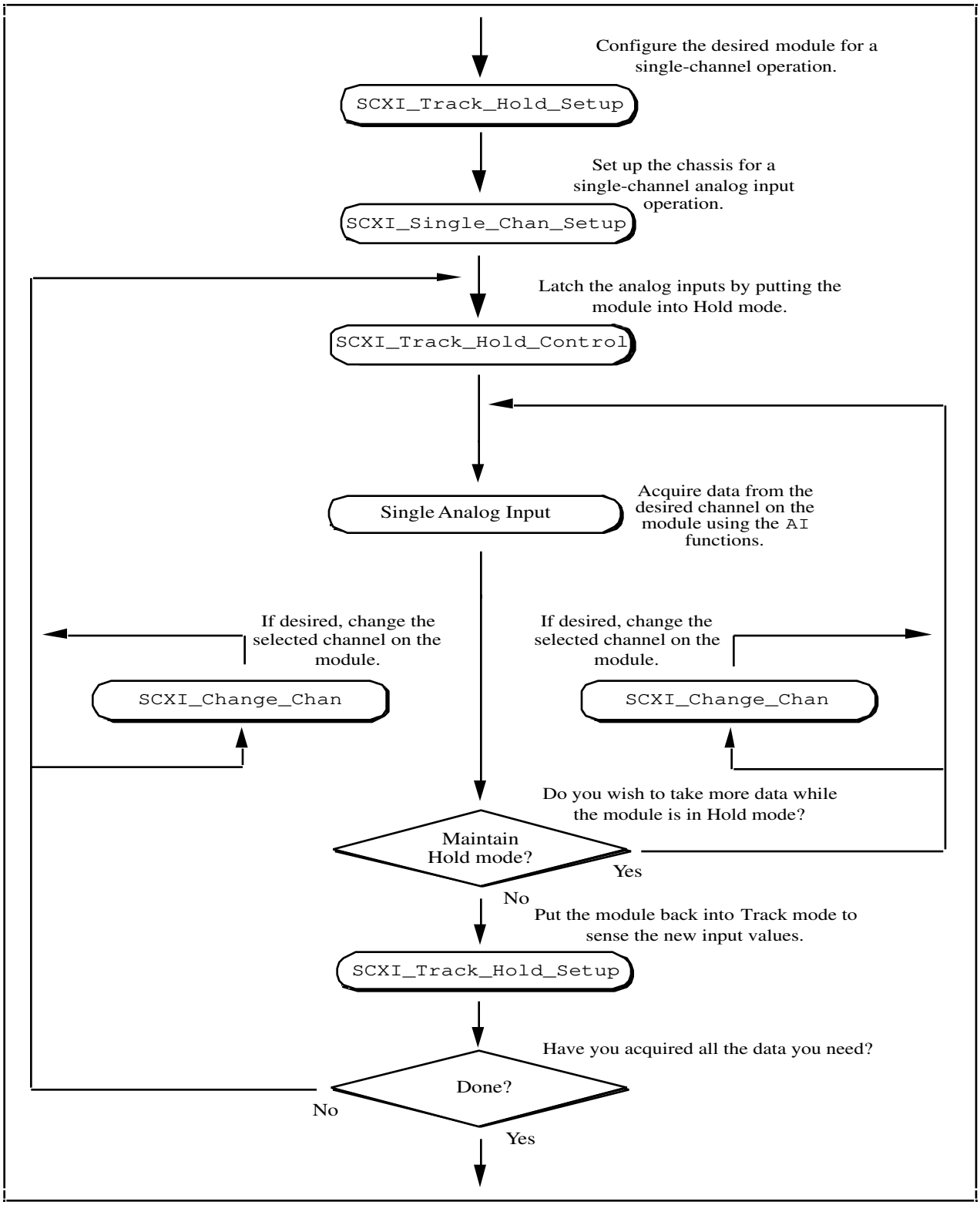

Figure 7-4. Single-Channel or Software-Scanning Operation Using the SCXI-1140 in Multiplexed Mode

The initial `SCXI_Track_Hold_Setup` call will signal the driver that the module will be used in a single-channel application, and will put the module into Track mode. The first `SCXI_Track_Hold_Control` call will latch, or sample, all the module inputs; subsequent `AI` calls will then read the voltages that were sampled. It is important to realize that all `AI` operations that occur between the first `SCXI_Track_Hold_Control` call, which puts the module into Hold mode, and the second control call, which puts the module into Track mode, will be acquiring data that was sampled at the time of the first control call. One or more channels may be read while the module is in Hold mode. After the module is put back into Track mode, the user can repeat the process to acquire new data. An `SCXI_Single_Chan_Setup` call is required to select the multiplexed channel and route the output to the DAQ board appropriately. The `SCXI_Change_Chan` call can change the channel on the module either while it is in Hold mode, or after the module has been returned to Track mode.

Figure 7-5 shows the function call sequence of a channel-scanning application in Multiplexed mode. Any combination of module types can be used in a scanning operation, with the following restrictions–if any SCXI-1140 modules are to be scanned, interval scanning must be used, and the module that is directly connected to the DAQ board must be an SCXI-1140.

Note: ***The SCXI-1122 uses relays to switch the input channels; the relays require 10 ms to switch, so the sampling rate in a channel scanning operation cannot exceed 100 Hz. If you want to take many readings from each channel and average them to reduce noise, you should use the single-channel or software scanning method shown in Figure 7-3 instead of the channel-scanning method shown in Figure 7-5. This means you select one channel on the module, acquire many samples on that channel using the DAQ functions, select the next channel, and so on. This will increase the lifetime of your module relays. Once you have selected a particular channel, you can use the fastest sample rate your DAQ board supports with the DAQ functions.***

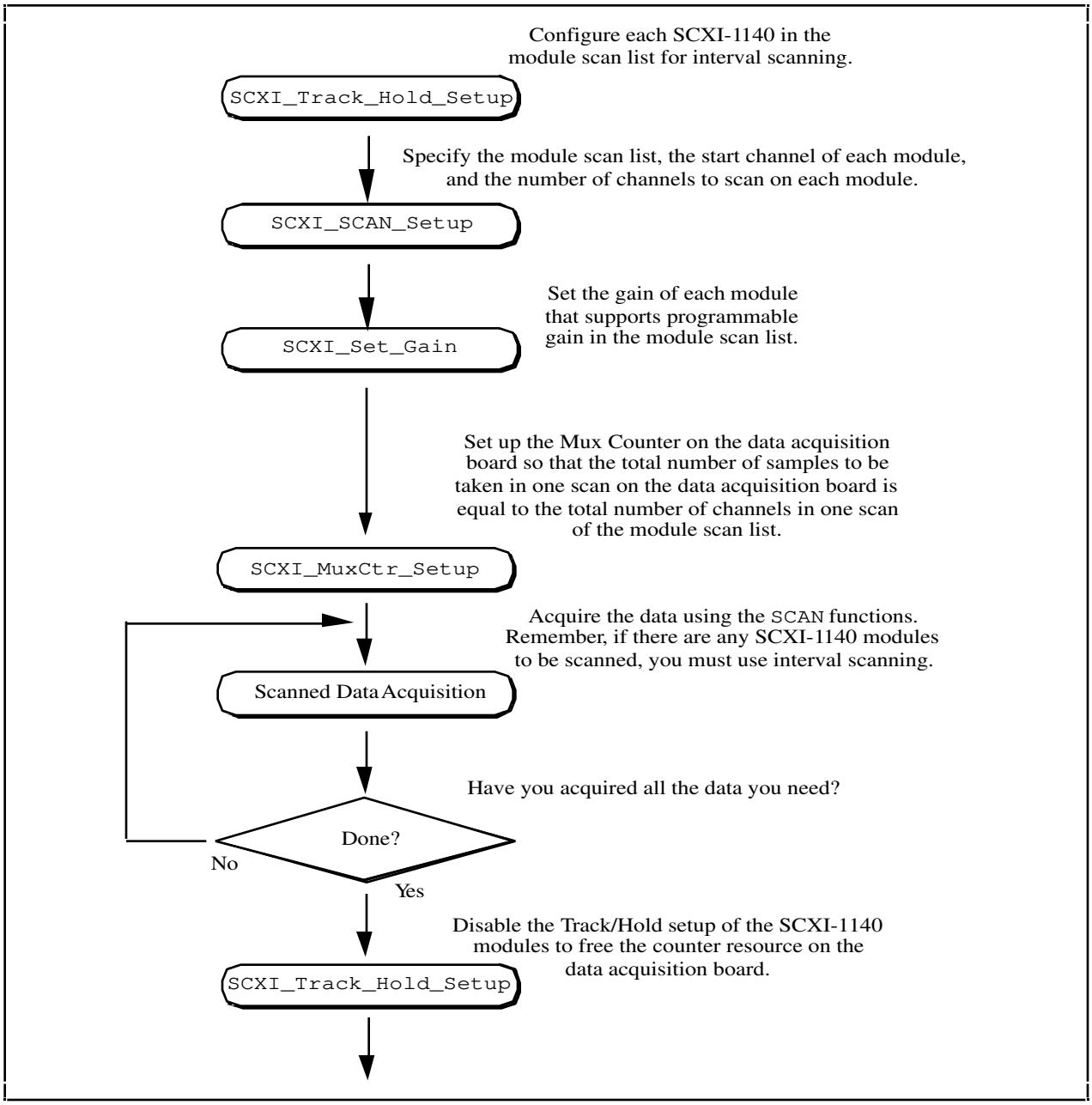

Figure 7-5. Channel-Scanning Operation Using Modules in Multiplexed Mode

If any of the modules to be scanned are SCXI-1140 modules, you must establish the Track/Hold setup of each one. If you want to synchronize multiple SCXI-1140 modules, you can configure the module that is receiving the Track/Hold control signal to send the Track/Hold signal on the SCXIbus so that any other SCXI-1140 modules can use it. The Track/Hold signal can either be from the DAQ board counter or from an external source.

The `SCXI_SCAN_Setup` call establishes the module scan list, which is downloaded to Slot 0. Each module is programmed for automatic scanning starting at its given start channel. If the SCXIbus will be needed during the scan to route the outputs of multiple modules, this function will resolve any contention.

In many of the data acquisition function descriptions in Chapter 6, the **count** parameter descriptions specify that **count** must be an integer multiple of the total number of channels scanned. When building channel-scanning

acquisitions in Multiplexed mode, the total number of channels scanned is the sum of all the elements in the **numChansList** array in the `SCXI_SCAN_Setup` function call.

If any of the modules in the module scan list support programmable gain, you can use the `SCXI_Set_Gain` to change the gain setting on each module. You can also use the `SCXI_Configure_Filter` and `SCXI_Set_Input_Mode` functions at this time if they are appropriate for your module.

The `SCXI_MuxCtr_Setup` call is used to synchronize the module scan list with the DAQ board scan list. In most cases (especially when interval scanning is used), it will be desirable to ensure that the number of samples to be taken in one pass through the module scan list is the same as the number of samples to be taken in one pass through the DAQ board scan list. Counter 1 on the MIO-16 is used to achieve this synchronization.

After the SCXI chassis and modules have been set up, more than one channel-scanning operation can be performed using the `SCAN` functions without reconfiguring the SCXI chassis or modules.

Building Analog Input Applications in Parallel Mode

When the SCXI-1120, SCXI-1121, and SCXI-1141 modules are operated in Parallel mode, no further `SCXI` function calls are required beyond those shown in Figure 7-2 to set up the modules for analog input operations. After the SCXI chassis and modules have been initialized and reset, you are ready use the `AI`, `DAQ`, `SCAN`, or `Lab_SCAN` functions. Remember that the **channel** and **gain** parameters of the `AI`, `DAQ`, `SCAN`, and `Lab_SCAN` functions refer to the DAQ board channels and gains.

For example, to acquire a single reading from channel 0 on the module, call the `AI_Read` function with the **channel** parameter set to zero. The **gain** parameter refers to the DAQ board gain. The `SCXI_Scale` function can then be used to convert the binary reading to a voltage.

To build a channel-scanning application using the SCXI-1120, SCXI-1121, or SCXI-1141 in Parallel mode, use the `SCAN` and `Lab_SCAN` functions to scan the channels on the DAQ board that correspond to the desired channels on the module. For example, to scan channels 0, 1, and 3 on the module using an NB-MIO-16X board, call the `SCAN_Setup` function with the **channel** vector set to {0, 1, 3}. The **gain** vector should contain the NB-MIO-16X channel gains. After the data is acquired, it can be demultiplexed and the data for each channel can be sent to the `SCXI_Scale` function.

In many of the data acquisition function descriptions in Chapter 6, the **count** parameter descriptions specify that **count** must be an integer multiple of the total number of channels scanned. When building channel-scanning acquisitions in Parallel mode, the total number of channels scanned is determined by the **num_chans** parameter in the `SCAN_Setup` or `Lab_SCAN_Start` call.

The SCXI-1140 module requires the use of `SCXI` functions to configure and control the Track/Hold state of the module before the `AI` and `SCAN` functions can be used to acquire the data. Figure 7-6 shows the function call sequence of a single-channel (or software-scanning) operation using the SCXI-1140 module in Parallel mode.

The SCXI-1100, SCXI-1102, and SCXI-1122 modules do not support Parallel mode.

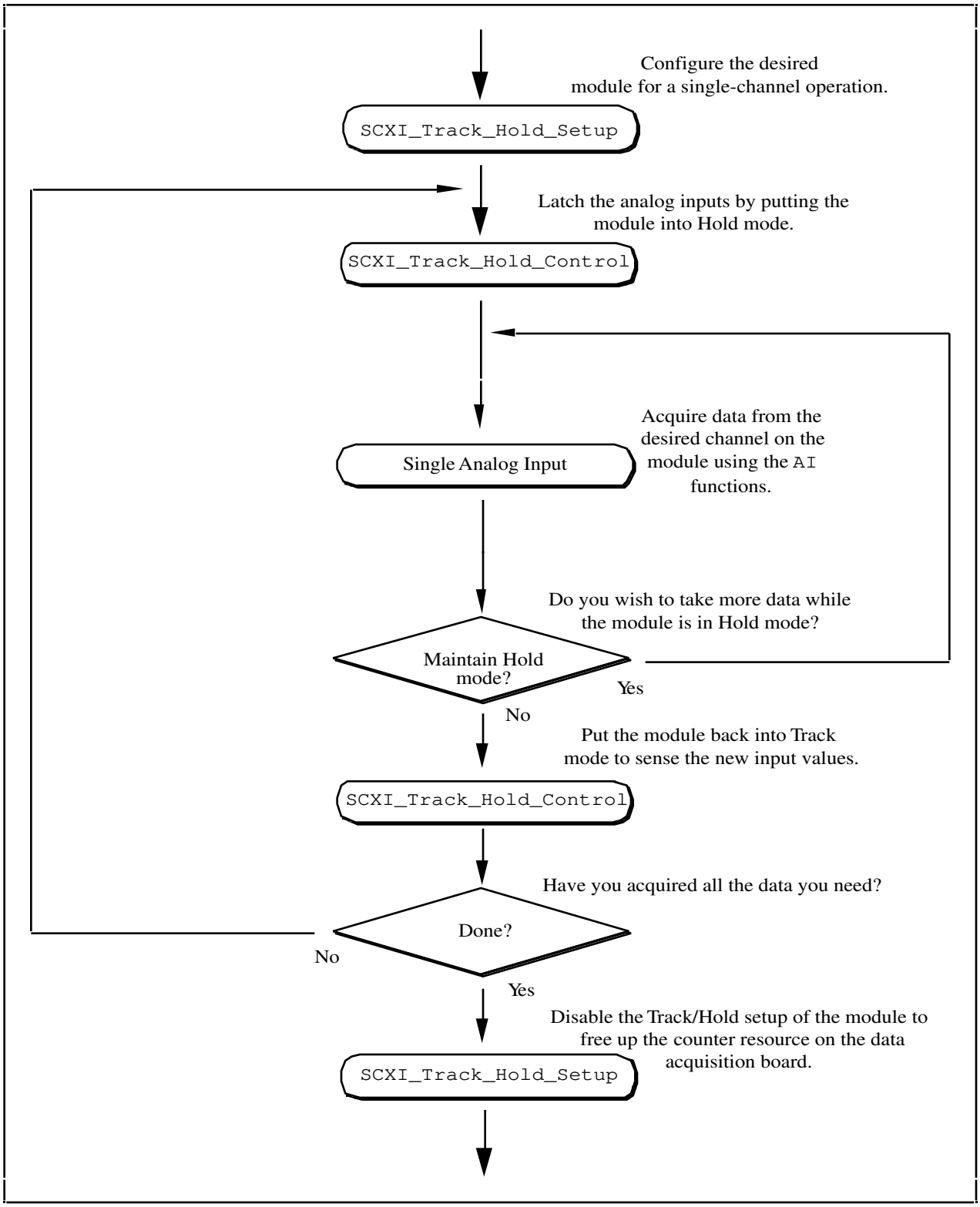

Figure 7-6. Single-Channel or Software-Scanning Operation Using the SCXI-1140 in Parallel Mode

The initial `SCXI_Track_Hold_Setup` call will signal the driver that the module will be used in a single-channel application, and will put the module into Track mode. The first `SCXI_Track_Hold_Control` call will latch, or sample, all the module inputs; subsequent `AI` calls will then read the voltages that were sampled. It is important to realize that all `AI` operations that occur between the first `SCXI_Track_Hold_Control` call, which puts the module into Hold mode, and the second control call, which puts the module into Track mode, will be acquiring data that was sampled at the time of the first control call. One or more channels may be read while the module is in Hold mode. After the module is put back into Track mode, the user can repeat the process to acquire new data.

Remember that the **channel** and **gain** parameters of the `AI` function calls refer to the DAQ board channels and gains. Simply use the data acquisition channels that correspond to the desired module channels as described earlier in this section. You must also be aware of the SCXI-1140 Track/Hold timing requirements that were described in *The SCXI-1140* section earlier in this chapter.

Figure 7-7 shows the function call sequence of a channel-scanning application using the SCXI-1140 in Parallel mode.

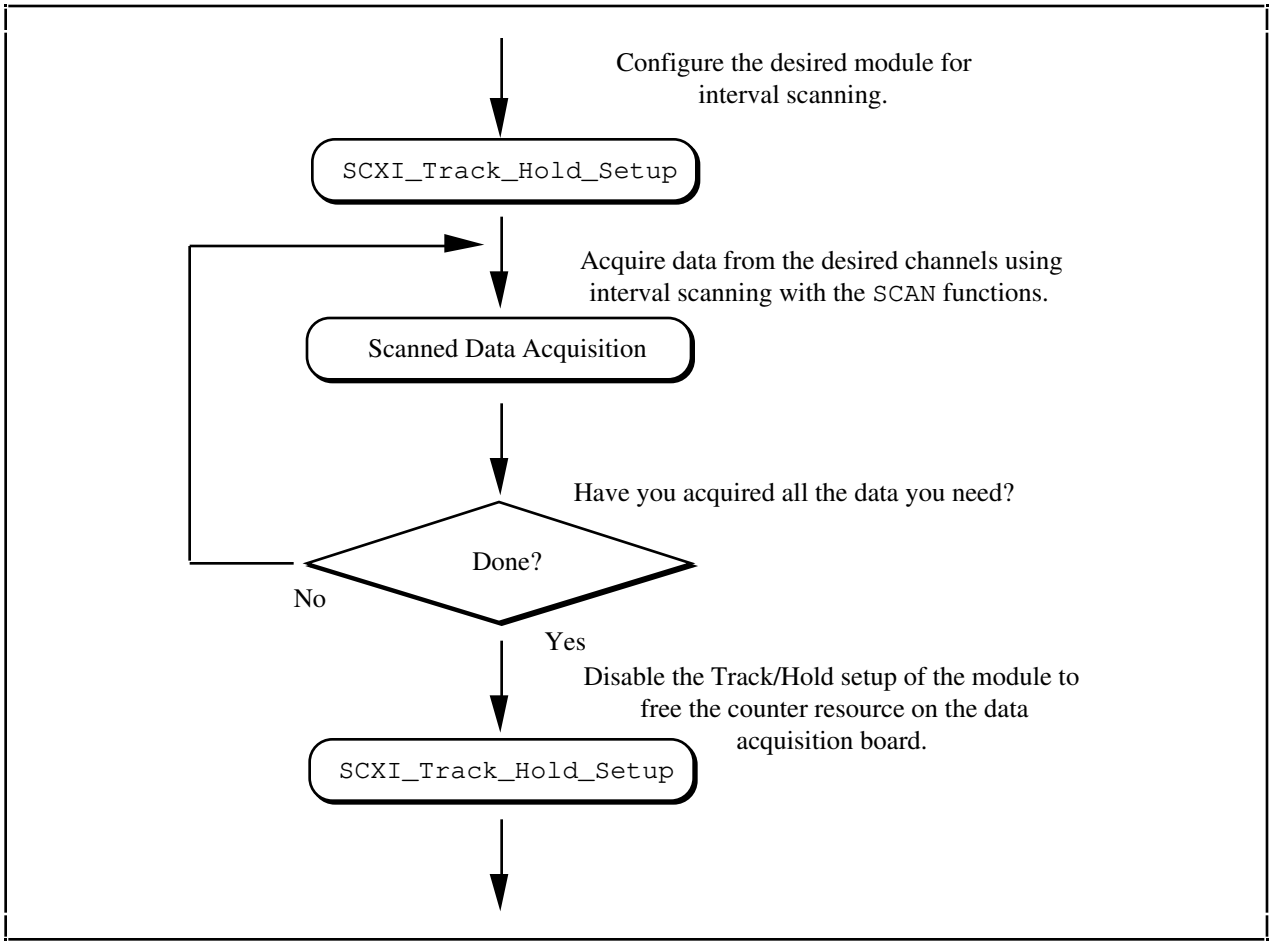

Figure 7-7. Channel-Scanning Operation Using the SCXI-1140 in Parallel Mode

The call sequence is much simpler because the scan interval timer will control the Track/Hold state of the module automatically during the interval-scanning operation. Remember that only the NB-MIO-16 and NB-MIO-16X boards support channel-scanning using the SCXI-1140 module.

Analog Output Applications

Using the SCXI-1124 analog output module with the NI-DAQ functions is very simple. Just call the `SCXI_AO_Write` function to write your desired voltages to the DAC channels on the module. You can use the `SCXI_Get_Status` function, if you wish, to determine when the DAC channels have settled to their final analog output voltages.

If you want to calculate new calibration constants for `SCXI_AO_Write` to use for the voltage to binary conversion instead of the factory calibration constants that are shipped in the module EEPROM, follow the procedure outlined in the `SCXI_Cal_Constants` function description.

Digital Applications

If you configured your digital or relay modules for Multiplexed mode, use the `SCXI_Set_State` and `SCXI_Get_State` functions to access your digital or relay channels.

If you are using the SCXI-1160 module, you may wish to use the `SCXI_Get_Status` function after calling the `SCXI_Set_State` function. `SCXI_Get_Status` will tell you when the SCXI-1160 relays have finished switching.

If you are using the SCXI-1162 or SCXI-1162HV module, `SCXI_Get_State` will read the module digital input channels. For the other digital and relay modules, `SCXI_Get_State` will return a software copy of the current state that is maintained by NI-DAQ. However, if you are using the SCXI-1163 or SCXI-1163R in Parallel mode, `SCXI_Get_State` will read the hardware states.

If you are using the SCXI-1162, SCXI-1162HV, SCXI-1163, or SCXI-1163R in Parallel mode, you can use the SCXI functions as described earlier, or you can call the `DIG_In_Port` and `DIG_Out_Port` functions using the correct DAQ board port numbers that correspond to the SCXI module digital channels. *The DIO-24 and DIO-96* and *The DIO-32F* sections earlier in this chapter list the onboard port numbers that are used for each type of board if the SCXI-1162, SCXI-1162HV, SCXI-1163, or SCXI-1163R is configured for Parallel mode. The MIO and Lab Series boards cannot use the SCXI-1162, SCXI-1162HV, SCXI-1163, or SCXI-1163R in Parallel mode.

Transducer Conversions

Appendix D, *Transducer Conversion Routines*, contains information regarding transducer conversion functions for RTDs, thermocouples, thermistors, and strain gauges. The source code for these functions is provided with NI-DAQ for Macintosh.

SCXI_AO_Write

Function

Sets the DAC channel on the SCXI-1124 module to the specified voltage or current output value. You can also use this function to write a binary value directly to the DAC channel, or to translate a voltage or current value to the corresponding binary value.

Synopsis

C Syntax	`locus i32 SCXI_AO_Write(u32 chassisID, u32 moduleSlot, u32 DACchannel, u32 opCode, u32 rangeCode, f64 voltCurrentData, i32 binaryData, i16 *binaryWritten);`
Pascal Syntax	`function SCXI_AO_Write(chassisID : i32; moduleSlot : i32; DACchannel : i32; opCode : i32; rangeCode : i32; voltCurrentData : f64; binaryData : i32; var binaryWritten : i16) : i32;`
BASIC Syntax	`FN SCXI_AO_Write(chassisID&, moduleSlot&, DACchannel&, opCode&, rangeCode&, voltCurrentData#, binaryData&, binaryWritten&)`

C Syntax	`SCXI_AO_Write (SCXIchassisID, moduleSlot, channel, opCode, rangeCode, voltCurrentData, binaryData, binaryWritten);`
	`int SCXIchassisID, moduleSlot, channel, opCode, rangeCode;`
	`double voltCurrentData;`
	`int binaryData, *binaryWritten;`

Pascal Syntax	`SCXI_AO_Write (SCXIchassisID, moduleSlot, channel, opCode, rangeCode : integer; voltCurrentData : double; binaryData : integer; var binaryWritten : integer) : integer;`

Parameters

channel is the number of the analog output channels on the module.

Range: 0 to 5.

opCode specifies the type of data to write to the DAC channel. You can also use **opCode** to tell `SCXI_AO_Write` to translate a voltage or current value and return the corresponding binary pattern in **binaryWritten** without writing anything to the module.

0: Write a voltage or current to **channel**.
1: Write a binary value directly to **channel**.
2: Translate a voltage or current value to binary, return in **binaryWritten**.\

rangeCode is the voltage or current range to be used for the analog output channel.

0: 0 to 1 V.
1: 0 to 5 V.
2: 0 to 10 V.
3: -1 to 1 V.
4: -5 to 5 V.
5: -10 to 10 V.
6: 0 to 20 mA.

voltCurrentData is the voltage or current you want to produce at the DAC channel output. If **opCode** = 1, NI-DAQ ignores this parameter. If **opCode** = 2, this is the voltage or current value you want to translate to binary. If the value is out of range for the given **rangeCode**, `SCXI_AO_Write` returns an error.

binaryData is the binary value you want to write directly to the DAC. If **opCode** is not 1, NI-DAQ ignores this parameter.

Range: 0 to 4,095

binaryWritten returns the actual binary value that NI-DAQ wrote to the DAC. `SCXI_AO_Write` uses a formula given later in this section using calibration constants that are stored on the module EEPROM to calculate the appropriate binary value that will produce the given voltage or current. If **opCode** = 1, **binaryWritten** is equal to **binaryData**. If **opCode** = 2, `SCXI_AO_Write` calculates the binary value but does not write anything to the module.

Description

`SCXI_AO_Write` uses the following equation to translate voltage or current values to binary:

$$B_w = B_l + (V_w - V_l) * (B_h - B_l) / (V_h - V_l)$$

where

B_l = binary value that produces the low value of the range
B_h = binary value that produces the high value of the range
V_h = high value of the range
V_l = low value of the range
V_w = desired voltage or current
B_w = the binary value which will generate Vw

NI-DAQ loads a table of calibration constants from the SCXI-1124 EEPROM load area. The calibration table contains values for B_l and B_h for each channel and range.

The SCXI-1124 is shipped with a set of factory calibration constants in the factory EEPROM area, and a copy of the factory constants in the EEPROM load area. You can recalibrate your module and store your own calibration constants in the EEPROM load area using the `SCXI_Cal_Constants` function. Please refer to the `SCXI_Cal_Constants` function description for calibration procedures and information about the module EEPROM.

If you want to write a binary value directly to the output channel, use **opCode** = 1. `SCXI_AO_Write` will not use the calibration constants or the conversion formula; it will simply write your **binaryData** value to the DAC.

SCXI_Cal_Constants

Function

Calculates calibration constants for the given channel and range or gain using measured voltage/binary pairs. You can use this function with any SCXI analog input or analog output module. The constants can be stored and retrieved from NI-DAQ memory or the module EEPROM (if your module has an EEPROM). The driver uses the calibration constants to more accurately scale analog input data when you use the `SCXI_Scale` function and output data when you use `SCXI_AO_Write`.

Synopsis

C Syntax	`locus i32 SCXI_Cal_Constants(u32 chassisID, u32 moduleSlot, i32 SCXIchannel, u32 opCode, u32 calArea, u32 rangeCode, f64 SCXIgain, u32 DAQdevice, u32 DAQchannel, u32 DAQgain, f64 TBgain, f64 volt1, f64 binary1, f64 volt2, f64 binary2, f64 *binEEprom1, f64 *binEEprom2);`
Pascal Syntax	`function SCXI_Cal_Constants(chassisID : i32; moduleSlot : i32; SCXIchannel : i32; opCode : i32; calArea : i32; rangeCode : i32; SCXIgain : f64; DAQdevice : i32; DAQchannel : i32; DAQgain : i32; TBgain : f64; volt1 : f64; binary1 : f64; volt2 : f64; binary2 : f64; var binEEprom1 : f64; var binEEprom2 : f64) : i32;`
BASIC Syntax	`FN SCXI_Cal_Constants(chassisID&, moduleSlot&, SCXIchannel&, opCode&, calArea&, rangeCode&, SCXIgain#, DAQdevice&, DAQchannel&, DAQgain&, TBgain#, volt1#, binary1#, volt2#, binary2#, binEEprom1&, binEEprom2&)`

Parameters

SCXIchannel is the number of the channel on the module.

Range: 0 to *n*-1, where *n* is the number of channels available on the module.

-1: all channels on the module. For instance, the SCXI-1100 and SCXI-1122 modules have one amplifier for all channels, so calibration constants for those modules apply to all channels on the module.

-2: the voltage (**calConst2**) and current (**calConst1**) excitation channels on the module. This is valid for the SCXI-1122 only, and only when **opCode** = 0.

opCode specifies the type of calibration operation to be performed.

0: Retrieve calibration constants for the given channel and range or gain from **calArea** and return them in **calConst1** and **calConst2**.

1: Do one point offset calibration calculation using (**volt1**, **binary1**) for the given channel and range or gain and write calibration constants to **calArea** (analog input modules only).

2: Do two point calibration calculation using (**volt1**, **binary1**) and (**volt2**, **binary2**) for the given channel and range or gain and write calibration constants to **calArea**.

3: Write the calibration constants passed in **calConst1** and **calConst2** to **calArea** for the given channel and range or gain.

4: Copy entire calibration table in **calArea** to the module EEPROM default load area so that it will be loaded automatically into NI-DAQ memory during subsequent application runs.

5: Copy entire calibration table in **calArea** to driver memory so it can be used in subsequent scaling operations in the current NI-DAQ session.

calArea is the location used for the calibration constants. Please read the discussion below in the *Description* section for an explanation of the calibration table stored in NI-DAQ memory and the SCXI-1102, SCXI-1122, and SCXI-1124 and SCXI-1141 EEPROM organization.

0: NI-DAQ memory. NI-DAQ maintains a calibration table in memory for use in scaling operations for the module.

1: Default EEPROM load area. NI-DAQ will also update the calibration table in memory when you write to the default load area.

2: Factory EEPROM area. You cannot write to this area, but you can read or copy from this area.

3: User EEPROM area.

rangeCode is the voltage or current range of the analog output channel. This parameter is only used for analog output modules.

0: 0 to 1 V.
1: 0 to 5 V.
2: 0 to 10 V.
3: -1 to 1 V.
4: -5 to 5 V.
5: -10 to 10 V.
6: 0 to 20 mA.

SCXIgain is the SCXI module or channel gain setting. This parameter is only used for analog input modules. Valid **SCXIgain** values depend on the module type:

SCXI-1100: 1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, 2,000.
SCXI-1102: 1, 100.
SCXI-1120: 1, 2, 5, 10, 20, 50, 100, 200, 250, 500, 1,000, 2,000.
SCXI-1121: 1, 2, 5, 10, 20, 50, 100, 200, 250, 500, 1,000, 2,000.
SCXI-1140: 1, 10, 100, 200, 500.
SCXI-1122: 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, 2,000.
SCXI-1141: 1, 2, 5, 10, 20, 50, 100

DAQdevice is the number of the device you are using to sample the channels on an analog input module. This only applies to analog input modules and only when **opCode** is 0, 1, 2, or 3. Otherwise, set **DAQdevice** to 0.

DAQchannel is the channel number on the DAQ board used to sample the channels on an analog input module. **DAQchannel** is usually 0. This only applies to analog input modules and only when **opCode** is 0, 1, 2, or 3.

gain is the gain code for the gain used for **DAQchannel** on **DAQdevice**. This only applies to analog input modules and only when **opCode** is 0, 1, 2, or 3.

TBgain is the terminal block gain applied to the SCXI channel, if any. Currently, the SCXI-1327 terminal block is the only terminal block that applies gain to your SCXI channels. The SCXI-1327 has switches that you use to select either a gain of 1.0 or a gain of 0.01. You can use this terminal block with an SCXI-1120 or SCXI-1121 module. For terminal blocks that do not apply gain to your SCXI channels, set **TBgain** =1.0.

volt1, **binary1** is a measured voltage/binary pair you have taken for the given channel and range or gain if **opCode** = 1 or 2. If the module is analog output, **volt1** is the voltage or current you measured at the output channel after writing the binary value **binary1** to the output channel.

If the module is analog input, **binary1** is the binary value you read from the input channel with a known voltage of **volt1** applied at the input. The **binary1** parameter is floating point, so you may take multiple binary readings from **volt1** and average them to be more accurate and reduce the effects of noise.

volt2, **binary2** is a second measured voltage/binary pair you have taken for the given channel and range or gain if **opCode** = 1 or 2. If the module is analog output, **volt2** is the voltage or current you measured when the binary value **binary2** was written to the output channel. If the module is analog input, **binary2** is the binary reading from the input channel with a known voltage of **volt2** applied at the input.

calConst1 is the first calibration constant. For analog output modules, **calConst1** is the binary value that will generate the voltage or current at the lower end of the voltage or current range. For analog input modules, **calConst1** is the binary zero offset; that is, the binary reading that would result from an input voltage of zero. If **opCode** = 1 or 2, **calConst1** is a return value calculated from the voltage/binary pairs. If **opCode** = 0 **calConst1** is a return constant retrieved from the **calArea**. If **opCode**= 0 and **channel**= -2, **calConst1** is the actual voltage excitation value returned in units of volts. If **opCode** = 3, you should pass your first calibration constant in **calConst1** for NI-DAQ to store in **calArea**.

calConst2 is the second calibration constant. For analog output modules, **calConst2** is the binary value that generates the voltage or current at the upper end of the voltage or current range. For analog input modules, **calConst2** is the gain adjust factor; that is, the ratio of the real gain to the ideal gain setting. If **opCode** = 1 or 2, **calConst2** is a return value calculated from the voltage/binary pairs. If **opCode** = 0, **calConst2** is a return constant retrieved from the **calArea**. If **opCode** = 0 and **channel**= -2, **calConst** is the actual current excitation value returned in units of milliamperes. If **opCode** = 3, you should pass your second calibration constant in **calConst2** for NI-DAQ to store in **calArea**.

Description

Analog Input Calibration

When you call `SCXI_Scale` to scale binary analog input data, NI-DAQ will use the binary offset and gain adjust calibration constants loaded for the given module, channel, and gain setting to scale the data to voltage. Please refer to the `SCXI_Scale` function description for the equations used.

By default, NI-DAQ will load calibration constants for the SCXI-1102, SCXI-1122 and SCXI-1141 from the module EEPROM (the *EEPROM Organization* section below explains the EEPROM in detail). The SCXI-1141 has only gain adjust constants in EEPROM and does not have binary zero offset in EEPROM. All other analog input modules have no calibration constants by default; NI-DAQ assumes no binary offset and ideal gain settings for those modules *unless* you use the procedure outlined below to store calibration constants for your module.

You can determine calibration constants based specifically on your application setup, which includes your type of DAQ board, your DAQ board settings, and your cable assembly, all combined with your SCXI module and its configuration settings.

Note: ***NI-DAQ will store constants in a table for each SCXI module gain setting. If your module has independent gains on each channel, NI-DAQ will store constants for each channel at each gain setting. When you use the procedure below, you are also calibrating for your DAQ board settings, so you must use the same DAQ board settings whenever you use the new calibration constants. The factory EEPROM constants apply only to the amplifiers on the SCXI modules, so you can use those with any DAQ board setup.***

To perform a two-point analog input calibration, use the following steps:

1. Make sure the SCXI gain is set to the gain you will be using in your application.

2. Use `SCXI_Single_Chan_Setup` to program the module for a single channel operation (as opposed to a channel scanning operation).

3. Ground your SCXI input channel. If you are using an SCXI-1100, SCXI-1122, or SCXI-1141, you can use the `SCXI_Calibrate_Setup` function to internally ground the module's amplifier inputs. For other analog input modules, you need to wire the positive and negative channel inputs together at the terminal block. Refer to your module user manual for terminal block wiring instructions.

4. Take several readings using the `DAQ` functions and average them for greater accuracy. You should use the DAQ board gain settings you will be using in your application. You should average over an integral number of 60 Hz or 50 Hz power line cycles to eliminate line noise.

You now have your first volt/binary pair: **volt1** = 0.0, and **binary1** is your binary reading or binary average.

5. Now apply a known, stable, non-zero voltage to your input channel at the terminal block. Preferably, your input voltage should be close to the upper limit of your input voltage range for the given gain setting.

6. Take another binary reading or average. If your binary reading is the maximum binary reading for your DAQ board, you should try a smaller input voltage. This is your second volt/binary pair: **volt2** and **binary2**.

7. Call `SCXI_Cal_Constants` with your two volt/binary pairs and **opCode** = 2. Make sure you pass the correct **SCXIgain** you used, and pass the gain code you used in `AI_Read` or `DAQ_Op` in the **gain** parameter.

You can save the constants in the module EEPROM, if your module has one (**calArea** = 1 or 3). Refer to the *EEPROM Organization* section below for information about constants in the EEPROM. We recommend that you use **calArea** = 3 (user EEPROM area) as you are calibrating, and then call `SCXI_Cal_Constants` again at the end of your calibration sequence with **opCode** = 4 to copy the user EEPROM area to the default EEPROM load area. That way there will be two copies of your new constants, and you can revert to the factory constants using **opCode** = 4 without wiping out your new constants entirely.

For analog input modules without an EEPROM, you must specify **calArea** = 0 (NI-DAQ memory). Unfortunately, calibration constants stored in NI-DAQ memory will be lost at the end of the current NI-DAQ session. You may wish to create a file and save the constants returned in **calConst1** and **calConst2** so that you can load them again in subsequent application runs using `SCXI_Cal_Constants` with **opCode** = 3.

Now, any subsequent calls to `SCXI_Scale` for the given module, channel, and gain setting will use the new calibration constants when scaling. You can repeat steps 1 through 7 for any other channel or gain settings you wish to calibrate.

You may use a different voltage for the first measurement instead of grounding the input channel. For instance, if you know you will be using a specific input voltage range, you might use the endpoints of your expected input voltage range as **volt1** and **volt2**. Then you would be specifically calibrating your expected input voltage range.

If you are using an SCXI-1100, SCXI-1122, or SCXI-1141, you may wish to just perform a one-point calibration to determine the binary offset; you can do this easily without external hookups using the `SCXI_Calibrate_Setup` function to internally ground the module amplifier. Use the procedure above, skipping steps 5 and 6, and using **opCode** = 1 for the `SCXI_Cal_Constants` function.

If you are storing calibration constants in the EEPROM, your binary offset and gain adjust factors must not exceed the ranges given in your module user manual. The constant format in the EEPROM will not allow for larger constants. If your constants exceed these specifications, the function will return a **-10086 badExtRefErr** error. If this error occurs, you should make sure your **SCXIgain**, **gain**, and **TBgain** values are the actual settings you used to measure the volt/binary pairs, and you may wish to recalibrate your DAQ board, if applicable.

Analog Output Calibration

When you call `SCXI_AO_Write` to output a voltage or current to your SCXI-1124 module, NI-DAQ uses the calibration constants loaded for the given module, channel, and output range to scale the voltage or current value to the appropriate binary value to write to the output channel. NI-DAQ will load calibration constants into memory for the SCXI-1124 from the module EEPROM load area the first time you access the module using an NI-DAQ function call (the *EEPROM Organization* section below explains the EEPROM in detail).

You can recalibrate your SCXI-1124 module to create your own calibration constants using the following procedure:

1. Use the `SCXI_AO_Write` function with **opCode** = 1. If you are calibrating a voltage output range, pass the parameter **binaryData** = 0. If you are calibrating the 0 to 20 mA current output range (**rangeCode** = 6), pass the parameter **binaryData** = 255.

2. Measure the output voltage or current at the output channel. This is your first volt/binary pair: **binary1** = 0 or 255, and **volt1** is the voltage or current you measured at the output.

3. Use the `SCXI_AO_Write` function with **opCode** = 1 to write the **binaryData** = 4,095 to the output DAC.

4. Measure the output voltage or current at the output channel. This is your second volt/binary pair: **binary2** = 4,095 and **volt2** is the voltage or current you measured at the output.

5. Call `SCXI_Cal_Constants` with your voltage/binary pairs and **opCode** = 2. You can save the constants on the module EEPROM (**calArea** = 1 or 3). Refer to the *EEPROM Organization* section below for information about constants in the EEPROM. We recommend that you use **calArea** = 3 (user EEPROM area) as you are calibrating, and then call `SCXI_Cal_Constants` again at the end of your calibration sequence with **opCode** = 4 to copy the user EEPROM area to the default load area. That way there will be two copies of your new constants, and you can revert to the factory constants using **opCode** = 4 without wiping out your new constants entirely.

Repeat the procedure above for each channel and range you wish to calibrate. Subsequent calls to `SCXI_AO_Write` will use your new constants to scale voltage or current to the correct binary value.

EEPROM Organization

The SCXI-1102, SCXI-1122, SCXI-1124, and SCXI-1141 modules have an onboard EEPROM to handle storage of calibration constants. The EEPROM is divided into three areas:

- The **factory area** is shipped with a set of factory calibration constants; you cannot write into the factory area, but you can read from it.
- The **default load area** is where NI-DAQ automatically looks to load calibration constants the first time you access the module during an NI-DAQ session using an NI-DAQ function call (such as `SCXI_Reset`, `SCXI_Single_Chan_Setup`, or `SCXI_AO_Write`). When the module is shipped, the default load area contains a copy of the factory calibration constants. When you write to the default load area using `SCXI_Cal_Constants`, NI-DAQ will also update the constants in NI-DAQ memory.
- The **user area** is an area provided for you to store your own calibration constants that you calculate by following the instructions above and using the `SCXI_Cal_Constants` function. You may also put a copy of your own constants in the default load area if you want NI-DAQ to automatically load your constants for subsequent NI-DAQ sessions.

SCXI_Calibrate_Setup

Function

Grounds the amplifier inputs of an SCXI-1100, SCXI-1122, or SCXI-1141 so that you can determine the amplifier offset. You can also use this function to switch a shunt resistor across your bridge circuit to test the circuit. Shunt calibration is supported for the SCXI-1122 and the SCXI-1121 when you use the SCXI-1321 terminal block.

Synopsis

C Syntax	`locus i32 SCXI_Calibrate_Setup(u32 chassisID, u32 moduleSlot, u32 calOp);`
Pascal Syntax	`function SCXI_Calibrate_Setup(chassisID : i32; moduleSlot : i32; calOp : i32) : i32;`
BASIC Syntax	`FN SCXI_Calibrate_Setup(chassisID&, moduleSlot&, calOp&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the module.
Range: 1 to *n*, where *n* is the number of slots in the chassis.

calOp indicates the desired Calibration mode.
0: disable calibration.
1: connect the positive and negative inputs of the module amplifier together and to analog reference.
2: switch the shunt resistor(s) across the bridge circuit(s) for the SCXI-1122, or the SCXI-1121 (Revision C or later) with the SCXI-1321 terminal block.

Description

The zero offset of the SCXI-1100, SCXI-1122, or SCXI-1141 amplifier varies with the module gain. Once you know the offset at a specific gain setting, that offset can be added to any readings acquired at that gain. In general, the procedure for determining the offset at a particular gain are as follows:

1. `SCXI_Single_Chan_Setup`–Enable the module output, route the module output on the SCXIbus if necessary, and resolve any SCXIbus contention if necessary. The module channel specified is irrelevant.

2. `SCXI_Set_Gain`–Set the module gain to the setting that will be used in your application.

3. `SCXI_Calibrate_Setup`–Ground the amplifier inputs.

4. Acquire data using the `DAQ` functions; many samples can be acquired and averaged, if desired. If you have enabled one of the filer settings on the module, you should wait for the amplifier to settle after you call `SCXI_Calibrate_Setup` before you acquire data. The SCXI-1100, SCXI-1122, and SCXI-1141 user manuals give settling time information for the filter settings.

5. `SCXI_Calibrate_Setup`–Disable calibration.

6. Continue with your application; subtract the binary offset value determined in Step 4 from any samples acquired from the module at the gain specified in Step 2 *before* passing the binary data to the `SCXI_Scale` function.

   Or, you can call `SCXI_Cal_Constants` to store the offset in NI-DAQ memory or the SCXI-1122 EEPROM. Then subsequent calls to `SCXI_Scale` for the given gain will automatically subtract the offset for you. Refer to the `SCXI_Cal_Constants` description for more information.

Refer to your SCXI-1321 and SCXI-1122 user manuals for more information about how the shunt resistor is applied when **calOp** = 2.

SCXI_Change_Chan

Function

Selects a new channel of a multiplexed module that has previously been set up for a single-channel analog input operation using the `SCXI_Single_Chan_Setup` function.

Synopsis

C Syntax	`locus i32 SCXI_Change_Chan(u32 chassisID, u32 moduleSlot, i32 channel);`
Pascal Syntax	`function SCXI_Change_Chan(chassisID : i32; moduleSlot : i32; channel : i32) : i32;`
BASIC Syntax	`FN SCXI_Change_Chan(chassisID&, moduleSlot&, channel&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the module that has been set up for a single-channel analog input operation.

Range: 1 to *n*, where *n* is the number of slots in the chassis.

channel is the channel number of the new input channel on the module that is to be read.

0 to *n*-1: where *n* is the number of input channels on the module.

-1: set up to read the temperature sensor on the terminal block connected to the module if the temperature sensor is in the MTEMP configuration.

Description

It is important to realize that this function affects only the channel selection on the module. It does not affect the module output enable or any analog signal routing on the SCXIbus; the `SCXI_Single_Chan_Setup` function is required to do that. `SCXI_Change_Chan` can be very useful in applications like those shown in Figures 7-4 and 7-5, especially when you are trying to read several channels on a module in a loop at relatively high speeds. However, you will need to call `SCXI_Single_Chan_Setup` again if you want to select a channel on a different module.

SCXI_Configure_Filter

Function

Configures the filter on any SCXI module that supports programmable filter settings. Currently, only the SCXI-1122 and SCXI-1141 have programmable filter settings; the other analog input modules have hardware-selectable filters.

Synopsis

C Syntax	`locus i32 SCXI_Configure_Filter(u32 chassisID, u32 moduleSlot, i32 channel, u32 filterMode, f64 freq, u32 cutoffDivDown, u32 outClkDivDown, f64 *actualFreq);`
Pascal Syntax	`function SCXI_Configure_Filter(chassisID : i32; moduleSlot : i32; channel : i32; filterMode : i32; freq : f64; cutoffDivDown : i32; outClkDivDown : i32; var actualFreq : f64) : i32;`
BASIC Syntax	`FN SCXI_Configure_Filter(chassisID&, moduleSlot&, channel&, filterMode&, freq#, cutoffDivDown&, outClkDivDown&, actualFreq&)`

Parameters

channel is the module channel for which you want to change the filter configuration If **channel** = -1, `SCXI_Configure_Filter` changes the filter configuration for all channels on the module.

filterMode indicates the filter configuration mode for the given **channel**:

0: Bypass the filter.

1: Set filter cutoff frequency to **freq**.

2: Configure the filter to use an external signal. The module divides the external signal by **cutoffDivDown** to determine the filter cutoff frequency. The module also divides the external signal by **outClkDivDown** and sends it to the module front connector OUTCLK pin. You can use this filter mode to configure a tracking filter. This mode is supported by the SCXI-1141 only.

3: Enable the filter. Reverse of 0.

freq is the cutoff frequency you want to select from the frequencies available on the module if **filterMode** = 1.

The SCXI-1122 has two possible cutoff frequencies:

4.0: -10 dB at 4 Hz

4,000.0: -3 dB at 4 kHz

The SCXI-1141 has a range of cutoff frequencies from 10 Hz to 25 kHz. `SCXI_Configure_Filter` produces the frequency you want as closely as possible by dividing an internal 10 MHz signal on the SCXI-1141. The function returns the exact cutoff frequency produced in the output parameter **actualFreq**.

If **filterMode** = 2, you should set **freq** to the *approximate* frequency of the external signal you will be using. Chapter 2 in the *SCXI-1141 User Manual* explains the impact of different signal frequencies on the filters.

If **filterMode** = 0, NI-DAQ ignores **freq**.

cutoffDivDown is an integer that the module divides the external signal by to determine the filter cutoff frequency when **filterMode** = 2. NI-DAQ ignores this parameter if **filterMode** is not 2.
Range: 2 to 65,535

outClkDivDown is an integer by which the module divides either the internal 10 MHz signal (if **filterMode** = 1) or the external signal (if **filterMode** = 2) to send back to the module front connector OUTCLK pin. This parameter is only used for the SCXI-1141.
Range: 2 to 65,535

actualFreq returns the actual cutoff frequency that the module will use.

Description

The SCXI-1122 has one filter setting applied to all channels on the module; therefore, you must set **channel** = -1. The SCXI-1122 only works with **filterMode** = 1; you cannot configure the SCXI-1122 to bypass the filter or to use an external signal to set the cutoff frequency. The default frequency setting for the SCXI-1122 is 4 Hz.

The SCXI-1141 also has one filter setting applied to all channels, so you must use **channel** = -1 when you select a cutoff frequency for that module. After you select the cutoff frequency for the entire module, you can configure one or more of the channels to enable the filter by calling `SCXI_Configure_Filter` again for each channel and setting **filterMode** = 3. By default, all the channel filters on the SCXI-1141 are bypassed.

SCXI_Get_Chassis_Info

Function

Returns current chassis configuration information.

Synopsis

C Syntax	`locus i32 SCXI_Get_Chassis_Info(u32 chassisID, u16 *chassisType, u16 *chassisAddr, u16 *commMode, u16 *commPath, u16 *numDeviceNumbers);`
Pascal Syntax	`function SCXI_Get_Chassis_Info(chassisID : i32; var chassisType : i16; var chassisAddr : i16; var commMode : i16; var commPath : i16; var numDeviceNumbers : i16) : i32;`
BASIC Syntax	`FN SCXI_Get_Chassis_Info(chassisID&, chassisType&, chassisAddr&, commMode&, commPath&, numDeviceNumbers&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

chassisType indicates what type of SCXI chassis is configured for the given **chassisID**.
0: SCXI-1000 4-slot chassis.
1: SCXI-1001 12-slot chassis.

chassisAddr is the hardware jumpered address of an SCXI chassis.
Range: 0 to 31.

commMode is the Communication mode that will be used when the driver communicates with the SCXI chassis and modules.

0: Communication mode is disabled. In effect, the chassis is disabled.
1: The SCXI-1000 and SCXI-1001 chassis support only one mode of communication–serial communication through a digital output port of a DAQ board that is cabled to a module in the chassis.

commPath is the communication path that will be used when the driver communicates with the SCXI chassis and modules. If **commMode** = 1, the **commPath** should be the device number of the DAQ board that is the designated communicator for the chassis. When **commMode** = 0, **commPath** is meaningless.

numDeviceNumbers is the number of plug-in module slots in the SCXI chassis.
4: for the SCXI-1000 chassis.
12: for the SCXI-1001 chassis.

Note to C Programmers: **chassisType, chassisAddr, commMode, commPath,** ***and*** **numDeviceNumbers** ***must*** ***be passed by reference.***

SCXI_Get_Module_Info

Function

Returns current configuration information for the given chassis slot number.

Synopsis

C Syntax	`locus i32 SCXI_Get_Module_Info(u32 chassisID, u32 moduleSlot, i32 *moduleType, u16 *opMode, u16 *DAQboard);`
Pascal Syntax	`function SCXI_Get_Module_Info(chassisID : i32; moduleSlot : i32; var moduleType : i32; var opMode : i16; var DAQboard : i16) : i32;`
BASIC Syntax	`FN SCXI_Get_Module_Info(chassisID&, moduleSlot&, moduleType&, opMode&, DAQboard&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the module.
Range: 1 to *n*, where *n* is the number of slots in the chassis.

moduleType indicates what type of module is present in the given slot.
-1: Empty slot; there is no module present in the given slot.
2: SCXI-1121.
4: SCXI-1120.
6: SCXI-1100.
8: SCXI-1140.
10: SCXI-1122
12: SCXI-1160.
14: SCXI-1161.
16: SCXI-1162.
18: SCXI-1163.
20: SCXI-1124.
24: SCXI-1162HV.
28: SCXI-1163R.
30: SCXI-1102.
32: SCXI-1141.

Any other return value for **moduleType** indicates that an unfamiliar module is in the given slot.

operatingMode indicates whether the module present in the given slot is being operated in Multiplexed or Parallel mode. Please refer to the *SCXI Operating Modes* section at the beginning of the chapter for an explanation of each operating mode. If the slot is empty the **operatingMode** is meaningless.

0: Multiplexed operating mode.
1: Parallel operating mode.
2: Parallel with secondary cable of the NB-DIO-96 and PCI-DIO-96.

DAQboard is the device number of the DAQ board in the Macintosh that is cabled to the module present in the given slot. If the slot is empty, **DAQboard** is meaningless.

0: no DAQ board is cabled to the module.
n: where *n* is the device number of the DAQ board cabled to the module.

Note to C Programmers: moduleType, operatingMode, *and* DAQboard *must be passed by reference.*

SCXI_Get_State

Function

Returns the state of a single channel or an entire port on any digital or relay module.

Synopsis

C Syntax	`locus i32 SCXI_Get_State(u32 chassisID, u32 moduleSlot, u32 port, i32 channel, u32 *data);`
Pascal Syntax	`function SCXI_Get_State(chassisID : i32; moduleSlot : i32; port : i32; channel : i32; var data : i32) : i32;`
BASIC Syntax	`FN SCXI_Get_State(chassisID&, moduleSlot&, port&, channel&, data&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the module.

1 to *n*: where *n* is the number of slots in the chassis.

port is the port number of the module that is read from. All SCXI modules support Port 0 only.

channel is the channel number on the specified port. Because all the modules support port 0 only, the channel number maps to the actual channel on the module. If **channel** = -1, the function reads the state pattern from the entire port.

Range: 0 through 15 for the SCXI-1160.
0 through 7 for the SCXI-1161.
0 through 31 for the SCXI-1162.
0 through 31 for the SCXI-1162HV.
0 through 31 for the SCXI-1163.
0 through 31 for the SCXI-1163R.
-1 to read from an entire port.

When **channel=** -1, **data** contains the pattern of an entire port. Bit 0 corresponds to the state of channel 0 in the port, and the states of the other channels are represented in ascending order in **data** so that bit n corresponds to channel n. If the port is less than 32 bits wide, the unused bits in **data** are set to zero.

When **channel=** n, the LSB (bit 0) of **data** contains the state of channel n on th specified port.

For relay modules, a 0 bit indicates that the relay is closed or in the normally closed position, and a 1 indicates that the module is open or in the normally open position. For SCXI digital modules, a 0 bit indicates that the line is low, and a 1 bit indicates that the line is high.

Note: ***For more information about the Normally Closed (NC) and Normally Open (NO) positions, refer to your SCXI user manual.***

Note to C Programmers: data ***must be passed by reference.***

Description

The SCXI-1160 is a latching module, that is, the module powers up with its relays in the position they were in at power down. Thus, the states of the relays are unknown at program execution. The driver remembers the state of a relay as soon as a hardware write takes place.

The SCXI-1161 is a nonlatching module, and powers up with its relays in the NC position. As with the SCXI-1160, after calling `SCXI_Load_Config` or `SCXI_Set_Config`, you must complete an actual hardware write to the relays for the driver to return state information for the relays. Do this by calling `SCXI_Reset`.

The SCXI-1162 and SCXI-1162HV are optically isolated digital output modules. The states are read from hardware.

The SCXI-1163 and SCXI-1163R are optically isolated digital output modules with 32 digital output channels and 32 solid-state relay channels, respectively. You can read the states of the digital output channels from one of the modules only if you have jumper-configured the SCXI-1163 and SCXI-1163R to operate in Parallel mode. When you operate the module in Multiplexed mode, the driver holds the states of the digital output lines in memory. Consequently, you must complete a hardware write before the driver can return the states on the module.

On the SCXI-1163 and SCXI-1163R in Parallel mode, the states are read from hardware. On both the SCXI-1160 and SCXI-1161, a software copy of the relay states is kept in memory by the driver.

We recommend that you call `SCXI_Reset` after calling `SCXI_Set_Config` or `SCXI_Load_Config` for the SCXI-1160, SCXI-1161, SCXI-1163, and SCXI-1163R modules.

SCXI_Get_Status

Function

Returns the data in the Status Register of the specified module. This function supports the SCXI-1160, SCXI-1102, SCXI-1122, or SCXI-1124 modules.

Synopsis

C Syntax	`locus i32 SCXI_Get_Status(u32 chassisID, u32 moduleSlot, u32 wait, u32 *data);`
Pascal Syntax	`function SCXI_Get_Status(chassisID : i32; moduleSlot : i32; wait : i32; var data : i32) : i32;`
BASIC Syntax	`FN SCXI_Get_Status(chassisID&, moduleSlot&, wait&, data&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the module.
moduleSlot = 1 to *n*, where *n* is the number of slots in the chassis.

wait determines if the function should poll the Status Register on the module, until the module is ready or timeout is reached. If the module does not become ready by timeout, NI-DAQ returns a timeout error.
1: the function will poll the status register on the module, until ready or timeout.
0: he function will read and return the status register on the module.

data contains the contents of the Status Register.

0: the module is busy. You should not perform any further operations on the module until the status bit goes high again. This means the SCXI-1122 or SCXI-1160 relays are still switching, the SCXI-1124 DACs are still settling, or the SCXI-1102 filters are still settling after the gain setting was changed.

1: the module is ready; the SCXI-1122 or SCXI-1160 relays are finished switching or the SCXI-1124 DACs are settled, or the SCXI-1102 filters have settled.

Description

If **wait** = 1, the function will wait for the module status to be ready. If timeout occurs while the Status Register is being polled, the current value of the Status Register is returned in the output parameter data.

The SCXI-1102, SCXI-1122, SCXI-1160, and SCXI-1124 Status Registers contain only one bit, so only the least significant bit of the **data** parameter is meaningful.

SCXI_Load_Config

Function

Loads the SCXI chassis configuration information that was established in the configuration utility. Sets the software states of the chassis and the modules present to their default states. No changes are made to the hardware state of the SCXI chassis or the SCXI modules. This function is called automatically when the Macintosh boots up. Thereafter, it is only needed if you have changed the configuration using the `SCXI_Set_Config` function and you want to revert to the configuration in the `NI-DAQ` Control Panel.

Synopsis

C Syntax	`locus i32 SCXI_Load_Config(u32 chassisID);`
Pascal Syntax	`function SCXI_Load_Config(chassisID : i32) : i32;`
BASIC Syntax	`FN SCXI_Load_Config(chassisID&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

Description

It is important to realize that this function makes no change to the hardware. To reset the hardware to its default state, you should use the `SCXI_Reset` or function. Refer to the `SCXI_Reset` function description for a listing of the default states of the chassis and modules.

It is possible to programmatically change the configuration that was established in the configuration utility using the `SCXI_Set_Config` function.

SCXI_MuxCtr_Setup

Function

Enables or disables a DAQ device counter to be used as a multiplexer counter during SCXI channel scanning to synchronize the MIO board scan list with the module scan list that NI-DAQ has downloaded to Slot 0 of the SCXI chassis.

Synopsis

C Syntax	`locus i32 SCXI_MuxCtr_Setup(u32 deviceNumber, u32 enable, u32 scanDiv, u32 muxCtrValue);`
Pascal Syntax	`function SCXI_MuxCtr_Setup(deviceNumber : i32; enable : i32; scanDiv : i32; muxCtrValue : i32) : i32;`
BASIC Syntax	`FN SCXI_MuxCtr_Setup(deviceNumber&, enable&, scanDiv&, muxCtrValue&)`

Parameters

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

enable indicates whether or not to enable a board counter to be a mux counter for subsequent SCXI channel scanning operations.

0: disable the mux counter; a board counter is freed.
1: enable a board counter to be a mux counter.

scanDiv indicates whether or not the mux counter will divide the scan clock during the acquisition.

0: the mux counter will not divide the scan clock; it will simply pulse after every n mux-gain entries on the DAQ board, where n is the **muxCtrValue**. The mux counter pulses are currently not used by the SCXI chassis or modules, so this mode is not yet useful.
1: the mux counter will divide the scan clock so that *n* conversions are performed for every mux-gain entry on the DAQ board, where *n* is the **muxCtrValue**.

muxCtrValue is the value to be programmed into the mux counter. If **enable** = 1 and **scanDiv** =1, **muxCtrValue** is the number of conversions to be performed on each mux-gain entry on the DAQ board. If **enable** = 0, this parameter is ignored.

Description

This function can be used to synchronize the scan list that has been loaded into the mux-gain memory of the DAQ board and the SCXI module scan list that has been loaded into Slot 0 of the SCXI chassis. The total number of samples to be taken in one pass through each scan list should be the same.

For example, for the following module scan list and NB-MIO-16X scan list, a **muxCtrValue** of 8 would cause eight samples to be taken for each NB-MIO-16X scan list entry. The first two entries in the module scan list will occur during the first entry of the NB-MIO-16X scan list, at an NB-MIO-16X gain of 5. The third module scan list entry will occur during the second entry of the NB-MIO-16X scan list, at an NB-MIO-16X gain of 10. Thus, the **muxCtrValue** here is used to distribute different DAQ board gains across the module scan list, as well as to make the scan list lengths equal at 16 samples each.

Module Scan List		**NB-MIO-16X Scan List**	
Module	**Number of Samples**	**Channel**	**Gain**
2	4	0	5
3	4	0	10
4	8		

Another example would use the same module scan list as above, but only one entry in the MIO-16 scan list. The appropriate **muxCtrValue** would then be 16.

SCXI_Reset

Function

Resets the specified module to its default state.

Synopsis

C Syntax	`locus i32  SCXI_Reset(u32 chassisID, u32 moduleSlot);`
Pascal Syntax	`function  SCXI_Reset(chassisID : i32; moduleSlot : i32) : i32;`
BASIC Syntax	`FN SCXI_Reset(chassisID&, moduleSlot&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the module that is to be reset.

1 to *n*: where *n* is the number of slots in the chassis.
0: reset Slot 0 of the chassis by resetting the module scan list and scanning circuitry.
-1: reset all modules present in the chassis and reset Slot 0.

Description

The default states of the SCXI modules are as follows:

- SCXI-1100, SCXI-1102, and SCXI-1122
  Module gain = 1.
  Module filter = 4 Hz (SCXI-1122 only).
  Channel 0 is selected.
  Multiplexed channel scanning is disabled.
  Module output is enabled if the module is cabled to a DAQ board.
  Calibration is disabled (SCXI-1100 and SCXI-1122).

Note: ***With an SCXI-1102, this function will not return until the module's gain amplifier has settled to within 0.01% upon a gain change. This can result in a noticeable amount of delay.***

- SCXI-1120, SCXI-1121, and SCXI-1140
  If the module is operating in Multiplexed mode:
    Channel 0 is selected.
    Multiplexed channel scanning is disabled.
    Module output is enabled if the module is cabled to a DAQ board.
    Hold count is 1 (SCXI-1140 only).
  If the module is operating in Parallel mode:
    All channels are enabled.
    Track/Hold signal is disabled (SCXI-1140 only).

- SCXI-1124
  No action.

- SCXI-1141
  If the module is in Multiplexed mode:
    Channel 0 is selected.
    Filters are bypassed.
    Muxed scanning is disabled.

Module output is enabled if module is cabled to a DAQ board.
Autozeroing is disabled.

If the module is in Parallel mode:
All channels are enabled.
Filters are bypassed.
Autozeroing is disabled.

- SCXI-1160
  All state information of the module in memory is set to unknown (see the `SCXI_Set_State` description).
  No hardware write to the module takes place.
- SCXI-1161
  All relays on the module are set to the NC position.
- SCXI-1163
  All digital lines are set to the high state.
- SCXI-1163R
  Initializes all of the solid-state relays to their open states.

SCXI_Scale

Function

Scales an array of binary data acquired from an SCXI channel to voltage. `SCXI_Scale` will use stored software calibration constants if applicable for the given module when it scales the data. The SCXI-1102, SCXI-1122, and SCXI-1141 have default software calibration constants loaded from the module EEPROM; all other analog input modules have no software calibration constants unless you follow the analog input calibration procedure outlined in the `SCXI_Cal_Constants` function description.

Synopsis

C Syntax	`locus i32 SCXI_Scale(u32 chassisID, u32 moduleSlot, i32 chan, f64 SCXIgain, f64 TBgain, u32 DAQboard, u32 DAQchan, u32 gain, u32 count, i16 *binaryArray, f64 *voltArray);`
Pascal Syntax	`function SCXI_Scale(chassisID : i32; moduleSlot : i32; chan : i32; SCXIgain : f64; TBgain : f64; DAQboard : i32; DAQchan : i32; gain : i32; count : i32; binaryArray : pi16; voltArray : pf64) : i32;`
BASIC Syntax	`FN SCXI_Scale(chassisID&, moduleSlot&, chan&, SCXIgain#, TBgain#, DAQboard&, DAQchan&, DAQgain&, count&, binaryArray&, voltArray&)`

Parameters

chan is the number of the channel on the SCXI module.
Range: 0 to *n*-1, where *n* is the number of channels available on the module.

SCXIgain is the SCXI module or channel gain setting. Valid **SCXIgain** values depend on the module type:
SCXI-1100: 1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, 2,000.
SCXI-1102: 1, 100.
SCXI-1120: 1, 2, 5, 10, 20, 50, 100, 200, 250, 500, 1,000, 2,000.
SCXI-1121: 1, 2, 5, 10, 20, 50, 100, 200, 250, 500, 1,000, 2,000.
SCXI-1122: 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, 2,000.
SCXI-1140: 1, 10, 100, 200, 500.
SCXI-1141: 1, 2, 5, 10, 20, 50, 100.

TBgain is the gain applied at the SCXI terminal block. Currently, only the SCXI-1327 terminal block can apply gain to your SCXI module channels; it has dip-switches to choose a gain of 1.0 or 0.01 for each input channel. You can use the SCXI-1327 with the SCXI-1120 and SCXI-1121 modules. For terminal blocks that do not apply gain to your SCXI channels, set **TBgain** = 1.0.

DAQboard is the device number of the DAQ board you used to acquire the binary data. This should be the same device number that you passed to the `DAQ` or `SCAN` function call, and the same **DAQboard** number you passed to `SCXI_Single_Chan_Setup` or `SCXI_SCAN_Setup`.

DAQchan is the DAQ board channel number you used to acquire the binary data. This should be the same channel number that you passed to the `DAQ` or `SCAN` function call. For most cases, you will be multiplexing all of your SCXI channels into DAQ board channel 0.

gain is the DAQ board gain you used to acquire the binary data. This should be the same gain code that you passed to the `DAQ` or `SCAN` function call. For most cases, you will use a DAQ board gain of 1, and you will set any gain you need at the SCXI module.

count is the number of data points you wish to scale for the given channel. The **binaryArray** and **voltArray** parameters must be arrays of length **count** (at least). If you acquired data from more than one SCXI channel, you must be careful to pass the number of points for this channel only, not the total number of points you acquired from *all* channels.

binaryArray is the array of binary data for the given channel. **binaryArray** should contain **count** data samples from the SCXI **chan**. If you acquired data from more than one SCXI channel, you will need to de-multiplex the binary data that was returned from the `SCAN` call before you call `SCXI_Scale`. You can use the `SCAN_Demux` call to do this. After demuxing the binary data, you should call `SCXI_Scale` once for *each* SCXI channel, passing in the appropriate demuxed binary data for each channel.

voltArray is the output array for the scaled voltage data. **voltArray** should be at least **count** elements long.

Description

`SCXI_Scale` uses the following equation to scale the binary data to voltage:

$$\mathbf{voltArray}[i] = \frac{(\mathbf{binArray}[i] - \text{binaryOffset})(\text{voltageResolution})}{(\mathbf{SCXIgain})(\mathbf{TBgain})(\mathbf{DAQgain})(\text{gainAdjust})}$$

The **voltageResolution** depends on your DAQ board and its range and polarity settings. For example, the NB-MIO-16 in bipolar mode with an input range of -10V to 10V has a voltage resolution of 4.88mV per LSB.

NI-DAQ automatically loads **binaryOffset** and **gainAdjust** parameters for the SCXI-1122 for all of its gain settings from the module EEPROM the first time you access the module using an NI-DAQ function call (such as `SCXI_Reset` or `SCXI_SCAN_Setup`). The SCXI-1102 and SCXI-1122 modules are shipped with factory calibration constants for binaryOffset and gainAdjust loaded in the EEPROM. You can calculate your own calibration constants and store them in the EEPROM and in NI-DAQ memory for `SCXI_Scale` to use. Please refer to the procedure outlined in the `SCXI_Cal_Constants` function description. The same is true for the SCXI-1141, except **binaryOffset** is not on the SCXI-1141 EEPROM and defaults to 0.0. However, you can calculate your own **binaryOffset** using the procedure outlined in the `SCXI_Cal_Constants` function description.

For other analog input modules, **binaryOffset** defaults to 0.0 and **gainAdjust** defaults to 1.0. However, you can calculate your own calibration constants and store them in NI-DAQ memory for NI-DAQ to use in the `SCXI_Scale` function by following the procedure outlined in the `SCXI_Cal_Constants` function description.

SCXI_SCAN_Setup

Function

Sets up the SCXI chassis for a multiplexed scanning data acquisition to be performed by the given DAQ board. Modules may be scanned in any order; channels on each module must be scanned in consecutive order. A module scan list is downloaded to Slot 0 in the SCXI chassis that will determine the sequence of modules that will be scanned and how many channels on each module will be scanned. Each module is programmed with its given start channel.

Synopsis

C Syntax	`locus i32 SCXI_SCAN_Setup(u32 chassisID, u32 numModules, u16 *moduleList, u16 *numChansList, i16 *startChansList, u32 DAQboard, u32 scanMode);`
Pascal Syntax	`function SCXI_SCAN_Setup(chassisID : i32; numModules : i32; moduleList : pi16; numChansList : pi16; startChansList : pi16; DAQboard : i32; scanMode : i32) : i32;`
BASIC Syntax	`FN SCXI_SCAN_Setup(chassisID&, numModules&, moduleList&, numChansList&, startChansList&, DAQboard&, scanMode&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

numModules is the number of modules to be scanned, and the length of the **moduleList**, **numChansList**, and the **startChansList** arrays.

Range: 1 to 256.

moduleList is an array of length **numModules** containing the list of module slot numbers corresponding to the modules to be scanned.

moduleList[i**]** = 1 to *n*, where *n* is the number of slots in the chassis.

Any element greater than the number of slots in the chassis indicates a *dummy* entry in the module scan list. Dummy entries can be useful in multi-chassis scanning as placeholders in each Slot 0 scan list to indicate when the DAQ board is scanning channels on another chassis.

numChansList is an array of length **numModules** that indicates how many channels to scan on each module represented in the **moduleList** array. If the number of channels specified for a module exceeds the number of input channels available on the module, the channel scanning will wrap around after the last input channel and continue with the first input channel. If a module is represented more than once in the **moduleList** array, there can be different **numChansList** values for each entry. The total number of channels scanned is the sum of all the elements in this array.

numChansList[i**]** = 1 to 128.

startChansList is an array of length **numModules** that contains the start channels for each module represented in the **moduleList** array. If a module is represented more than once in the **moduleList** array, the corresponding elements in the **startChansList** array should contain the same value; *there can only be one start channel for each module*. If the temperature sensor is chosen as the start channel for a module, all readings from that module will be readings of the temperature sensor; channel scanning is not possible.

startChansList[i**]** = 0 to *n*-1, where n is the number of input channels available on the corresponding module.

startChansList[i**]** = -1 select the temperature sensor on the terminal block.

DAQboard is the device of the DAQ board that performs the channel scanning operation.

scanMode indicates the scanning mode to be used. Only one scanning mode is currently supported, so this parameter should always be set to zero.

Note: ***The SCXI-1122 uses relays to switch the input channels; the relays require 10 ms to switch, so the sampling rate in a channel scanning operation cannot exceed 100 Hz. If you want to take many readings from each channel and average them to reduce noise, you should use the single-channel or software scanning method shown in Figure 7-3 instead of the channel-scanning method shown in Figure 7-5. This means you select one channel on the module, acquire many samples on that channel using the DAQ functions, select the next channel, and so on. This will increase the lifetime of your module relays. Once you have selected a particular channel, you can use the fastest sample rate your DAQ board supports with the DAQ functions.***

SCXI_Set_Config

Function

Allows you to change the software configuration of the SCXI chassis that was established in the configuration utility. Sets the software states of the chassis and the modules specified to their default states. No changes are made to the hardware state of the SCXI chassis or the SCXI modules.

Synopsis

C Syntax	`locus i32 SCXI_Set_Config(u32 chassisID, u32 chassisType, u32 chassisAddr, u32 commMode, u32 commPath, u32 numSlots, i32 *modules, u16 *opModes, u16 *DAQboardMap);`
Pascal Syntax	`function SCXI_Set_Config(chassisID : i32; chassisType : i32; chassisAddr : i32; commMode : i32; commPath : i32; numSlots : i32; modules : pi32; opModes : pi16; DAQboardMap : pi16) : i32;`
BASIC Syntax	`FN SCXI_Set_Config(chassisID&, chassisType&, chassisAddr&, commMode&, commPath&, numSlots&, modules&, opModes&, DAQboardMap&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

chassisType indicates what type of SCXI chassis is configured for the given **chassisID**.

0: SCXI-1000 4-slot chassis.
1: SCXI-1001 12-slot chassis.

chassisAddr is the hardware jumpered address of an SCXI chassis.

Range: 0 to 31.

commMode is the Communication mode that will be used when the driver communicates with the SCXI chassis and modules.

0: Communication mode is disabled. In effect, this disables the chassis.
1: The SCXI-1000 and SCXI-1001 chassis support only one mode of communication–serial communication through a digital output port of a DAQ board that is cabled to a module in the chassis.

commPath is the communication path that will be used when the driver communicates with the SCXI chassis and modules. When **commMode** = 1, the path should be the device number of the DAQ board that is the designated communicator for the chassis. If only one DAQ board is connected to the chassis, the **commPath** should be the device number of that board. If more than one DAQ board is connected to modules in the chassis, one board must be designated as the communicator board, and its device number should be the **commPath**. Refer to the **DAQboardMap** array description below; the **commPath** should be one of the slot numbers specified in that array. When **commMode** = 0, **commPath** is ignored.

numSlots is the number of plug-in module slots in the SCXI chassis.
- 4: for the SCXI-1000 chassis.
- 12: for the SCXI-1001 chassis.

modules is an array of length **numSlots** that indicates what type of module is present in each slot. The first element of the array corresponds to Slot 1 of the chassis, and so on.

modules[*i*] =	-1:	Empty slot; there is no module present in the corresponding slot.
modules[*i*] =	2:	SCXI-1121.
modules[*i*] =	4:	SCXI-1120.
modules[*i*] =	6:	SCXI-1100.
modules[*i*] =	8:	SCXI-1140.
modules[*i*] =	10:	SCXI-1122.
modules[*i*] =	12:	SCXI-1160.
modules[*i*] =	14:	SCXI-1161.
modules[*i*] =	16:	SCXI-1162.
modules[*i*] =	18:	SCXI-1163.
modules[*i*] =	20:	SCXI-1124.
modules[*i*] =	24:	SCXI-1162HV.
modules[*i*] =	28:	SCXI-1163R.
modules[*i*] =	30:	SCXI-1102.
modules[*i*] =	32:	SCXI-1141.

Any other value for **modules** indicates a module that is unfamiliar to NI-DAQ.

opModes is an array of length **numSlots** that indicates the operating mode of each module in the **modules** array: multiplexed or parallel. Please refer to the *SCXI Operating Modes* section at the beginning of the chapter for an explanation of each operating mode. If any of the slots are empty (indicated by a value of -1 in the corresponding element of the **modules** array), the corresponding element in the **opModes** array is ignored.

opModes[*i*] =	0:	Multiplexed operating mode.
opModes[*i*] =	1:	Parallel operating mode.
opModes[*i*] =	2:	Parallel operating mode with secondary connector of the DAQ board.

DAQboardMap is an array of length **numSlots** that describes the connections between the SCXI chassis and the DAQ boards in the Macintosh. For each module present in the chassis, you must specify the device number of the DAQ board that is cabled to the module, if there is one. If any of the slots are empty (indicated by a value of -1 in the corresponding element of the **modules** array), the corresponding element of the **DAQboardMap** array is ignored. The **commPath** parameter value must be one of the data acquisition device numbers specified in this array.

DAQboardMap[*i*] =	0:	no DAQ board is cabled to the module.
DAQboardMap[*i*] =	*n:*	where *n* is the device number of the DAQ board cabled to the module.[[

Description

The configuration information that was saved by the configuration utility will remain unchanged; only the configuration in the current session is changed. Any subsequent calls to `SCXI_Load_Config` will reload the configuration from the configuration utility.

Remember, the hardware state of the chassis is not affected by this function; you should use the `SCXI_Reset` function to reset the hardware states. Refer to the `SCXI_Reset` function description for a listing of the default states of the chassis and modules.

SCXI_Set_Gain

Function

Sets the specified channel to the given gain setting on any SCXI module that supports programmable gain settings. Currently, the SCXI-1100, SCXI-1102, SCXI-1122, and SCXI-1141 have programmable gains; the other analog input modules have hardware-selectable gains.

Synopsis

C Syntax	`locus i32 SCXI_Set_Gain(u32 chassisID, u32 moduleSlot, i32 channel, f64 SCXIgain);`
Pascal Syntax	`function SCXI_Set_Gain(chassisID : i32; moduleSlot : i32; channel : i32; SCXIgain : f64) : i32;`
BASIC Syntax	`FN SCXI_Set_Gain(chassisID&, moduleSlot&, channel&, SCXIgain#)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the module.
Range: 1 to *n*, where *n* is the number of slots in the chassis.

channel is the module channel you wish to change the gain setting for. If **channel** = -1, `SCXI_Set_Gain` will change the gain for all channels on the module. The SCXI-1100 and SCXI-1122 have one gain amplifier, so all channels have the same gain setting; therefore, you must set **channel** = -1 for those modules.

SCXIgain is the gain setting you wish to use. Notice that **SCXIgain** is a double-precision floating point parameter. Valid gain settings depend on the module type:
SCXI-1100: 1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, 2,000.
SCXI-1102: 1, 100.
SCXI-1122: 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, 2,000.
SCXI-1141: 1, 2, 5, 10, 20, 50, 100.

Description

With an SCXI-1102, this function will not return until the module's gain amplifier has settled to within 0.01% upon a gain change. This can result in a noticeable amount of delay.

SCXI_Set_Input_Mode

Function

Configures the SCXI-1122 channels for two-wire mode or four-wire mode.

Synopsis

C Syntax	`locus i32 SCXI_Set_Input_Mode(u32 chassisID, u32 moduleSlot, u32 inputMode);`
Pascal Syntax	`function SCXI_Set_Input_Mode(chassisID : i32; moduleSlot : i32; inputMode : i32) : i32;`
BASIC Syntax	`FN SCXI_Set_Input_Mode(chassisID&, moduleSlot&, inputMode&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the module.

Range: 1 to *n*, where *n* is the number of slots in the chassis.

inputMode is the channel configuration you wish to use.
0: two-wire mode (module default)
1: four-wire mode

Description

When the SCXI-1122 is in two-wire mode (module default setting), the module is configured for 16 differential input channels.

When the SCXI-1122 is in four-wire mode, channels 0 through 7 are configured to be differential input channels, and channels 8 through 15 are configured to be current excitation channels. The SCXI-1122 has a current excitation source that will switch to drive the corresponding excitation channel 8 through 15 whenever an input channel 0 through 7 is selected. Channel 8 will produce the excitation when input channel 0 is selected, channel 9 will produce the excitation when input channel 1 is selected, and so on. You can use four-wire mode for single point data acquisition, or for multiple channel scanning acquisitions. During a multiple channel scan, the excitation channels will switch simultaneously with the input channels.

You can hook up an RTD or thermistor to your input channel that uses the corresponding excitation channel to drive the transducer.

You can call the `SCXI_Set_Input_Mode` function to enable four-wire mode at any time before you start the acquisition; and you can call `SCXI_Set_Input_Mode` again after the acquisition to return the module to normal two-wire mode.

SCXI_Set_State

Function

Sets the state of a single channel, or an entire port on any digital output or relay module.

Synopsis

C Syntax	`locus i32 SCXI_Set_State(u32 chassisID, u32 moduleSlot, u32 port, i32 channel, u32 data);`
Pascal Syntax	`function SCXI_Set_State(chassisID : i32; moduleSlot : i32; port : i32; channel : i32; data : i32) : i32;`
BASIC Syntax	`FN SCXI_Set_State(chassisID&, moduleSlot&, port&, channel&, data&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the module.
Range: 1 to *n*, where *n* is the number of slots in the chassis.

port is the port number of the module that is written to. All SCXI modules currently support Port 0 only.

channel is the channel number on the specified port. Because all the modules currently support Port 0 only, the channel number maps to the actual channel on the module. If **channel** = -1, the function writes the pattern contained in **data** to the entire port.

Range: 0 through 15 for the SCXI-1160.
0 through 7 for the SCXI-1161.
0 through 31 for the SCXI-1163.

0 through 31 for the SCXI-1163R.
-1 to write to an entire port.

When **channel=** -1, **data** contains the pattern of an entire port. Bit 0 corresponds to the state of channel 0 in the port, and the states of the other channels are represented in ascending order in **data** so that bit n corresponds to channel n. If the port is less than 32 bits wide, the unused bits in **data** are ignored.

When **channel=** n, the LSB (bit 0) of **data** contains the state of channel n on the specified port.

For relay modules, a 0 bit indicates that the relay is closed or in the normally closed position, and a 1 indicates that the module is open or in the normally open position. For SCXI digital modules, a 0 bit indicates that the line is low, and a 1 bit indicates that the line is high.

Note: ***For more information about the Normally Closed (NC) and Normally Open (NO) positions, refer to your SCXI module user manual.***

Description

Because the relays on the SCXI-1160 module have a finite lifetime, the driver will maintain a software copy of the relay states as you write to them; this allows the driver to excite the relays only when you specify a *new* relay state. If you call this function to specify the current relay state again, NI-DAQ will not excite the relay again. When the SCXI-1160 powers up, the relays remain in the same position as they were at power down. However, when you start an application, the driver does not know the states of the relays; it will excite all of the relays the first time you write to them and then remember the states for the remainder of the application. When you call the `SCXI_Reset` function, the driver will mark all relay states as unknown.

The SCXI-1161 powers up with its relays in the NC position. The SCXI-1163 powers up with its output lines high when you operate the module in Multiplexed mode. The SCXI-1163R powers up with its relays open. If you operate the SCXI-1163 or the SCXI-1163R in Parallel mode, the states of the output lines or relays are determined by the states of the corresponding lines on the DAQ board.

SCXI_Single_Chan_Setup

Function

Sets up a multiplexed module for a single channel analog input operation to be performed by the given DAQ board. Sets the module channel, enables the module output, and routes the module output on the SCXIbus if necessary. Resolves any contention on the SCXIbus by disabling the output of any module that was previously driving the SCXIbus. This function can also be used to read the temperature sensor on a terminal block connected to the front of a module.

Synopsis

C Syntax	`locus i32 SCXI_Single_Chan_Setup(u32 chassisID, u32 moduleSlot, i32 channel, u32 DAQboard);`
Pascal Syntax	`function SCXI_Single_Chan_Setup(chassisID : i32; moduleSlot : i32; channel : i32; DAQboard : i32) : i32;`
BASIC Syntax	`FN SCXI_Single_Chan_Setup(chassisID&, moduleSlot&, channel&, DAQboard&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the module that is to be read.
Range: 1 to *n*, where *n* is the number of slots in the chassis.

channel is the channel number of the input channel on the module that is to be read.
Range: 0 to *n*-1, where *n* is the number of input channels on the module.
-1: set up to read the temperature sensor on the terminal block connected to the module if the temperature sensor is in the MTEMP configuration.

DAQboard is the device number of the DAQ board in the Macintosh that will be used to read the input channel.

SCXI_Track_Hold_Control

Function
Controls the Track/Hold state of an SCXI-1140 module that has been set up for a single-channel operation.

Synopsis

C Syntax	`locus i32 SCXI_Track_Hold_Control(u32 chassisID, u32 moduleSlot, u32 state, u32 DAQboard);`
Pascal Syntax	`function SCXI_Track_Hold_Control(chassisID : i32; moduleSlot : i32; state : i32; DAQboard : i32) : i32;`
BASIC Syntax	`FN SCXI_Track_Hold_Control(chassisID&, moduleSlot&, state&, DAQboard&)`

Parameters
chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the desired SCXI-1140 module.
Range: 1 to *n*, where *n* is the number of slots in the chassis.

state indicates whether to put the module into Track or Hold mode.
0: put the module into Track mode.
1: put the module into Hold mode.

DAQboard is the device number of the DAQ board in the Macintosh that will be used to read the input channel.

Description
Please refer to the *SCXI Applications* discussion at the beginning of this chapter for information about how to use the SCXI-1140 for single-channel and channel-scanning operations. This function is only needed for single-channel applications; the scan interval timer controls the Track/Hold state of the SCXI-1140 during a channel-scanning operation. Figures 7-2 and 7-5 show flowcharts for single-channel applications, which use the SCXI-1140 with this function.

SCXI_Track_Hold_Setup

Function
Establishes the track/hold behavior of an SCXI-1140 module, and sets up the module for either a single-channel operation or an interval-scanning operation.

Synopsis

C Syntax	`locus i32 SCXI_Track_Hold_Setup(u32 chassisID, u32 moduleSlot, u32 inputMode, u32 source, u32 send, u32 holdCount, u32 DAQboard);`
Pascal Syntax	`function SCXI_Track_Hold_Setup(chassisID : i32; moduleSlot : i32; inputMode : i32; source : i32; send : i32; holdCount : i32; DAQboard : i32) : i32;`
BASIC Syntax	`FN SCXI_Track_Hold_Setup(chassisID&, moduleSlot&, inputMode&, source&, send&, holdCount&, DAQboard&)`

Parameters

chassisID is the logical ID that was assigned to the SCXI chassis in the configuration utility.

moduleSlot is the chassis slot number of the SCXI-1140 module.
Range: 1 to *n*, where *n* is the number of slots in the chassis.

inputMode indicates what type of analog input operation.
0: none; any resources that were previously reserved for the module (such as a DAQ board counter or an SCXIbus trigger line) are freed.
1: single-channel operation.
2: interval channel-scanning operation.

source indicates what signal will control the Track/Hold state of the module. If the **inputMode** is zero, this parameter is ignored.
0: A counter of the DAQ board that is cabled to the module will be the source (Am9513-based MIO-16 Counter 2, an E Series dedicated DAQ-STC counter, a Lab and 1200 series Counter B1, or DAQCard-700 Counter 2 will be reserved and used for this purpose). This **source** is only valid if the module is cabled to a DAQ board.
1: An external source connected to the HOLDTRIG pin on the front connector of the module will control the Track/Hold state of the module. There is a hardware connection between the HOLDTRIG pin and the counter output of the DAQ board, so if **source**=1 the appropriate counter (listed above) is driven by the external source and will be reserved. Note that if **inputMode** = 2, this external source will drive the scan interval timer. If you are using a Lab or 1200 series board or DAQCard-700, you must change the jumper setting on the SCXI-1341 adapter card to prevent the external source from damaging the timer chip on the DAQ board. This mode is not supported for the NB-MIO-16.
2: A signal routed on an SCXIbus trigger line from another SCXI-1140 module will be used to control the Track/Hold state of the module.

send indicates where else to send the signal specified by **source** for synchronization purposes. This parameter is also ignored if the **inputMode** is zero.
0: nowhere.
1: make the **source** signal drive the DAQ board counter output and the HOLDTRIG pin on the module front connector (if the **source** is not already one of those signals). If you are using a Lab or 1200 series board or DAQCard-700, you must change the jumper setting on the SCXI-1341 adapter card to prevent the external signal source from damaging the timer chip on the DAQ board.
2: make the **source** signal drive an SCXIbus trigger line so that other SCXI-1140 modules can use it (if the **source** is not from the SCXIbus). Only one SCXI-1140 module can drive that trigger line; an error will occur if you attempt to configure more than one SCXI-1140 to drive it.

holdCount is the number of times the module is enabled during an interval scan before going back into Track mode. Each time Slot 0 encounters an entry for the module in the module scan list, the module is enabled, and it remains enabled until the sample count in that module scan list entry expires. If there is only one entry for the module in the module scan list, **holdCount** should be one (this will almost always be the case.)
Range: 1 to 256

DAQboard is the device number of the DAQ board in the Macintosh that will be used in the analog input operation. If the **DAQboard** specified is a Lab-NB or a Lab-LC, **inputMode** 2 is not supported.

Application Hints

For single channel operations (**inputMode**=1) the module is level-sensitive to the **source** signal; that is, when the **source** signal is low the module is in Track mode, and when the **source** signal is high the module is in Hold mode. If **source** = 0, calls to the `SCXI_Track_Hold_Control` function can be used to put the module into Track or Hold mode by toggling the output of the appropriate counter on the DAQ board (see Figure 7-2). If the SCXI-1140 you wish to read is not cabled to the DAQ board, you will have to configure the SCXI-1140 module that *is* cabled to the DAQ board to send the counter output on the SCXIbus to the desired module. Then the `SCXI_Track_Hold_Control` call will be able to put the desired module into Track or Hold mode. The `SCXI_Track_Hold_Setup` parameters for each module would be:

For the SCXI-1140 that is cabled to the DAQ board:
inputMode = 1.
source = 0.
send = 2.
For the SCXI-1140 module to be read:
inputMode = 1.
source = 2.
send = 0.

Using an external signal source (**source**=1) for single channel operations is not normally useful because NI-DAQ has no way of determining when the module has gone into Hold mode and it is appropriate to read the channels.

For interval channel scanning operations (**inputMode** = 2), which is only supported for MIO boards, the module will be configured to go into Hold mode on the rising edge of the source signal. If **source**=0, that will happen when Counter 2 pulses at the beginning of each scan interval; if **source**=1, that will happen on the rising edge of the external signal connected to HOLDTRIG on the module front connector. In the latter case, the external signal will also trigger the start of each scan. If more than one SCXI-1140 is to be scanned, you can send the **source** signal from the module that is receiving it (either from Counter 2 or from HOLDTRIG) to the other modules over the SCXIbus. Note that the module that is cabled to the board can receive the **source** signal from the SCXIbus and drive the scan interval timer of the board, if desired; or it can use OUT2 from the board and send it on the SCXIbus, *even if that module is not in the module scan list.*

For example, two SCXI-1140 modules are to be scanned, one of which is cabled to an NB-MIO-16X that is to perform the acquisition. An external signal connected to the HOLDTRIG pin of the module that is not cabled to the DAQ board is to control the Track/Hold state of both modules and the scan interval during the acquisition. The `SCXI_Track_Hold_Setup` parameters would be:

For the SCXI-1140 that is cabled to the NB-MIO-16X :
inputMode = 2.
source = 2.
send = 1.
For the other SCXI-1140 module to be scanned:
inputMode = 2.
source = 1.
send = 2.

If the NB-MIO-16X is used, the module will go back into Track mode after n module scan list entries for that module have occurred, where n is the holdCount. Usually, each module is represented in the module scan list only once, so a holdCount of one is appropriate. However, if an SCXI-1140 module is represented more than once in the module scan list and you want the module to remain in Hold mode until after the last scan list entry for that module, you will need to set the module holdCount to equal the number of times the module is represented in the module scan list.

If the NB-MIO-16 is used, the holdCount has no effect and is ignored. The module always goes back into track mode at the end of each NB-MIO-16 scan interval. An external source (**source** = 1) may not be used with the NB-MIO-16.

Chapter 8
Counter/Timer Functions

This chapter describes the functions that perform timing I/O and counter operations such as pulse generation, frequency generation, and event counting. The chapter is divided into three sections to describe the Counter (CTR) functions, Interval Counter (ICTR), and General-Purpose Counter (GPCTR) functions for the National Instruments boards for the Macintosh. See Appendix A to determine which function set works with your board.

The Counter and Interval Counter functions perform timing I/O (counter) functions such as pulse generation, frequency generation, and event counting. These functions are used for frequency and pulse measurement, event counting, and timed process control.

Counter/Timer Operations (CTR Functions)

The 16-bit counters available on the NB-MIO-16, NB-MIO-16X, NB-DMA-8-G, NB-DMA2800, NB-A2000, and NB-TIO-10 can be diagrammed as shown in Figure 8-1.

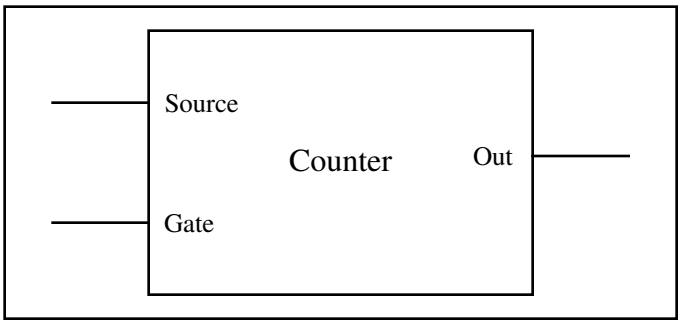

Figure 8-1. Counter Block Diagram

Each counter has a SOURCE input, a GATE input, and an output labeled OUT. The SOURCE, GATE, and OUT pins for Counters 1, 2, and 5 of the onboard Am9513 are available at the NB-MIO-16 and NB-MIO-16X I/O connectors. The SOURCE, GATE, and OUT pins for Counters 1 through 10 of the onboard Am9513 are available at the NB-TIO-10 I/O connector.

The counters can use several timebases for counting operations. A counter can use the signal supplied at any of the Am9513 five SOURCE or five GATE inputs for counting operations. The Am9513 also has five internal timebases that any counter can use. On the NB-MIO-16, NB-MIO-16X, NB-DMA2800, NB-DMA-8-G, NB-A2000, and NB-TIO-10, these timebases are as follows:

1: 1-MHz clock (1-µs resolution)

2: 100-kHz clock (10-µs resolution)

3: 10-kHz clock (100-µs resolution)

4: 1-kHz clock (1-ms resolution)

5: 100-Hz clock (10-ms resolution)

In addition, the counter can be programmed to use the output of the next lower-order counter as a signal source. This arrangement is useful for counter concatenation. For example, Counter 2 can be programmed to count the output of Counter 1, thus creating a 32-bit counter.

A counter can be configured to count either falling or rising edges of the selected internal timebase, SOURCE input, or next lower-order counter signal.

The counter GATE input is used to gate counting operations. When a counter is configured through software for an operation, a signal at the GATE input can be used to start and stop counter operation. There are nine gating modes available in the Am9513:

- No Gating — Counter is started and stopped by software.
- High-Level Gating — Counter starts when gate input is at a high-logic state. The counter is suspended when gate input is at a low-logic state.
- Low-Level Gating — Counter starts when gate input is at a low-logic start. The counter is suspended when gate input is at a high-logic state.
- Rising-Edge Gating — Counter starts counting when it receives a low-to-high edge at the gate input.
- Falling-Edge Gating — Counter starts counting when it receives a high-to-low edge at the gate input.
- Active high on terminal count of next lower-order counter — Counter starts when the TC (not toggler) output of the next lower-order counter is at a high-logic state. The counter is suspended when the TC output of the next lower-order counter is at a low-logic state.
- Active high on gate of next higher-order counter — Counter starts when the gate input of the next higher-order counter is at a high-logic state. The counter is suspended when the gate input of the next higher-order counter is at a low-logic state.
- Active high on gate of next lower-order counter — Counter starts when the gate input of the next lower-order counter is at a high-logic state. The counter is suspended when the gate input of the next lower-order counter is at a low-logic state.
- Special Gating — The gate input selects the reload source, but does not start counting. The counter uses the value stored in its internal Hold register when the gate input is high, and uses the value stored in its internal Load register when the gate input is low.

Counter operation starts and stops relative to the selected timebase. When a counter is configured for no gating, the counter starts at the first timebase/source edge (rising or falling, depending on selection) after the counter is configured by the software. When a counter is configured for gating modes, gate signals take effect at the next timebase/source edge. For example, if a counter is configured to count rising edges and to use the falling edge gating mode, the counter starts counting on the next rising edge after it receives a high-to-low edge on its GATE input. Thus, some time is spent synchronizing the GATE input with the timebase/source. This synchronization time creates a time lapse uncertainty of between 0 and 1 timebase periods between the application of the signal at the GATE input and the start of the counter operation.

The counter generates timing signals at its OUT output. If the counter is not operating, its output can be set to one of three states: high-impedance state, low-logic state, or high-logic state.

The counters generate two types of output signals during counter operation: terminal count (TC) pulse output, and TC toggle output. A counter reaches TC when it counts up to the maximum or down to 0 and rolls over. In many counter applications, the counter reloads from an internal register when it reaches TC. In TC pulse output mode, the counter generates a pulse during the cycle in which it reaches TC and reloads. In TC toggle output mode, the counter output changes state on the next source edge after it reaches TC and reloads. In addition, the counters can be configured for positive logic output or negative (inverted) logic output. Examples of the four types of output signals generated are shown in Figure 8-2.

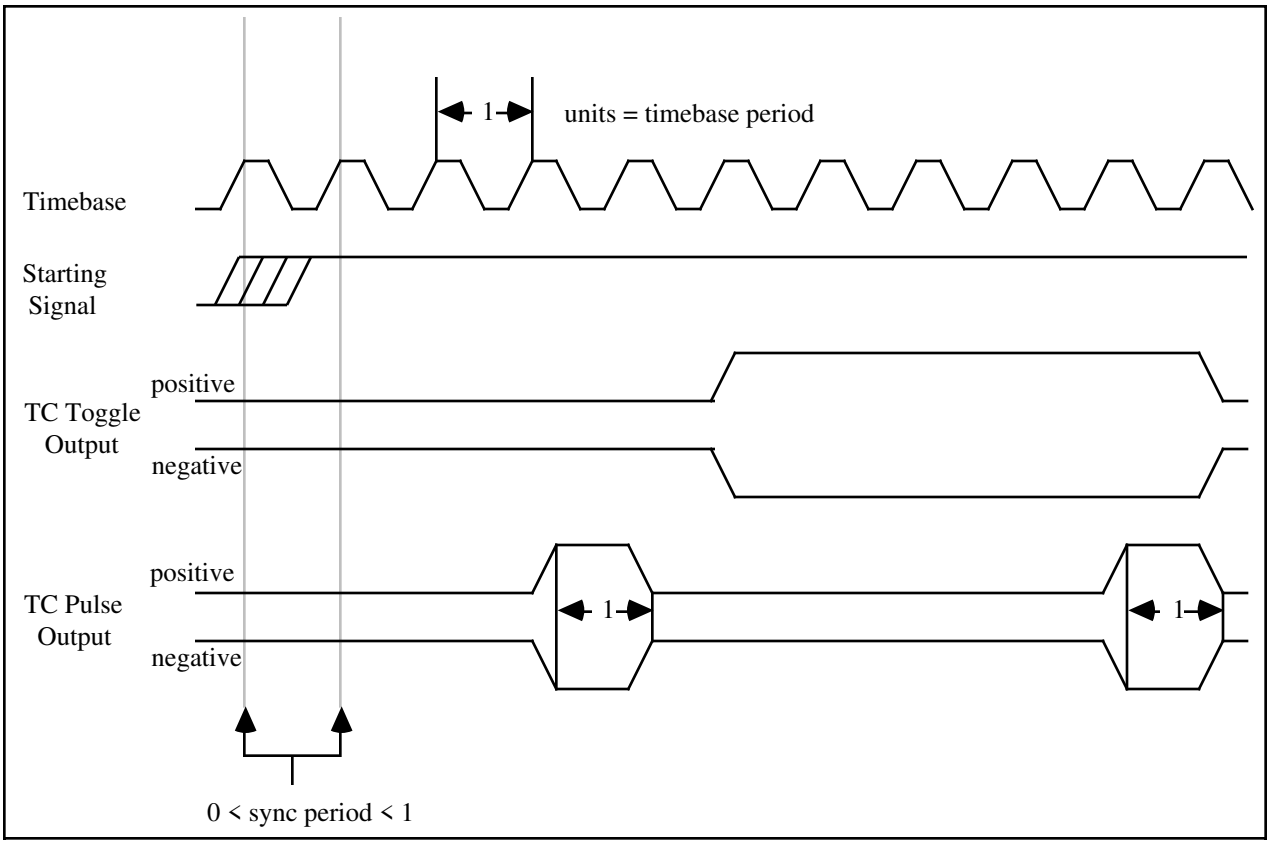

Figure 8-2. Counter Timing and Output Types

Figure 8-2 represents a counter generating a delayed pulse (see `CTR_Pulse`) and demonstrates the four forms the output pulse could take given the four different types of output signal available. The TC toggle positive logic output looks like what would be expected when generating a pulse. For most of the Counter/Timer functions, TC toggle output is the preferred output configuration; however, the other signal types are also available.

The starting signal shown in Figure 8-2 represents either a software starting of the counter (for the no-gating case), or some sort of signal at the GATE input. The signal could be either a rising-edge gate or a high-level gate. If a low-level or falling-edge gate, the starting signal simply appears inverted. In Figure 8-1, the counter is configured to count the rising edges of the timebase; therefore, the starting signal takes effect on the rising edge of the timebase, and the signal output changes state with respect to the rising edge of the timebase.

Programmable Frequency Output Operation

The NB-MIO-16, the NB-MIO-16X, NB-TIO-10, NB-DMA2800, and the NB-DMA-8-G have 4-bit programmable output signals. These signals are a divided-down version of the selected timebase. Any of five internal timebases and any of the counter SOURCE inputs can be selected as the FOUT source. (See the `CTR_FOUT_Config` function description for FOUT use and timing.)

NB-MIO-16 Counter/Timers

From an onboard Am9513 System Timing chip, the NB-MIO-16 uses three independent 16-bit counter/timers and a 4-bit programmable frequency output. The connection of the Am9513 Counter/Timer signals to the NB-MIO-16 I/O connector and to the RTSI is shown in Figure 8-3.

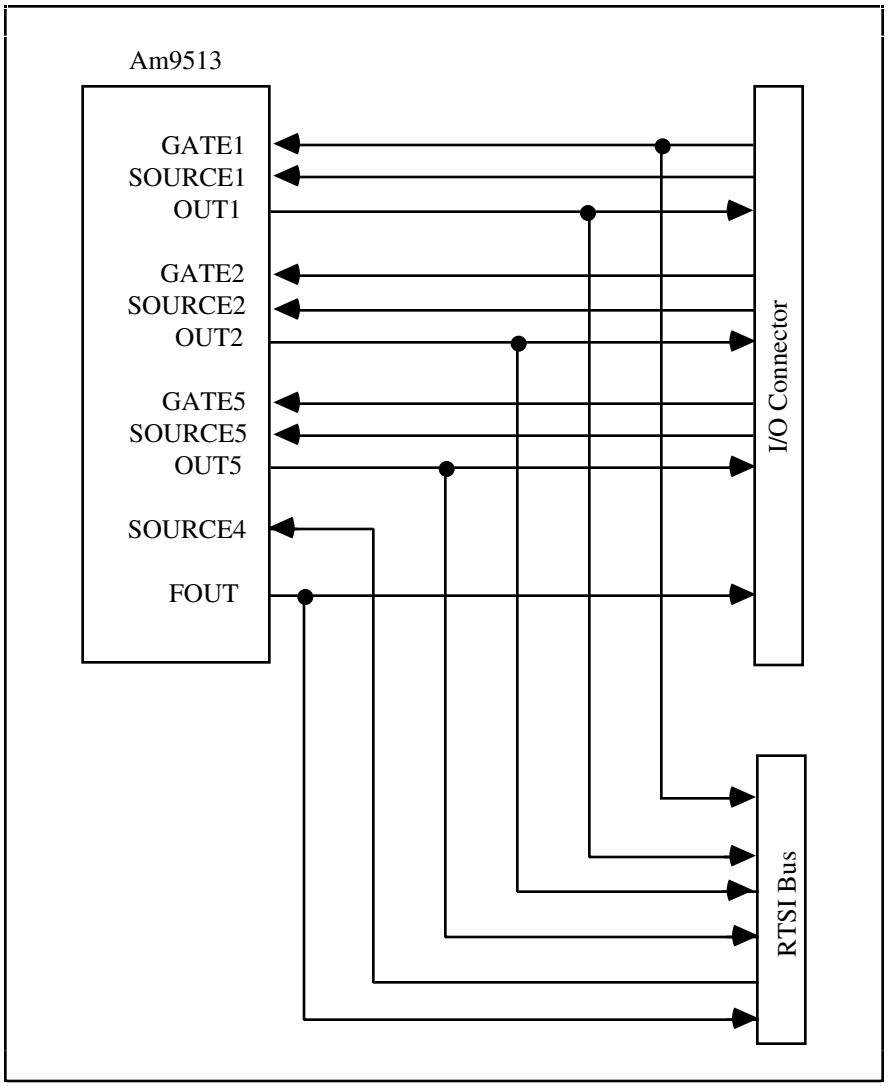

Figure 8-3. NB-MIO-16 Counter/Timer Signal Connections

Counters 1, 2, and 5 are available for counting operations on the NB-MIO-16. Counters 3 and 4 (not shown) are reserved for data acquisition operations. As shown in Figure 8-3, the FOUT, GATE, SOURCE, and OUT signals for Counters 1, 2, and 5 are available at the I/O connector. The signals GATE1, FOUT, OUT1, OUT2, and OUT5 are also available for connection to the RTSI bus trigger lines. (See Chapter 9, *RTSI Bus Trigger Functions*, for more information.) SOURCE4 can be driven from the RTSI bus.

Although Counter 1 and Counter 5 are made available for general use, they are reserved by NI-DAQ for Macintosh when performing certain operations. Counter 1 is reserved when acquiring data with an AMUX-64T. Counter 5 is reserved when the sample count in a data acquisition is greater than 65,536.

The operation of the counter/timers and the programmable frequency output is discussed later in this chapter.

NB-MIO-16X Counter/Timers

From an onboard Am9513 System Timing chip, the NB-MIO-16X uses three independent 16-bit counter/timers and a 4-bit programmable frequency output. The connection of the Am9513 Counter/Timer signals to the NB-MIO-16X I/O connector and to the RTSI bus is shown in Figure 8-4.

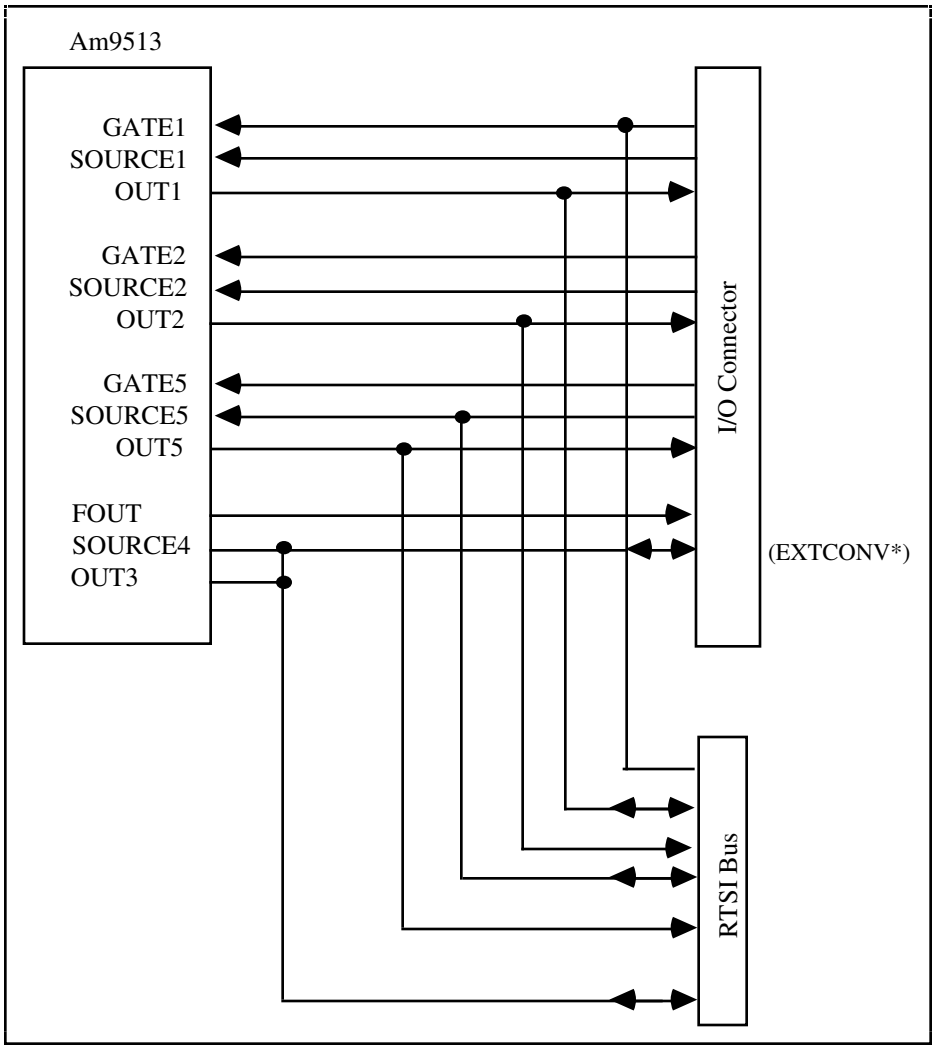

Figure 8-4. NB-MIO-16X Counter/Timer Signal Connections

Counters 1, 2, and 5 are available for counting operations on the NB-MIO-16X. Counters 3 and 4 are reserved for data acquisition operations. As shown in Figure 8-4, the FOUT, GATE, SOURCE, and OUT signals for Counters 1, 2, and 5 are available at the I/O connector. The signals GATE1, SOURCE5, OUT1, OUT2, OUT5, OUT3, and SOURCE4 are also available for connection to the RTSI bus trigger lines. (See Chapter 9, *RTSI Bus Trigger Functions*, for more information.)

Although Counters 1, 2, and 5 are made available for general use, they are reserved by NI-DAQ for Macintosh when performing certain operations. Counter 1 is reserved when acquiring data with an AMUX-64T. Counter 2 is reserved when performing an interval scanning operation initiated by `SCAN_IntStart`. Counter 5 is reserved when the sample count in a data acquisition is greater than 65,536.

The operation of the counter/timers and the programmable frequency output is discussed later in this chapter.

NB-DMA-8-G and NB-DMA2800 Counter/Timers

The NB-DMA-8-G and NB-DMA2800 contain an onboard Am9513 System Timing chip that has five independent 16-bit counter/timers and a 4-bit programmable frequency output. The connection of the Am9513 Counter/Timer signals to the RTSI bus is shown in Figure 8-5.

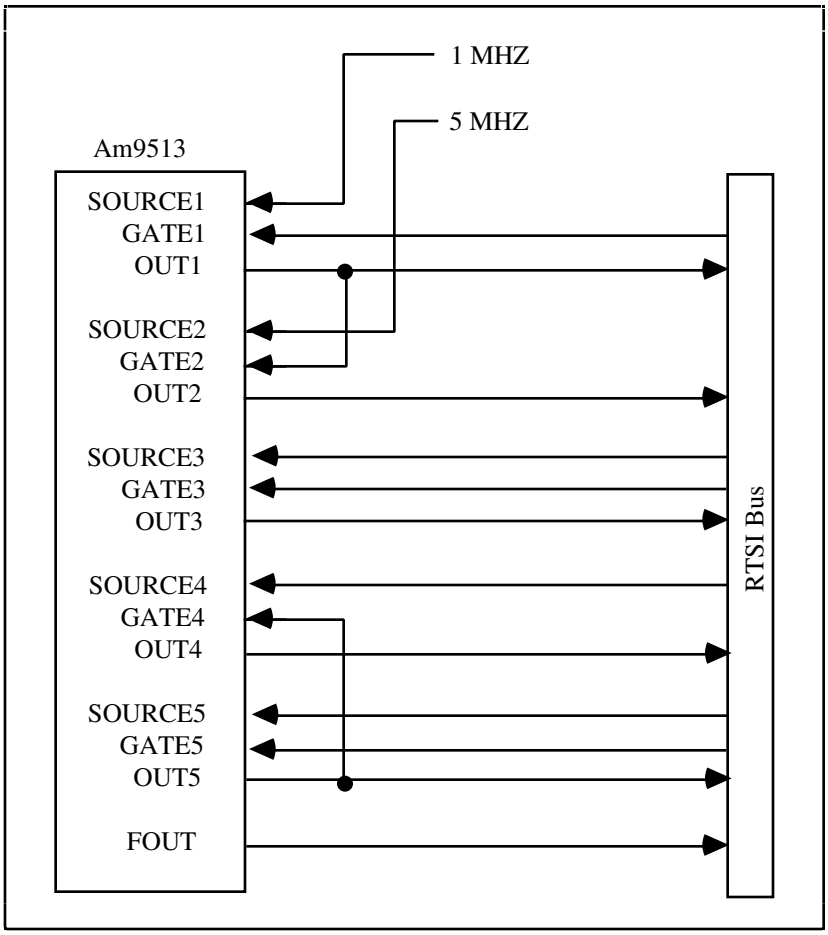

Figure 8-5. NB-DMA-8-G and NB-DMA2800 Counter/Timer Signal Connections

Counters 1 through 5 are available for counting operations on the NB-DMA-8-G and NB-DMA2800. The input and output signals for these counters are available for connection to the RTSI bus trigger lines for system timing operations. (See Chapter 9, *RTSI Bus Trigger Functions*, for more information.) As shown in Fig. 8-5, FOUT, SOURCE3 through 5, GATE1, GATE3, GATE5, and OUT1 through 5 can be connected to the RTSI bus. SOURCE1 and SOURCE2 are not currently connected to anything. The GATE2 signal is hardwired to OUT1, and the GATE4 signal is hardwired to OUT5. SOURCE4 can be driven from the RTSI bus.

The operation of the counter/timers and the programmable frequency output is discussed later in this chapter.

NB-A2000 Counter/Timers

The NB-A2000 has one unused 16-bit counter/timer from the onboard Am9513A System Timing chip. This counter, Counter 2, is made available for general use via the RTSI bus. The onboard RTSI switch connects the GATE2, SOURCE2, and OUT2 signals to the RTSI bus as shown in Figure 8-6 (see Chapter 9, *RTSI Bus Trigger Functions*, for more information).

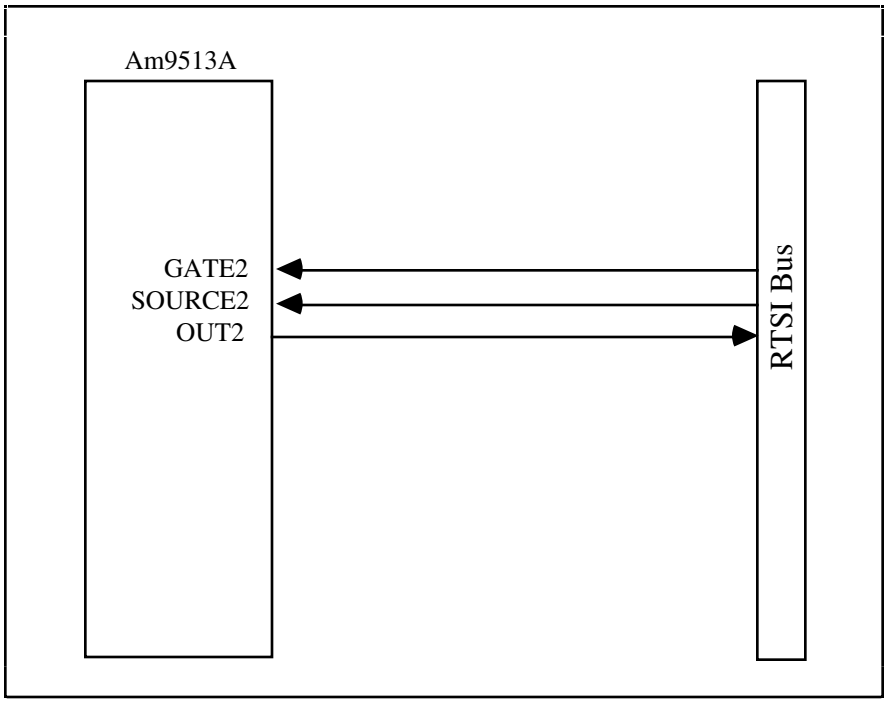

Figure 8-6. NB-A2000 Counter/Timer Signal Connections

The remaining counters, Counter 1, Counter 3, Counter 4, and Counter 5, are reserved for NB-A2000 Data Acquisition functions.

NB-TIO-10 Counter/Timers

The NB-TIO-10 has two onboard Am9513 System Timing chips that provide ten independent 16-bit counters and two 4-bit programmable frequency output. The connection of the Am9513 counter/timer signals to the NB-TIO-10 I/O connector and to the RTSI bus is shown in Figure 8-7.

Counters 1 through 10 are made available for counting operations on the NB-TIO-10. As shown in Figure 8-7, the FOUT1 and FOUT2 signals for Counters 1 through 10 are made available at the I/O connector. The signals GATE1, SOURCE1, OUT1, GATE6, SOURCE6, OUT6, SOURCE2, OUT2, SOURCE7, GATE5, OUT5, GATE10, OUT10, and FOUT1 signals are also available for connection to the RTSI bus trigger lines. See Chapter 9, *RTSI Bus Trigger VIs*, for more information.

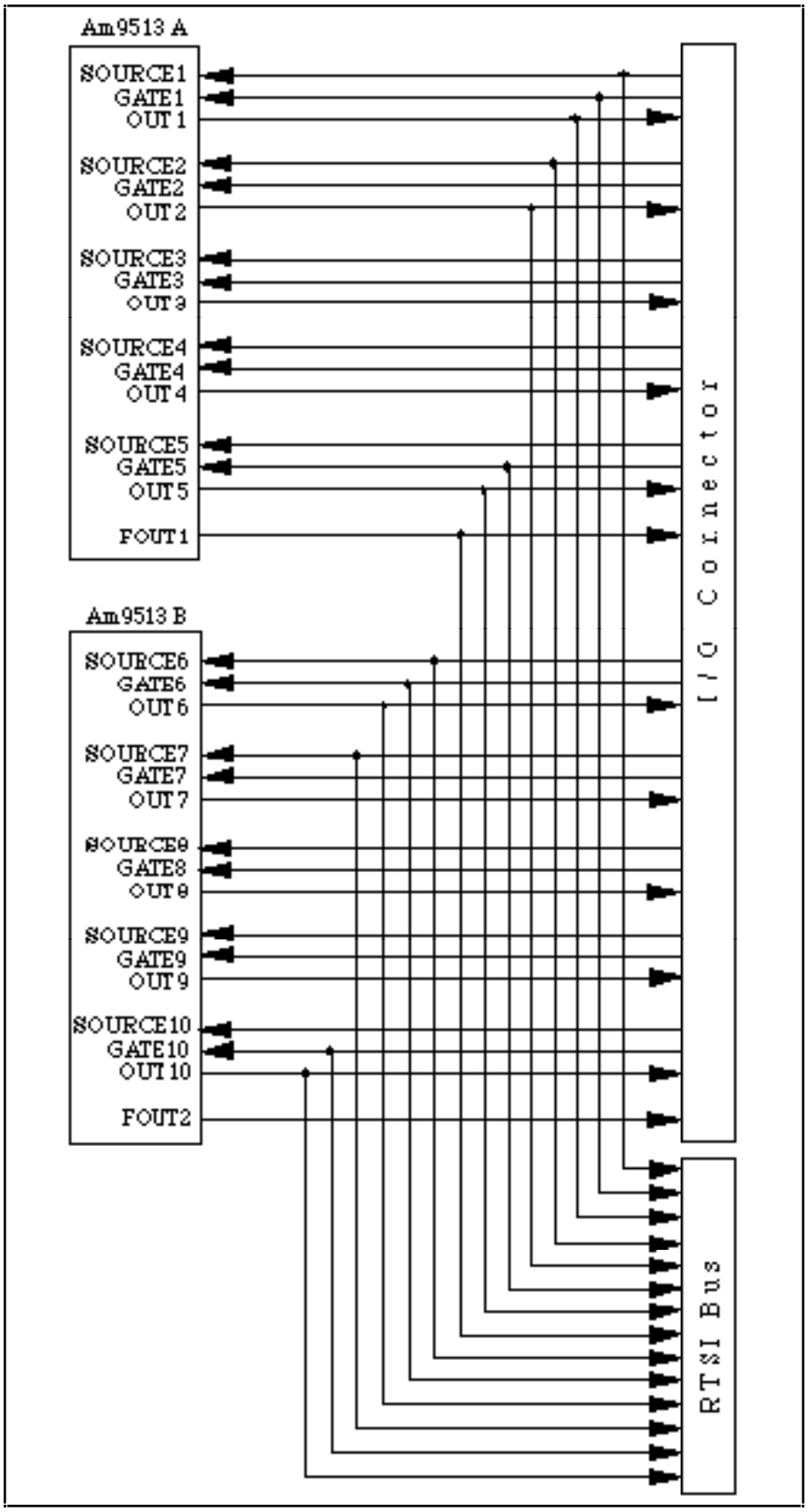

Figure 8-7. NB-TIO-10 Counter/Timer Signal Connections

Counter/Timer Function Summary

Use the following Counter/Timer functions on the NB-MIO-16, NB-MIO-16X, NB-DMA-8-G, NB-DMA2800, NB-A2000, and NB-TIO-10:

`CTR_Config`	Saves information about which configuration to use for a specified counter.
`CTR_EvCount`	Configures the specified counter for an event-counting operation and starts the counter.
`CTR_EvRead`	Reads the current counter without disturbing the event-counting operation and returns the count and overflow conditions.
`CTR_Period`	Configures the specified counter for period or pulse width measurement and starts the counter.
`CTR_Pulse`	Sets up the specified counter to generate an output pulse with programmable delay and pulse width.
`CTR_Reset`	Places counter output drivers in the specified output state.
`CTR_Restart`	Restarts the operation of the specified counter.
`CTR_Square`	Causes the specified counter to generate a continuous square wave output of specified duty cycle and frequency.
`CTR_State`	Returns the OUT logic level of the specified counter.
`CTR_Stop`	Suspends the operation of the specified counter in such a way that the counter operation can be restarted.

Use the following function for programming the frequency output on the NB-MIO-16, NB-MIO-16X, NB-TIO-10, NB-DMA-8-G, and NB-DMA2800:

`CTR_FOUT_Config`	Disables or enables and sets the frequency of the 4-bit programmable FOUT.

Counter/Timer Function Application Hints

The two basic types of functions are event counting and timing signal generation. For all of these functions, `CTR_Config` configures the counter modes; `CTR_Stop` suspends the function; `CTR_Restart` restarts or repeats the function; `CTR_State` returns the state of the counter output signal; and `CTR_Reset` stops the counter, clears the counter's mode, and places the output in a specified state.

Event Counting

`CTR_EvCount` initiates the event-counting process, and `CTR_EvRead` returns counter values. These two functions perform event-counting and frequency, pulse-width, or time-lapse measurement. Details on these applications are given under *Event-Counting Applications*, which follows the `CTR_EvRead` description in this chapter.

Timing Signal Generation

`CTR_Pulse` initiates single-pulse generation. `CTR_Square` initiates counter square wave (also known as pulse-train) generation. You can use `CTR_Square` with special gating (**gateMode** = 5) to perform gate-controlled pulse generation. When the gate input is low, NI-DAQ uses period1 to generate the pulses. When the gate input is high, NI-DAQ uses period2 to generate the pulses. If the output mode is TC toggle, the result is two 50% duty square

waves of different frequencies. If the output mode is TC pulse, the result is two pulse trains of different frequencies. `CTR_FOUT_Config` generates a clock signal at the FOUT Port.

The Pulse Generator function included in the NI-DAQ for Macintosh Examples folder illustrates the use of the Counter/Timer functions. This function generates a specified number of pulses at a selected frequency and duty cycle.

CTR_Config

Function

Saves information about which configuration to use for the specified counter.

Synopsis

C Syntax	`locus i32 CTR_Config(u32 deviceNumber, u32 counter, u32 edgeMode, u32 gateMode, u32 outputType, u32 outputPolarity);`
Pascal Syntax	`function CTR_Config(deviceNumber : i32; counter : i32; edgeMode : i32; gateMode : i32; outputType : i32; outputPolarity : i32) : i32;`
BASIC Syntax	`FN CTR_Config(deviceNumber&, counter&, edgeMode&, gateMode&, outputType&, outputPolarity&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.

Range: 1, 2, or 5 for the NB-MIO-16 or NB-MIO-16X.
1 through 5 for the NB-DMA-8-G and NB-DMA2800.
1 through 10 for the NB-TIO-10.
2 for the NB-A2000.

edgeMode indicates the edge of the input signal that the counter counts.

0: count rising edges.
1: count falling edges.

gateMode selects the gating mode used by the counter. The five different gating modes numbered 0 through 4 are:

0: no gating used.
1: high-level gating used.
2: low-level gating used.
3: edge-triggered gating used—rising edge.
4: edge-triggered gating used—falling edge.
10: active high on terminal count of next lower-order counter.
11: active high on gate of next higher-order counter.
12: active high on gate of next lower-order counter.
13: special gating used (See *Counter/Timer Operations (CTR Functions)* section earlier in this chapter for more information.

outputType selects the type of output generated by the counter. The counters generate two types of output signals: TC toggle output, and TC pulse output.

0: TC toggle output type used.
1: TC pulse output type used.

outputPolarity selects the output polarity used by the counter.
0: positive logic output.
1: negative logic (inverted) output.

If TC pulse output type is selected, then **outputPolarity** = 0 means that active logic-high TC pulses are generated. **outputPolarity** = 1 means that active logic-low TC pulses are generated. Similarly, if TC toggle output type is selected, then **outputPolarity** = 0 means that the OUT signal toggles from low to high on the first TC. **outputPolarity** = 1 means that the OUT signal toggles from high to low on the first TC.

`CTR_Config` saves the parameters in the configuration table for the specified counter. This configuration table is used when the counter is set up for an event-counting, pulse output, or frequency output operation. `CTR_Config` lets you take advantage of the many modes supported by the counter.

The default settings for counter configuration modes after system startup are as follows:
edgeMode = 0: count rising edges.
gateMode = 0: no gating used.
outputType = 0: TC toggle output type used.
outputPolarity = 0: positive logic output used.

If you want a counter configuration different from this default setting, then you must call `CTR_Config` with the configuration you want before any counter operation is initiated. Call `CTR_Config` only when the counter configuration is changed.

CTR_EvCount

Function
Configures the specified counter for an event-counting operation and starts the counter.

Synopsis

C Syntax	`locus i32 CTR_EvCount(u32 deviceNumber, u32 counter, u32 timebase, u32 mode);`
Pascal Syntax	`function CTR_EvCount(deviceNumber : i32; counter : i32; timebase : i32; mode : i32) : i32;`
BASIC Syntax	`FN CTR_EvCount(deviceNumber&, counter&, timebase&, mode&)`

Description
deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.
Range: 1, 2, or 5 for the NB-MIO-16 or NB-MIO-16X.
1 through 5 for the NB-DMA-8-G and NB-DMA2800.
1 through 10 for the NB-TIO-10.
2 for the NB-A2000.

timebase selects the resolution used by the counter. **timebase** has the following possible values:

0: TC signal of **counter**-1 used as timebase.
1: Internal 1-MHz clock used as timebase (1-µs resolution).
2: Internal 100-kHz clock used as timebase (10-µs resolution).
3: Internal 10-kHz clock used as timebase (100-µs resolution).
4: Internal 1-kHz clock used as timebase (1-ms resolution).
5: Internal 100-Hz clock used as timebase (10-ms resolution).
6: SOURCE1 used as timebase if 1≤ **counter** ≤ 5 or SOURCE6 used as timebase if 6 ≤ **counter** ≤ 10.
7: SOURCE2 used as timebase if 1≤ **counter** ≤ 5 or SOURCE7 used as timebase if 6 ≤ **counter** ≤ 10.
8: SOURCE3 used as timebase if 1≤ **counter** ≤ 5 or SOURCE8 used as timebase if 6 ≤ **counter** ≤ 10.
9: SOURCE4 used as timebase if 1≤ **counter** ≤ 5 or SOURCE9 used as timebase if 6 ≤ **counter** ≤ 10.
10: SOURCE5 used as timebase if 1≤ **counter** ≤ 5 or SOURCE10 used as timebase if 6 ≤ **counter** ≤ 10.
11: GATE 1 used as timebase if 1 ≤ **counter** ≤5 or GATE 6 used as timebase if 6 ≤ **counter** ≤ 10.
12: GATE 2 used as timebase if 1 ≤ **counter** ≤5 or GATE 7 used as timebase if 6 ≤ **counter** ≤ 10.
13: GATE 3 used as timebase if 1 ≤ **counter** ≤5 or GATE 8 used as timebase if 6 ≤ **counter** ≤ 10.
14: GATE 4 used as timebase if 1 ≤ **counter** ≤5 or GATE 9 used as timebase if 6 ≤ **counter** ≤ 10.
15: GATE 5 used as timebase if 1 ≤ **counter** ≤5 or GATE 10 used as timebase if 6 ≤ **counter** ≤ 10.

If **timebase** is 0, counters are concatenated. Counters 5 and 6 on the NB-TIO-10 cannot be concatenated because Counters 1 through 5 are on one chip and Counters 6 through 10 are on the other chip.

cont indicates whether counting continues when the counter rolls over to 0. **cont** can be either 0 or 1. If **cont** is 0, event counting stops when the counter counts up to 65,535 and rolls over to 0, in which case an overflow condition is registered. If **cont** is 1, event counting continues when the counter rolls over to 0, and no overflow condition is registered. Setting **cont** to 1 is useful when more than one counter is concatenated for event counting.

`CTR_EvCount` configures the specified counter for an event-counting operation. The counter is configured to count up from 0 and to use the gating mode, edge mode, output type, and polarity as indicated by the `CTR_Config` call.

Note: ***Edge-triggered gating mode does not operate properly during event counting if* cont *is 1. If* cont *is 1, use level-gating mode or no-gating mode.***

Applications for `CTR_EvCount` are discussed in *Event-Counting Applications* later in this chapter.

CTR_EvRead

Function

Reads the current count without disturbing the counting process and returns the count and overflow conditions.

Synopsis

C Syntax	`locus i32 CTR_EvRead(u32 deviceNumber, u32 counter, u16 *overflow, u32 *count);`
Pascal Syntax	`function CTR_EvRead(deviceNumber : i32; counter : i32; var overflow : i16; var count : i32) : i32;`
BASIC Syntax	`FN CTR_EvRead(deviceNumber&, counter&, overflow&, count&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.

Range: 1, 2, or 5 for the NB-MIO-16 or NB-MIO-16X.
1 through 5 for the NB-DMA-8-G and NB-DMA2800.
1 through 10 for the NB-TIO-10.
2 for the NB-A2000.

overflow returns the overflow state of the counter. A counter overflows if it counts up to 65,535 and rolls over to 0 on the next count. If **overflow** is 0, no overflow has occurred. If **overflow** is 1, an overflow occurred. See the *Special Considerations for Overflow Detection* section later in this function description.

count returns the current value of the specified counter. **count** can be between 0 and 65,535. **count** represents the number of edges (either falling or rising edges, not both) that have occurred because the counter started counting.

`CTR_EvRead` reads the current value of the counter without disturbing the counting process and returns the value in **count**. `CTR_EvRead` also performs overflow detection and returns the overflow status in **overflow**. Overflow detection and the significance of **count** depend on the counter configuration.

Pascal Note: ***The event count returned in* count *is a 16-bit unsigned number. Because Pascal does not support unsigned representation, a* count *greater than 32,767 is treated as a negative by Pascal. You can use the* `UToL` *conversion function to convert* count *to a Pascal long integer. (See Chapter 11,*** *NI-DAQ for Macintosh Examples****, for a complete description of the*** `UToL` ***function.)***

Special Considerations for Overflow Detection

To detect an overflow condition the counter must be configured for TC toggle output type and positive output polarity, and the counter must be configured to stop counting on overflow (**cont** = 0 in the `CTR_EvCount` call). If these conditions are not met, then the value of **overflow** is meaningless. If more than one counter is concatenated for event-counting applications, the lower-order counters should be configured to continue counting when overflow occurs (overflow detection is meaningful only for the highest order counter). The value of **count** returned by `CTR_EvRead` for the lower-order counters, then, represents the module 65,536 event count. (For more information, see the *Event Counting Applications* section later in this chapter.)

Event-Counting Applications

The four types of event-counting/timing measurements utilized by `CTR_EvCount` and `CTR_EvRead` are event counting, pulse-width measurement, time-lapse measurement, and frequency measurement. `CTR_EvCount` also permits concatenation of counters such that 32-bit or 48-bit resolution can be used for these measurements.

For event-counting applications, the events counted are the signal transition or *edges* of an input SOURCE signal; therefore, you should set **timebase** to a value from 6 through 10. Either low-to-high or high-to-low edges can be counted (this feature is selected by **edgeMode** in the `CTR_Config` function). In addition, the various gating modes given by `CTR_Config` can be used to control counting.

For pulse-width measurement, a counter is configured to count during the duration of a pulse. For this application, any timebase can be used, including an external clock connected to the counter SOURCE input. Level gating modes should be used for pulse-width measurements, where the pulse measured is connected to the counter GATE input. Pulse width is equal to (event count) $*$ (timebase period).

For time-lapse measurement, a counter is configured to count from the occurrence of some event. For this application, any timebase can be used, including an external clock connected to the counter SOURCE input. Edge-

triggered gating modes can be used if a single counter performs the event-counting and if **cont** is 0. In this case, the starting event is an edge applied to the GATE input of the counter. The time lapse from the edge is equal to (event count) ∗ (timebase period). If counters are concatenated for time-lapse measurement, use level-gating where the GATE input signal goes active at the starting event and stays active.

Frequency measurement is a special case of event-counting; that is, the frequency of an input signal can be measured by counting the number of edges of a signal that occur during a fixed amount of time. For this application, **timebase** should be set to 0, and the signal measured should be connected to the SOURCE input of the counter. Either low-to-high or high-to-low edges can be counted (this feature is selected by **edgeMode** in the `CTR_Config` function). Event-counting is constrained to a fixed amount of time by using level-gating and by applying a gate pulse of known fixed duration to the GATE input of the counter. The average frequency of the incoming signal is then (event count) / (gate period). The gating pulse for frequency measurement can be supplied by another counter (see `CTR_Pulse`).

For 16-bit resolution event-counting and time-lapse, only one counter need be used. **cont** should be set to 0 so that you are notified if the counter overflows (see `CTR_EvRead`). Any gating mode can be used. In addition, TC toggle output type and positive output polarity should be selected during the `CTR_Config` call so that overflow detection operates properly.

For greater than 16-bit resolution, two or more counters can be concatenated. A low-order counter is configured to count the incoming edges. The OUT signal of the low-order counter is connected to the SOURCE input of the next high-order counter. The next high-order counter is configured to count once every time the low-order counter rolls over. The OUT signal of the next high-order counter is connected to the SOURCE input of an additional counter. The last counter (referred to as the high-order counter) is the counter that performs overflow detection. The lower-order counters increment continuously and generate output pulses whenever they roll over.

For 32-bit counting, use two counters. For 48-bit counting, use three counters, and so on. The counter configurations for concatenated event-counting are as follows:

- Low-order counter configuration:

  edgeMode: count rising edges or falling edges.
  gateMode: any value.
  outputType: TC pulse output type.
  outputPolarity: positive polarity.
  timebase: any value.
  cont = 1: (continuous counting).

- Intermediate counter configuration:

  edgeMode: count rising edges (indicates low-order counter rolled over).
  gateMode: no gating.
  outputType: TC pulse output type.
  outputPolarity: positive polarity.
  timebase = 0: (counts lower-order counter output).
  cont = 1: (continuous counting).

- High-order counter configuration:

  edgeMode: count rising edges (indicates low-order counter rolled over).
  gateMode: no gating.
  outputType: TC toggle output type (for proper overflow detection).
  outputPolarity: positive polarity.
  timebase = 0: so that it counts lower-order counter output.
  cont = 0: so that counter stops on overflow.

Period Measurements Applications

With the proper use of `CTR_Config`, `CTR_Period`, and `CTR_EvRead`, you can configure a counter for the following:

- Period measurement
- Continuous pulse width measurement

To make a period measurement, call `CTR_Config` with **gateMode** set to either rising or falling edge-triggered gating (3 or 4). With rising edge-triggered gating, a counter can measure the time interval between two rising edges of the gate signal. With falling edge-triggered gating, a counter can measure the time interval between two falling edges of the gate signal. When you execute `CTR_Config`, and you apply the signal being measured to the appropriate gate, you can execute `CTR_Period` to initiate period measurement. The specified counter starts counting on the first gate edge and latches the counter value to the onboard Hold Register once the counter detects a second gate edge. After each period measurement, the counter re-loads itself with a 0 and starts a new measurement.

While the measurement is occurring, call `CTR_EvRead` to retrieve the counter value saved in the Hold Register. The period is then equal to the value returned by `CTR_EvRead` * **timebase**.

If you choose an improper timebase frequency, `CTR_EvRead` retrieves a smaller count value. A small count indicates that the timebase frequency is either too low or too high compared to the gate signal. If the timebase frequency is too low, the counter can only count a few source edges. However, if the timebase frequency is too high, the counter counts too many source edges, causing counter overflow. In case of counter overflow, a small count (typically 1 or 2) is saved on the Hold Register, and the counter reloads itself with a 0 and waits for a new gate trigger to make a new measurement. If the value returned by `CTR_EvRead` is 0, the period measurement reading is not available.

For a pulse width measurement, use the same NI-DAQ for Macintosh calls used for period measurement, except **gateMode** should be set to high-level or low-level gating (1 or 2). With high-level gating, a counter can measure the duration of a positive pulse, and with low-level gating, a counter can measure the duration of a negative pulse. When `CTR_Period` is called, the counter starts counting once the gate goes active. When the gate goes inactive, the counter value latches to the Hold Register. `CTR_EvRead` can then be called to retrieve the saved value. Pulse width is then equal to the value returned by `CTR_EvRead` * **timebase**. When the counter value is latched to the Hold Register, the counter reloads itself with a 0 and waits for the gate to go active to begin a new measurement.

For measuring pulse width, a rough estimate is needed for the duration of the pulse being measured. When a counter is configured to measure pulse width, the counter continues counting in case of overflow. No counter value is latched to the Hold Register until the gate signal becomes inactive. To detect the counter overflow, feed the output of the pulse width measurement counter to the source input of an event-counting counter. If the event-counting counter value is not 0 after the pulse width measurement, the pulse width measurement is not correct.

For more information on event-counting, see `CTR_EvCount` and *Event-Counting Application*s earlier in this chapter.

CTR_FOUT_Config

Function

Disables or enables and sets the frequency of the 4-bit programmable FOUT.

Synopsis

C Syntax	`locus i32 CTR_FOUT_Config(u32 deviceNumber, u32 FOUT_port, u32 mode, u32 timebase, u32 division);`
Pascal Syntax	`function CTR_FOUT_Config(deviceNumber : i32; FOUT_port : i32; mode : i32; timebase : i32; division : i32) : i32;`
BASIC Syntax	`FN CTR_FOUT_Config(deviceNumber&, FOUT_port&, mode&, timebase&, division&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

FOUT_port is the frequency output port programmed.

1: for FOUT1 on the NB-TIO-10 or FOUT on the NB-MIO-16, NB-MIO-16X, NB-DMA-8-G, and NB-DMA2800.
2: for FOUT2 on the NB-TIO-10.

mode indicates whether to enable or disable the programmable frequency output.

0: the frequency output signal is turned off to a logic low level.
1: the frequency output signal is enabled.

If **clock** is 0, none of the following parameters apply.

timebase selects the resolution used by the programmable frequency output. **timebase** has the following possible values:

1: Internal 1-MHz clock used as timebase (1-µs resolution).
2: Internal 100-kHz clock used as timebase (10-µs resolution).
3: Internal 10-kHz clock used as timebase (100-µs resolution).
4: Internal 1-kHz clock used as timebase (1-ms resolution).
5: Internal 100-Hz clock used as timebase (10-ms resolution).
6: SOURCE1 used as timebase if **FOUT_port** = 1 or SOURCE6 used as timebase if **FOUT_port** = 2.
7: SOURCE2 used as timebase if **FOUT_port** = 1 or SOURCE7 used as timebase if **FOUT_port** = 2.
8: SOURCE3 used as timebase if **FOUT_port** = 1 or SOURCE8 used as timebase if **FOUT_port** = 2.
9: SOURCE4 used as timebase if **FOUT_port** = 1 or SOURCE9 used as timebase if **FOUT_port** = 2.
10: SOURCE5 used as timebase if **FOUT_port** = 1 or SOURCE10 used as timebase if **FOUT_port** = 2.
11: GATE 1 used as timebase if **FOUT_port** = 1 or GATE 6 used as timebase if **FOUT_port** = 2.
12: GATE 2 used as timebase if **FOUT_port** = 1 or GATE 7 used as timebase if **FOUT_port** = 2.
13: GATE 3 used as timebase if **FOUT_port** = 1 or GATE 8 used as timebase if **FOUT_port** = 2.
14: GATE 4 used as timebase if **FOUT_port** = 1 or GATE 9 used as timebase if **FOUT_port** = 2.
15: GATE 5 used as timebase if **FOUT_port** = 1 or GATE 10 used as timebase if **FOUT_port** = 2.

division is the divide-down factor for generating the clock. The clock frequency is then equal to (**timebase** frequency) divided by **division.**

Range: 1 through 16.

`CTR_FOUT_Config` generates an output clock at the programmable frequency output signal FOUT if **mode** is 1; otherwise, the FOUT signal is a logic low level. The frequency of the FOUT signal is the **timebase** frequency divided by the **division** factor. FOUT provides a 50 percent duty-cycle clock signal.

Note: `CTR_FOUT_Config` ***function is only available on the NB-MIO-16, NB-MIO-16X, NB-TIO-10, NB-DMA-8-G, and NB-DMA2800.***

Note: `CTR_FOUT_Config` ***replaces the*** `CTR_Clock` ***function used in previous versions of NI-DAQ for Macintosh.***

CTR_Period

Function

Configures the specified counter for period or pulse width measurement and starts the counter.

Synopsis

C Syntax	`locus i32 CTR_Period(u32 deviceNumber, u32 counter, u32 timebase);`
Pascal Syntax	`function CTR_Period(deviceNumber : i32; counter : i32; timebase : i32) : i32;`
BASIC Syntax	`FN CTR_Period(deviceNumber&, counter&, timebase&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.

Range: 1, 2, or 5 for the NB-MIO-16 or NB-MIO-16X.
1 through 5 for the NB-DMA-8-G and NB-DMA2800.
1 through 10 for the NB-TIO-10.
2 for the NB-A2000.

timebase selects the resolution used by the counter. **timebase** has the following possible values:

0: TC signal of **counter**-1 used as timebase.
1: Internal 1-MHz clock used as timebase (1-µs resolution).
2: Internal 100-kHz clock used as timebase (10-µs resolution).
3: Internal 10-kHz clock used as timebase (100-µs resolution).
4: Internal 1-kHz clock used as timebase (1-ms resolution).
5: Internal 100-Hz clock used as timebase (10-ms resolution).
6: SOURCE1 used as timebase if 1≤ **counter** ≤ 5 or SOURCE6 used as timebase if 6 ≤ **counter** ≤ 10.
7: SOURCE2 used as timebase if 1≤ **counter** ≤ 5 or SOURCE7 used as timebase if 6 ≤ **counter** ≤ 10.
8: SOURCE3 used as timebase if 1≤ **counter** ≤ 5 or SOURCE8 used as timebase if 6 ≤ **counter** ≤ 10.
9: SOURCE4 used as timebase if 1≤ **counter** ≤ 5 or SOURCE9 used as timebase if 6 ≤ **counter** ≤ 10.
10: SOURCE5 used as timebase if 1≤ **counter** ≤ 5 or SOURCE10 used as timebase if 6 ≤ **counter** ≤ 10.
11: GATE 1 used as timebase if 1 ≤ **counter** ≤5 or GATE 6 used as timebase if 6 ≤ **counter** ≤ 10.
12: GATE 2 used as timebase if 1 ≤ **counter** ≤5 or GATE 7 used as timebase if 6 ≤ **counter** ≤ 10.
13: GATE 3 used as timebase if 1 ≤ **counter** ≤5 or GATE 8 used as timebase if 6 ≤ **counter** ≤ 10.
14: GATE 4 used as timebase if 1 ≤ **counter** ≤5 or GATE 9 used as timebase if 6 ≤ **counter** ≤ 10.
15: GATE 5 used as timebase if 1 ≤ **counter** ≤5 or GATE 10 used as timebase if 6 ≤ **counter** ≤ 10.

Set **timebase** to 1 through 5 for the counter to count one of the five available internal signals. Set **timebase** to 6 through 15 (except 10 for the NB-TIO-10) if you plan to provide an external signal to a counter. This external signal is then the signal counted for counting.

`CTR_Period` configures the specified counter for period and pulse width measurement. The counter is configured to count up from 0 and to use gating mode, edge mode, output type, and polarity as specified by the `CTR_Config` call.

Applications for `CTR_Period` are discussed in the *Period Measurement Applications* section later in this chapter.

CTR_Pulse

Function

Sets up the specified counter to generate an output pulse with programmable delay and pulse width.

Synopsis

C Syntax	`locus i32 CTR_Pulse(u32 deviceNumber, u32 counter, u32 timebase, u32 delay, u32 pulseWidth);`
Pascal Syntax	`function CTR_Pulse(deviceNumber : i32; counter : i32; timebase : i32; delay : i32; pulseWidth : i32) : i32;`
BASIC Syntax	`FN CTR_Pulse(deviceNumber&, counter&, timebase&, delay&, pulseWidth&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.

Range: 1, 2, or 5 for the NB-MIO-16 or NB-MIO-16X.
1 through 5 for the NB-DMA-8-G and NB-DMA2800.
1 through 10 for the NB-TIO-10.
2 for the NB-A2000.

timebase selects the resolution used by the counter. **timebase** has the following possible values:

0: TC signal of **counter**-1 used as timebase.
1: Internal 1-MHz clock used as timebase (1-µs resolution).
2: Internal 100-kHz clock used as timebase (10-µs resolution).
3: Internal 10-kHz clock used as timebase (100-µs resolution).
4: Internal 1-kHz clock used as timebase (1-ms resolution).
5: Internal 100-Hz clock used as timebase (10-ms resolution).
6: SOURCE1 used as timebase if 1≤ **counter** ≤ 5 or SOURCE6 used as timebase if 6 ≤ **counter** ≤ 10.
7: SOURCE2 used as timebase if 1≤ **counter** ≤ 5 or SOURCE7 used as timebase if 6 ≤ **counter** ≤ 10.
8: SOURCE3 used as timebase if 1≤ **counter** ≤ 5 or SOURCE8 used as timebase if 6 ≤ **counter** ≤ 10.
9: SOURCE4 used as timebase if 1≤ **counter** ≤ 5 or SOURCE9 used as timebase if 6 ≤ **counter** ≤ 10.
10: SOURCE5 used as timebase if 1≤ **counter** ≤ 5 or SOURCE10 used as timebase if 6 ≤ **counter** ≤ 10.
11: GATE 1 used as timebase if 1 ≤ **counter** ≤5 or GATE 6 used as timebase if 6 ≤ **counter** ≤ 10.
12: GATE 2 used as timebase if 1 ≤ **counter** ≤5 or GATE 7 used as timebase if 6 ≤ **counter** ≤ 10.
13: GATE 3 used as timebase if 1 ≤ **counter** ≤5 or GATE 8 used as timebase if 6 ≤ **counter** ≤ 10.
14: GATE 4 used as timebase if 1 ≤ **counter** ≤5 or GATE 9 used as timebase if 6 ≤ **counter** ≤ 10.
15: GATE 5 used as timebase if 1 ≤ **counter** ≤5 or GATE 10 used as timebase if 6 ≤ **counter** ≤ 10.

If **timebase** is 0, counters are concatenated. Set **timebase** to 6 through 15 if you plan to provide an external clock to the counter.

delay is the delay before the pulse is generated. **delay** can be between 3 and 65,536. The actual time period **delay** represents is determined by the following formula:

delay ∗ (**timebase** resolution)

pulseWidth is the width of the pulse generated. **pulseWidth** can be between 0 and 65,536. The actual time **pulseWidth** represents is determined by the following formula:

pulseWidth ∗ (**timebase** resolution)

for 1≤**pulseWidth**≤65,536. **pulseWidth** = 0 is a special case of pulse generation and actually generates a pulse of infinite duration (see the timing diagrams in Figures 6-6 and 6-7).

`CTR_Pulse` sets up the counter to generate a pulse of the duration indicated by **pulseWidth** after a time delay of the duration indicated by **delay**. If no gating is selected, `CTR_Pulse` starts the counter; otherwise, counter operation is controlled by the gate input. Timing of pulse generation is determined by the timebase selected and is shown in Figure 8-8.

Successive identical pulses can be generated by calling `CTR_Restart`. Be sure that the previous pulse generation is complete; otherwise, the call is ignored. In the case where **pulseWidth** is 0 and TC toggle output is used, the delay period toggles polarity after every `CTR_Restart` call.

Pulse Generation Timing Considerations

Figure 8-8 shows pulse generation timing for both the TC toggle output and TC pulse output cases. These signals are positive polarity output signals.

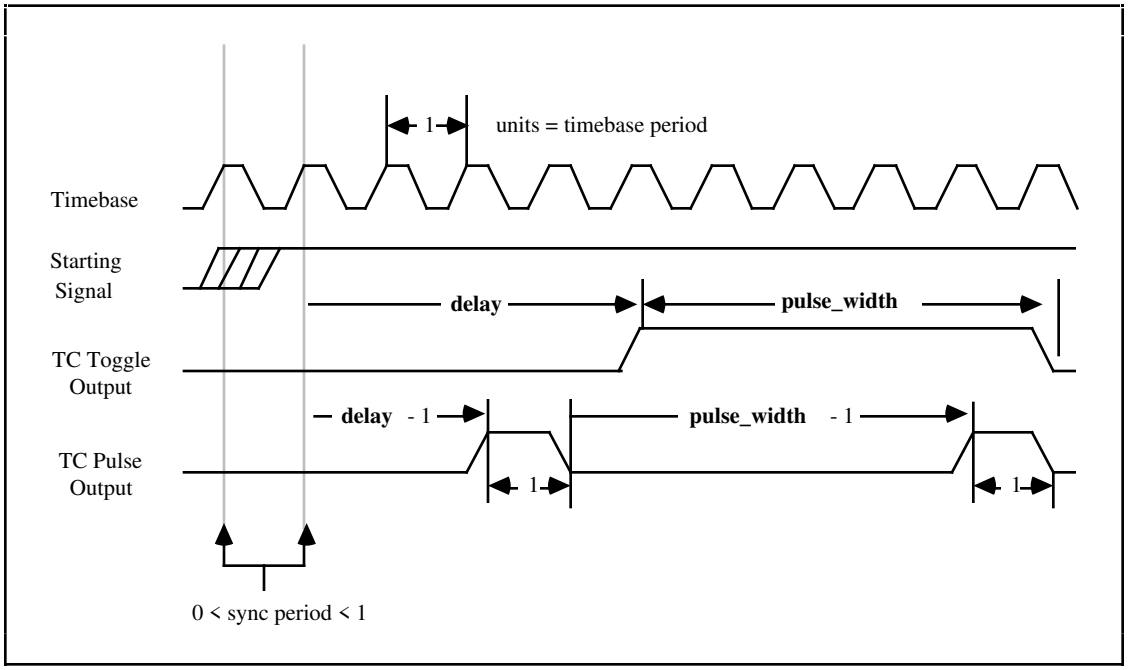

Figure 8-8. Pulse Generation Timing

An uncertainty is associated with the delay period due to counter synchronization. Counting starts on the first timebase edge *after* the starting signal is applied. The delay between receipt of the starting pulse and start of pulse generation can last between (**delay**) and (**delay**+1) units of the timebase.

pulseWidth = 0 generates a special case signal as shown in Figure 8-9.

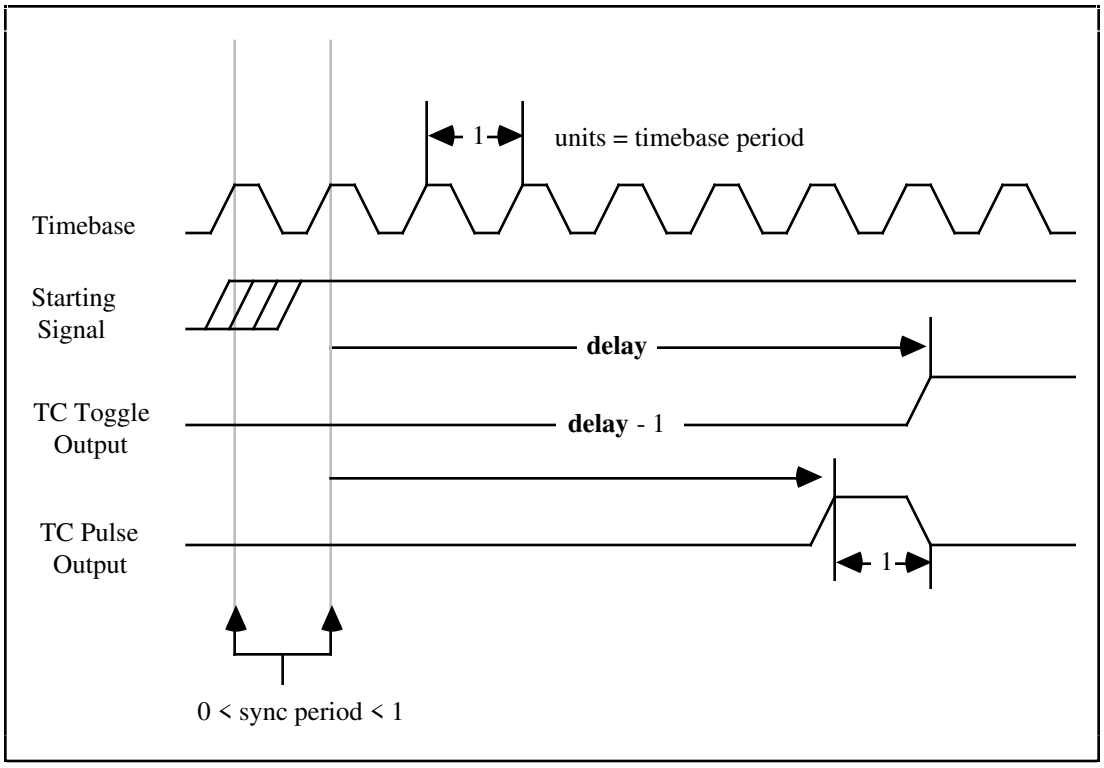

Figure 8-9. Pulse Timing for pulse_width = 0

CTR_Reset

Function

Places counter output drives in the specified output state.

Synopsis

C Syntax	`locus i32 CTR_Reset(u32 deviceNumber, u32 counter, u32 outputState);`
Pascal Syntax	`function CTR_Reset(deviceNumber : i32; counter : i32; outputState : i32) : i32;`
BASIC Syntax	`FN CTR_Reset(deviceNumber&, counter&, outputState&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.

Range: 1, 2, or 5 for the NB-MIO-16 or NB-MIO-16X.
1 through 5 for the NB-DMA-8-G and NB-DMA2800.
1 through 10 for the NB-TIO-10.
2 for the NB-A2000.

outputState indicates the output state of the counter OUT signal driver.

0: set OUT signal driver to high-impedance state.
1: set OUT signal driver to low-logic level.
2: set OUT signal driver to high-logic level.

CTR_Reset causes the specified counter to terminate its current operation, clears the counter mode, and places the counter OUT driver in the specified output state. All counters are reset and have their OUT drivers set to the high-impedance state after system startup. The counters are then ready to perform counter operations. CTR_Reset must be used to stop and clear a counter before it is set up for any subsequent operations. CTR_Reset can also be used to change the output state of an idle counter.

CTR_Restart

Function

Restarts the operation of the specified counter.

Synopsis

C Syntax	`locus i32 CTR_Restart(u32 deviceNumber, u32 counter);`
Pascal Syntax	`function CTR_Restart(deviceNumber : i32; counter : i32) : i32;`
BASIC Syntax	`FN CTR_Restart(deviceNumber&, counter&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.

Range: 1, 2, or 5 for the NB-MIO-16 or NB-MIO-16X.
1 through 5 for the NB-DMA-8-G and NB-DMA2800.
1 through 10 for the NB-TIO-10.
2 for the NB-A2000.

CTR_Restart can be used after a CTR_Stop operation to resume the suspended counter operation. If the specified counter was never set up for an operation, CTR_Restart returns an error.

CTR_Restart can also be used after a CTR_Pulse operation to generate additional pulses. CTR_Pulse generates the first pulse. In this case, CTR_Restart should not be called until the previous pulse has completed.

CTR_Square

Function

Causes the specified counter to generate a continuous square wave output of specified duty cycle and frequency.

Synopsis

C Syntax	`locus i32 CTR_Square(u32 deviceNumber, u32 counter, u32 timebase, u32 period1, u32 period2);`
Pascal Syntax	`function CTR_Square(deviceNumber : i32; counter : i32; timebase : i32; period1 : i32; period2 : i32) : i32;`
BASIC Syntax	`FN CTR_Square(deviceNumber&, counter&, timebase&, period1&, period2&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.
Range: 1, 2, or 5 for the NB-MIO-16 or NB-MIO-16X.
1 through 5 for the NB-DMA-8-G and NB-DMA2800.
1 through 10 for the NB-TIO-10.
2 for the NB-A2000.

timebase selects the resolution used by the counter. **timebase** has the following possible values:
0: TC signal of **counter**-1 used as timebase.
1: Internal 1-MHz clock used as timebase (1-µs resolution).
2: Internal 100-kHz clock used as timebase (10-µs resolution).
3: Internal 10-kHz clock used as timebase (100-µs resolution).
4: Internal 1-kHz clock used as timebase (1-ms resolution).
5: Internal 100-Hz clock used as timebase (10-ms resolution).
6: SOURCE1 used as timebase if 1≤ **counter** ≤ 5 or SOURCE6 used as timebase if 6 ≤ **counter** ≤ 10.
7: SOURCE2 used as timebase if 1≤ **counter** ≤ 5 or SOURCE7 used as timebase if 6 ≤ **counter** ≤ 10.
8: SOURCE3 used as timebase if 1≤ **counter** ≤ 5 or SOURCE8 used as timebase if 6 ≤ **counter** ≤ 10.
9: SOURCE4 used as timebase if 1≤ **counter** ≤ 5 or SOURCE9 used as timebase if 6 ≤ **counter** ≤ 10.
10: SOURCE5 used as timebase if 1≤ **counter** ≤ 5 or SOURCE10 used as timebase if 6 ≤ **counter** ≤ 10.
11: GATE 1 used as timebase if 1 ≤ **counter** ≤5 or GATE 6 used as timebase if 6 ≤ **counter** ≤ 10.
12: GATE 2 used as timebase if 1 ≤ **counter** ≤5 or GATE 7 used as timebase if 6 ≤ **counter** ≤ 10.
13: GATE 3 used as timebase if 1 ≤ **counter** ≤5 or GATE 8 used as timebase if 6 ≤ **counter** ≤ 10.
14: GATE 4 used as timebase if 1 ≤ **counter** ≤5 or GATE 9 used as timebase if 6 ≤ **counter** ≤ 10.
15: GATE 5 used as timebase if 1 ≤ **counter** ≤5 or GATE 10 used as timebase if 6 ≤ **counter** ≤ 10.

If **timebase** is 0, counters are concatenated. Set **timebase** to a value between 6 and 15 if you plan to provide an external clock to the counter.

period1 and **period2** indicate the two periods making up the square wave generated. For TC toggle output type and positive output polarity, **period1** indicates the duration of the on-cycle (logic-high state) and **period2** specifies the duration of the off-cycle (logic-low state).
Range: 1 through 65,536.

`CTR_Square` sets up the counter to generate a square wave of duration and frequency determined by **period1**, **period2**, and **timebase**. If no gating is selected, square wave generation is started by the `CTR_Square` call; otherwise, counter operation is controlled by the gate input.

You can use special gating (**gateMode** = 5) to perform gate-controlled pulse generation. When the gate input is low, NI-DAQ uses period1 to generate the pulses. When the gate input is high, NI-DAQ uses period2 to generate the pulses. If the output mode is TC toggle, the result is two 50% duty square waves of different frequencies. If the output mode is TC pulse, the result is two pulse trains of different frequencies.

The total period of the square wave is determined by the following formula:

(**period1** + **period2**) ∗ (**timebase** period)

This implies that the frequency of the square wave is as follows:

$$\frac{1}{(\textbf{period1} + \textbf{period2}) * (\textbf{timebase}\ \text{period})}$$

The percent duty cycle of the square wave is determined by the following formula:

$$\frac{\textbf{period 1}}{(\textbf{period1} + \textbf{period2})} * 100\%$$

Figure 8-10 shows the timing of square wave generation for both TC toggle output and TC pulse output. For this example, **period1** = 3 and **period2** = 2. The output signals shown are positive polarity output signals.

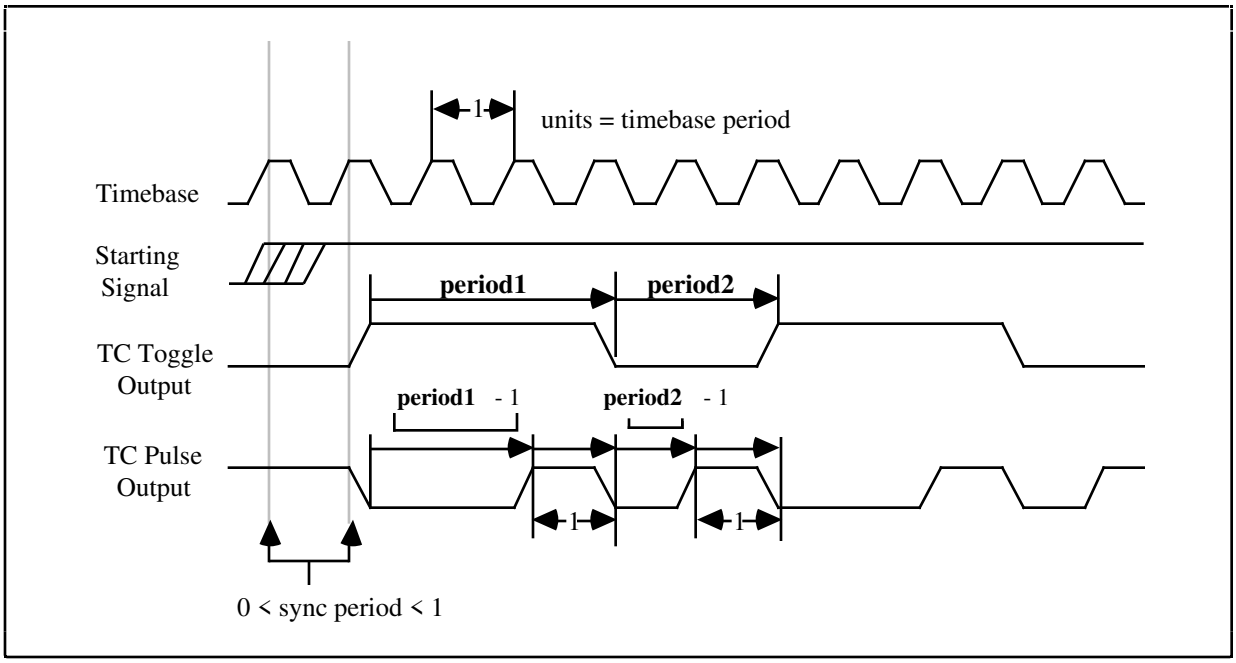

Figure 8-10. Square Wave Timing

Square Wave Generation Timing Considerations

There is an uncertainty associated with the beginning of square wave generation due to counter synchronization. Square wave generation starts on the first timebase edge *after* the starting signal is applied. The delay between receipt of the starting signal and the start of the square wave generation can last between 0 and 1 units of the timebase.

Edge-triggered gating should not be used with square wave generation. If edge-triggered gating is used, the waveform stops after **period1** expires and continues for one total period (**period2 + period1**) only after another edge is applied. This delay may or may not be useful. For continuous square wave generation, use level or no gating.

CTR_State

Function

Returns the OUT logic level of the specified counter.

Synopsis

C Syntax	`locus i32 CTR_State(u32 deviceNumber, u32 counter, u16 *outputState);`
Pascal Syntax	`function CTR_State(deviceNumber : i32; counter : i32; var outputState : i16) : i32;`
BASIC Syntax	`FN CTR_State(deviceNumber&, counter&, outputState&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.
Range: 1, 2, or 5 for the NB-MIO-16 or NB-MIO-16X.
1 through 5 for the NB-DMA-8-G and NB-DMA2800.
1 through 10 for the NB-TIO-10.
2 for the NB-A2000.

outputState returns the logic state of the counter OUT signal.
0: indicates that OUT is at a low-logic state.
1: indicates that OUT is at a high-logic state.

`CTR_State` reads the logic state of the OUT signal for the specified counter and returns the state in **outputState**. If the counter OUT driver is set to the high-impedance state, **outputState** is determinate and can be either 0 or 1.

CTR_Stop

Function
Suspends the operation of the specified counter in such a way that the counter operation can be restarted.

Synopsis

C Syntax	`locus i32 CTR_Stop(u32 deviceNumber, u32 counter);`
Pascal Syntax	`function CTR_Stop(deviceNumber : i32; counter : i32) : i32;`
BASIC Syntax	`FN CTR_Stop(deviceNumber&, counter&)`

Description
deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.
Range: 1, 2, or 5 for the NB-MIO-16 or NB-MIO-16X.
1 through 5 for the NB-DMA-8-G and NB-DMA2800.
1 through 10 for the NB-TIO-10.
2 for the NB-A2000.

`CTR_Stop` suspends the operation of the counter in such a way that the counter can be restarted by `CTR_Restart` and continue in its operation. For example, if a counter is set up for frequency output, issuing `CTR_Stop` causes the counter to stop generating a square wave, and `CTR_Restart` allows it to resume. `CTR_Stop` causes the counter output to remain at the state it was when `CTR_Stop` was issued.

Interval Counter/Timer Operation (ICTR Functions)

The 16-bit counters available on the DAQCard-500, DAQCard-700, and Lab and 1200 Series boards can be diagrammed as shown in Figure 8-11.

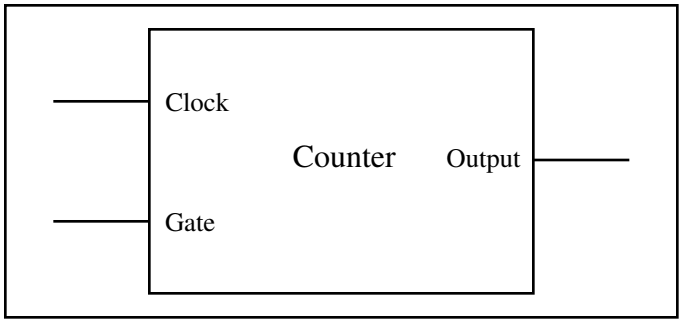

Figure 8-11. Interval Counter Block Diagram

Each counter has a Clock input, a Gate input, and an output labeled CLK, GAT, and OUT, respectively. The CLK pin for Counters B1 and B2 and the GAT and OUT pins for Counters B0, B1, and B2 are available on the Lab and 1200 Series I/O connector.

A counter can be used to count the falling edges of the signal applied to the CLK input. The counter GAT input is used to gate counting operations. Refer to the 8253 data sheet included in your board user manual for information on how the GAT input affects the counting operation in different counting modes.

Interval Counter/Timer Function Summary

Use the following functions for interval counter operations on the Lab and 1200 Series boards:

`ICTR_Read`	Returns the current contents of the selected interval counter without disturbing the counting process.
`ICTR_Reset`	Resets the interval counter output to the specified state.
`ICTR_Setup`	Configures the selected interval counter to operate in the specified mode.

Interval Counting Function Application Hints

Lab and 1200 Series

Note: ***All of the Lab and 1200 series boards have onboard 82C53 Programmable Interval Chips (CTRs), except the Lab-NB, which has onboard 8253 CTRs. Operation modes for your board's CTRs are documented in our Lab or 1200 series user manual.***

The Lab and 1200 series boards contain two onboard CTRs that provide three independent 16-bit counter/timers each. One of these, CTR-A, is reserved for data acquisition and waveform generation operations. The three counters on the other chip, CTR-B, are available for counting/timing operations. The connection of the CTR-B counter/timer signals to the Lab-NB I/O connector is shown in Figure 8-12.

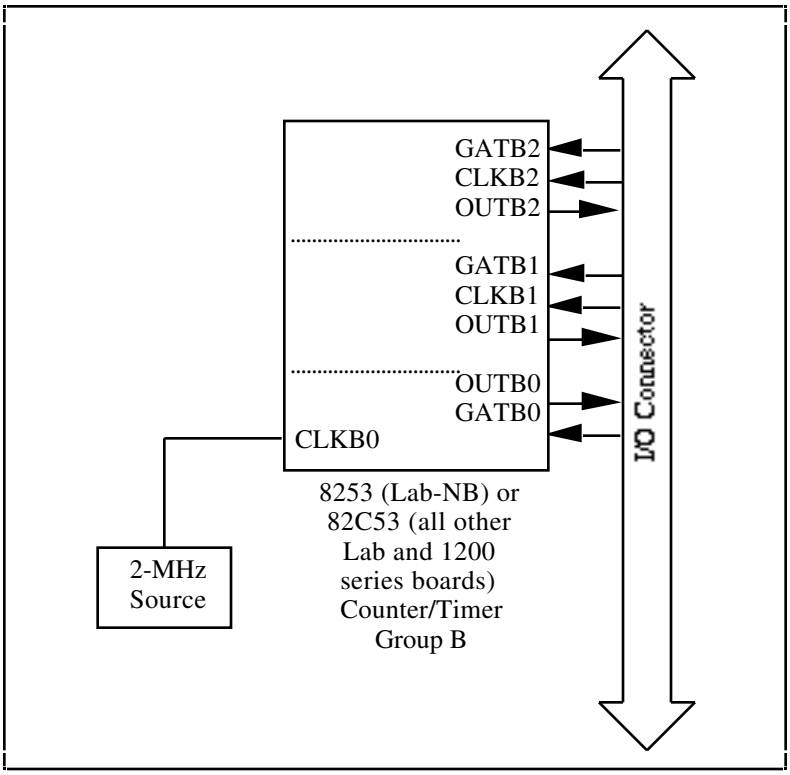

Figure 8-12. Lab and 1200 Series Counter/Timer Signal Connections

CLKB0 is driven by an internal 2 MHz clock. You should supply the signals for CLKB1, CLKB2, GATB1, and GATB2 through the I/O connector.

Counter B0 is used either as a general-purpose counter/timer or for data acquisition/waveform generation timing. Counter B0 is used for data acquisition (waveform generation) if the total sample (update) interval is greater than 65,535 µs. If Counter B0 is used for data acquisition/waveform generation, then it is not available as a general-purpose counter/timer through an `ICTR_Setup` or `ICTR_Reset` call. Similarly, when the counter is used as a general-purpose counter/timer through an `ICTR_Setup` call, it is no longer available for data acquisition/waveform generation. In this case, an `ICTR_Reset` call on Counter B0 makes it available for data acquisition timing.

Counters B1 and B2 are always available for counting/timing operations.

DAQCard-500 and DAQCard-700

The DAQCard-500 and DAQCard-700 contain an onboard MSM82C53 Programmable Interval Timer chip that has three independent 16-bit counter/timers. Counter 0 is used for data acquisition operations. Counter 1 is available for counting/timing operations. Counter 2 can be reserved for Track*/Hold manipulation for the SCXI-1140 (with the DAQCard-700 only), and is otherwise available for counting/timing operations.

Figure 8-13 shows the connections of the MSM82C53 Counter/Timer signals to the DAQCard-500 and DAQCard-700 I/O connector.

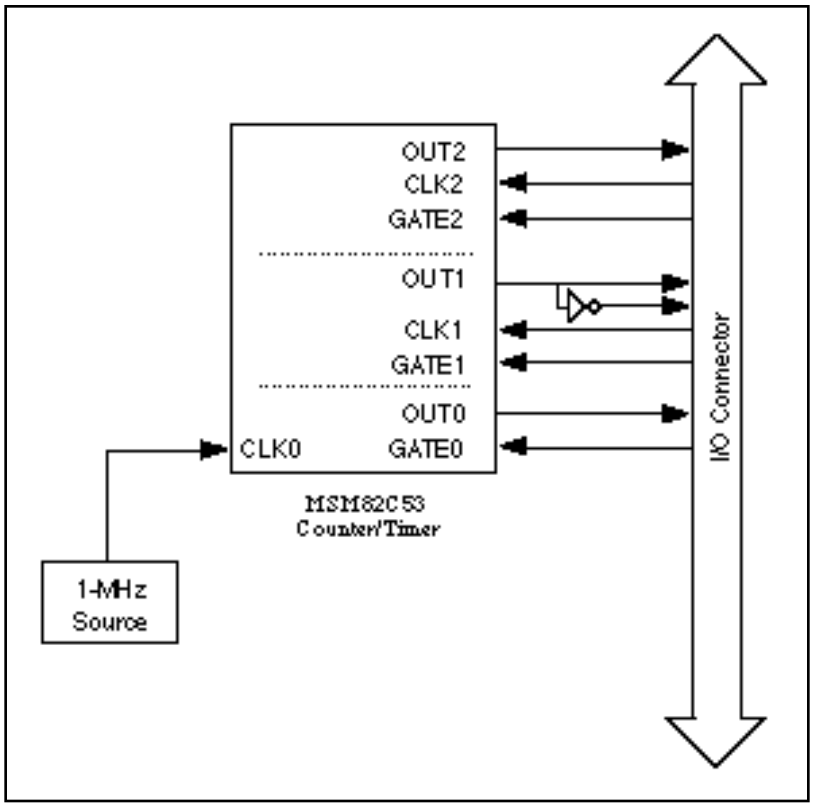

Figure 8-13. DAQCard-500 and DAQCard-700 I/O Counter/Timer Signal Connections

The counter has a clock input, a gate input, and an output labeled CLK, GATE, and OUT, respectively. The CLK pin for counters 1 and 2 and the GATE and OUT pins for counters 0, 1, and 2 are available on the device I/O connector. The inverted OUT1 signal is also available on the DAQCard-700 I/O connector.

The inverted OUT1 signal is not available on the DAQCard-500.

ICTR_Read

Function

Returns the current contents of the selected interval counter without disturbing the counting process.

Synopsis

C Syntax	`locus i32 ICTR_Read(u32 deviceNumber, u32 counter, u32 *count);`
Pascal Syntax	`function ICTR_Read(deviceNumber : i32; counter : i32; var count : i32) : i32;`
BASIC Syntax	`FN ICTR_Read(deviceNumber&, counter&, count&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.
Range: 0 through 2.

count returns the current count of the specified counter while the counter is counting down. **count** can be between 0 and 65,535 if `ICTR_Setup` has not been called since startup or if the last call to `ICTR_Setup` configured **counter** in binary counting mode. **count** can be between 0 and 9,999 if the last call to `ICTR_Setup` configured **counter** in binary-coded decimal (BCD) counting mode.

ICTR_Reset

Function

Resets the interval counter output to the specified state.

Synopsis

C Syntax	`locus i32 ICTR_Reset(u32 deviceNumber, u32 counter, u32 outputState);`
Pascal Syntax	`function ICTR_Reset(deviceNumber : i32; counter : i32; outputState : i32) : i32;`
BASIC Syntax	`FN ICTR_Reset(deviceNumber&, counter&, outputState&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.
Range: 0 through 2.

outputState is the logic state to which the counter is reset.
Range: 0 or 1.

If **outputState** is 0, the counter output is forced low by programming the specified counter in Mode 0. The count register is *not* loaded; thus, the output remains low until the counter is programmed in another mode.

If **outputState** is 1, the counter output is forced high by programming the given counter in Mode 2. The count register is not loaded; thus, the output remains high until the counter is programmed in another mode.

ICTR_Setup

Function

Configures the selected interval counter to operate in the specified mode.

Synopsis

C Syntax	`locus i32 ICTR_Setup(u32 deviceNumber, u32 counter, u32 mode, u32 count, u32 counterType);`
Pascal Syntax	`locus i32 ICTR_Setup(u32 deviceNumber, u32 counter, u32 mode, u32 count, u32 counterType);`
BASIC Syntax	`FN ICTR_Setup(deviceNumber&, counter&, mode&, count&, counterType&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

counter is the counter number.
Range: 0 through 2.

mode is the mode in which the counter is to operate.
Range: 0: Toggles low-to-high on terminal count.
1: Programmable one-shot
2: Rate generator.
3: Square wave rate generator.
4: Software-triggered strobe.
5: Hardware-triggered strobe.

In Mode 0, the output goes low after the `ICTR_Setup` operation, and the counter begins to count down while the gate input is high. The output goes high when the terminal count is reached (that is, the counter has decremented to 0) and stays high until the selected counter is set to a different mode.

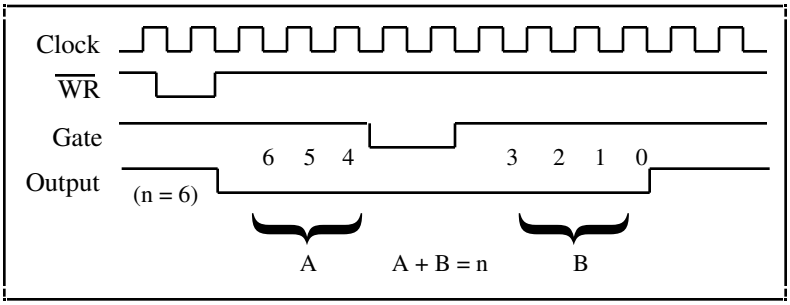

Figure 8-14. Mode 0 Timing Diagram

In Mode 1, the output goes low on the count following the rising edge of the gate input and goes high on terminal count.

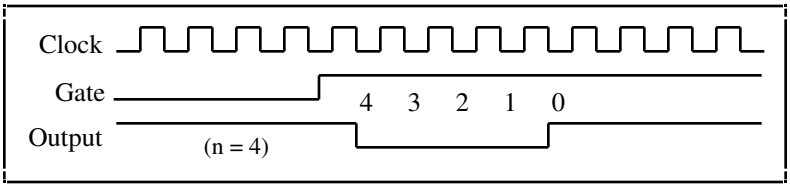

Figure 8-15. Mode 1 Timing Diagram

In Mode 2, the output goes low for one period of the clock input. **count** indicates the period from one output pulse to the next.

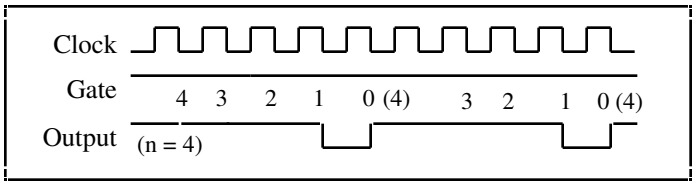

Figure 8-16. Mode 2 Timing Diagram

In Mode 3, the output stays high for one half of the **count** clock pulses and stays low for the other half.

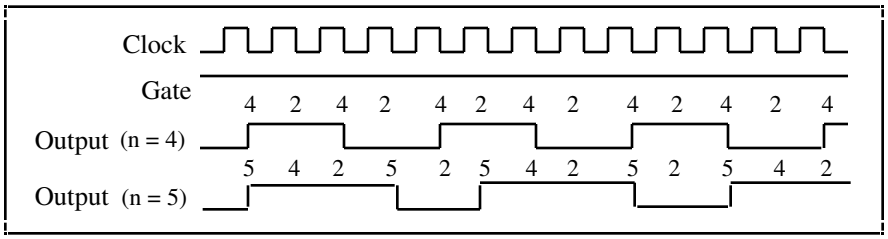

Figure 8-17. Mode 3 Timing Diagram

In Mode 4, the output is initially high, and the counter begins to count down while the gate input is high. On terminal count, the output goes low for one clock pulse, then goes high again.

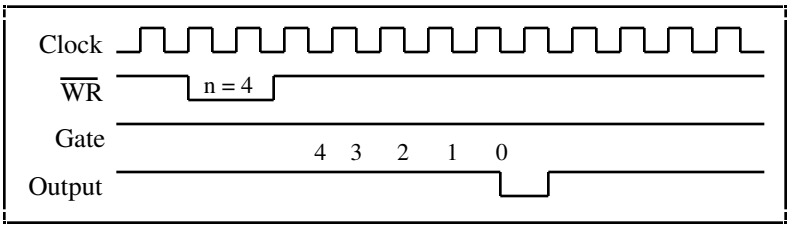

Figure 8-18. Mode 4 Timing Diagram

Mode 5 is similar to Mode 4 except that the gate input is used as a trigger to start counting.

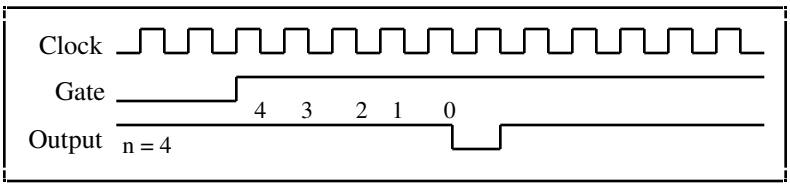

Figure 8-19. Mode 5 Timing Diagram

Note: ***All of the Lab and 1200 series boards have onboard 82C53 CTRs except the Lab-NB, which has onboard 8253 CTRs. Operation modes for your board's CTRs are further documented in your Lab or 1200 series user manual.***

count is the period from one output pulse to the next.
Range for Modes 0, 1, 4 and 5: 0 through 65,535 in binary counter operation
0 through 9,999 in BCD counter operation
Range for Modes 2 and 3: 2 through 65,535 and 0 in binary counter operation
2 through 9,999 and 0 in BCD counter operation

Note: ***0 is equivalent to 65,536 in binary counter operations and 10,000 in BCD counter operations.***

After the **count** value is written by the `ICTR_Setup` call, the **count** value is loaded into the counter register on the falling edge following a rising edge. Any read of the counter prior to that falling clock edge can yield invalid data.

counterType controls whether the counter operates as a 16-bit binary counter or as a 4-decade BCD counter.
0: 4-decade BCD counter
1: 6-bit binary counter

General-Purpose Counter/Timer Function Summary

The General-Purpose Counter/Timer (GPCTR) functions perform counting and timing operations on the E Series devices:

`GPCTR_Change_Parameter`	Customizes the counter operation to fit the requirements of your application by selecting a specific parameter setting.
`GPCTR_Configure_Buffer`	Assigns the buffer that NIDAQ will use for a buffered counter operation.
`GPCTR_Control`	Controls the operation of the general-purpose counter.
`GPCTR_Set_Application`	Selects the application for which you will use the general-purpose counter. The function description contains many application hints.
`GPCTR_Watch`	Monitors the state of the general-purpose counter and its operation.

The General-Purpose Counter/Timer Application Hints

The General-Purpose Counter/Timer (`GPCTR`) functions perform a variety of event counting, time measurement, and pulse and pulse train generation operations, including buffered operations. To start learning about the GPCTR functions, read the introduction to the `GPCTR_Set_Application` description and the sections that pertain to the applications you want to perform with the counter. You can then refer to the descriptions of other GPCTR functions for more details.

GPCTR_Change_Parameter

Function

Selects a specific parameter setting for the general-purpose counter (E Series devices only).

Synopsis

C Syntax	`locus i32 GPCTR_Change_Parameter(u32 deviceNumber, u32 gpctrNum, u32 paramID, u32 paramValue);`
Pascal Syntax	`function GPCTR_Change_Parameter(deviceNumber : i32; gpctrNum : i32; paramID : i32; paramValue : i32) : i32;`
BASIC Syntax	`FN GPCTR_Change_Parameter(deviceNumber&, gpctrNum&, paramID&, paramValue&)`

Description

Legal ranges for **gpctrNum**, **paramID**, and **paramValue** are given in terms of constants defined in a header file. The header file you should use depends on the language you are using:

- C programmers—`nidaqcns.h`
- Pascal programmers—`nidaq.p`
- BASIC programmers—`nidaq.bas`

Use **gpctrNum** to indicate to NI-DAQ which counter you want to program. Legal values for this parameter are `ND_COUNTER_0` and `ND_COUNTER_1`.

Legal values for **paramValue** depend on **paramID**. The following paragraphs list legal values for **paramID** with explanations and corresponding legal values for **paramValue**:

paramID = `ND_SOURCE`

The general-purpose counter counts transitions of this signal. Corresponding legal values for **paramValue** are as follows:

- `ND_PFI_0` through `ND_PFI_9`—the 10 I/O connector pins
- `ND_RTSI_0` through `ND_RTSI_6`—the seven RTSI lines
- `ND_INTERNAL_20_MHZ` and `ND_INTERNAL_100_KHZ`—the internal timebases
- `ND_OTHER_GPCTR_TC`—terminal count of the other general-purpose counter

Use this function with **paramID** = `ND_SOURCE_POLARITY` to select polarity of transitions to use for counting.

paramID = `ND_SOURCE_POLARITY`

The general-purpose counter counts the transitions of the signal selected by paramID = `ND_SOURCE`. Corresponding legal values for **paramValue** are as follows:

- `ND_LOW_TO_HIGH`—counter counts the low-to-high transitions of the source signal
- `ND_HIGH_TO_LOW`—counter counts the high-to-low transitions of the source signal

paramID = `ND_GATE`

This signal controls the operation of the general-purpose counter in some applications. Corresponding legal values for **paramValue** are as follows:

- `ND_PFI_0` through `ND_PFI_9`—the 10 I/O connector pins
- `ND_RTSI_0` through `ND_RTSI_6`—the seven RTSI lines
- `ND_IN_START_TRIGGER` and `ND_IN_STOP_TRIGGER`—the input section triggers
- `ND_OTHER_GPCTR_OUTPUT`—output of the other general-purpose counter

Use this function with **paramID** = ND_GATE_POLARITY to select polarity of the gate signal.

paramID = ND_GATE_POLARITY

This gate signal controls the operation of the general-purpose counter in some applications. In those applications, you can use polarity of the gate signals to modify behavior of the counter. Corresponding legal values for **paramValue** are as follows:

- ND_POSITIVE
- ND_NEGATIVE

The meaning of the two ND_GATE_POLARITY selections is described in the GPCTR_Set_Application function.

paramID = ND_INITIAL_COUNT

The general-purpose counter starts counting from this number when the counter is configured for one of the simple event counting and time measurement applications. Corresponding legal values for **paramValue** are 0 through $2^{24-1.}$

paramID = ND_COUNT_1, ND_COUNT_2, ND_COUNT_3, ND_COUNT_4

The general-purpose counter uses these numbers for pulse width specification when the counter is configured for one of the simple pulse and pulse train generation applications. For example, when you use the counter for frequency shift keying (FSK), ND_COUNT_1 and ND_COUNT_2 specify the durations of low and high output states for one gate state and ND_COUNT_3 and ND_COUNT_4 specify them for the other gate state. Corresponding legal values for **paramValue** are 2 through $2^{24-1.}$

paramID = ND_AUTOINCREMENT_COUNT

The value specified by ND_COUNT_1 is incremented by the value selected by ND_AUTOINCREMENT_COUNT every time the counter is reloaded with the value specified by ND_COUNT_1**.**

For example, with this feature you can generate retriggerable delayed pulses with incrementally increasing delays. You can then use these pulses for applications such as equivalent time sampling (ETS). Corresponding legal values for **paramValue** are 0 through $2^{8-1.}$

paramID = ND_UP_DOWN

When the application is ND_SIMPLE_EVENT_CNT or ND_BUFFERED_EVENT_CNT, you can use the up or down control options of the DAQ-STC general-purpose counters. The up or down control can be performed by software or hardware.

Software Control

The software up or down control is available by default; if you do not use the GPCTR_Change_Parameter function with **paramID** set to ND_UP_DOWN, the counter will be configured for the software up or down control and will start counting up. To make the counter use the software up or down control and start counting down, use the GPCTR_Change_Parameter function with the **paramID** set to ND_UP_DOWN and the **paramValue** set to ND_COUNT_DOWN. To change the counting direction during counting, use the GPCTR_Control function with the action set to ND_COUNT_UP or ND_COUNT_DOWN.

Hardware Control

If you want to use hardware to control the counting direction, use digital I/O line 6 (GPCTR 0) or 7 (GPCTR 1); the counter will count down when the DIO line is in the low state and up when it is in the high state. Use the GPCTR_Change_Parameter function with the **paramID** set to ND_UP_DOWN and the **paramValue** set to ND_HARDWARE to take advantage of this counter feature.

paramID = ND_OUTPUT_MODE

This value changes the output mode from default toggle (the output of the counter toggles on each terminal count) to pulsed (the output of the counter makes a pulse on each terminal count). The corresponding settings of **paramValue** are ND_TOGGLE and ND_PULSE. This **paramID** is rarely used.

paramID = ND_OUTPUT_POLARITY

This **paramID** allows you to change the output polarity from default positive (the normal state of the output is TTL low) to negative (the normal state of the output is TTL high). The corresponding settings of **paramValue** are ND_POSITIVE and ND_NEGATIVE.

This function lets you customize the counter for your application. You can use this function after the GPCTR_Set_Application function, and before GPCTR_Control with **action** = ND_PREPARE or **action** = ND_PROGRAM. You can call this function as many times as you need to.

GPCTR_Config_Buffer

Function

Assigns a buffer that NI-DAQ will use for a buffered counter operation (E Series devices only).

Synopsis

C Syntax	`locus i32 GPCTR_Config_Buffer(u32 deviceNumber, u32 gpctrNum, u32 reserved, u32 numPoints, u32 *buffer);`
Pascal Syntax	`function GPCTR_Config_Buffer(deviceNumber : i32; gpctrNum : i32; reserved : i32; numPoints : i32; buffer : pi32) : i32;`
BASIC Syntax	`FN GPCTR_Config_Buffer(deviceNumber&, gpctrNum&, reserved&, numPoints&, buffer&)`

Description

The legal range for **gpctrNum** is given in terms of constants defined in a header file. The header file you should use depends on the language you are using:

- C programmers—nidaqcns.h
- Pascal programmers—nidaq.p
- BASIC programmers—nidaq.bas

Use **gpctrNum** to indicate to NI-DAQ which counter you want to program. Legal values for this parameter are ND_COUNTER_0 and ND_COUNTER_1.

numPoints is the number of data points the **buffer** can hold. The definition of a data point depends on the application the counter is used for. Legal range is 2 through 2^{32-1}.

When you use the counter for one of the buffered event counting or buffered time measurement operations, a data point is a single counted number.

buffer is an array of U32 or an NI_DAQ_Mem array.

You need to use this function if you want to use a general-purpose counter for buffered operation. You should call this function after calling the GPCTR_Set_Application function.

NI-DAQ transfers counted values into the **buffer** assigned by this function when you are performing a buffered counter operation.

If you are using the general-purpose counter for ND_BUFFERED_PERIOD_MSR, ND_BUFFERED_SEMI_PERIOD_MSR, or ND_BUFFERED_PULSE_WIDTH_MSR, you should wait for the operation to be completed before accessing the buffer.

GPCTR_Control

Function

Controls the operation of the general-purpose counter (E Series devices only).

Synopsis

C Syntax	`locus i32 GPCTR_Control(u32 deviceNumber, u32 gpctrNum, u32 action);`
Pascal Syntax	`function GPCTR_Control(deviceNumber : i32; gpctrNum : i32; action : i32) : i32;`
BASIC Syntax	`FN GPCTR_Control(deviceNumber&, gpctrNum&, action&)`

Description

Legal ranges for the **gpctrNum** and **action** are given in terms of constants defined in a header file. The header file you should use depends on the language you are using:

- C programmers—`nidaqcns.h`
- Pascal programmers—`nidaq.p`
- BASIC programmers—`nidaq.bas`

Use **gpctrNum** to indicate to NI-DAQ which counter you want to program. Legal values for this parameter are `ND_COUNTER_0` and `ND_COUNTER_1`.

action is what you want NI-DAQ to perform with the counter. Legal values for this parameter are as follows:

Action	Description
`ND_PREPARE`	Prepare the general-purpose counter for the operation selected by invocations of the `GPCTR_Set_Application` and (optionally) `GPCTR_Change_Parameter` function. Do not arm the counter.
`ND_ARM`	Arm the general-purpose counter.
`ND_PROGRAM`	`ND_PREPARE` and then `ND_ARM` the counter.
`ND_RESET`	Reset the general-purpose counter.
`ND_COUNT_UP`	Change the counting direction to UP.
`ND_COUNT_DOWN`	Change the counting direction to DOWN.

You need to use this function with **action** = `PROGRAM` after completing the configuration sequence consisting of calling `GPCTR_Set_Application` followed by optional calls to `GPCTR_Change_Parameter` and `GPCTR_Config_Buffer`.

Use the `ND_PREPARE` and `ND_ARM` actions to program the counter before arming. You may find this useful if it is critical to minimize time between a software event (a call to `GPCTR_Control`) and a hardware action (counter starts counting).

You can use this function with **action** = ND_RESET whenever you want to halt the operation the general-purpose counter is performing.

Use actions ND_COUNT_UP and ND_COUNT_DOWN to change the counting direction. You can do this only when your application is ND_SIMPLE_EVENT_CNT or ND_BUFFERED_EVENT_CNT and the counter is configured for software control of the counting direction (up/down).

GPCTR_Set_Application

Function

Selects the application for which you will use the general-purpose counter (E Series devices only).

Synopsis

C Syntax	`locus i32 GPCTR_Set_Application(u32 deviceNumber, u32 gpctrNum, u32 application);`
Pascal Syntax	`function GPCTR_Set_Application(deviceNumber : i32; gpctrNum : i32; application : i32) : i32;`
BASIC Syntax	`FN GPCTR_Set_Application(deviceNumber&, gpctrNum&, application&)`

Description

Legal ranges for **gpctrNum** and **application** are given in terms of constants that are defined in a header file. The header file you should use depends on the language you are using:

- C programmers—nidaqcns.h
- Pascal programmers—nidaq.p
- BASIC programmers—nidaq.bas

Use **gpctrNum** to indicate to NI-DAQ which counter you want to program. Legal values for this parameter are ND_COUNTER_0 and ND_COUNTER_1.

application can be one of the following:

Group	Application	Description
Simple Counting and Time Measurement	ND_SIMPLE_EVENT_CNT	Simple event counting
	ND_SINGLE_PERIOD_MSR	Simple single period measurement
	ND_SINGLE_PULSE_WIDTH_MSR	Simple single pulse-width measurement
	ND_TRIG_PULSE_WIDTH_MSR	Pulse-width measurement you can use for recurring pulses.
Simple Pulse and Pulse Train Generation	ND_SINGLE_PULSE_GNR	Generation of a single pulse
	ND_SINGLE_TRIG_PULSE_GNR	Generation of a single triggered pulse
	ND_RETRIG_PULSE_GNR	Generation of a retriggerable single pulse
	ND_PULSE_TRAIN_GNR	Generation of pulse train
	ND_FSK	Frequency Shift-Keying
Buffered Counting and Time Measurement	ND_BUFFERED_EVENT_CNT	Buffered, asynchronous event counting
	ND_BUFFERED_PERIOD_MSR	Buffered, asynchronous period measurement
	ND_BUFFERED_SEMI_PERIOD_MSR	Buffered, asynchronous semi-period measurement
	ND_BUFFERED_PULSE_WIDTH_MSR	Buffered, asynchronous pulse-width measurement
_CNT stands for *Counting* _MSR stands for *Measurement* _GNR stands for *Generation*		

NI-DAQ requires you to select a set of parameters so that it can program the counter hardware. Those parameters include, for example, signals to be used as counter source and gate and the polarities of those signals. A full list of the parameters is given in the description of the GPCTR_Change_Parameter function. By using the GPCTR_Set_Application function, you assign specific values to all of those parameters. If you do not like some of the settings used by this function, you can alter them by using the GPCTR_Change_Parameter function.

The behavior of the counter you are preparing for an application with this function will depend on **application**, your future calls of the GPCTR functions, and the signals supplied to the counter. The following paragraphs illustrate typical scenarios.

application = ND_SIMPLE_EVENT_CNT

In this application, the counter is used for simple counting of events. By default, the events are low-to-high transitions on the PFI8/GPCTR0_SOURCE I/O connector pin for general-purpose counter 0 and the PFI3/GPCTR1_SOURCE I/O connector pin for general-purpose counter 1. The counter counts up starting from 0, and it is not gated.

Figure 8-20 shows one possible scenario of a counter used for ND_SIMPLE_EVENT_CNT after the following programming sequence:

```
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum, ND_SIMPLE_EVENT_CNT)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-20:

- Source is the signal present at the counter source input
- Count is the value you would read from the counter if you called the GPCTR_Watch function with **entityID** = ND_COUNT. The different numbers illustrate behavior at different times.

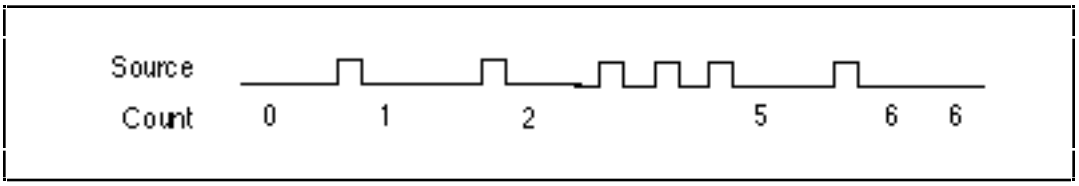

Figure 8-20. Simple Event Counting

The following pseudo-code continuation of the example given earlier illustrates what you can do if you want to continuously read the counter value (GPCTR_Watch function with **entityID** = ND_COUNT does this) and print it:

```
Repeat Forever
{
GPCTR_Watch(deviceNumber, gpctrNum, COUNT, counterValue)
Output counterValue.
}
```

When the counter reaches 2^{24}-1 (Terminal Count) it rolls over and keeps counting. If you want to check if this occurred, use GPCTR_Watch function with **entityID** set to ND_TC_REACHED.

Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_SIMPLE_EVENT_CNT. You can change the following:

- ND_SOURCE to any value
- ND_SOURCE_POLARITY to ND_HIGH_TO_LOW

You can use the GPCTR_Change_Parameter function after calling GPCTR_Set_Application and before calling GPCTR_Control with **action** = ND_PROGRAM or ND_PREPARE.

application = ND_SINGLE_PERIOD_MSR

In this application, the counter is used for a single measurement of the time interval between two transitions of the same polarity of the gate signal. By default, the events are low-to-high transitions on the PFI9/GPCTR0_GATE I/O connector pin for general-purpose counter 0 and the PFI4/GPCTR1_GATE I/O connector pin for general-purpose counter 1. The counter counts the 20 MHz internal timebase (ND_INTERNAL_20_MHZ), so the resolution of measurement is 50 ns. The counter counts up starting from 0.

With the default 20 MHz timebase, combined with the counter width (24 bits), you can measure a time interval between 100 ns and 0.8 s long.

Figure 8-21 shows one possible scenario of a counter used for ND_SINGLE_PERIOD_MSR after the following programming sequence:

```
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum, ND_SINGLE_PERIOD_MSR)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-21:

- Gate is the signal present at the counter gate input.
- Source is the signal present at the counter source input.
- Count is the value you would read from the counter if you called the GPCTR_Watch function with **entityID** = ND_COUNT. The different numbers illustrate behavior at different times.
- Armed is the value you would read from the counter if you called the GPCTR_Watch function with **entityID** = ND_ARMED. The different values illustrate behavior at different times.

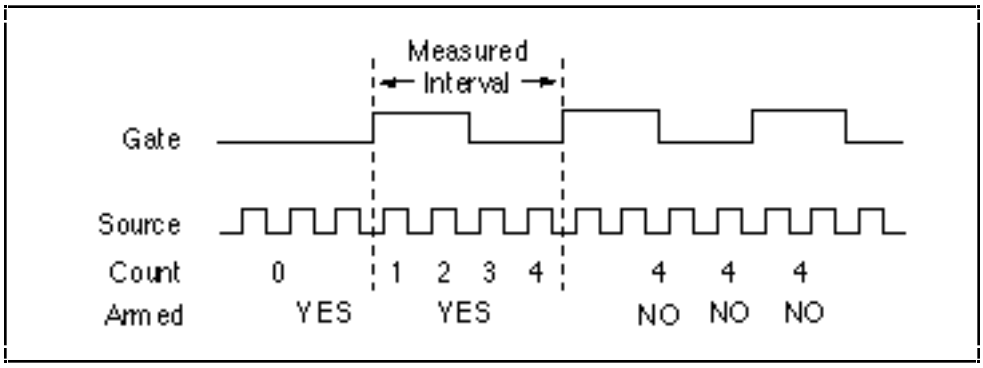

Figure 8-21. Single Period Measurement

Use the GPCTR_Watch function with **entityID** = ND_ARMED to monitor the progress of the counting process. This measurement completes when **entityValue** becomes ND_NO. When counter is no longer armed, you can retrieve the counted value by using GPCTR_Watch with **entityID** = ND_COUNT. You can do this as follows:

```
Create U32 variable counter_armed.
Create U32 variable counted_value.
repeat
{
GPCTR_Watch(deviceNumber, gpctrNumber, ND_ARMED, counter_armed)
}
until (counter_armed = ND_NO)
GPCTR_Watch(deviceNumber, gpctrNumber, ND_COUNT, counted_value)
```

To calculate the measured interval, you need to multiply the counted value by the period corresponding to the timebase you are using. For example, if your ND_SOURCE is ND_INTERNAL_20_MHZ, the interval will be 1/(20 MHz) = 50 ns. If the ND_COUNT is 4 (Figure 8-21), the actual interval is 4 * 50 ns = 200 ns.

When the counter reaches 2^{24}-1 (Terminal Count), it rolls over and keeps counting. If you want to check if this occurred, use the GPCTR_Watch function with **entityID** set to ND_TC_REACHED.

Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_SINGLE_PERIOD_MSR. You can change the following:

- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can measure the time interval between 20 µs and 160 s. The resolution will be lower than if you are using the ND_INTERNAL_20_MHZ timebase.
- ND_SOURCE_POLARITY to ND_HIGH_TO_LOW.
- ND_GATE to any legal value listed in the GPCTR_Change_Parameter function description.
- ND_GATE_POLARITY to ND_NEGATIVE. The interval will be measured from a high-to-low to the next high-to-low transition of the gate signal.

You can use the GPCTR_Change_Parameter function after calling GPCTR_Set_Application and before calling GPCTR_Control with **action** = ND_PROGRAM or ND_PREPARE.

If you want to provide your timebase, you can connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.

You can also configure the other general-purpose counter for ND_PULSE_TRAIN_GNR and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to measure intervals longer than 160 s.

application = ND_SINGLE_PULSE_WIDTH_MSR

In this application, the counter is used for a single measurement of the time interval between two transitions of the opposite polarity of the gate signal. By default, the measurement is performed between a low-to-high and a high-to-low transition on the PFI9/GPCTR0_GATE I/O connector pin for general-purpose counter 0 and the PFI4/GPCTR1_GATE I/O connector pin for general-purpose counter 1. The counter counts the 20 MHz internal timebase (INTERNAL_20_MHZ), so the resolution of measurement is 50 ns. The counter counts up starting from 0.

The default 20 MHz timebase, combined with the counter width (24 bits), lets you measure the duration of a pulse between 100 ns and 0.8 s long.

Figure 8-22 shows one possible scenario of a counter used for ND_SINGLE_PULSE_WIDTH_MSR after the following programming sequence:

```
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum, ND_SINGLE_PULSE_WIDTH_MSR)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-22:

- Gate is the signal present at the counter gate input.
- Source is the signal present at the counter source input.
- Count is the value you would read from the counter if you called the GPCTR_Watch function with **entityID** = ND_COUNT. The different numbers illustrate behavior at different times.

- Armed is the value you would read from the counter if you called the `GPCTR_Watch` function with **entityID** = `ND_ARMED`. The different values illustrate behavior at different times.

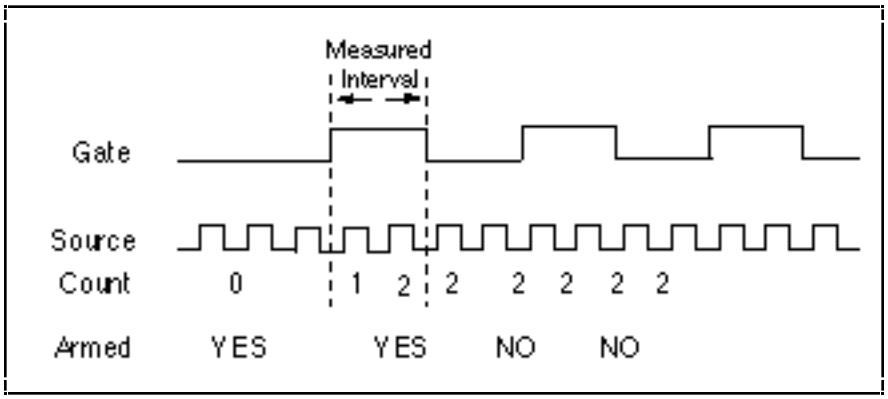

Figure 8-22. Single-Pulse Width Measurement

Use the `GPCTR_Watch` function with **entityID** = `ND_ARMED` to monitor the progress of the counting process. This measurement completes when **entityValue** becomes `ND_NO`. When the counter is no longer armed, you can retrieve the counted value by using `GPCTR_Watch` with **entityID** = `ND_COUNT`, as shown in the following example code:

```
Create U32 variable counter_armed.
Create U32 variable counted_value.
repeat
{
GPCTR_Watch(deviceNumber, gpctrNumber, ND_ARMED, counter_armed)
}
until (counter_armed = ND_NO)
GPCTR_Watch(deviceNumber, gpctrNumber, ND_COUNT, counted_value)
```

To calculate the measured interval, multiply the counted value by the period corresponding to the timebase you are using. For example, if your `ND_SOURCE` is `ND_INTERNAL_20_MHZ`, the interval will be 1/(20 MHz) = 50 ns. If the `ND_COUNT` is 4 (Figure 8-21), the actual interval is 4 * 50 ns = 200 ns.

When the counter reaches 2^{24}-1 (Terminal Count), it rolls over and keeps counting. If you want to check if this occurred, use the `GPCTR_Watch` function with **entityID** set to `ND_TC_REACHED`.

Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_SINGLE_PULSE_WIDTH_MSR. You can change the following:

- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can measure pulse widths between 20 µs and 160 s. The resolution will be lower than if you are using the ND_INTERNAL_20_MHZ timebase.
- ND_SOURCE_POLARITY to ND_HIGH_TO_LOW.
- ND_GATE to any legal value listed in the GPCTR_Change_Parameter function description.
- ND_GATE_POLARITY to ND_NEGATIVE. The pulse width will be measured from a high-to-low to the next low-to-high transition of the gate signal.

You can use the GPCTR_Change_Parameter function after calling GPCTR_Set_Application and before calling GPCTR_Control with **action** = ND_PROGRAM or ND_PREPARE.

If you want to provide your timebase, connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.

You can also configure the other general-purpose counter for ND_PULSE_TRAIN_GNR and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to measure pulse widths longer than 160 s.

Warning: ***Application*** ND_SINGLE_PULSE_WIDTH_MSR ***will work as described only if the gate signal stays in the low state when*** ND_GATE_POLARITY ***is*** ND_POSITIVE***, or if the signal stays in the high state when*** ND_GATE_POLARITY ***is*** ND_NEGATIVE ***while*** GPCTR_Control ***is executed with*** **action =** ND_ARM ***or*** **action =** ND_PROGRAM***. If this criterion is not met, executing*** GPCTR_Control ***with*** **action =** ND_ARM ***or*** **action =** ND_PROGRAM ***returns an error. If this happens, you should not rely on values returned by*** GPCTR_Watch**.**

application = ND_TRIG_PULSE_WIDTH_MSR

In this application, the counter is used for a single measurement of the time interval between two transitions of the opposite polarity of the gate signal. By default, the measurement is performed between a low-to-high and a high-to-low transition on the PFI9/GPCTR0_GATE I/O connector pin for general-purpose counter 0 and the PFI4/GPCTR1_GATE I/O connector pin for general-purpose counter 1. The counter counts the 20 MHz internal timebase (INTERNAL_20_MHZ), so the resolution of measurement is 50 ns. The counter counts up starting from 0.

Unlike ND_SINGLE_PULSE_WIDTH_MSR, your gate signal can change state during counter arming. However, the counter will start counting only after a high-to-low edge on the gate if the gate polarity is positive, or after a low-to-high edge on the gate if the gate polarity is negative. This transition is the trigger from this application's name.

The default 20 MHz timebase, combined with the counter width (24 bits), lets you measure the duration of a pulse between 100 ns and 0.8 s long.

Figure 8-23 shows one possible scenario of a counter used for ND_TRIG_PULSE_WIDTH_MSR after the following programming sequence:

```
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum, ND_TRIG_PULSE_WIDTH_MSR)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-23:

- Gate is the signal present at the counter gate input.
- Source is the signal present at the counter source input.
- Count is the value you would read from the counter if you called the `GPCTR_Watch` function with **entityID** = `ND_COUNT`. The different numbers illustrate behavior at different times.
- Armed is the value you would read from the counter if you called the `GPCTR_Watch` function with **entityID** = `ND_ARMED`. The different values illustrate behavior at different times.

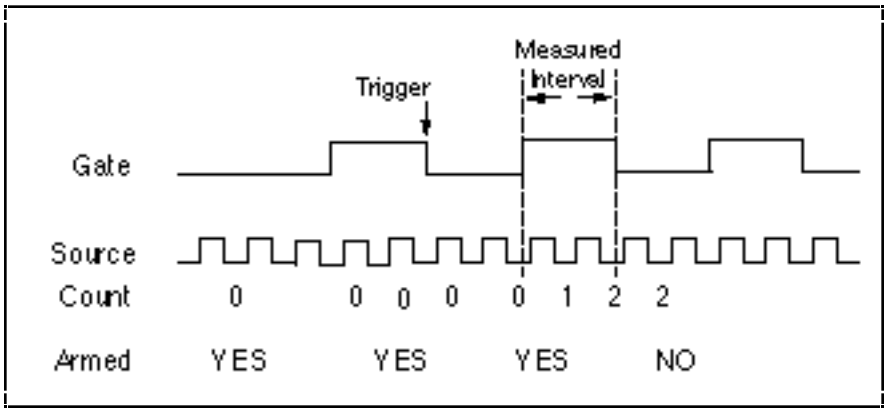

Figure 8-23. Single Triggered Pulse Width Measurement

Use the `GPCTR_Watch` function with **entityID** = `ND_ARMED` to monitor the progress of the counting process. This measurement completes when **entityValue** becomes `ND_NO`. When the counter is no longer armed, you can retrieve the counted value by using `GPCTR_Watch` with **entityID** = `ND_COUNT`, as shown in the following example code:

```
Create U32 variable counter_armed.
Create U32 variable counted_value.
repeat
{
GPCTR_Watch(deviceNumber, gpctrNumber, ND_ARMED, counter_armed)
}
until (counter_armed = ND_NO)

GPCTR_Watch(deviceNumber, gpctrNumber, ND_COUNT, counted_value)
```

To calculate the measured interval, multiply the counted value by the period corresponding to the timebase you are using. For example, if your `ND_SOURCE` is `ND_INTERNAL_20_MHZ`, the interval will be 1/(20 MHz) = 50 ns. If the `ND_COUNT` is 4 (Figure 8-21), the actual interval is 4 * 50 ns = 200 ns.

When the counter reaches 2^{24}-1 (Terminal Count), it rolls over and keeps counting. If you want to check if this occurred, use `GPCTR_Watch` function with **entityID** set to `ND_TC_REACHED`.

Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_TRIG_PULSE_WIDTH_MSR. You can change the following:

- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can measure pulse widths between 20 µs and 160 s. The resolution will be lower than if you are using the ND_INTERNAL_20_MHZ timebase.
- ND_SOURCE_POLARITY to ND_HIGH_TO_LOW.
- ND_GATE to any legal value listed in the GPCTR_Change_Parameter function description.
- ND_GATE_POLARITY to ND_NEGATIVE. The pulse width will be measured from a high-to-low to the next low-to-high transition of the gate signal.

You can use the GPCTR_Change_Parameter function after calling GPCTR_Set_Application and before calling GPCTR_Control with **action** = ND_PROGRAM or ND_PREPARE.

If you want to provide your timebase, connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.

You can also configure the other general-purpose counter for ND_PULSE_TRAIN_GNR and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to measure pulse widths longer than 160 s.

application = ND_SINGLE_PULSE_GNR

In this application, the counter is used for the generation of single delayed pulse. By default, you get this through the 20 MHz internal timebase (ND_INTERNAL_20_MHZ), so the resolution of timing is 50 ns. By default, the counter counts down from ND_COUNT_1 = 5 million to 0 for the delay time and then down from ND_COUNT_2 = 10 million to 0 for the pulse generation time to generate a 0.5 s pulse after 0.25 s of delay.

The default 20 MHz timebase, combined with the counter width (24 bits), lets you generate pulses with a delay and length between 100 ns and 0.8 s (each).

For example, assume that you want to generate a pulse 200 ns long after 150 ns of delay. You need to set ND_COUNT_1 to 150 ns/50 ns = 3 and ND_COUNT_2 to 200 ns/50 ns = 4. Figure 8-24 shows a counter used for ND_SINGLE_PULSE_GNR after the following programming sequence:

```
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum, ND_SINGLE_PULSE_GRN)
GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_1, 3)
GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_2, 4)
Select_Signal(deviceNumber, gpctrNumOut, gpctrNumOut,ND_LOW_TO_HIGH)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-24:

- Source is the signal present at the counter source input.
- Output is the signal present at the counter output.
- Armed is the value you would read from the counter if you called the GPCTR_Watch function with **entityID** = ND_ARMED. The different values illustrate behavior at different times.

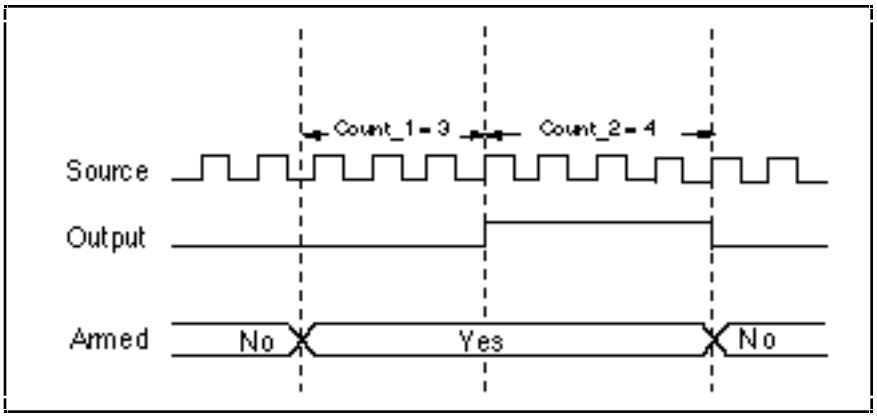

Figure 8-24. Single Pulse Generation

Use the GPCTR_Watch function with **entityID** = ND_ARMED to monitor the progress of the pulse generation process. The generation completes when **entityValue** becomes ND_NO.

You will typically find modification of the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_SINGLE_PERIOD_MSR. You can change the following:

- ND_COUNT_1 and ND_COUNT_2 to any value between 2 and 2^{24} - 1. The defaults are given for illustrative purposes only.
- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can generate pulses with delay and length between 20 µs and 160 s. The timing resolution will be lower than if you are using the ND_INTERNAL_20_MHZ timebase.

You can use the GPCTR_Change_Parameter function after calling GPCTR_Set_Application and before calling GPCTR_Control with **action** = ND_PROGRAM or ND_PREPARE.

If you want to provide your timebase, connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.

You can also configure the other general-purpose counter for ND_PULSE_TRAIN_GNR and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to generate pulses with delays and intervals longer than 160 s.

application = ND_SINGLE_TRIG_PULSE_GRN

In this application, the counter is used for the generation of single delayed pulse after a transition on the gate input. By default, this is achieved by using the 20 MHz internal timebase (ND_INTERNAL_20_MHZ), so the resolution of timing is 50 ns. By default, the counter counts down from ND_COUNT_1 = 5 million to 0 for the delay time, and then down from ND_COUNT_2 = 10 million to 0 for the pulse generation time to generate a 0.5 s pulse after 0.25 s of delay. By default, the gate is PFI9/GPCTR0_GATE I/O connector pin for general-purpose counter 0 and the PFI4/GPCTR1_GATE I/O connector pin for general-purpose counter 1, and the transition which initiates the pulse generation is low-to-high. Only the first transition of the gate signal after you arm the counter initiates pulse generation; all subsequent transitions are ignored.

The default 20 MHz timebase, combined with the counter width (24 bits), lets you generate pulses with delay and length between 100 ns and 0.8 s.

Assume that you want to generate a pulse 200 ns long after 150 ns of delay from the transition of the gate signal. You need to set ND_COUNT_1 to 150 ns/50 ns = 3 and ND_COUNT_2 to 200 ns/50 ns = 4. Figure 8-25

shows the scenario of a counter used for ND_SINGLE_TRIG_PULSE_GRN after the following programming sequence:

```
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum, ND_SINGLE_TRIG_PULSE_GRN)
GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_1, 3)
GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_2, 4)
Select_Signal(deviceNumber, gpctrNumOut, gpctrNumOut,ND_LOW_TO_HIGH)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-25 :

- Gate is the signal present at the counter gate input.
- Source is the signal present at the counter source input.
- Output is the signal present at the counter output.
- Armed is the value you would read from the counter if you called the GPCTR_Watch function with **entityID** = ND_ARMED. The different values illustrate behavior at different times.

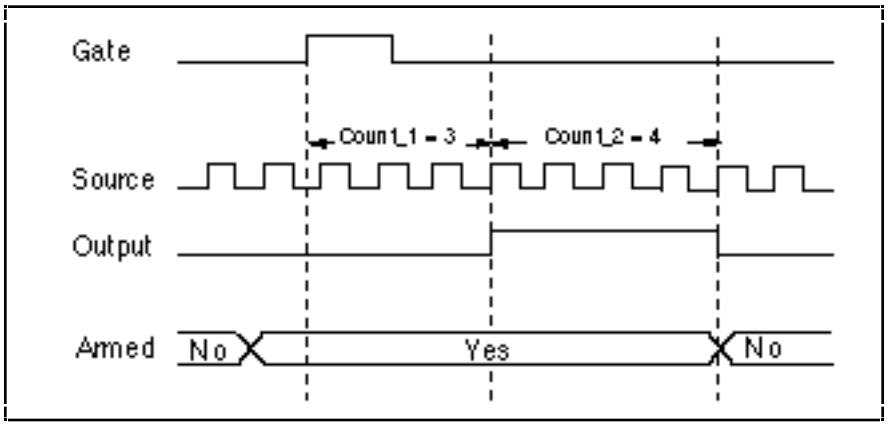

Figure 8-25 . Single Triggered Pulse Generation

Use the GPCTR_Watch function with **entityID** = ND_ARMED to monitor the progress of the pulse generation process. The generation completes when **entityValue** becomes ND_NO.

Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_SINGLE_TRIG_PULSE_GNR. You can change the following:

- ND_COUNT_1 and ND_COUNT_2 to any value between 2 and 2^{24} - 1. The defaults are given for illustrative purposes only.
- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can generate pulses with a delay and length between 20 µs and 160 s. The timing resolution will be lower than if you are using ND_INTERNAL_20_MHZ timebase.
- ND_GATE to any legal value listed in the GPCTR_Change_Parameter function description.
- ND_GATE_POLARITY to ND_NEGATIVE. A high-to-low transition of the gate signal initiates the pulse generation timing.

You can use the GPCTR_Change_Parameter function after calling GPCTR_Set_Application and before calling GPCTR_Control with **action** = ND_PROGRAM or ND_PREPARE.

If you want to provide your timebase, you can connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.

You can also configure the other general-purpose counter for ND_SINGLE_TRIG_PULSE_GRN and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to generate pulses with delays and intervals longer than 160 s.

application = ND_RETRIG_PULSE_GNR

In this application, the counter is used for the generation of a retriggerable delayed pulse after each transition on the gate input. By default, you get this by using the 20 MHz internal timebase (ND_INTERNAL_20_MHZ), so the resolution of timing is 50 ns. By default, the counter counts down from ND_COUNT_1 = 5 million to 0 for the delay time and then down from ND_COUNT_2 = 10 million to 0 for the pulse generation time to generate a 0.5 s pulse after 0.25 s of delay. By default, the gate is the PFI9/GPCTR0_GATE I/O connector pin for general-purpose counter 0 and the PFI4/GPCTR1_GATE I/O connector pin for general-purpose counter 1, and the transition which initiates the pulse generation is low-to-high. All transitions of the gate signal after you arm the counter initiate pulse generation.

The default 20 MHz timebase, combined with the counter width (24 bits), lets you generate pulses with a delay and length between 100 ns and 0.8 s.

For example, assume that you want to generate a pulse 200 ns long after 150 ns of delay from every transition of the gate signal. You need to set ND_COUNT_1 to 150 ns/50 ns = 3 and ND_COUNT_2 to 200 ns/50 ns = 4. Figure 8-26 shows a counter used for ND_RETRIG_PULSE_GNR after the following programming sequence:

```
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum, ND_RETRIG_PULSE_GNR)
GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_1, 3)
GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_2, 4)
Select_Signal(deviceNumber, gpctrNumOut, gpctrNumOut,ND_LOW_TO_HIGH)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-26 :

- Gate is the signal present at the counter gate input.
- Source is the signal present at the counter source input.
- Output is the signal present at the counter output.

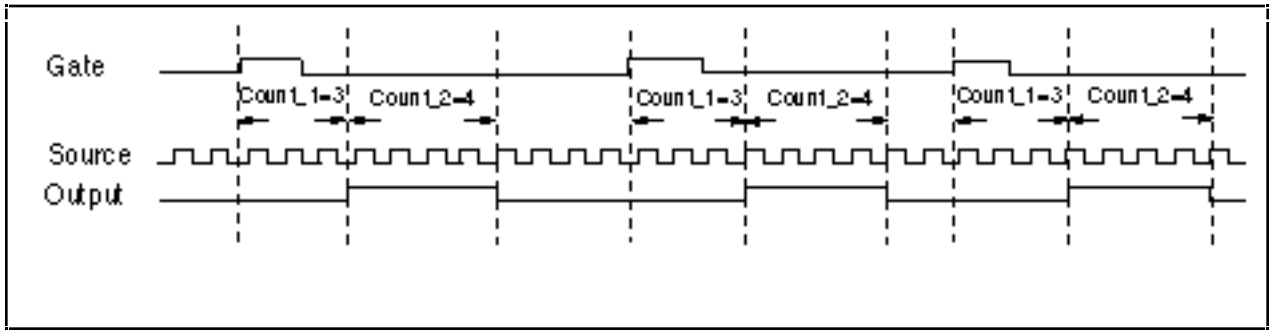

Figure 8-26 . Retriggerable Pulse Generation

Use the GPCTR_Control function with **action** = ND_RESET to stop the pulse generation.
Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_RETRIG_PULSE_GNR. You can change the following:

- ND_COUNT_1 and ND_COUNT_2 to any value between 2 and 2^{24} - 1. The defaults are given for illustrative purposes only.
- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can generate pulses with delay and length between 20 µs and 160 s. The timing resolution will be lower than if you are using ND_INTERNAL_20_MHZ timebase.
- ND_GATE to any legal value listed in the GPCTR_Change_Parameter function description.
- ND_GATE_POLARITY to ND_NEGATIVE. A high-to-low transition of the gate signal initiates the pulse generation timing.

You can use the GPCTR_Change_Parameter function after calling GPCTR_Set_Application and before calling GPCTR_Control with **action** = ND_PROGRAM or ND_PREPARE.

If you want to provide your timebase, you can connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.

You can also configure the other general-purpose counter for ND_RETRIG_PULSE_GNR, and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to generate pulses with delays and intervals longer than 160 s.

application = ND_PULSE_TRAIN_GNR

In this application, the counter is used for generation of a pulse train. By default, you get this by using the 20 MHz internal timebase (ND_INTERNAL_20_MHZ), so the resolution of timing is 50 ns. By default, the counter repeatedly counts down from ND_COUNT_1 = 5 million to 0 for the delay time and then down from ND_COUNT_2 = 10 million to 0 for the pulse generation time to generate a train 0.5 s pulses separated by 0.25 s of delay. Pulse train generation starts as soon as you arm the counter. You must reset the counter to stop the pulse train.

The default 20 MHz timebase, combined with the counter width (24 bits), lets you generate trains consisting of pulses with delay and length between 100 ns and 0.8 s.

Assume that you want to generate a pulse train with the low period 150 ns long and the high period 200 ns long. You need to set ND_COUNT_1 to 150 ns/50 ns = 3 and ND_COUNT_2 to 200 ns/50 ns = 4. This corresponds to a 20 MHz : (3 + 4) = 2.86 MHz signal with (3/7)/(4/7) = 43/57 duty cycle. Figure 8-27 shows the scenario of a counter used for ND_PULSE_TRAIN_GNR after the following programming sequence:

```
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)

   GPCTR_Set_Application(deviceNumber, gpctrNum, ND_PULSE_TRAIN_GNR)
   GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_1, 3)
   GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_2, 4)
   Select_Signal(deviceNumber, gpctrNumOut, gpctrNumOut,ND_LOW_TO_HIGH)
   GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-27

- Source is the signal present at the counter source input.
- Output is the signal present at the counter output.

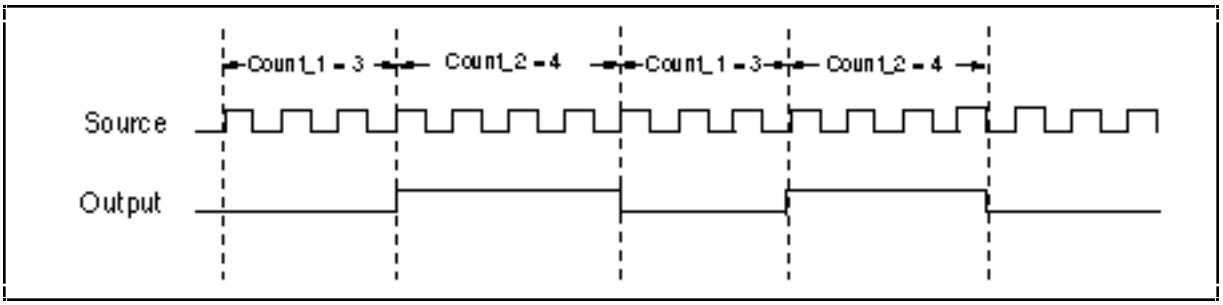

Figure 8-27. Pulse Train Generation

Use the GPCTR_Control function with **action** = ND_RESET to stop the pulse generation.
Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_PULSE_TRAIN_GNR. You can change the following:

- ND_COUNT_1 and ND_COUNT_2 to any value between 2 and 2^{24} - 1. The defaults are given for illustrative purposes only.

- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can generate pulses with delay and length between 20 µs and 160 s. The timing resolution will be lower than if you are using the ND_INTERNAL_20_MHZ timebase.

You can use the GPCTR_Change_Parameter function after calling GPCTR_Set_Application and before calling GPCTR_Control with **action** = ND_PROGRAM or ND_PREPARE.

If you want to provide your timebase, you can connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.

You can also configure the other general-purpose counter for ND_PULSE_TRAIN_GNR, and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to generate pulses with delays and intervals longer than 160 s.

application = ND_FSK

In this application, the counter is used for generation of frequency shift keyed signals. The counter generates a pulse train of one frequency and duty cycle when the gate is low, and a pulse train with different parameters when the gate is high. By default, you get this by using the 20 MHz internal timebase (ND_INTERNAL_20_MHZ), so the resolution of timing is 50 ns. By default, when the gate is low, the counter repeatedly counts down from ND_COUNT_1 = 5 million to 0 for the delay time, and then down from ND_COUNT_2 = 10 million to 0 for the pulse generation time, to generate a train 0.5 s pulses separated by 0.25 s of delay. Also by default, when the gate is high, the counter repeatedly counts down from ND_COUNT_3 = 4 million to 0 for the delay time, and then down from ND_COUNT_4 = 6 million to 0 for the pulse generation time, to generate a train 0.3 s pulses separated by 0.2 s of delay. The FSK pulse generation starts as soon as you arm the counter. You must reset the counter to stop the pulse generation.
The default 20 MHz timebase, combined with the counter width (24 bits), lets you generate pulses with a delay and length between 100 ns and 0.8 s.

Assume that you want to generate a pulse train with 100 ns low and 150 ns high time when the gate is low and with 300 ns low time and 200 ns high time when the gate is high. You need to set ND_COUNT_1 to 100 ns/50 ns = 2, ND_COUNT_2 to 150 ns/50 ns = 3, ND_COUNT_3 to 300 ns/50 ns = 6, and ND_COUNT_4 to 200 ns/50 ns = 4. Figure 8-28 shows a counter used for ND_FSK after the following programming sequence:

```
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum, ND_FSK)
```

```
GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_1, 2)
GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_2, 3)
GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_3, 6)
GPCTR_Change_Parameter(deviceNumber, gpctrNum, ND_COUNT_4, 4)
Select_Signal(deviceNumber, gpctrNumOut, gpctrNumOut,ND_LOW_TO_HIGH)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-28:

- Gate is the signal present at the counter gate input.
- Source is the signal present at the counter source input.
- Output is the signal present at the counter output.

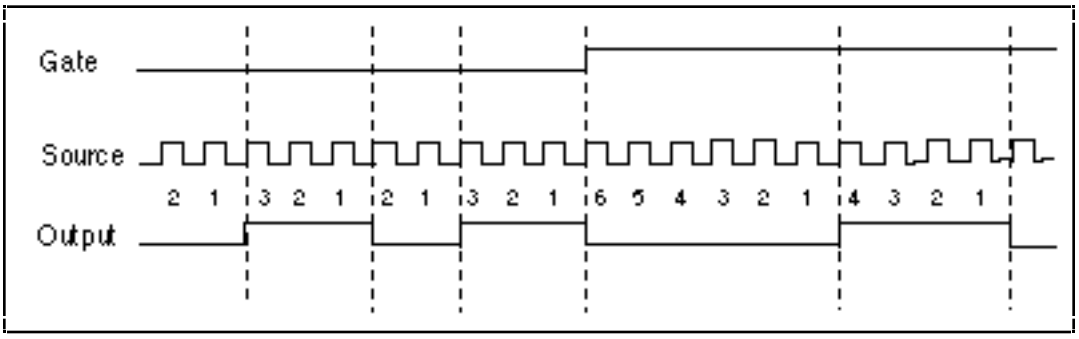

Figure 8-28. Frequency Shift Keying

Use the GPCTR_Control function with **action** = ND_RESET to stop the pulse generation.

Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_RETRIG_PULSE_GNR. You can change the following:

- ND_COUNT_1, ND_COUNT_2, ND_COUNT_3, and ND_COUNT_4 to any value between 2 and 2^{24} - 1. The defaults are given for illustrative purposes only.
- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can generate pulses with a delay and length between 20 µs and 160 s. The timing resolution will be lower than if you are using the ND_INTERNAL_20_MHZ timebase.
- ND_GATE to any legal value listed in the GPCTR_Change_Parameter function description.

You can use the GPCTR_Change_Parameter function after calling GPCTR_Set_Application and before calling GPCTR_Control with **action** = ND_PROGRAM or ND_PREPARE.

If you want to provide your timebase, connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.
You can also configure the other general-purpose counter for ND_FSK, and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to generate pulses with delays and intervals longer than 160 s.

application = ND_BUFFERED_EVENT_CNT

In this application, the counter is used for continuous counting of events. By default, the events are low-to-high transitions on the PFI8/GPCTR0_SOURCE I/O connector pin for general-purpose counter 0 and the PFI3/GPCTR1_SOURCE I/O connector pin for general-purpose counter 1. Counts present at specified events of

the signal present at the gate are saved in a buffer. By default, those events are the low-to-high transitions of the signal on the PFI9/GPCTR0_GATE I/O connector pin for general-purpose counter 0 and the PFI4/GPCTR1_GATE I/O connector pin for general-purpose counter 1. The counter counts up starting from 0; its contents are placed in the buffer after an edge of appropriate polarity is detected on the gate; the counter keeps counting without interruption. NI-DAQ transfers data from the counter into the buffer until the buffer is filled; the counter is disarmed at that time.

The counter width (24 bits) lets you count up to 2^{24}-1 events. Figure 8-29 shows one possible scenario of a counter used for `ND_BUFFERED_EVENT_CNT` after the following programming sequence:

Make buffer be a 100-element array of U32.

```
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum, ND_BUFFERED_EVENT_CNT)
GPCTR_Config_Buffer(deviceNumber, gpctrNum, 0, 100, buffer)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-29:

- Gate is the signal present at the counter gate input.
- Source is the signal present at the counter source input.
- Buffer is the contents of the buffer; you can retrieve data from the buffer when the counter becomes disarmed.

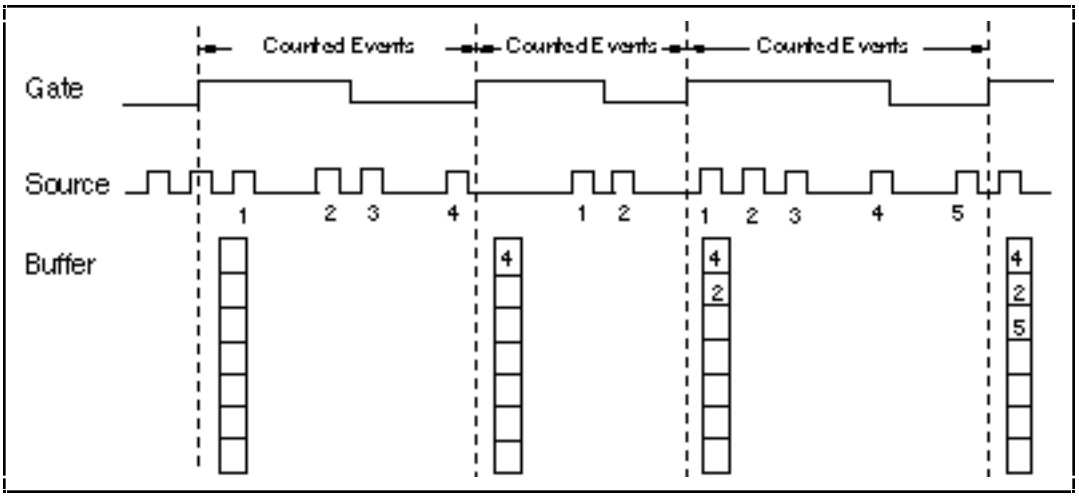

Figure 8-29. Buffered Event Counting

Use the `GPCTR_Watch` function with **entityID** = `ND_ARMED` to monitor the progress of the counting process. This measurement completes when **entityValue** becomes `ND_NO`. You can do this as follows:

```
Create U32 variable counter_armed.
repeat
{
GPCTR_Watch(deviceNumber, gpctrNumber, ND_ARMED, counter_armed)
}
until (counter_armed = ND_NO)
```

When the counter is disarmed, you can safely access data in the buffer.

Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_BUFFERED_PERIOD_MSR. You can change the following:

- ND_SOURCE to any legal value listed in the GPCTR_Change_Parameter function description.
- ND_SOURCE_POLARITY to ND_HIGH_TO_LOW.
- ND_GATE to any legal value listed in the GPCTR_Change_Parameter function description.
- ND_GATE_POLARITY to ND_NEGATIVE. Counts will be captured on every high-to-low transition of the signal present at the gate.

Note: ***The counter will start counting as soon as you arm it. However, it will not count if the gate signal stays in low state when*** ND_GATE_POLARITY ***is*** ND_POSITIVE ***or if it stays in high state when*** ND_GATE_POLARITY ***is*** ND_NEGATIVE ***while*** GPCTR_Control ***is executed with*** **action =** ND_ARM ***or*** **action =** ND_PROGRAM***. Be aware of this when you interpret the first count in your buffer.***

application = ND_BUFFERED_PERIOD_MSR

In this application, the counter is used for continuous measurement of the time interval between successive transitions of the same polarity of the gate signal. By default, those are the low-to-high transitions of the signal on the PFI9/GPCTR0_GATE I/O connector pin for general-purpose counter 0 and the PFI4/GPCTR1_GATE I/O connector pin for general-purpose counter 1. The counter counts the 20 MHz internal timebase (ND_INTERNAL_20_MHZ), so the resolution of measurement is 50 ns. The counter counts up starting from 0; its contents are placed in the buffer after an edge of appropriate polarity is detected on the gate; the counter then starts counting up from 0 again. NI-DAQ transfers data from the counter into the buffer until the buffer is filled; the counter is disarmed at that time.

The default 20 MHz timebase, combined with the counter width (24 bits), lets you measure the width of a pulse between 100 ns and 0.8 s long.

Figure 8-30 shows one possible scenario of a counter used for ND_BUFFERED_PERIOD_MSR after the following programming sequence:

```
Make buffer be a 100-element array of U32.
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum, ND_BUFFERED_PERIOD_MSR)
GPCTR_Config_Buffer(deviceNumber, gpctrNum, 0, 100, buffer)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-30:

- Gate is the signal present at the counter gate input.
- Source is the signal present at the counter source input.
- Buffer is the contents of the buffer; you can retrieve data from the buffer when the counter becomes disarmed.

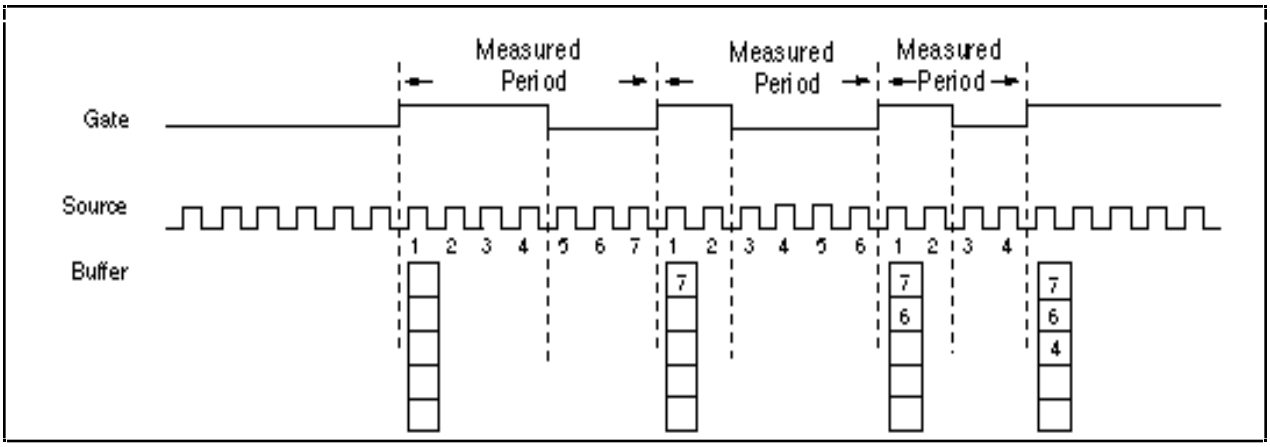

Figure 8-30. Buffered Period Measurement

Use the GPCTR_Watch function with **entityID** = ND_ARMED to monitor the progress of the counting process. This measurement completes when **entityValue** becomes ND_NO, as shown in the following example:

```
Create U32 variable counter_armed.

repeat
{
GPCTR_Watch(deviceNumber, gpctrNumber, ND_ARMED, counter_armed
}
until (counter_armed = ND_NO)
```

When the counter is disarmed, you can safely access data in the buffer.

Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_BUFFERED_PERIOD_MSR. You can change the following:

- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can measure intervals between 20 µs and 160 s long. The resolution will be lower than if you are using ND_INTERNAL_20_MHZ timebase.
- ND_SOURCE_POLARITY to ND_HIGH_TO_LOW.
- ND_GATE to any legal value listed in the GPCTR_Change_Parameter function description.
- ND_GATE_POLARITY to ND_NEGATIVE. Measurements will be performed between successive high-to-low transitions of the signal present at the gate.

If you want to provide your timebase, you can connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.

You can also configure the other general-purpose counter for ND_PULSE_TRAIN_GNR, and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to measure intervals longer than 160 s.

Note: ***The counter will start counting as soon as you arm it. Be aware of this when you interpret the first count in your buffer.***

application = ND_BUFFERED_SEMI_PERIOD_MSR

In this application, the counter is used for the continuous measurement of the time interval between successive transitions of the gate signal. By default, those are all transitions of the signal on the PFI9/GPCTR0_GATE I/O

connector pin for general-purpose counter 0 and the PFI4/GPCTR1_GATE I/O connector pin for general-purpose counter 1. The counter counts the 20 MHz internal timebase (ND_INTERNAL_20_MHZ), so the resolution of measurement is 50 ns. The counter counts up starting from 0; its contents are placed in the buffer after an edge is detected on the gate; the counter then starts counting up from 0 again. NI-DAQ transfers data from the counter into the buffer until the buffer is filled; the counter is disarmed at that time.

The default 20 MHz timebase, combined with the counter width (24 bits), lets you measure the width of a pulse between 100 ns and 0.8 s long.

Figure 8-31 shows one possible scenario of a counter used for ND_BUFFERED_SEMI_PERIOD_MSR after the following programming sequence:

```
Make buffer be a 100-element array of U32.
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum,
ND_BUFFERED_SEMI_PERIOD_MSR)
GPCTR_Config_Buffer(deviceNumber, gpctrNum, 0, 100, buffer)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-31:

- Gate is the signal present at the counter gate input.
- Source is the signal present at the counter source input.
- Buffer is the contents of the buffer; you can retrieve data from the buffer when the counter becomes disarmed.

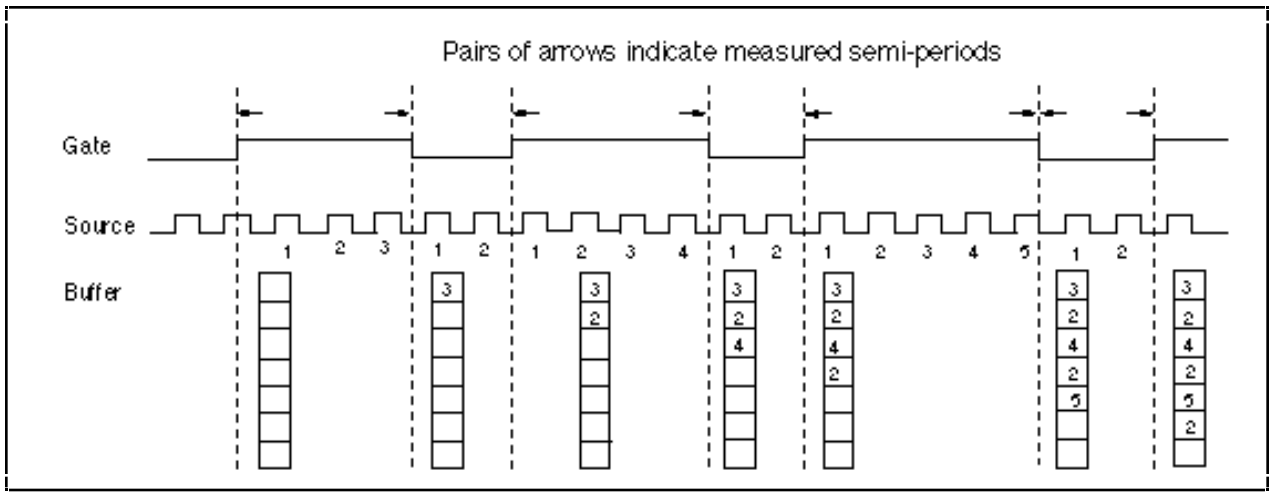

Figure 8-31. Buffered Semi-Period Measurement

Use the GPCTR_Watch function with **entityID** = ND_ARMED to monitor the progress of the counting process. This measurement completes when **entityValue** becomes ND_NO. The following code example shows this process:

```
Create U32 variable counter_armed.
repeat
{
GPCTR_Watch(deviceNumber, gpctrNumber, ND_ARMED, counter_armed)
}
until (counter_armed = ND_NO)
```

When the counter is disarmed, you can safely access data in the buffer.

Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_BUFFERED_SEMI_PERIOD_MSR. You can change the following:

- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can measure intervals between 20 µs and 160 s long. The resolution will be lower than if you are using the ND_INTERNAL_20_MHZ timebase.
- ND_SOURCE_POLARITY to ND_HIGH_TO_LOW.
- ND_GATE to any legal value listed in the GPCTR_Change_Parameter function description.

If you want to provide your timebase, you can connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.

You can also configure the other general-purpose counter for ND_PULSE_TRAIN_GNR and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to measure intervals longer than 160 s.

Note: ***The counter will start counting as soon as you arm it. Be aware of this when you interpret the first count in your buffer.***

application = ND_BUFFERED_PULSE_WIDTH_MSR

In this application, the counter is used for continuous measurement of width of pulses of selected polarity present at the counter gate. By default, those pulses are active high pulses present on the signal on the PFI9/GPCTR0_GATE I/O connector pin for general-purpose counter 0 and the PFI4/GPCTR1_GATE I/O connector pin for general-purpose counter 1. The counter counts the 20 MHz internal timebase (ND_INTERNAL_20_MHZ), so the resolution of measurement is 50 ns. The counter counts up starting from 0; its contents are placed in the buffer after a pulse completes; the counter then starts counting up from 0 again when the next pulse appears. NI-DAQ transfers data from the counter into the buffer until the buffer is filled; the counter is disarmed at that time.

The default 20 MHz timebase, combined with the counter width (24 bits), lets you measure the width of a pulse between 100 ns and 0.8 s long.

When using the buffered counter operations in applications involving period/pulse-width measurements, the first acquired point may represent bad data. The first data point is the measured interval between the instant when the counter is armed and when the first edge transition takes place on the counter GATE. Because there is no deterministic way of specifying when the counter is actually armed, the first value may be incorrect. Subsequent data points acquired will not have this problem.

Figure 8-32 shows one possible scenario of a counter used for ND_BUFFERED_PULSE_WIDTH_MSR after the following programming sequence:

```
Make buffer be a 100-element array of U32.
GPCTR_Control(deviceNumber, gpctrNum, ND_RESET)
GPCTR_Set_Application(deviceNumber, gpctrNum,
ND_BUFFERED_PULSE_WIDTH_MSR)
GPCTR_Config_Buffer(deviceNumber, gpctrNum, 0, 100, buffer)
GPCTR_Control(deviceNumber, gpctrNum, ND_PROGRAM)
```

In Figure 8-32:

- Gate is the signal present at the counter gate input.
- Source is the signal present at the counter source input.

- Buffer is the contents of the buffer; you can retrieve data from the buffer when the counter becomes disarmed.

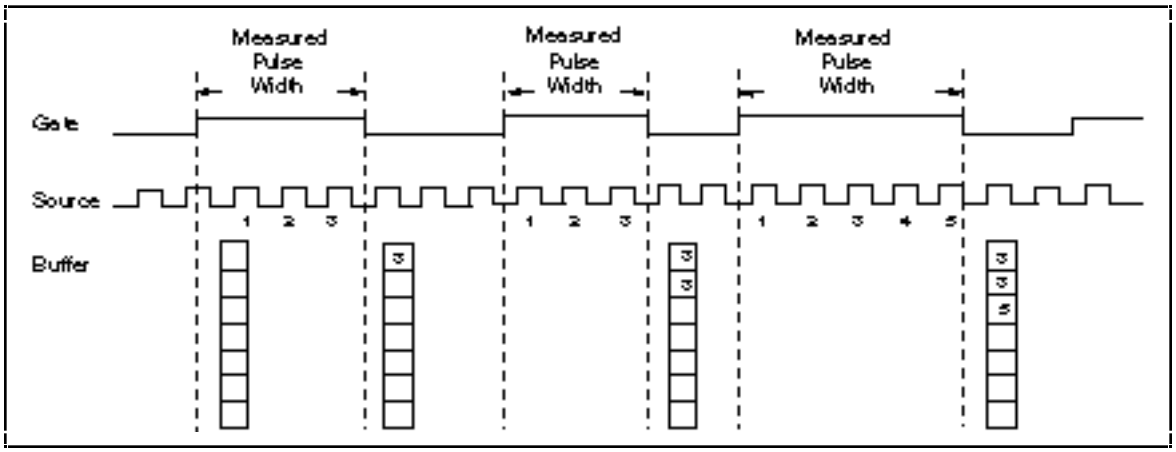

Figure 8-32. Buffered Pulse Width Measurement

Use the GPCTR_Watch function with **entityID** = ND_ARMED to monitor the progress of the counting process.

This measurement completes when **entityValue** becomes ND_NO. You can do this as follows:

```
Create U32 variable counter_armed.

repeat
{
GPCTR_Watch(deviceNumber, gpctrNumber, ND_ARMED, counter_armed)
}
until (counter_armed = ND_NO)
```

When the counter is disarmed, you can safely access data in the buffer.

Typically, you will find modifying the following parameters through the GPCTR_Change_Parameter function useful when the counter **application** is ND_BUFFERED_PULSE_WIDTH_MSR. You can change the following:

- ND_SOURCE to ND_INTERNAL_100_KHZ. With this timebase, you can measure intervals between 20 µs and 160 s long. The resolution will be lower than if you are using ND_INTERNAL_20_MHZ timebase.
- ND_SOURCE_POLARITY to ND_HIGH_TO_LOW.
- ND_GATE to any legal value listed in the GPCTR_Change_Parameter function description.
- ND_GATE_POLARITY to ND_NEGATIVE. Measurements will be performed on the active low pulses.

If you want to provide your timebase, you can connect your timebase source to one of the PFI pins on the I/O connector and change ND_SOURCE and ND_SOURCE_POLARITY to the appropriate values.
You can also configure the other general-purpose counter for ND_PULSE_TRAIN_GNR, and set ND_SOURCE of this counter to ND_OTHER_GPCTR_TC if you want to measure intervals longer than 160 s.

Note: ***You must make sure that there is at least one source transition during the measured pulse in order for this application to work properly.***

GPCTR_Watch

Function

Monitors state of the general-purpose counter and its operation (E Series devices only).

Synopsis

C Syntax	`locus i32 GPCTR_Watch(u32 deviceNumber, u32 gpctrNum, u32 entityID, u32 *entityValue);`
Pascal Syntax	`function GPCTR_Watch(deviceNumber : i32; gpctrNum : i32; entityID : i32; var entityValue : i32) : i32;`
BASIC Syntax	`FN GPCTR_Watch(deviceNumber&, gpctrNum&, entityID&, entityValue&)`

Description

Legal ranges for the **gpctrNum**, **entityID**, and **entityValue** are in terms of constants defined in a header file. The header file you should use depends on the language you are using:

- C programmers—`nidaqcns.h`
- Pascal programmers—`nidaq.p`
- BASIC programmers—`nidaq.bas`

Use **gpctrNum** to indicate to NI-DAQ which counter you want to program. Legal values for this parameter are `ND_COUNTER_0` and `ND_COUNTER_1`.

Use **entityID** to indicate to NI-DAQ which feature you are interested in. Legal values are listed in the following paragraphs, along with the corresponding values you can expect for **entityValue**. **entityValue** will be given either in terms of constants from the header file, or as numbers, as appropriate.

entityID = `ND_COUNT`
This is the counter contents. **entityValue** can be between 0 and 2^{24}-1.

entityID = `ND_ARMED`
Indicates whether the counter is armed. **entityValue** can be `ND_YES` or `ND_NO`. You can use this in applications such as `ND_SINGLE_PULSE_WIDTH_MSR` for finding out when the pulse width measurement completes.

entityID = `ND_TC_REACHED`
Indicates whether the counter has reached terminal count **entityValue** can be `ND_YES` or `ND_NO`. You can use this in applications such as `ND_SINGLE_PULSE_WIDTH_MSR` for detecting overflow (pulse was too long to be measured using the selected timebase).

entityID = `ND_DONE`
When the application is `ND_SINGLE_TRIG_PULSE_GNR`, this indicates that the pulse has completed. When the application is `ND_RETRIG_PULSE_GNR`, this indicates that an individual pulse has completed. In this case, the indication that an individual pulse has completed will be returned only once per pulse by the `GPCTR_Watch` function.

entityID = `ND_OUTPUT_STATE`
You can use this to read the value of the counter output; the range is `ND_LOW` and `ND_HIGH`.

entityID = `ND_AVAILABLE_POINTS`
This is useful for buffered operations. You can use this to find the number of sample points available in your buffer.

entityID = ND_COUNT_AVAILABLE

If the application is ND_TRIG_PULSE_WIDTH_MSR, ND_SINGLE_PULSE_WIDTH_MSR, or ND_SINGLE_PERIOD_MSR, this indicates whether your measurement has completed. **entityValue** can be ND_YES (the measurement has completed) and ND_NO (the measure has not completed).

Note to C Programmers—entityValue ***is a pass-by-reference parameter.***

Chapter 9
RTSI Bus Trigger Functions

The RTSI Bus Trigger functions connect and disconnect signals over the RTSI bus trigger lines. See Appendix A to determine which functions are available for your board.

The RTSI Bus

The RTSI bus is implemented via a 50-pin ribbon cable connector on the top edge of the National Instruments boards on the Macintosh. Fourteen of the RTSI bus lines are dedicated to a 8-wire trigger bus. Each board that uses a RTSI bus interface contains a number of useful signals that can be driven onto, or received from, the trigger lines. Each board is equipped with a switch that can connect any onboard signal to any one of the RTSI bus trigger lines through software control. If you program one board to drive a given trigger line and another board to receive from the same trigger line, a hardware connection can be made between the two boards. The RTSI Bus Trigger functions explained in this chapter support this type of programmable signal interconnection between boards.

The RTSI bus trigger lines can be used to trigger one board from another board, to share clocks and signals between boards, and to synchronize boards to the same signals.

To specify the signals on each board that can be connected to the RTSI bus trigger lines, each board signal is assigned a signal code number. All references to that signal are made by using the signal code number in the RTSI Bus Trigger function calls. The signal codes for each board that uses the RTSI bus trigger lines are given later in this chapter.

Each signal listed in this chapter also has a signal direction. If a signal is listed with a source direction, then that signal is able to drive the trigger lines. If a signal is listed with a receiver direction, then that signal must be received from the trigger lines. A bidirectional signal direction means that the signal can act as either a source or receiver, depending on the application.

NB-MIO-16 RTSI Connections

The NB-MIO-16 contains nine signals that you can connect to the RTSI bus trigger. These signals are shown in Table 9-1.

Table 9-1. NB-MIO-16 RTSI Bus Signals

Signal Name	Signal Direction	Signal Code
GATE1	bidirectional	0
FOUT	source	1
EXTCONV*	receiver	2
OUT2	source	3
SOURCE4	bidirectional	4
EXTGATE	receiver	5
OUT5	source	6
OUT1	bidirectional	7
EXTTRIG*	bidirectional	8

The signals GATE1, SOURCE4, OUT1, OUT2, OUT5, and FOUT are input and output signals from the Am9513 Counter/Timer on the NB-MIO-16 board. OUT1, OUT2, and OUT5 are outputs of Counters 1, 2, and 5, respectively. FOUT is the Am9513 programmable frequency output. GATE1 is the gating signal for Counter 1, and SOURCE4 is a counter source input. The counters and frequency output are programmed via the Counter functions (see Chapter 8, *Counter/Timer Functions*). GATE1, OUT1, OUT2, OUT5, and FOUT are also available on the NB-MIO-16 I/O connector.

The signals EXTCONV*, EXTGATE, and EXTTRIG* are used for data acquisition timing. These signals are explained in Chapter 6, *Data Acquisition Functions*.

Some restrictions apply for connection of onboard signals to the RTSI bus. OUT2 and EXTCONV* should not be connected to the RTSI bus at the same time. Doing so results in the OUT2 signal driving the EXTCONV* line. Similarly, OUT5 and EXTGATE should not be connected to the RTSI bus at the same time. Doing so results in the OUT5 signal driving the EXTGATE line. For more information about the NB-MIO-16 signals, see the *NB-MIO-16 User Manual*.

NB-MIO-16X RTSI Connections

The NB-MIO-16X contains nine signals that you can connect to the RTSI bus trigger. These signals are shown in Table 9-2.

Table 9-2. NB-MIO-16X RTSI Bus Signals

Signal Name	Signal Direction	Signal Code
EXTCONV*	bidirectional	0
RTSIWG	receiver	1
GATE1	receiver	2
OUT2	source	3
SOURCE5	bidirectional	4
STOPTRIG	receiver	5
OUT5	source	6
OUT1	bidirectional	7
STRTTRIG*	bidirectional	8

The signals GATE1, SOURCE5, OUT1, OUT2, and OUT5 are input and output signals from the Am9513 Counter/Timer on the NB-MIO-16X board. OUT1, OUT2, and OUT5 are outputs of Counters 1, 2, and 5, respectively. GATE1 is the gating signal for Counter 1, and SOURCE5 is a counter source input. The counters and frequency output are programmed via the Counter functions (see Chapter 8, *Counter/Timer Functions*). GATE1, OUT1, OUT2, and OUT5 are also available on the NB-MIO-16X I/O connector.

The signals EXTCONV*, STOPTRIG, and STRTTRIG* are used for data acquisition timing. These signals are explained in Chapter 6, *Data Acquisition Functions*.

Some restrictions apply for connection of onboard signals to the RTSI bus. OUT2 and GATE1 should not be connected to the RTSI bus at the same time. Doing so results in the OUT2 signal driving the GATE1 line. Similarly, OUT5 and STOPTRIG should not be connected to the RTSI bus at the same time. Doing so results in the OUT5 signal driving the STOPTRIG line. For more information about the NB-MIO-16X signals, see the *NB-MIO-16X User Manual*.

E Series Boards RTSI Connections

For more information regarding signals on the E Series boards that you can connect to the RTSI bus, refer to the `Select_Signal` function description in Chapter 2, *Board-Specific Functions*.

NB-DMA-8-G and NB-DMA2800 RTSI Connections

The NB-DMA-8-G and NB-DMA2800 contain 14 signals that you can connect to the RTSI bus trigger. These signals are shown in Table 9-3.

Table 9-3. NB-DMA-8-G and NB-DMA2800 RTSI Bus Signals

Signal Name	Signal Direction	Signal Code
FOUT	source	0
SOURCE5	receiver	1
SOURCE4	receiver	2
SOURCE3	receiver	3
GATE5	receiver	4
GATE3	receiver	5
GATE1	receiver	6
TOUT3	source	7
TOUT2	source	8
OUT5	source	9
OUT4	source	10
OUT3	source	11
OUT2	source	12
OUT1	source	13

The signals FOUT, SOURCE5 through SOURCE3, GATE5, GATE3, GATE1, and OUT5 through OUT1 are input and output signals from the Am9513 Counter/Timer on the NB-DMA-8-G and NB-DMA2800 boards. OUT5 through OUT1 signals are outputs of Counters 5 through 1, respectively. FOUT is the Am9513 programmable frequency output. GATE5, GATE3, and GATE1 are the gating signals for Counters 5, 3, and 1, respectively. SOURCE5 through SOURCE3 are counter clock source inputs. The counters and frequency output are programmed via the Counter functions (see Chapter 8, *Counter/Timer Functions*).

The signal TOUT3 supplies a 5 MHz clock signal, and TOUT2 supplies a 1 MHz clock signal. For more information about the NB-DMA-8-G or NB-DMA2800 signals, see the *NB-DMA-8-G User Manual* or the *NB-DMA2800 User Manual*.

NB-DIO-32F RTSI Connections

The NB-DIO-32F contains six signals that you can connect to the RTSI bus trigger. These signals are shown in Table 9-4.

Table 9-4. NB-DIO-32F RTSI Bus Signals

Signal Name	Signal Direction	Signal Code
RRQ1	source	0
RRQ2	source	1
RQNI1	receiver	2
RQNI2	receiver	3
XAK1	source	4
XAK2	source	5

The signals RRQ1 and RRQ2 are request signals received from the I/O connector. These signals are driven by an external device during handshaking. RQNI1, RQNI2, XAK1, and XAK2 permit handshaking with the NB-DIO-32F over the RTSI bus. For more information about the NB-DIO-32F signals, see the *NB-DIO-32F User Manual*.

NB-AO-6 RTSI Connections

The NB-AO-6 contains two signals connected to the RTSI bus trigger. These signals are shown in Table 9-5.

Table 9-5. NB-AO-6 RTSI Bus Signals

Signal Name	Signal Direction	Signal Code
TRIGUP	receiver	0
UPDATE	source	1

The signal TRIGUP can be used to trigger an update on the NB-AO-6 for updating of the DACs. UPDATE is the update signal generated. For more information about the NB-AO-6 signals, see the *NB-AO-6 User Manual*.

NB-A2000 RTSI Connections

The NB-A2000 contains seven signals that can be connected to the RTSI bus trigger lines. These signals are shown in Table 9-6.

Table 9-6. NB-A2000 RTSI Bus Signals

Signal Name	Signal Direction	Signal Code
START*	bidirectional	0
TRIGGER*	bidirectional	1
CLOCKO	source	2
CLOCKI	receiver	3
GATE2	receiver	4
SOURCE2	receiver	5
OUT2	source	6

The signals GATE2, SOURCE2, and OUT2 are input and output signals from the Am9513A Counter/Timer on the NB-A2000 board. GATE2 is the gating signal for Counter 2, SOURCE2 is the source signal for Counter 2, and OUT2 is the output of Counter 2. Counter 2 is programmed via the Counter/Timer functions (see Chapter 8, *Counter/Timer Functions*).

The signals START*, TRIGGER*, CLOCKO, and CLOCKI are used for data acquisition timing. These signals may be generated locally on the NB-A2000 or may be controlled from the RTSI bus. These signals are explained as follows.

Signal Name	Description
START*	START* is an active-low signal that initiates a data acquisition sequence. If data acquisition is locally controlled, the START* signal pulses low when a trigger is generated from software (pretrigger mode) or from the NB-A2000 analog trigger or digital trigger circuitry (posttrigger or pretrigger mode). When locally generated, START* is a signal source. Alternatively, if the NB-A2000 is configured to receive START* from the RTSI bus, it initiates a data acquisition sequence when a low pulse is received.
TRIGGER*	TRIGGER* is an active-low signal that activates the sample counter. If data acquisition is locally controlled, TRIGGER* pulses low when a trigger is received either through software (posttrigger mode only) or from the NB-A2000 analog trigger or digital trigger circuitry (posttrigger or pretrigger mode). In pretrigger mode, TRIGGER* pulses sometime after START*. In posttrigger mode, START* drives the TRIGGER* signal directly. The TRIGGER* signal may be driven from the RTSI switch when the NB-A2000 is configured for pretrigger mode only. In this case, the NB-A2000 begins to acquire posttrigger samples after a low pulse is received on the TRIGGER* input rather than from the local analog or digital circuitry.
CLOCKO	CLOCKO is the active-high, sample-clock output signal. The rising edge of this signal initiates a scanning sequence in which all active channels are sampled simultaneously. Any locally generated sample clock or any clock received from the I/O connector SAMPCLK* input will appear on the CLOCKO signal.
CLOCKI	CLOCKI is the active-high, sample-clock input signal. If CLOCKI is driven from the RTSI Switch, the rising edge of this signal initiates a scanning sequence in which all active channels are sampled simultaneously. The locally generated sample clock and I/O connector SAMPCLK* signal are ignored when CLOCKI is driven by the RTSI switch. With the exception of master/slave clock operation, the NB-A2000 must be configured to use external sample clock if CLOCKI is to be driven from the RTSI switch (see `A2000_Config`).

Note: ***If the RTSI switch drives any of the START*, TRIGGER*, or CLOCKI signals, then locally generated signals are overwritten. TRIGGER* should be driven from the RTSI switch only if the NB-A2000 is configured for pretrigger mode.***

NB-A2100 RTSI Connections

The NB-A2100 has three signals that you can connect to the RTSI bus trigger lines. These signals are shown in Table 9-7.

Table 9-7. NB-A2100 RTSI Bus Signals

Signal Name	Signal Direction	Signal Code
EXTTRIG*	bidirectional	0
WCAD	source	1
WCDA	source	2

The RTSI bus trigger lines share the signal EXTTRIG* with the RCA jack on the I/O connector panel. So, if you configure EXTTRIG* as a source, the signal applied at the I/O connector appears at the RTSI trigger lines. Similarly, if you configure EXTTRIG* as a receiver, the signal applied at EXTTRIG* through a RTSI trigger line also appears at the I/O connector.

The signals labeled WCAD and WCDA are the A/D and D/A sampling clock signals.

NB-A2150 RTSI Connections

The NB-A2150 has five signals that you can connect to the RTSI bus trigger lines. These signals are shown in Table 9-8.

Table 9-8. NB-A2150 RTSI Bus Signals

Signal Name	Signal Direction	Signal Code
RTSITRIG*	bidirectional	0
WCAD	source	1
RTSI_A2	source	2
SWSTART*	source	3
RTSISTART*	receiver	4

The signal WCAD is the A/D word clock signal. The signal RTSI_A2 is controlled through one of the registers on the board and is always low.

You can use the signals RTSITRIG*, SWSTART*, and RTSISTART* for data acquisition triggering. These signals are explained as follows:

Signal Name	Description
RTSITRIG*	RTSITRIG* is the signal that activates the sample counter when a high-to-low edge is received at the signal. In posttrigger mode, a high-to-low edge starts the data acquisition on the NB-A2150. In pretrigger mode, the NB-A2150 begins to acquire posttrigger samples after a high-to-low pulse is received. Alternatively, if the RTSITRIG* signal is configured to be the source, it is connected to the local trigger signal of the board. The local trigger signal is connected to the EXTTRIG* pin on the I/O connector if the external digital trigger is enabled or to the analog trigger circuitry if analog level triggering is enabled. The analog triggering circuitry generates an active low pulse when the analog trigger conditions are met.
SWSTART*	SWSTART* is the signal that generates an active low pulse when a trigger is generated from software (pretrigger or posttrigger mode). This signal can be driven to the RTSISTART* signal on other NB-A2150 boards to simultaneously start data acquisition on all boards when a software trigger is generated at the board that is driving this signal.
RTSISTART*	RTSISTART* is the signal that starts the data acquisition when a high-to-low edge is received at the signal. In posttrigger mode, this signal has the same effect as the RTSITRIG* signal. In pretrigger mode, a high-to-low edge at this signal starts continuous data acquisition. However, the sample counter is not started until a trigger is received from the RTSITRIG* signal, the EXTTRIG* pin on the I/O connector, or the analog triggering circuitry.

NB-TIO-10 RTSI Connections

The NB-TIO-10 has 14 signals that you can connect to the RTSI bus trigger lines. These signals are shown in Table 9-9.

Table 9-9. NB-TIO-10 RTSI Bus Signals

Signal Name	Signal Direction	Signal Code
SOURCE1	bidirectional	0
GATE1	bidirectional	1
OUT1	bidirectional	2
SOURCE2	bidirectional	3
OUT2	bidirectional	4
GATE5	bidirectional	5
OUT5	bidirectional	6
SOURCE6	bidirectional	7
GATE6	bidirectional	8
OUT6	bidirectional	9
SOURCE7	bidirectional	10
GATE10	bidirectional	11
OUT10	bidirectional	12
FOUT1	source	13

All 14 signals are input and output signals from the Am9513 on the NB-TIO-10 board. OUT1, OUT2, OUT5, OUT6, and OUT10 are outputs of Counters 1, 2, 5, 6, and 10, respectively. FOUT1 is the Am9513 programmable frequency output. GATE1, GATE5, GATE6, and GATE10 are the gating signals for Counters 1, 5, 6, and 10, respectively. SOURCE1, SOURCE2, SOURCE6, and SOURCE7 are counter clock source inputs. The counters and frequency output are programmed via the Counter/Timer Functions (see Chapter 8, *Counter/Timer Functions*).

Warning: ***If you configure the output of a counter as an input over the RTSI bus, you must call*** `CTR_Reset` ***and put the counter back in high impedance mode to avoid multiple sources driving the signal and possibly causing damage to the board.***

For more information about these signals, refer to the *NB-TIO-10 User Manual*.

RTSI Bus Trigger Function Summary

Use the following RTSI Bus Trigger functions to make signal connections from NB Series boards to the RTSI bus trigger lines:

`RTSI_Clear`	Disconnects all RTSI trigger connections from the specified board.
`RTSI_Conn`	Connects a signal from the specified board to the specified trigger line.
`RTSI_DisConn`	Disconnects a signal on the specified board from a specified trigger line.

RTSI Bus Trigger Function Application Hints

Note: ***For information regarding signals on the E Series boards that you can connect to the RTSI bus, refer to the*** `Select_Signal` ***function in Chapter 2,*** *Board-Specific Functions.*

To connect signals across the RTSI bus trigger lines, you must execute `RTSI_Conn` at least twice: once for the signal transmitter and once for the signal receiver. Additional executions of `RTSI_Conn` can add more signal receivers to the connection.

Signals can be disconnected in one of two ways: either execute `RTSI_DisConn` for each connection made via `RTSI_Conn`, or use `RTSI_Clear` to disconnect an entire board from the RTSI bus trigger lines.

RTSI_Clear

Function

Disconnects all RTSI bus trigger line connections from the specified board.

Synopsis

C Syntax	`locus i32 RTSI_Clear(u32 deviceNumber);`
Pascal Syntax	`function RTSI_Clear(deviceNumber : i32) : i32;`
BASIC Syntax	`FN RTSI_Clear(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`RTSI_Clear` clears all RTSI bus trigger line connections from the specified board. After you execute `RTSI_Clear`, the board does not drive signals onto any trigger line nor does it receive signals from any trigger line. This function can be used to reset the RTSI interface of the board. After system startup, the RTSI bus interface for each board is reset in this manner.

RTSI_Conn

Function

Connects a signal from the specified board to the specified trigger line.

Synopsis

C Syntax	`locus i32 RTSI_Conn(u32 deviceNumber, u32 signalCode, u32 triggerLine, u32 direction);`
Pascal Syntax	`function RTSI_Conn(deviceNumber : i32; signalCode : i32; triggerLine : i32; direction : i32) : i32;`
BASIC Syntax	`FN RTSI_Conn(deviceNumber&, signalCode&, triggerLine&, direction&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

signalCode is the signal code number of the signal to be connected to the trigger line. Signal code numbers for each board type are given at the beginning of this chapter.

triggerLine is the RTSI bus trigger line that is to be connected to the signal.
Range: 0 through 6.

direction is the direction of the connection. The value of **direction** corresponds to the direction of the signal.
0: receive signal (input, receiver) from the RTSI bus trigger line.
1: transmit signal (output, source) to the RTSI bus trigger line.

`RTSI_Conn` programs the RTSI interface on the specified **deviceNumber** such that the signal identified by **signalCode** is connected to the trigger line specified by **triggerLine**. For example, if the specified board is an NB-DMA-8-G or NB-DMA2800 board, **signalCode** is 13, **triggerLine** is 3, and **direction** is 1, then the NB-DMA-8-G or NB-DMA2800 RTSI interface is configured to drive the signal for OUT1 of the onboard Am9513 onto trigger line 3.

You need two `RTSI_Conn` calls to make a connection between two boards. For example, a second call can access an NB-MIO-16 board with **signalCode** set to 2, **triggerLine** set to 3, and **direction** set to 0. This call configures the NB-MIO-16 board RTSI interface to receive a signal from trigger line 3 and drive it onto the NB-MIO-16 EXTCONV* signal. The total effect of these two calls is that the NB-MIO-16 EXTCONV* signal is controlled by the OUT1 signal on the NB-DMA-8-G and NB-DMA2800 boards, thus controlling A/D conversions on the NB-MIO-16 by timers on the NB-DMA-8-G and NB-DMA2800.

Rules for RTSI Bus Connections

Observe the following rules when routing signals over the RTSI bus trigger lines:

- Any signal can be connected to any trigger line.

- RTSI connections should have only one source signal but can have multiple receiver signals. Connection of two or more source signals causes bus contention over the trigger line.
- You can connect two or more signals on the same board together via a RTSI bus trigger line as long as the above rules are followed.
- RTSI connections can be disconnected by using either `RTSI_DisConn` or `RTSI_Clear`.

RTSI_DisConn

Function

Disconnects a signal on the specified board from the specified trigger line.

Synopsis

C Syntax	`locus i32 RTSI_DisConn(u32 deviceNumber, u32 signalCode, u32 triggerLine);`
Pascal Syntax	`function RTSI_DisConn(deviceNumber : i32; signalCode : i32; triggerLine : i32) : i32;`
BASIC Syntax	`FN RTSI_DisConn(deviceNumber&, signalCode&, triggerLine&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

signalCode is the signal code number of the signal to be disconnected from the trigger line. Signal code numbers for each board type are given at the beginning of this chapter.

triggerLine is the RTSI bus trigger line that is to be disconnected from the signal.
Range: 0 through 6.

`RTSI_DisConn` programs the RTSI interface on the specified **deviceNumber** such that the signal identified by **signalCode** and the trigger line specified by **triggerLine** are disconnected.

Note: ***The same number of*** `RTSI_DisConn` ***calls are needed to disconnect a connection as were needed to make the connection in the first place. (See the*** `RTSI_Conn` ***description for further explanation.)***

Chapter 10
Waveform Generation Functions

This chapter describes the functions for generating analog waveform signals at the analog output channels. The three sets of functions described are as follows:

Synchronous Waveform Generation (`WF_Grp`) **Functions–**Control analog output boards for single-buffered or, if the board can use DMA, double-buffered waveform generation on a group of analog output channels sharing a common update clock.

Asynchronous Waveform Generation (`WF`) **Functions**–Control the following analog output boards for single-buffered or double-buffered waveform generation on a single individually updated channel.

Buffered Waveform Generation (`BWF`) **Functions**–Control the following analog output board for buffered waveform generation in which output data can be updated while the waveform generation is in progress.

See Appendix A to determine which function set works with your board.

Waveform Generation Hardware

System Timing for Waveform Generation

Some analog output boards require an NB-DMA-8-G or the NB-DMA2800 in the system for waveform generation timing. Table 10-1 lists the waveform generation boards that require DMA and the boards that will automatically use DMA to service data requests, if available. When DMA is not used, an interrupt routine services the requests.

Table 10-1. Waveform Generation DMA Requirements

Waveform Generation Board	DMA Required	DMA Used If Available
NB-AO-6	Yes	Yes
NB-MIO-16	Yes	Yes
NB-MIO-16X	No	Yes
Lab and 1200 series	No	No
NB-A2100	No	Yes
MIO E Series	No	No

Waveform Generation Using DMA

The NB-DMA-8-G and NB-DMA2800 connect to the other NB Series boards via the RTSI connector. Each DMA board has three independent update clocks that can be used for controlling waveform generation. These clock signals are generated by programmable counter/timers. These clocks can generate DMA requests that cause new values to be loaded into the DACs used for waveform generation during each update interval. In addition, the update clock is sent over the RTSI bus to clock the DACs on the waveform generation board.

One of the waveform generation update clocks on each DMA board can handle multiple-channel waveform generation. You can generate up to six different waveforms on six different analog output channels synchronized to

this update clock. The other two waveform clocks can handle one channel of waveform generation each. Each DMA board can generate up to seven waveforms in parallel.

Waveform generation on the NB-A2100 uses an internal clock to generate DMA requests that are serviced by DMA, if a DMA channel is available.

Waveform Generation Without DMA

Some waveform generation boards use their own internal update clock for controlling waveform generation if DMA is not available. The update clocks generate interrupts that cause new values to be loaded into the DACs during each update interval. In addition, the update clock clocks the DACs on the analog output board. Both DACs on each analog output board are updated simultaneously with this clock to perform multiple-channel waveform generation. The NB-MIO-16X uses counter 2; the Lab and 1200 series use counters A2 and B0; and the NB-A2100 and MIO E Series use internal clocks to control waveform generation.

Synchronous Waveform Generation

Synchronous waveform generation allows several analog output channels to be loaded one at a time, and then all the analog output voltages can be updated simultaneously with the update clock. Synchronous waveform generation provides two benefits–the analog output channel voltage changes precisely with the update clock edge; and for multiple-channel waveform generation, the multiple-channel clock ensures that the analog output channel voltages change at the same time. This type of waveform generation requires double-buffered DACs. Double-buffered DACs are capable of storing a new output value when written to, and they wait to change the output voltage value when a clock edge is received.

Asynchronous Waveform Generation

Asynchronous waveform generation updates only one analog output channel per update clock. The analog output voltage still changes once every update clock pulse; however, the delay between the analog output changing state and the update clock pulse is variable. This variable delay contributes some delay and phase jitter with respect to the update clock. The delay is due to the DAC changing its output voltage value whenever it is written to and not waiting for a clock edge to change state. The phase jitter results from variations in DMA servicing times for the DAC. This phase jitter is on the order of a few hundred nanoseconds and therefore may not be noticeable for lower frequency update rates. This type of waveform generation can be used with any type of DAC–double-buffered, or single-buffered. When a board with double-buffered DACs is used in the asynchronous waveform generation mode, the DACs are simply configured to operate as if they were single-buffered DACs.

Synchronous Versus Asynchronous Waveform Generation

Obviously, synchronous waveform generation using double-buffered DACs is the preferred way to generate waveforms. It results in a high precision update rate and allows synchronization between waveforms. However, the NB-MIO-16 does not have double-buffered DACs. In addition, a board with double-buffered DACs has one update clock shared by all channels. Therefore, either all channels must operate in synchronous mode with a single update clock, or each channel must operate asynchronously, each with its own update clock.

Finally, the following combinations of asynchronous and synchronous waveform generation are possible when using the NB-DMA-8-G or NB-DMA2800 to control waveform generation:

Asynchronous	Synchronous
Three channels (any board)	No channels
Two channels (any board other than the board being used for synchronous waveform generation)	Up to four channels on a single NB-AO-6 board *or* Up to two channels on a single NB-MIO-16X board
One channel (any board other than the board being used for synchronous waveform generation)	Up to six channels on one NB-AO-6 *or* Up to two channels on one NB-MIO-16X *or* Up to four channels on one NB-AO-6 or up to two channels on an NB-MIO-16X board and one channel on a separate NB-AO-6 or NB-MIO-16X board
No channel boards	Up to six channels on one NB-AO-6 or two channels on one NB-MIO-16X board and one channel each on a separate NB-AO-6 or NB-MIO-16X board *or* Up to four channels on one NB-AO-6 or two channels on one NB-MIO-16X board and up to one channel on each of two separate NB-AO-6 or NB-MIO-16X

Notice that for asynchronous waveform generation, each channel has its own update clock, but for synchronous waveform generation, all channels on the same board share the update clock. In addition, only one clock is available for multiple-channel waveform generation.

NB-MIO-16 Waveform Generation

The NB-MIO-16 contains two analog voltage output channels numbered 0 and 1. Each analog output channel contains a single-buffered DAC. The NB-MIO-16 can use asynchronous waveform generation only. Up to two waveforms can be generated simultaneously on one NB-MIO-16 board, with each channel using a separate update clock. See *Asynchronous Waveform Generation Call Sequences* under *Waveform Generation Application Hints* later in this chapter.

Analog output on the NB-MIO-16 can be hardware jumper-configured for the following output ranges:

- 0 to +10 V (unipolar, internal reference)
- -10 to +10 V (bipolar, internal reference)
- 0 to Vref (unipolar, external reference)
- -Vref to Vref (bipolar, external reference)

NB-MIO-16X Waveform Generation

The NB-MIO-16X contains two analog voltage output channels numbered 0 and 1. Each analog output channel contains a double-buffered DAC. The NB-MIO-16X can use synchronous waveform generation only. Up to two waveforms can be generated simultaneously on one NB-MIO-16X board.

Both channels on the NB-MIO-16X share the same update clock. A single NB-MIO-16X can generate up to two different waveforms with synchronous waveform generation. If the waveform generation operation is configured for external timing, an external clock signal can be applied to OUT2 on the I/O connector to clock the DACs and control

the update interval. See *Synchronous Waveform Generation* under *Waveform Generation Application Hints* later in this chapter.

The NB-MIO-16X uses the NB-DMA-8-G board or the NB-DMA2800 board for waveform generation if one of the DMA boards is detected in the system. If a DMA board is not present, then the NB-MIO-16X internal Counter 2 is used to clock the DACs on the board.

Analog output on the NB-MIO-16X is configured for the following output ranges:

- 0 to +10 V (unipolar, internal reference)
- -10 to +10 V (bipolar, internal reference)
- 0 to Vref (unipolar, external reference)
- -Vref to Vref (bipolar, external reference)

PCI-MIO-16XE-50 Waveform Generation

Because of large interrupt latencies on PCI Macintosh computers, waveform generation will halt and an underflow error will be returned at rates above a few hundred hertz.

The PCI-MIO-16XE-50 contains two analog voltage output channels numbered 0 and 1. Each analog output channel contains a double-buffered DAC. The PCI-MIO-16XE-50 can use synchronous waveform generation only. Up to two waveforms can be generated simultaneously on one PCI-MIO-16XE-50 board.

Both channels on the PCI-MIO-16XE-50 share the same update clock. A single PCI-MIO-16XE-50 can generate up to two different waveforms with synchronous waveform generation. If the waveform generation operation is configured for external timing, an external clock signal can be applied to a pin on the I/O connector to clock the DACs and control the update interval. See *Synchronous Waveform Generation* under *Waveform Generation Application Hints* later in this chapter.

Analog output on the PCI-MIO-16XE-50 is configured for the following output range:

- -10 to +10 V (bipolar, internal reference)

NB-AO-6 Waveform Generation

The NB-AO-6 contains six analog output channels numbered 0 through 5. Each channel provides both a voltage and current output. Each analog output channel contains a double-buffered DAC. All channels on the NB-AO-6 share the same update clock. The NB-AO-6 can be operated such that either all channel outputs change synchronously with respect to this update clock, or each output changes individually when written to. The NB-AO-6 supports both synchronous and asynchronous waveform generation. Synchronous waveform generation allows up to six different waveforms to be generated by a single NB-AO-6. If asynchronous waveform generation is used, up to three waveforms can be generated simultaneously on one NB-AO-6 board, with each channel using a separate update clock. See the discussions *Synchronous Waveform Generation Call Sequences* and *Asynchronous Waveform Generation Call Sequences* under *Waveform Generation Application Hints* later in this chapter.

Analog voltage outputs on the NB-AO-6 can be hardware jumper-configured for the following output ranges:

- 0 to +10 V (unipolar, internal reference)
- -10 to +10 V (bipolar, internal reference)
- 0 to Vref (unipolar, external reference)
- -Vref to Vref (bipolar, external reference)

Lab and 1200 Series Waveform Generation

Because of large interrupt latencies on PCI Macintosh computers, waveform generation timing is unreliable on PCI-1200s at rates above a few hundred hertz.

These boards contain two analog output channels numbered 0 and 1. Each analog output channel contains a 12-bit double-buffered DAC. Each analog output channel can be hardware jumper configured for unipolar or bipolar voltage with the following output ranges:

- 0 to +10 V (unipolar)
- -5 to +5 V (bipolar)

The DACs can be updated immediately upon writing to the DAC (asynchronous update mode) or updated at a later time by applying an update pulse (synchronous update mode). Waveform generation on the Lab-NB or Lab-LC uses the synchronous update mode for generating waveforms. Up to two waveforms can be generated simultaneously on one Lab-NB or Lab-LC board, with each channel using the same update clock so that the outputs of both channels change synchronously with respect to the update clock. Synchronous waveform generation has two benefits:

- The analog output channel voltage changes precisely with the update clock edge.
- For two-channel waveform generation, the analog output voltages change at the same time.

Lab and 1200 Series Counter/Timer Signals

Waveform generation uses the onboard Counter A2 to produce the total update interval for waveform generation. However, if the total update interval is greater than 65,535 µs, Counter B0 is used to generate the clock for a slower timebase, which is used by Counter A2 to provide the total update interval. Counter B0 then cannot be used by the `ICTR_Setup` and `ICTR_Reset` functions for the duration of the waveform generation operation. Counter B0 also cannot be used by the Data Acquisition Functions `DAQ_Start` and `Lab_ISCAN_Start` if the total sample interval for data acquisition is also greater than 65,535 µs and Counter B0 is required to produce a timebase for data acquisition different from the timebase being produced by Counter B0 for waveform generation. Counter B0 is considered available for waveform generation if any of the following is true:

- If data acquisition is not in progress and no `ICTR_Setup` call has been made on Counter B0 since startup.
- If data acquisition is not in progress and an `ICTR_Reset` call has been made on Counter B0.
- If data acquisition is in progress and is using Counter B0 to obtain the timebase required to produce the total sample interval, this timebase is the same as required by Waveform Generation Functions to produce the total update interval. In this case, Counter B0 is used to provide the same timebase for both data acquisition and waveform generation.

NB-A2100 Waveform Generation

The NB-A2100 contains two simultaneously-updated analog output channels numbered 0 and 1. These 16-bit resolution D/A channels use 8-times oversampling digital anti-imaging filters for extremely high fidelity data output. Each channel also has a jumper to select AC or DC coupling.

The output range for each channel is ±3 V (or about 2.12 Vrms).

The DACs can be run at 16, 22.05, 24, 32, 44.1, or 48 kHz conversion rates. A 32-bits wide, 16 words deep FIFO memory on the board serves as a buffer to the DAC and can store 32 conversion values if one channel is being output or 16 conversion values for each channel if both channels are being output.

The D/A conversion data can be received serially over the RTSI bus from other National Instruments boards such as the NB-DSP2300 digital signal processing board.

Waveform generation can be started by applying a software trigger or by applying a high-to-low edge on the external digital trigger. The digital trigger may be received through the EXTTRIG* pin on the I/O connector or over the RTSI bus.

Synchronous and Asynchronous Waveform Generation Function Summary

The Waveform Generation Functions, listed as follows, are divided into two groups–those for asynchronous waveform generation and those for synchronous waveform generation. The asynchronous Waveform Generation Functions control individual channels, whereas the synchronous Waveform Generation Functions control a group of channels. For each set of Waveform Generation Functions, waveform loading and status checking are performed on an individual channel basis; therefore, `WF_Load` and `WF_Check` are included in both groups of functions.

Asynchronous Waveform Generation Functions

These functions can be used for single-channel waveform generation:

`WF_Check`	Checks to see if at least one cycle of the waveform has been generated.
`WF_Load`	Loads the waveform buffer to be used for waveform generation.
`WF_Offset`	Offsets the values in a waveform buffer for bipolar waveform generation (NB-MIO-16 only).
`WF_Reset`	Resets the synchronous waveform generation on the specified board by stopping waveform generation, and releases the DMA channels, analog output channels, and trigger line being used for synchronous waveform generation.
`WF_Setup`	Selects the update rate used for asynchronous waveform generation.
`WF_Start`	Initiates asynchronous waveform generation.
`WF_Stop`	Stops asynchronous waveform generation.

Each of the functions listed here operates on a single-channel basis. Waveform generation can be configured, started, and stopped on an individual channel basis. Each channel receives its own update clock. This type of waveform generation can be performed by both the NB-MIO-16 and NB-AO-6.

Synchronous Waveform Generation Functions

These functions can be used for single-channel or multiple-channel waveform generation synchronized to a single update clock:

`WF_Check`	Checks to see if at least one cycle of the waveform has been generated.
`WF_Grp_Reset`	Resets the synchronous waveform generation on the specified board by stopping waveform generation and releases any DMA channels, analog output channels, and trigger line being used for synchronous waveform generation.
`WF_Grp_Setup`	Configures the board for subsequent synchronous waveform generation.
`WF_Grp_Start`	Initiates synchronous waveform generation.

`WF_Grp_Stop` Stops synchronous waveform generation.

`WF_Load` Loads the waveform buffer to be used for waveform generation.

`WF_Offset` Offsets the values in a waveform buffer for bipolar waveform generation.

Several of the functions listed here operate on a group of analog output channels. Channels can be configured, started, and stopped on a group basis. This type of waveform generation can be performed by the NB-MIO-16X, NB-AO-6, Lab and 1200 series boards, and MIO E series boards.

Waveform Generation Application Hints

Waveform generation is performed by writing a buffer of values to an analog output channel controlled by an update clock. One value is written to the analog output channel every update interval, resulting in an output voltage change every update interval. The amplitude of the waveform is determined by the range of values in the buffer. The frequency of the waveform depends on the length of the buffer and the update interval.

The configuration information for the analog output circuitry is controlled by the `AO_Setup` function. After system startup, the analog output configuration defaults to bipolar output polarity. If you have changed the jumper configuration on your board, you must execute `AO_Setup` so that waveform generation functions work properly.

Fundamental Frequency

If you select continuous waveform generation, waveform generation repeatedly outputs the contents of the buffer until stopped. The fundamental frequency of the output waveform depends on three factors:

- update interval
- number of points in the buffer
- number of cycles in the buffer

The following formula determines the fundamental frequency of the generated waveform:

$$F = \frac{\text{number of cycles per buffer}}{\text{update interval} * \text{timebase} * \text{number of points in buffer}}$$

Timebase is the timebase selected for the update interval counter.

For example, if the update interval is 10, timebase is 1 µs, the number of points per cycle of the waveform is 500, and the buffer contains one cycle of the waveform, then the frequency of the output waveform is calculated as follows:

$$F = \frac{1}{(10 * 1\text{ E-}6 * 500)} = 200\text{ Hz}$$

Minimum Update Interval

A minimum update interval of 4 µs per channel is recommended for both synchronous and asynchronous waveform generation with DMA (an NB-DMA-8-G or NB-DMA2800 is present in the system). For example, if one channel is used, the update interval can be 4 µs. If two channels are used simultaneously, then the update interval for each channel should be no smaller than 8 µs, and so on. The update interval is limited by the bus bandwidth of the Macintosh computer when using DMA. If the update interval is made smaller than recommended, waveform points may be lost due to bus conflict.

A minimum update interval of 100 µs/channel is recommended for waveform generation without DMA (an NB-DMA-8-G or NB-DMA2800 is not present in the system). For waveform generation on the Lab-LC a minimum update interval of 125 µs/channel is recommended. When you use waveform generation on the NB-MIO-16X without DMA, interrupts are generated for *every* point in the waveform cycle. Since these interrupts are serviced by the Macintosh CPU, this large number of interrupts can result in sluggish response of the Macintosh user interface (mouse moving slowly, and so on). For generation of very high frequency waveforms, you may choose to use DMA. If an NB-DMA-8-G or NB-DMA2800 is present in the system, DMA is used to update the DACs on the NB-MIO-16X. The NB-MIO-16X waveform generation response is greatly improved with the use of DMA.

Minimum Buffer Size

There is no minimum buffer size restriction; however, an interrupt is generated at the end of each waveform cycle. In the case where one or more waveforms are generated at high update rates, a large number of interrupts are generated. Since these interrupts are serviced by the Macintosh CPU, this large number of interrupts may result in sluggish response of the Macintosh user interface (mouse moving slowly, and so on). For generation of very high frequency waveforms, you may choose to use a small update interval and very few points per cycle. In this case, it is recommended that you store several cycles of a waveform in a buffer to reduce the frequency of system interrupts and improve the overall responsiveness of the computer.

Asynchronous Waveform Generation Call Sequences

The NB-AO-6 and NB-MIO-16 boards perform asynchronous waveform generation using the functions `WF_Setup`, `WF_Load`, `WF_Start`, `WF_Stop`, `WF_Reset`, and `WF_Check`. These functions control waveform generation on an individual analog output channel basis.

To generate a waveform, you must create an array containing the waveform points. First, call `WF_Setup` to indicate the update interval for the analog output channel and to indicate single-cycle or continuous mode. Calling `WF_Load` passes the waveform to the channel. `WF_Start` can then generate a waveform.

Once waveform generation on a given channel is started, you can stop it by calling `WF_Stop`. You can then restart it at any time by calling `WF_Start`, that is, until you call `WF_Reset` for that channel. For continuous waveform generation, you must call `WF_Reset` before you exit the application. You can call `WF_Check` to determine whether at least one cycle (an entire buffer) of the waveform has been generated.

`WF_Reset` clears waveform generation on a given channel at any time. `WF_Reset` frees update clocks and DMA channels for other NI-DAQ for Macintosh activities.

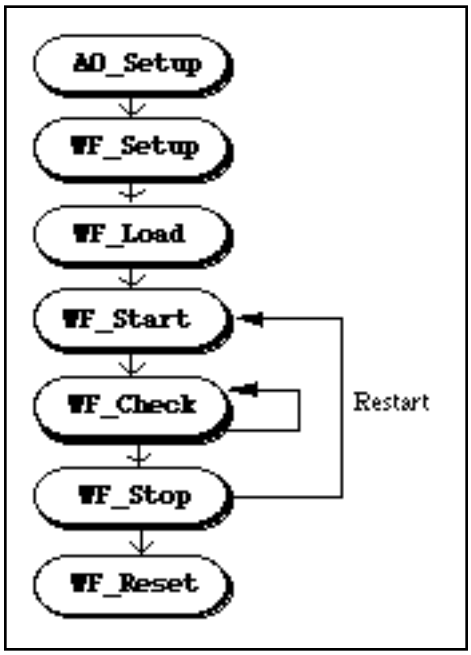

Figure 10-1. Asynchronous Waveform Generation Flowchart

Synchronous Waveform Generation Call Sequences

The NB-AO-6, NB-MIO-16X, MIO E Series, and Lab and 1200 series boards perform synchronous waveform generation using the functions `WF_Grp_Setup`, `WF_Load`, `WF_Grp_Start`, `WF_Grp_Stop`, `WF_Grp_Reset`, and `WF_Check`. These functions control waveform generation on a group of channels sharing a single update clock. These channels are updated simultaneously. The NB-MIO-16X, MIO E Series, and Lab and 1200 series boards perform only synchronous waveform generation. The NB-AO-6 performs both synchronous and asynchronous waveform generation. Synchronous waveform generation has more precise waveform generation timing than asynchronous waveform generation; therefore, even if only one waveform is generated from the NB-AO-6, you should use synchronous functions.

In order to generate a waveform, you must create one array containing waveform points for each channel. First, call `WF_Grp_Setup` to specify the group of channels and the update interval. You must then pass a waveform to each channel by calling `WF_Load` for *each* channel. You can then initiate waveform generation on all channels by calling `WF_Grp_Start`.

Once synchronous waveform generation starts, you can stop it by calling `WF_Grp_Stop`. You can then restart it at any time by calling `WF_Grp_Start`, that is, until you call `WF_Grp_Reset`. The MIO E Series devices do not support the `WF_Grp_Stop` function. If continuous waveform generation was specified on any channel, you must call `WF_Grp_Reset` before you exit the application. You can call `WF_Check` for each channel to determine whether at least one cycle (an entire buffer) of the waveform has been generated.

`WF_Grp_Reset` clears waveform generation on the channel group at any time. `WF_Grp_Reset` frees update clocks and DMA channels for other NI-DAQ for Macintosh activities.

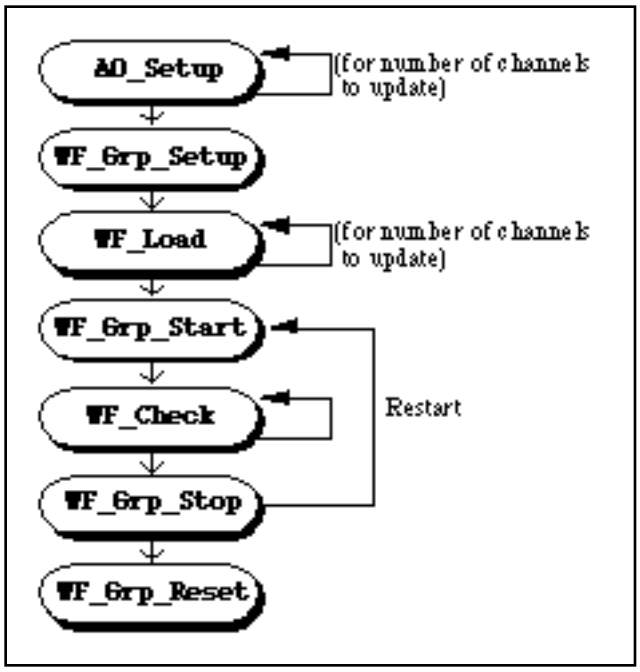

Figure 10-2. Synchronous Waveform Generation Flowchart

Double-Buffered Waveform Generation Using WF_DBLoad

Table 10-2 shows the conditions under which you can use double-buffered waveform generation.

Table 10-2. Conditions When You Can Use Double-Buffered Waveform Generation

Device	Conditions
NB-AO-6	• Can be used if a DMA board is installed in the system • Can be applied to asychronous or synchronous waveform generation
NB-MIO-16	• Can be used if a DMA board is installed in the system • Can be applied to asychronous or synchronous waveform generation
NB-MIO-16X	• Can be used if a DMA board is installed in the system and the external update clock is *not* used • Can be applied to asychronous or synchronous waveform generation
PCI-MIO-16XE-50	• No restrictions

Because the synchronous waveform generation case is more involved, it will be used in the example that follows; however, all the steps that are outlined apply to the asynchronous case as well.

In order to generate waveforms using the double-buffered mode, you must create two different buffers for every channel included in the group. One buffer, the buffer loaded during the initial call to `WF_DBLoad`, is called the master buffer, and the handler divides it into two equal halves called buffer *A* and buffer *B*, as shown in Figure 10-4 under the description of `WF_DBLoad`. The other buffer will be used as an intermediate buffer and will hold new data until that data has been copied into the next available half of the master buffer.

To configure the software for double-buffered waveform generation, you must run `WF_Grp_Setup` to configure the channels in the group and the update interval for the waveform. Then, for each channel in the group, you must run `WF_DBLoad` in the *load initial* mode (see the description of the `WF_DBLoad` function for more information). After all the channels have been loaded, run `WF_Grp_Start` to start the waveform operation. Then, for each channel in the group, you must run `WF_DBLoad` in the *load next* or *load last* mode; this step should be repeated until all data has been written to the master buffer. After the waveform generation ends, you should run `WF_Grp_Reset`.

The `WF_Grp_Stop` and `WF_Grp_Start` functions can be used to stop and start the waveform at any point during its generation. However, once the last buffer is loaded or the `WF_Grp_Reset` function is executed, the waveform operation must be reconfigured in order to generate more waveforms. The MIO E Series devices do not support the `WF_Grp_Stop` function.

A flowchart of the operations required for using double-buffered waveform generation is outlined in Figure 10-3.

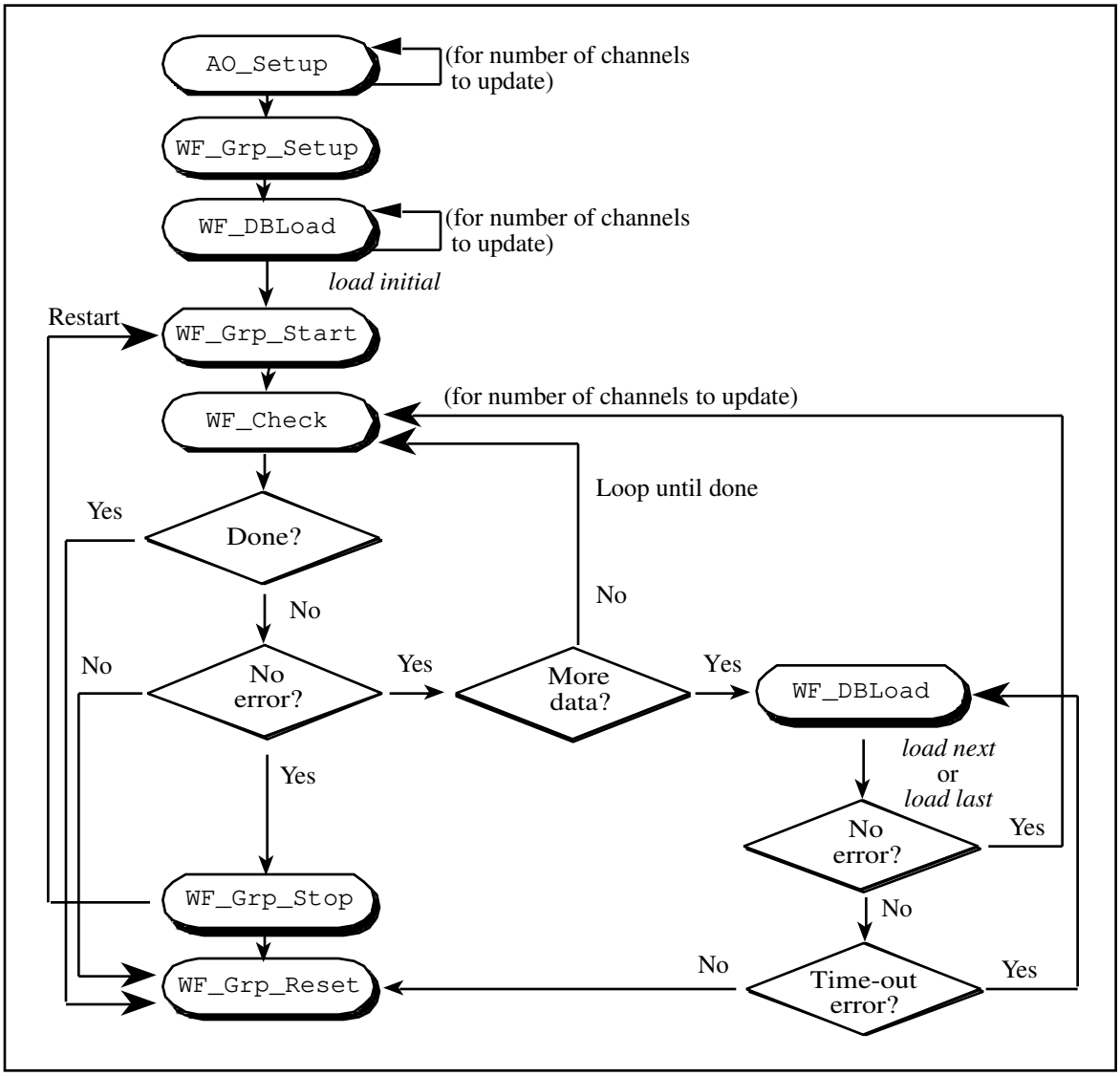

Figure 10-3. Double-Buffered Waveform Generation Flowchart

Externally Timed Waveform Generation

The NB-AO-6, NB-MIO-16X, MIO E Series and Lab and 1200 series can have an external clock signal to control waveform generation. You can configure a waveform generation operation for external timing by calling `WF_Grp_Setup`. Once configured for external timing, the external signal on the I/O connector clocks the DACs and controls the update interval. During an externally timed waveform generation operation on the NB-MIO-16X, DMA is disabled.

WF_Check

Function

Checks to see if at least one cycle of the output waveform has been generated on the specified output channel.

Synopsis

C Syntax	`locus i32 WF_Check(u32 deviceNumber, u32 channel, u16 *status);`
Pascal Syntax	`function WF_Check(deviceNumber : i32; channel : i32; var status : i16) : i32;`
BASIC Syntax	`FN WF_Check(deviceNumber&, channel&, status&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog output channel.

Range: 0 through *n*-1, where *n* is the number of analog output channels on the board.

status returns a value that reflects certain conditions of the waveform generation operation. If **count** is set to 0 in `WF_Load`, a **status** of 0 indicates the waveform operation has not completed, while a **status** of 1 indicates that the waveform operation has completed. If **count** is set to 1 in `WF_Load`, a **status** of 0 indicates that less than one full buffer of data has been generated, while a **status** of 1 indicates that the buffer has been generated at least once. For the double-buffered cases, a **status** of 0 indicates either that the waveform operation has not started or that it is still in progress, while a **status** of 1 indicates that the waveform operation has terminated. If the waveform operation has terminated, the error code indicates whether the process halted under normal conditions or aborted.

For single-cycle waveforms, the **buffer** parameter used in `WF_Load` should not be deallocated until `WF_Check` returns **status** is 1. For repetitive output or double-buffered waveforms, the **buffer** parameter used in `WF_Load` should not be deallocated until a waveform reset function is called.

WF_DBLoad

Function

Loads the waveform buffers to be used for single-buffered or double-buffered waveform generation. `WF_DBLoad` is a superset of `WF_Load`; you can use it in place of `WF_Load`. To configure double-buffered waveforms, you *must* use `WF_DBLoad`. Table 10-3 shows the conditions under which you can use double-buffered waveform generation.

Table 10-3. Conditions When You Can Use Double-Buffered Waveform Generation

Device	Conditions
NB-AO-6	• Can be used if a DMA board is installed in the system • Can be applied to asychronous or synchronous waveform generation
NB-MIO-16	• Can be used if a DMA board is installed in the system • Can be applied to asychronous or synchronous waveform generation
NB-MIO-16X	• Can be used if a DMA board is installed in the system and the external update clock is *not* used • Can be applied to asychronous or synchronous waveform generation
PCI-MIO-16XE-50	• No restrictions

WF_Load

Function

Loads the waveform buffers to be used for single-buffered waveform generation. `WF_Load` is a subset of `WF_DBLoad`; you can use it in place of `WF_DBLoad` *except* when using double-buffered waveforms.

WF_DBLoad Synopsis

C Syntax	`locus i32 WF_DBLoad(u32 deviceNumber, u32 channel, i16 *buffer, u32 count, u32 mode, u32 timeout);`
Pascal Syntax	`function WF_DBLoad(deviceNumber : i32; channel : i32; buffer : pi16; count : i32; mode : i32; timeout : i32) : i32;`
BASIC Syntax	`FN WF_DBLoad(deviceNumber&, channel&, buffer&, count&, mode&, timeout&)`

WF_Load Synopsis

C Syntax	`locus i32 WF_Load(u32 deviceNumber, u32 channel, i16 *buffer, u32 count, u32 mode);`
Pascal Syntax	`function WF_Load(deviceNumber : i32; channel : i32; buffer : pi16; count : i32; mode : i32) : i32;`
BASIC Syntax	`FN WF_Load(deviceNumber&, channel&, buffer&, count&, mode&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog output channel.

Range: 0 through *n*-1, where *n* is the number of analog output channels on the board.

buffer is an integer buffer of length **count**. The elements of **buffer** are the values constituting the output waveform.

For most boards, if the specified analog output channel is configured for bipolar output, each element of the buffer should range from -2,048 to +2,047. Otherwise, if the analog output channel is configured for unipolar output, each element of the buffer should range from 0 to 4,095.

When using the NB-MIO-16, each element of the buffer should range from 0 to 4,095 regardless of whether the specified analog output channel is configured for unipolar or bipolar output. `WF_Offset` can be called prior to `WF_Load` or `WF_DBLoad` to offset the buffer values by 2,048 if needed (see `WF_Offset`).

count is the number of elements in **buffer**.

Range: *m* through *n*, where *m* and *n* are defined by **count** as follows:

count	*m*	*n*
0	2	$2^{23} - 1$
1	2	$2^{23} - 1$
8† (*load initial*)	256	$2^{23} - 2$
9† (*load initial*)	256	$2^{23} - 2$
10 (*load next*)	128	$n_0 \div 2$
11 (*load next*)	128	$n_0 \div 2$
12 (*load last*)	128	$n_0 \div 2$
13 (*load last*)	128	$n_0 \div 2$

†**count** must be even

n_0 is the value specified for *n* when **count** was 8 or 9 (when waveform generation was started)

If no DMA board is installed, the range for *n* is 2 through 2^{31} (except the E Series boards) and 2 through 2^{24} (E Series boards).

count selects the generation mode for the waveform. The **count** parameter has the following possible values:

0: single-buffered waveform, single playback of buffer
1: single-buffered waveform, continuous playback of buffer
8: double-buffered waveform, load initial buffers, stop at end of second buffer if no new data is loaded
9: double-buffered waveform, load initial buffers, replay both buffers until new data is loaded
10: double-buffered waveform, load next buffer, stop at end of added buffer if no new data is loaded
11: double-buffered waveform, load next buffer, replay last two buffers until new data is loaded
12: double-buffered waveform, load last buffer, stop at end of added buffer
13: double-buffered waveform, load last buffer, replay last two buffers until stopped by a stop or reset call

A **count** setting of 0 or 1 selects a single-buffered waveform-generation mode–the former specifies a single playback of the loaded buffer, while the latter specifies continuous playback of the loaded buffer (until a stop or reset call is executed).

A **count** setting of 8, 9, 10, 11, 12, or 13 selects double-buffered waveform generation. When double-buffered waveform generation is used, there are three different cases to consider–*load initial*, *load next*, and *load last*. Each of these cases has, in turn, two different playback options: *stop at end* or *continuous*.

The *load initial* case is used to configure the initial buffers and to indicate to the software that double-buffered waveform generation should be used. In this case, the number of points loaded determines the size of the master buffer. The maximum size of each half of the master buffer is (**count** ÷ 2). For example, if you wish to continue adding 10,000-point buffers to the system, you should load at least 20,000 points during the *load initial* call.

The *load next* and *load last* cases are used to copy a new buffer into the next available half of the master buffer. Conceptually, there is no difference between these two modes; however, there is a functional difference associated with the amount of error checking that takes place—the *load last* mode makes additional checks that prevent any new buffers from being loaded.

The *stop at end* playback option is used to prevent stale data–data already used to generate an output–from being replayed. In the event that a new buffer of data is not loaded before the end of the last buffer is reached, waveform generation is terminated with an error.

The *continuous* playback option is used to replay the last two buffers in the event new data is not loaded before the end of the last buffer is reached. If this option is selected, *both* halves of the master buffer are continuously replayed–buffer *A*, buffer *B*, buffer *A*, buffer *B*, and so on. If the previous mode was *load initial* or *load next*, new data can be copied into the next available half of the master buffer.

Note: ***This function has no provision for specifying which half of the master buffer will be loaded. The function loads the halves sequentially, first*** *A*, ***then*** *B*, ***and so on.***

timeout specifies, in ticks, the maximum amount of time to wait before returning from the load call; there are 60 ticks in each second. This variable is of significance only if you are using double buffering. Figure 10-4 shows how new buffers are copied into the master buffer, which affects **timeout**.

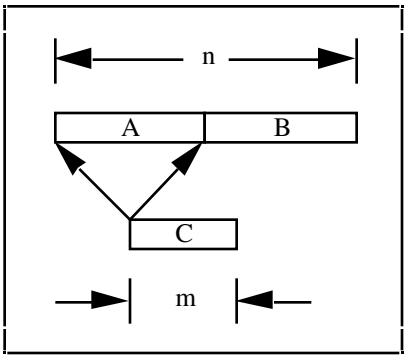

Figure 10-4. Waveform Master Buffer Scheme

When you make the *load initial* call, you have a single buffer (the master buffer) which is *n* values long. This buffer is divided into two buffers, buffer *A* and buffer *B*, each of which is (*n* ÷ 2) values long. When you make all subsequent calls, you are attempting to copy a buffer, buffer *C*, into the memory occupied by buffer *A* or buffer *B*. In this case, the length of buffer *C*, *m*, must be less than or equal to the length of buffer *A* (or buffer *B*). Therefore, $m \leq (n \div 2)$. Because buffer *C* will be copied into memory occupied by an earlier portion of your wave, the software must wait until the current buffer has been generated before it can copy buffer *C* into the master buffer. This is where **timeout** comes into play. Let us say that buffer *B* is being generated when you

try to load buffer *C*. In addition, let us say that the software just began generating buffer *B*, that buffer *A* has already been loaded with new data, and that it can take up to 30 seconds to finish generating buffer *B*. In this case, you may not want to wait all that time to load buffer *C* (which must be copied into master buffer *B*)–what you would do in this case is set a small **timeout**, perhaps 3 seconds (180 ticks), after which your program can do something else. At some later time, your program tries again to load buffer *C*. In this way, you can prevent your program from stopping operation for a long period of time whenever you try to load a new buffer of data.

You can use `WF_Load` or `WF_DBLoad` to load a waveform buffer for asynchronous and synchronous waveform generation on an analog output channel. To set up an asynchronous waveform generation, the desired channel and update interval must first be specified in a call to `WF_Setup`. `WF_Load` or `WF_DBLoad` is then called to specify a waveform buffer to be generated on the analog output channel. Once the waveform has been loaded, `WF_Start`, `WF_Stop`, `WF_Reset`, and `WF_Check` can be used to execute and complete the asynchronous waveform operation. An error code is returned if `WF_Setup` has not been called before `WF_Load` or `WF_DBLoad`. If you are using double buffering, `WF_DBLoad` is called in the *load initial* mode, `WF_Start` is called, then `WF_DBLoad` is called again to update the master buffer.

To set up a synchronous waveform generation, the desired channels must first be specified in a `WF_Grp_Setup` call. The update interval indicated in `WF_Grp_Setup` applies to all the channels in the group. After `WF_Grp_Setup`, `WF_Load` or `WF_DBLoad` is called once for each channel in the group. Once a waveform has been loaded for each channel, `WF_Grp_Start`, `WF_Grp_Stop`, `WF_Grp_Reset`, and `WF_Check` can be used to execute and complete the synchronous waveform operation. An error is returned if `WF_Grp_Setup` is not called before `WF_Load` or `WF_DBLoad` or if **channel** was not included in the group defined by `WF_Grp_Setup`. If you are using double buffering, `WF_DBLoad` is called in the *load initial* mode for each channel, `WF_Grp_Start` is called, then `WF_DBLoad` is called again to update the master buffer–you must make one call for each channel in the group.

There are several new error conditions that `WF_DBLoad` or `WF_Check` can return, as follows:

- **noDataErr**
- **dmaChainingErr**
- e**ndBlkLoadedErr**

A code of **noDataErr** indicates that no new data was available to replace the data that was just generated. This error occurs when the *stop at end* playback option halts the playback, and a subsequent `WF_DBLoad` is attempted. In this case, you can do one of two things to remedy the problem–either increase the update interval (which increases the time required to generate the buffer), or increase the number of points in the buffer.

A code of **dmaChainingErr** indicates that the software was unable to reprogram the DMA controller before it finished generating the current buffer. Usually, this error indicates that some system operation had a higher interrupt priority and was using the processor when the DMA controller requested an update. To remedy the problem, try running your waveform operation again. This should solve the problem in most cases. If the error occurs repeatedly, treat this error as if it were a **noDataErr**.

A code of **endBlkLoadedErr** indicates that the last buffer of data has already been loaded (a previous load call was made with mode set to *load last*). You must restart the waveform operation before another load call can be made.

In general, we suggest that you disable virtual memory when using the double-buffered waveform generation calls. The system uses interrupts to update the buffers; using virtual memory increases system interrupt latency to the extent that DMA may shut down because of a lack of new data before the waveform buffers can update.

For single-cycle waveforms, do not deallocate the **buffer** parameter used in `WF_Load` or `WF_DBLoad` until `WF_Check` returns 1 in **status**. For repetitive output waveforms or double-buffered waveforms, do not deallocate the **buffer** parameter used in `WF_Load` or `WF_DBLoad` until you call a waveform reset function.

WF_Grp_Reset

Function

Resets the synchronous waveform by stopping waveform generation, and releases any DMA channels, analog output channels, and trigger lines being used for synchronous waveform generation.

Synopsis

C Syntax	`locus i32  WF_Grp_Reset(u32 deviceNumber);`
Pascal Syntax	`function   WF_Grp_Reset(deviceNumber : i32) : i32;`
BASIC Syntax	`FN  WF_Grp_Reset(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`WF_Grp_Reset` first calls `WF_Grp_Stop` to stop synchronous waveform generation on the specified board and then releases any DMA channels on the NB-DMA-8-G or NB-DMA2800 board, analog output channels on the specified board, and trigger lines on the RTSI bus that were used for synchronous waveform generation.

WF_Grp_Setup

Function

Configures the board for subsequent synchronous waveform generation on a group of channels.

Synopsis

C Syntax	`locus i32  WF_Grp_Setup(u32 deviceNumber, u32 channelCount, u16 *channels, u32 interval, u32 timebase);`
Pascal Syntax	`function   WF_Grp_Setup(deviceNumber : i32; channelCount : i32; channels : pi16; interval : i32; timebase : i32) : i32;`
BASIC Syntax	`FN  WF_Grp_Setup(deviceNumber&, channelCount&, channels&, interval&, timebase&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channelCount is the number of analog output channels to be used for synchronous waveform generation. The following values are valid for **channelCount** on the NB-AO-6:

1, 2, 3, 4, 5, 6.

The following values are valid for **channelCount** on the NB-MIO-16X, MIO E Series, and Lab and 1200 series:

1, 2.

channels is an integer array of length **channelCount** that indicates the analog output channels to be used for synchronous waveform generation.

Range: 0 through *n*-1, where *n* is the number of analog output channels on the board

channels can contain any analog output channel number in any order. For example, if **channelCount** is 4, and if **channels**[*i*] is 2, then analog output channel 2 is one of the four channels on which a synchronous waveform can be generated.

interval is the update interval to be used for synchronous waveform generation.

Range: 3 through 65,536 (except the E Series boards).
2 through 2^{24} (E Series boards).

The **interval** is a function of the **timebase** used. The actual update interval in seconds is given by the following formula:

interval ∗ timebase resolution

where the timebase resolution for each value of **timebase** is indicated below. For example, if **interval** = 25 and **timebase** = 2, then the update interval is 25 ∗ 10 µs = 250 µs. The **interval** parameter is ignored if **timebase** is passed as 0.

timebase indicates the resolution of the timer used for the update interval counter. **timebase** has the following possible values:

-3: 20 MHz clock used as timebase (50-ns resolution).
0: External clock used to provide update interval.
1: 1-MHz clock used as timebase (1-µs resolution).
2: 100-kHz clock used as timebase (10-µs resolution).
3: 10-kHz clock used as timebase (100-µs resolution).
4: 1-kHz clock used as timebase (1-ms resolution).
5: 100-Hz clock used as timebase (10-ms resolution).
6: SOURCE1 used as timebase.
7: SOURCE2 used as timebase.
8: SOURCE3 used as timebase.
9: SOURCE4 used as timebase.
10: SOURCE5 used as timebase.
11: External timebase

Range: 0 to 10 for most devices.
-3, 0, 2, 11 for E Series devices.

SOURCE1 through SOURCE5 are timing signals available on the NB-DMA-8-G and NB-DMA2800. See the description of NB-DMA-8-G and NB-DMA2800 counters and timers in Chapter 8, *Counter/Timer Functions*, for more information about these signals. If the waveform generation is on an NB-MIO-16X board, and an NB-DMA-8-G or NB-DMA2800 is not present in the system, then SOURCE 1 through SOURCE 5 are timing signals available on the NB-MIO-16X. See the description of NB-MIO-16X counters and timers in Chapter 8, *Counter/Timer Functions* for more information about these signals.

Setting **timebase** to 0 indicates that a signal connected to the I/O connector supplies the total update interval. For example, if the frequency of the clock signal connected to the I/O connector is 5000 Hz, then the total update interval is 1/5000, or 200 µs. The external signal is connected to the EXTUPDATE* line on the Lab and 1200 series I/O connector. The external signal is connected to the OUT2 signal on the NB-MIO-16X I/O connector. The external signal is connected to the EXT.UPD signal on the NB-AO-6 I/O connector. Use the `AO_Setup` call to configure the edge used for updating the DACs when using EXT.UPD. Connect your external signal to the PFI5 pin, by default, or use the `Select_Signal` function to specify a different source for E Series devices.

`WF_Grp_Setup` marks the analog output channels indicated in **channels** *busy*. If DMA is available, this function allocates the appropriate number of DMA channels on an NB-DMA-8-G or NB-DMA2800 present in the system. Waveform generation on the NB-AO-6 requires DMA. Waveform generation on the NB-MIO-16X uses DMA if it is available; waveform generation on the PCI-MIO-16XE-50 and Lab and 1200 series never uses DMA. The appropriate error code is returned if DMA is needed and an NB-DMA-8-G or NB-DMA2800 board is not detected in the system or one or more of the required DMA channels are not available. Synchronous waveform generation using DMA also requires the use of a trigger line on the RTSI bus; therefore, at least one

RTSI bus trigger line should be available. The DMA channels, analog output channels, and one RTSI bus trigger line used by the synchronous waveform generation operation are allocated by the `WF_Grp_Setup` call and are deallocated by a call to `WF_Grp_Reset`.

WF_Grp_Start

Function

Initiates synchronous waveform generation.

Synopsis

C Syntax	`locus i32 WF_Grp_Start(u32 deviceNumber);`
Pascal Syntax	`function WF_Grp_Start(deviceNumber : i32) : i32;`
BASIC Syntax	`FN WF_Grp_Start(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`WF_Grp_Start` initiates synchronous waveform generation on the specified board. The number of channels, update interval, and channel specification are selected in a `WF_Grp_Setup` call, and the waveform buffer or buffers to be used are specified in `WF_Load` calls.

If **count** is 0 in a call to `WF_Load` for an analog output channel, then a single cycle of the waveform is generated for that channel. In single-cycle waveform generation, the contents of the buffer are output only once. Otherwise, the waveform is generated repeatedly until `WF_Grp_Stop` or `WF_Grp_Reset` is called.

A call to `WF_Grp_Start` can immediately follow a call to `WF_Grp_Stop` in order to restart waveform generation. Waveform generation restarts at the position in the buffer where `WF_Grp_Stop` stopped the waveform generation. The MIO E Series devices do not support `WF_Grp_Stop`.

WF_Grp_Stop

Function

Stops synchronous waveform generation.

Synopsis

C Syntax	`locus i32 WF_Grp_Stop(u32 deviceNumber);`
Pascal Syntax	`function WF_Grp_Stop(deviceNumber : i32) : i32;`
BASIC Syntax	`FN WF_Grp_Stop(deviceNumber&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

`WF_Grp_Stop` stops synchronous waveform generation on the specified board. A `WF_Grp_Start` call can immediately follow a `WF_Grp_Stop` call to restart the waveform generation on the specified channel.

Waveform generation restarts at the position in the buffer where waveform generation was stopped by `WF_Grp_Stop`. Notice that any DMA channels, analog output channels, and RTSI bus trigger lines used for synchronous waveform generation remain allocated after a `WF_Grp_Stop` call. `WF_Grp_Reset`, on the other hand, stops synchronous waveform generation and releases any DMA channels, analog output channels, and RTSI trigger lines used. The MIO E Series devices do not support `WF_Grp_Stop`.

WF_Offset

Function

Offsets the values in a waveform buffer for bipolar waveform generation (NB-MIO-16 only).

Synopsis

C Syntax	`locus i32 WF_Offset(u32 deviceNumber, u32 channel, u32 count, i16 *buffer);`
Pascal Syntax	`function WF_Offset(deviceNumber : i32; channel : i32; count : i32; buffer : pi16) : i32;`
BASIC Syntax	`FN WF_Offset(deviceNumber&, channel&, count&, buffer&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog output channel.

Range: 0 through *n*-1, where *n* is the number of analog output channels on the board.

buffer is an integer buffer of length **count**. The elements of the buffer are the values constituting the output waveform. If the DAC for the specified channel is configured for bipolar output, each element of the buffer should range from -2,048 to +2,047. Otherwise, if the DAC is configured for unipolar output, each element of the buffer should range from 0 to 4,095.

count is the number of elements in **buffer.**

Range: 1 through 2^{23}-1.

When using the NB-MIO-16, each element of the buffer passed to `WF_Load` should always range from 0 to 4,095, regardless of whether the DAC for the specified channel is configured for unipolar or bipolar output. `WF_Offset` can be called prior to `WF_Load` to offset each element in the buffer by 2,048 if needed.

`WF_Offset` first checks if **deviceNumber** is an NB-MIO-16 and if **channel** has been configured for bipolar output. If both of these conditions are met, `WF_Offset` simply adds an offset of 2,048 to each element in **buffer**. This addition results in **buffer** containing values ranging from 0 to 4,095. If either of the conditions is not met, `WF_Offset` returns **buffer** unchanged. `WF_Load` can then be called to select this waveform buffer for waveform generation on the analog output channel.

WF_Reset

Function

Resets the asynchronous waveform by stopping waveform generation, and releases any DMA channel and analog output channel being used for asynchronous waveform generation.

Synopsis

C Syntax	`locus i32 WF_Reset(u32 deviceNumber, u32 channel);`
Pascal Syntax	`function WF_Reset(deviceNumber : i32; channel : i32) : i32;`
BASIC Syntax	`FN WF_Reset(deviceNumber&, channel&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog output channel.

Range: 0 through *n*-1, where *n* is the number of analog output channels on the board.

`WF_Reset` first calls `WF_Stop` to stop asynchronous waveform generation on the specified board and then releases the DMA channel on the NB-DMA-8-G or NB-DMA2800 board and the analog output channel on the specified board.

WF_Setup

Function

Selects the update rate used for asynchronous waveform generation.

Synopsis

C Syntax	`locus i32 WF_Setup(u32 deviceNumber, u32 channel, u32 interval, u32 timebase);`
Pascal Syntax	`function WF_Setup(deviceNumber : i32; channel : i32; interval : i32; timebase : i32) : i32;`
BASIC Syntax	`FN WF_Setup(deviceNumber&, channel&, interval&, timebase&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog output channel.

Range: 0 through *n*-1, where *n* is the number of analog output channels on the board.

interval is the update interval to be used for waveform generation on the specified channel.

Range: 3 through 65,536.

The **interval** is a function of the **timebase** used. The following formula gives the actual update interval in seconds:

interval * timebase resolution

where the timebase resolution for each value of **timebase** is indicated below. For example, if you set **interval** to 25 and **timebase** to 2, then the update interval is 25 ∗ 10 µs = 250 µs.

timebase indicates the resolution of the timer used for the update interval counter. The **timebase** parameter has the following possible values:

1: 1-MHz clock used as timebase (1-µs resolution).
2: 100-kHz clock used as timebase (10-µs resolution).
3: 10-kHz clock used as timebase (100-µs resolution).
4: 1-kHz clock used as timebase (1-ms resolution).
5: 100-Hz clock used as timebase (10-ms resolution).
6: SOURCE1 used as timebase.
7: SOURCE2 used as timebase.
8: SOURCE3 used as timebase.
9: SOURCE4 used as timebase.
10: SOURCE5 used as timebase.

SOURCE1 through SOURCE5 are timing signals available on the NB-DMA-8-G and NB-DMA2800. See the description of NB-DMA-8-G and NB-DMA2800 counters and timers in Chapter 8, *Counter/Timer Functions*, for more information about these signals.

`WF_Setup` selects the update interval and timebase to be used for asynchronous waveform generation. If the given channel has been configured as part of a synchronous group through a `WF_Grp_Setup` call, an error is returned, indicating that the specified channel is not available.

WF_Start

Function

Initiates asynchronous waveform generation.

Synopsis

C Syntax	`locus i32 WF_Start(u32 deviceNumber, u32 channel);`
Pascal Syntax	`function WF_Start(deviceNumber : i32; channel : i32) : i32;`
BASIC Syntax	`FN WF_Start(deviceNumber&, channel&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog output channel.
Range: 0 through *n*-1, where *n* is the number of analog output channels on the board.

`WF_Start` initiates asynchronous waveform generation on the specified channel. The waveform buffer must be specified in a `WF_Load` call. If `WF_Start` is called without calling `WF_Load` first, an error code is returned.

If **count** is set to 0 in a call to `WF_Load` for this analog output channel, then a single cycle of the waveform is generated. In single-cycle waveform generation, the contents of the buffer are output once and then the operation stops. Otherwise, the waveform is generated repeatedly until `WF_Stop` or `WF_Reset` is called.

A call to `WF_Start` can immediately follow a call to `WF_Stop` in order to restart the waveform generation. Waveform generation restarts at the position in the buffer where waveform generation was stopped by `WF_Stop`.

WF_Stop

Function

Stops asynchronous waveform generation.

Synopsis

C Syntax	`locus i32 WF_Stop(u32 deviceNumber, u32 channel);`
Pascal Syntax	`function WF_Stop(deviceNumber : i32; channel : i32) : i32;`
BASIC Syntax	`FN WF_Stop(deviceNumber&, channel&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

channel is the analog output channel.

Range: 0 through *n*-1, where *n* is the number of analog output channels on the board.

`WF_Stop` stops asynchronous waveform generation on the specified board. A call to `WF_Start` can immediately follow a call to `WF_Stop` in order to restart waveform generation on the specified channel. Notice that the DMA channel and analog output channel used for asynchronous waveform generation remain allocated after a call to `WF_Stop`. `WF_Reset`, on the other hand, stops the asynchronous waveform generation and releases the DMA channel and analog output channel used.

Buffered Waveform Generation Function Summary

Use the following functions for buffered waveform generation operations on the NB-A2100.

`BWF_BlkLoad`	Updates the data in the waveform buffer without interrupting the background waveform generation.
`BWF_BufLoad`	Initializes a waveform buffer for waveform generation. Indicates the data values and the rate at which the waveform buffer will be output. Also disables or enables block update mode for the waveform generation.
`BWF_Check`	Reports the completion status of the waveform generation, the buffer number being output, the number of cycles that have been output, the number of points that have been output in the current cycle (optionally), and the current waveform generation rate.
`BWF_Clear`	Stops the waveform generation in progress and deallocates the resources being used for waveform generation.
`BWF_Rate`	Changes the current waveform generation rate at which the output channels are updated.
`BWF_Resume`	Resumes the waveform generation stopped by `BWF_Stop`.
`BWF_Start`	Starts waveform generation using the selected waveform buffer.
`BWF_Stop`	Stops the waveform generation in progress.

Buffered Waveform Generation Terminology

The following terminology is used to describe the Buffered Waveform Generation Functions.

Name	Description
Group	A group of channels configured to share the same update clock.
Channel	An analog output channel number on which a waveform is being generated.
Waveform Buffer	The one-dimensional array that contains the data to be output at a selected channel.
Points	The number of data points that a waveform buffer contains.
Cycles	The number of times the waveform buffer is cyclically output.
Update Interval	The amount of time that elapses between the output of two consecutive points to the output channel. If the output buffer contains 100 points and the update interval is 20 µs, the total time to output one cycle would be 100 ∗ 20 = 2000 µs.
Blocks	A division of a waveform buffer. A waveform buffer can be divided into two or more blocks for block update mode waveform generation.
Block Update Mode	The waveform generation mode in which you can update data in one block of a waveform buffer while another block of that waveform buffer is being generated.
Interleaved Buffer Data	The data for channels 0 and 1 alternated in the same waveform buffer–channel 0's first point, channel 1's first point, channel 0's second point, channel 1's second point, and so on. The two channels are tightly coupled and cannot be controlled independently. Therefore, channels 0 and 1 are treated as a single channel (channel 2) with one shared interleaved buffer.

Buffered Waveform Generation Application Hints

Use the buffered waveform generation (BWF) functions for generation of analog output waveforms on the NB-A2100. The BWF functions use a circular waveform buffer to continuously generate a waveform at the analog output channels. You can use the BWF functions to update the circular buffer and change the output waveform data or output rate while waveform generation is in progress. With a circular buffer, you can seamlessly generate an unlimited amount of data without requiring an unlimited amount of memory. Buffered waveform generation is useful for data streaming from disk and for changing the output data in the circular waveform buffer on the fly. Application flowcharts shown below step through creating a function generator or stream to disk application.

Buffered Waveform Generation Call Sequence

You can begin building a buffered waveform generation application by using `BWF_BufLoad` to load and initialize a circular output buffer. `BWF_BufLoad` selects the analog output channels, output rate, and the update mode. A circular buffer created by `BWF_BufLoad` can be divided into individual blocks. These block divisions are used later for updating blocks of data without interrupting the waveform generation.

Call `BWF_Start` to start the waveform generation.

While the waveform generation is in progress, you can optionally call five BWF functions repeatedly to control or monitor the waveform generation: `BWF_Check`, `BWF_Rate`, `BWF_BlkLoad`, `BWF_Stop`, and `BWF_Resume`. `BWF_Check` monitors the status of the waveform generation. `BWF_Rate` changes the output rate on the fly. `BWF_BlkLoad` changes the output waveform values using various update methods described below. `BWF_Stop` suspends waveform generation. `BWF_Resume` continues waveform output from where it was suspended by `BWF_Stop`.

Call `BWF_Clear` to complete the waveform generation operation.

If changing the output waveform data or output rate while waveform generation is in progress is not necessary, then BWF_BufLoad, BWF_Start, and BWF_Clear are the only functions needed to output a cycle of a waveform or a continuous waveform.

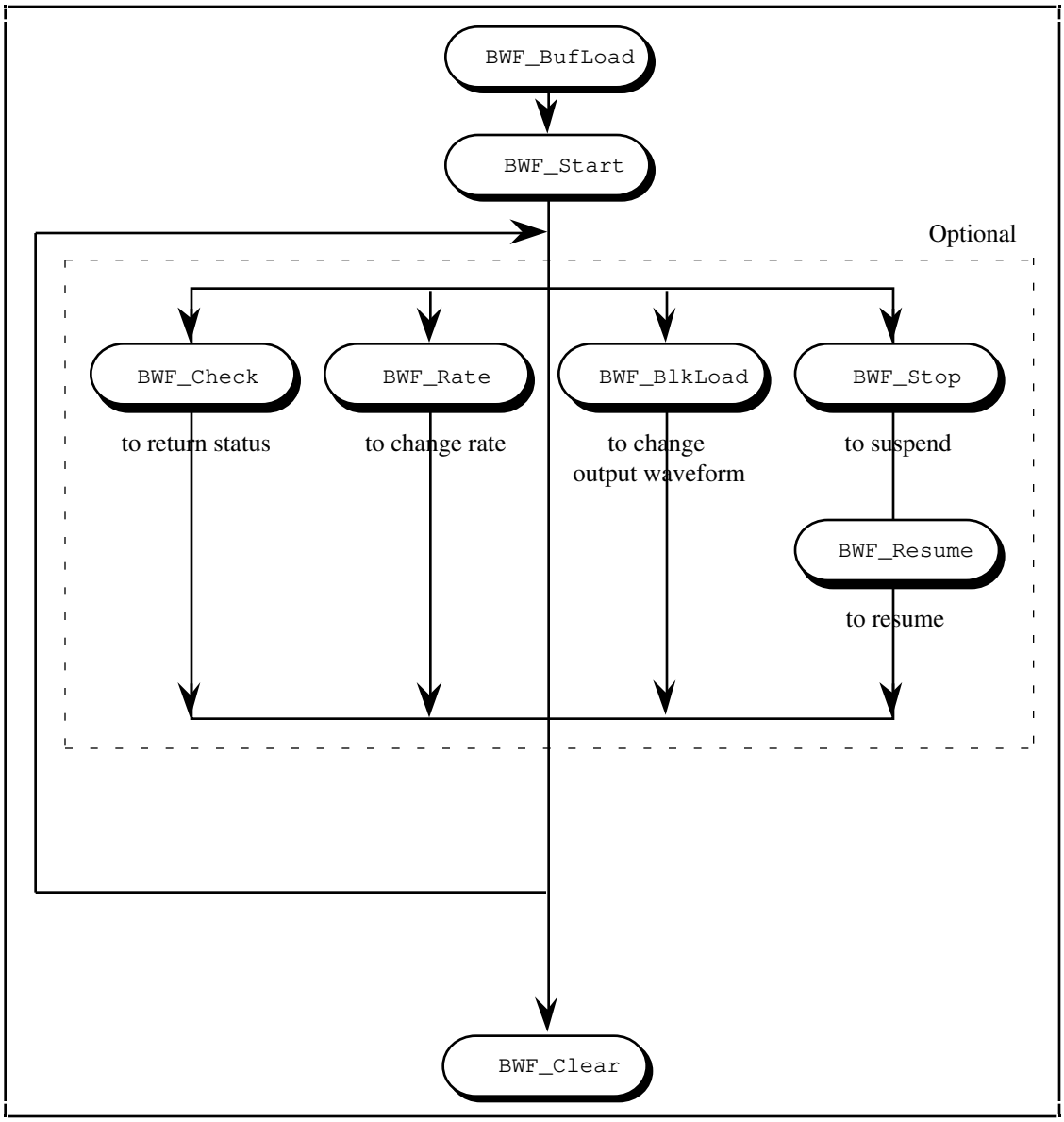

Figure 10-5. BWF Function Flowchart

Initializing Buffered Waveform Generation

BWF_BufLoad selects the analog output channels, output rate, and the update modes. BWF_BufLoad also initializes a circular output waveform buffer for waveform generation. The data in the circular buffer is cyclically generated continuously. When the end of the circular buffer is reached, data from the beginning of the buffer is generated until the specified number of cycles of the circular buffer have been generated or until the waveform generation operation is cleared.

If you want to synchronize the update of the waveform output values as the waveform generation is in progress, you should use BWF_BufLoad to divide the circular buffer into individual *blocks*. These block divisions are the update divisions used later by BWF_BlkLoad for updating data without interrupting the waveform generation. The background waveform generation can be thought of as actually being performed in continuous fragments that are the size of the smaller blocks specified. Figure 10-6 represents blocks in the large circular buffer. If updates do not need to be synchronized, then specify a block size of 0 in BWF_BufLoad and use immediate update in BWF_BlkLoad.

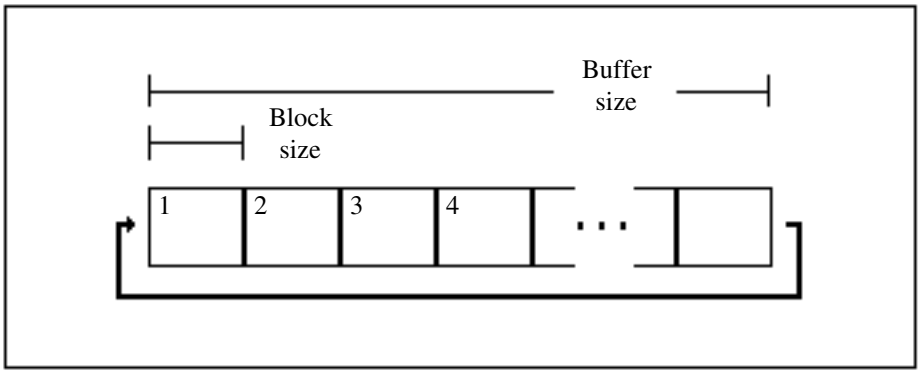

Figure 10-6. Circular Waveform Buffer and Blocks

If you have divided the circular buffer into blocks and you do not want regeneration of old data to occur, you should set **regenerationMode** in BWF_BufLoad to 1. This insures that the waveform generation stops and data that has been generated once is not generated a second time if the data has not been updated in the circular waveform buffer with BWF_BlkLoad.

After initializing buffered waveform generation with BWF_BufLoad, you can start waveform generation by executing BWF_Start. This action begins the D/A conversion operations and the generation of data from the circular waveform buffer. If changing the output waveform data or output rate while waveform generation is in progress is not necessary, then BWF_BufLoad, BWF_Start, and BWF_Clear are the only functions needed to output a cycle of a waveform or a continuous waveform.

Updating Waveform Output During Waveform Generation

Use BWF_BlkLoad to update data in the circular output buffer during waveform generation. You can update a block of data or a range of points. BWF_BlkLoad updates data on-the-fly in either of two modes: block update and immediate. Block update mode synchronizes data updates and provides underwrite error detection. Immediate mode updates a range of points immediately with no error checking performed.

Note: ***DMA is required for block update mode.***

Block Update of the Output Waveform

Setting **copyMode** equal to 1 or 2 in BWF_BlkLoad selects block update for updating the circular waveform buffer during waveform generation.

If **copyMode** is 1, the **count** selected in BWF_BlkLoad must be an integral multiple of the **blockSize** specified in BWF_BufLoad. This means that you can update one or more blocks in the block update mode with one BWF_BlkLoad execution. When multiple blocks are being updated, BWF_BlkLoad waits until all the blocks have been updated before returning. If **copyMode** is 2, the **count** selected in BWF_BlkLoad does not have to be an integral multiple of block size specified in BWF_BufLoad. When **copyMode** is set to 2, waveform generation

stops when data being copied into the circular waveform buffer by BWF_BlkLoad is generated once. This mode is useful for cleanly stopping waveform generation after a data file's last fragment which may not be the exact multiple of **blockSize**. You can use the **timeout** control in BWF_BlkLoad to set the maximum amount of time that BWF_BlkLoad should wait to update the circular waveform buffer.

You can use two mechanisms for updating circular waveform buffer data during block update mode of waveform generation. The *sequential* update mechanism blocks of data in the order that they are generated. The sequential update is selected by setting **startPoint** to 0 in BWF_BlkLoad. The *selected* update mechanism lets you update any specified block. The selected update is used if **startPoint** is set to a nonzero value in BWF_BlkLoad.

Sequential Block Update

Setting the **startPoint** parameter in BWF_BlkLoad to 0 selects the *sequential* update mode. Data passed to BWF_BlkLoad is copied into the next circular waveform buffer block, wrapping back to the beginning if needed. An internal pointer cycles through the buffer to keep track of the next block. You can use sequential block update to easily buffer sequential output blocks without keeping track of their position in the circular buffer.

Selected Block Update

Setting **startPoint** in BWF_BlkLoad to a block boundary chooses the *selected* update mode. A block boundary is defined by the number of points in the circular waveform buffer and the block size. So, if the circular buffer size is 10,000 and the block size is 2,500, the block boundaries would be at 1, 2,501, 5,001 and 7,501. When **startPoint** is at a block boundary, data in the block passed to BWF_BlkLoad is copied into the block of the circular waveform buffer which starts at the block boundary specified by **startPoint.** After the data is copied into the circular buffer, the internal pointer is not updated as it was in the *sequential* update mode.

For both the *sequential* and the *selected* update modes, BWF_BlkLoad will not update a block buffer until it has been generated as specified by the **blockRepititions** control in BWF_BlkLoad. If **blockRepititions** is 1, BWF_BlkLoad will not update a buffer block until it has been generated at least once. This mode is useful for applications which are generating data from a large disk file. If some of the data in the circular waveform generation data has been generated more than once, BWF_BlkLoad still does the copy, but it also returns an **underWriteErr** which indicates this error condition. If **blockRepititions** is 0, BWF_BlkLoad first will not update the buffer block until the circular waveform buffer block is completely generated. This mode is useful for changing the waveform buffer data without any glitches. If some data in the circular waveform buffer block is generated while the new data is being copied into it, BWF_BlkLoad will return an underWriteErr. If multiple blocks of the circular waveform buffer are being updated, an underWriteErr is also returned if one of the blocks is generated more times than one of the blocks that has already been updated during the multiple block update. If an underWriteErr occurs while copying one of the multiple blocks, BWF_BlkLoad will wait until all the blocks have been updated or a timeout occurs. Then it will return the **underWriteErr**.

Immediate Update of the Output Waveform

You can also do an immediate update of the output waveform using BWF_BlkLoad. Setting the **copyMode** parameter in BWF_BlkLoad to 0 selects the immediate update mode. The immediate update mode immediately copies the new data passed to BWF_BlkLoad into the waveform buffer and does not perform any **underWriteErr** detection described above for block update waveform generation. The immediate update mode does not require that the waveform buffer be divided into blocks in BWF_BufLoad, therefore **blockSize** can be 0. In the immediate update mode, the **startPoint** and the **count** parameters to BWF_BlkLoad must be between 1 and the **count** parameter that was passed to BWF_BufLoad.

Writing a Stream-from-Disk Application

The flowchart in Figure 10-7 illustrates the sequence of BWF functions needed to create a stream-from-disk application.

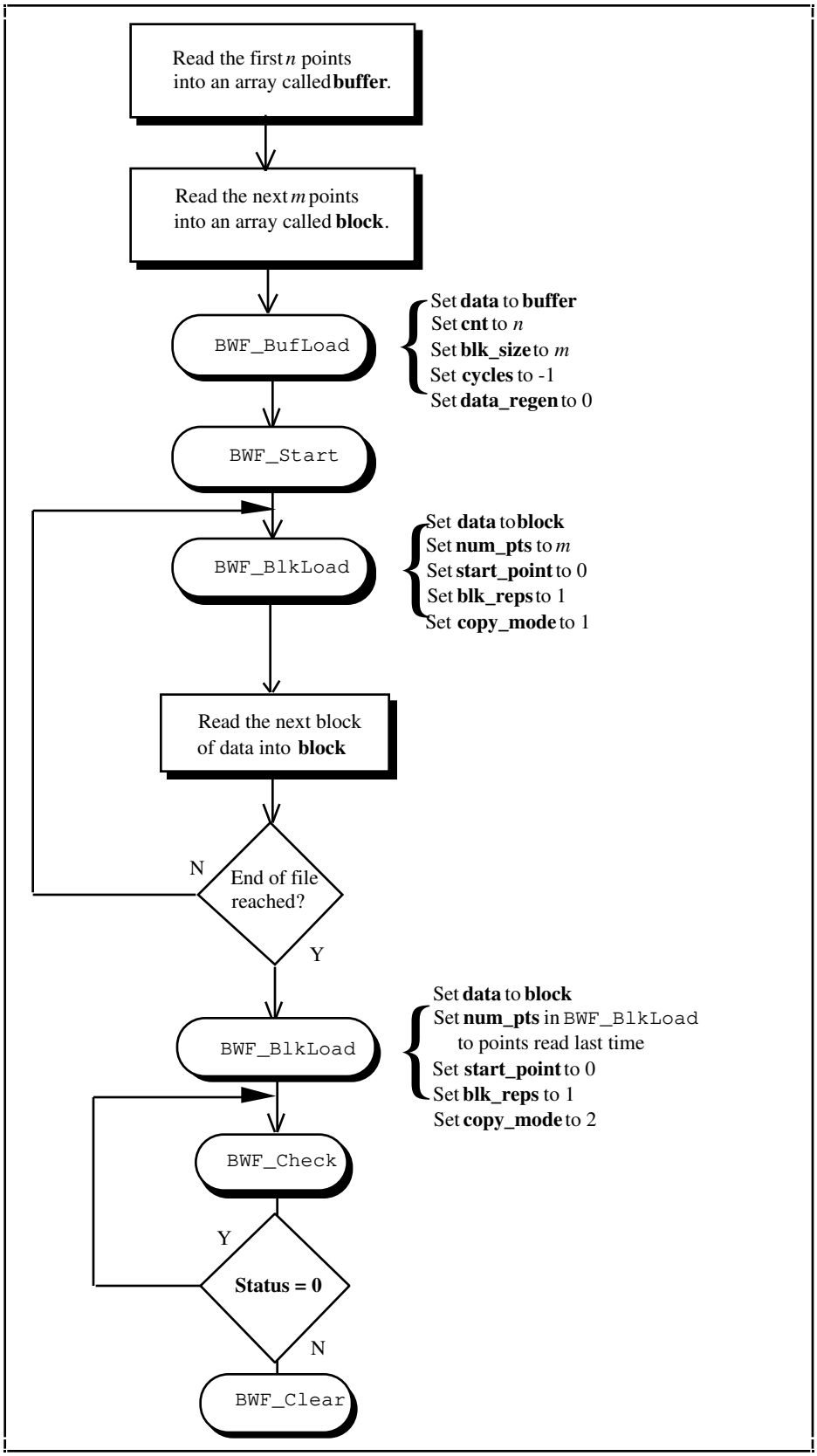

Figure 10-7. Streaming from Disk Application Hints

Writing a Function Generator Application

The flowchart in Figure 10-8 illustrates the sequence of `BWF` functions needed to create a function generator application. The function generator continuously generates an output waveform and can switch to a new output waveform at any time. The waveform output changes with no delay between the last point of the previous waveform and the first point of the new waveform. This function generator example imposes the restriction that the number of points in the output waveforms must be equal.

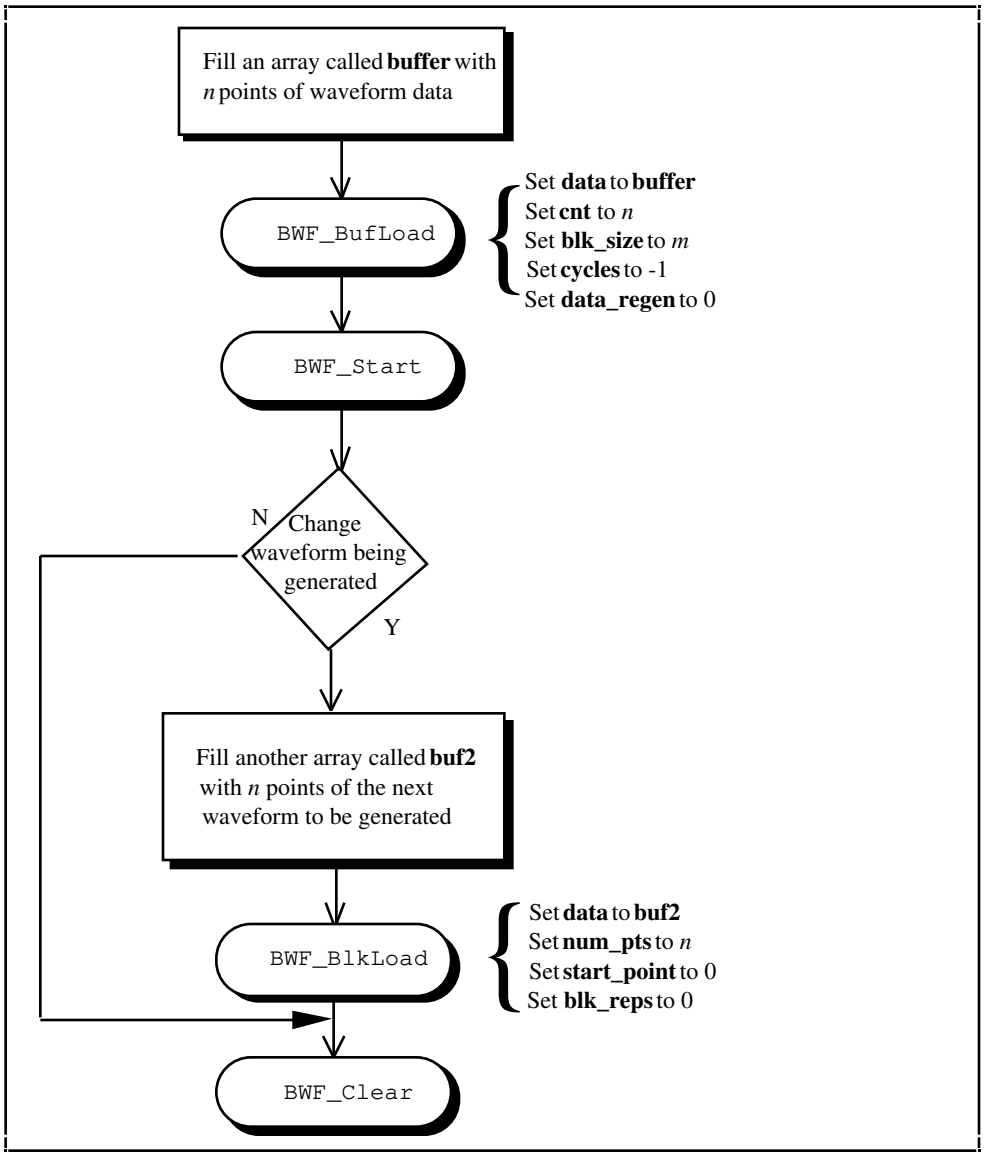

Figure 10-8. Function Generator Application Hints

BWF_BlkLoad

Function

Updates the data in the waveform buffer without interrupting the background waveform generation.

Synopsis

C Syntax	`locus i32 BWF_BlkLoad(u32 deviceNumber, u32 group, u32 channel, u32 bufferNumber, u32 copyMode, i16 *buffer, u32 count, u32`
Pascal Syntax	`function BWF_BlkLoad(deviceNumber : i32; group : i32; channel : i32; bufferNumber : i32; copyMode : i32; buffer : pi16; count : i32; startPoint : i32; blockRepititions : i32; timeout : i32; var pointsCopied : i32; var lastPoint : i32; var blockRepititionsDone : i32) : i32;`
BASIC Syntax	`FN BWF_BlkLoad(deviceNumber&, group&, channel&, bufferNumber&, copyMode&, buffer&, count&, startPoint&, blockRepititions&, timeout&, pointsCopied&, lastPoint&, blockRepititionsDone&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is analog output channel group number. **group** must be 0 for NB-A2100 because the NB-A2100 does not allow configuration of channel groups.

channel is the analog output channel number(s) selected in `BWF_BufLoad`.

Range: 0: Channel 0.
1: Channel 1.
2: Both channels 0 and 1.

bufferNumber selects the waveform buffer to be updated. **bufferNumber** is always 1 for the NB-A2100.

copyMode selects the update mode.

Range: 0: Updates a range of data points in the waveform buffer using immediate update mode.
1: Updates a block of data in the waveform buffer using block update mode and continues waveform generation.
2: Updates a block of data in the waveform buffer using block update mode and stops waveform generation after the data block has been generated.

buffer contains the block data to be copied into the waveform buffer.

count is the total number of data points in **buffer**.

If **copyMode** is 0 or 2, **count** can range from 1 through **count** selected for **bufferNumber** in `BWF_BufLoad`.

If **copyMode** is 1, **count** must be an integer multiple of **blockSize** selected for **bufferNumber** in `BWF_BufLoad`.

startPoint selects an index into the waveform buffer where **buffer** should be copied.

If **copyMode** is 0, **startPoint** must be between 1 and the **count** selected for **bufferNumber** in `BWF_BufLoad`.

If **copyMode** is 1 or 2, **startPoint** must be either at 0 or at a block boundary of **bufferNumber** . So, if **count** in `BWF_BufLoad` is 100 and **blockSize** was defined in `BWF_BufLoad` to be 25, **startPoint** can be 1, 26, 51, 76 or 0. If **startPoint** is 0, a *sequential* block update is performed. If **startPoint** is at a block boundary, *selected* block update is performed.

blockRepititions indicates when the data is copied from **buffer** into the circular buffer of **bufferNumber**.
Range: 0: `BWF_BlkLoad` updates the buffer block immediately. If the block to be updated is currently being generated, `BWF_BlkLoad` copies **buffer** into the waveform block after generation of the block is completed.
1: `BWF_BlkLoad` waits until the data in the buffer block has been generated at least once before copying **buffer** into the waveform buffer.

blockRepititions is ignored for immediate update mode (**copyMode** = 0).

timeout is the number of clock ticks ($^1/_{60}$ s) to wait to copy the data. There are two special cases for **timeout**:
-1: Wait indefinitely.
0: Return immediately if the block to be updated has not been generated.

timeout is ignored for immediate update mode (**copyMode** = 0).

pointsCopied indicates the number of points copied.

lastPoint indicates the index number of the last point where **buffer** was copied.
Range: 1 to the number of points in the waveform buffer.

blockRepititionsDone returns the number of times the data in the last updated block had been generated since the last block update if **copyMode** is 1 or 2. **blockRepititionsDone** always returns 0 if **copyMode** is 0.

See the *Buffered Waveform Generation Application Hints* section earlier in this chapter for more information on how `BWF_BlkLoad` is used.

BWF_BufLoad

Function

Initializes a waveform buffer for waveform generation. Indicates the data values and the rate at which the waveform buffer will be output. Also disables or enables block update mode for the waveform generation.

Synopsis

C Syntax	`locus i32 BWF_BufLoad(u32 deviceNumber, u32 group, u32 channel, u32 bufferNumber, u32 cycles, i16 *buffer, u32 count, u32 interval, u32 timebase, u32 triggerMode, u32 blockSize, u32 regenerationMode);`
Pascal Syntax	`function BWF_BufLoad(deviceNumber : i32; group : i32; channel : i32; bufferNumber : i32; cycles : i32; buffer : pi16; count : i32; interval : i32; timebase : i32; triggerMode : i32; blockSize : i32; regenerationMode : i32) : i32;`
BASIC Syntax	`FN BWF_BufLoad(deviceNumber&, group&, channel&, bufferNumber&, cycles&, buffer&, count&, interval&, timebase&, triggerMode&, blockSize&, regenerationMode&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is analog output channel group number. **group** must be 0 for NB-A2100 because the NB-A2100 does not allow configuration of channel groups.

channel is the analog output channel number(s).
Range: 0: Channel 0.
1: Channel 1.
2: Both channels 0 and 1.

bufferNumber is the number to be assigned to the buffer being loaded. Currently only one buffer can be loaded at a time, so **bufferNumber** must be 1.

cycles is the number of cycles of the waveform buffer to generate.
Range: -1: Generate infinite number of cycles, that is, continue waveform generation indefinitely.
≥1: Generate the selected number of cycles.

buffer is the integer buffer that contains the values defining the output waveform. If **channel** is 2 on the NB-A2100, **buffer** must contain interleaved data for channels 0 and 1.

count is the total number of data points in **buffer**. **count** must be an even number.
Range: 2 through 2^{31}.

If **cycles** is not = -1, the total number of points to be output, determined by **count** * **cycles**, must be greater than 32.

interval is the length of the update interval (that is, the amount of time to elapse between updates on the channel).
Range: 0: Use the default rate or the rate parameters defined by the latest `BWF_Rate` call since startup. At startup, the default **interval** on the NB-A2100 is set to 1 and default **timebase** is set to 1.
1: Use an update interval of 1. The actual update interval in seconds can be calculated as follows:

$$\frac{\textbf{interval}}{\text{timebase frequency}}$$

where the timebase frequency is selected by **timebase**.

timebase is the frequency of the timer used for the update interval. On the NB-A2100, timebase has the following possible values:
1: 48 kHz.
2: 44.1 kHz.
3: 32 kHz.
4: 24 kHz.
5: 22.05 kHz.
6: 6 kHz.

timebase is ignored if **interval** is 0.

triggerMode enables or disables the digital trigger input (EXTTRIG*) on the NB-A2100 I/O connector and determines when `BWF_Start` starts the waveform output.
Range: 0: Start immediately.
1: Wait for a falling edge on EXTTRIG* input.
2: Wait for a falling edge on EXTTRIG* input to start waveform output for every cycle.

If **triggerMode** is 1, and waveform output for the waveform buffer starts after a trigger is applied, the subsequent cycles of the waveform buffer are output with a software trigger. If **triggerMode** is 2, each cycle of the waveform buffer is output after a trigger is received on the EXTTRIG* input.

blockSize enables or disables block update mode. If **blockSize** is not 0, the `BWF_BufLoad` sets the number of samples in a block of data and has the following possible values:

0: Disables block update mode. Does not divide the buffer into blocks. If **blockSize** is 0, you cannot use `BWF_BlkLoad` in the block update mode. However, you can still use `BWF_BlkLoad` to update a range of points in the buffer in the immediate update mode.

16 through **count**/2: Enables block update mode. **blockSize** is used to divide the waveform buffer data into **count/blockSize** number of blocks of data. In this mode, you can use `BWF_BlkLoad` in block update mode to update the output waveform. **blockSize** should be at least 16 and such that **count** is an integer multiple of **blockSize**.

Block update mode can be enabled only if a DMA channel is available.

regenerationMode determines if waveform generation should stop after data in the waveform buffer has been generated once.

Range: 0: Continue cycling through the buffer after data has been generated once.
1: Stop after data has been generated once. This mode is only used in block update mode (**blockSize** is not 0).

If **regenerationMode** is 0, waveform generation continues even if `BWF_BlkLoad` is not called in time to replace generated data in the waveform buffer. If **regenerationMode** is 1, waveform generation stops if `BWF_BlkLoad` is not called in time to replace generated data in the waveform buffer. **blockSize** should be less than or equal to **count**/3 if you want to set **regenerationMode** to 1. Setting **regenerationMode** to 1 ensures seamless waveform generation.

The **channel** selected in `BWF_BufLoad` defines the channel(s) for waveform generation. The channel number selected in subsequent buffered waveform generation functions must be the same as **channel**. To switch to a different channel after a `BWF_BufLoad` has been made, you must call `BWF_Clear` and then call `BWF_BufLoad` with the new channel number.

BWF_Check

Function

Reports the completion status of the waveform generation, the buffer number being output, the number of cycles that have been output, the number of points that have been output in the current cycle (optionally), and the current waveform generation rate.

Synopsis

C Syntax	`locus i32 BWF_Check(u32 deviceNumber, u32 group, u32 channel, u32 checkMode, u16 *status, u32 *bufferNumber, u32 *cyclesOutput, u32 *pointsOutput, u32 *interval, u16 *timebase);`
Pascal Syntax	`function BWF_Check(deviceNumber : i32; group : i32; channel : i32; checkMode : i32; var status : i16; var bufferNumber : i32; var cyclesOutput : i32; var pointsOutput : i32; var interval : i32; var timebase : i16) : i32;`
BASIC Syntax	`FN BWF_Check(deviceNumber&, group&, channel&, checkMode&, status&, bufferNumber&, cyclesOutput&, pointsOutput&, interval&, timebase&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is analog output channel group number. **group** must be 0 for NB-A2100 because the NB-A2100 does not allow configuration of channel groups.

channel is the analog output channel number(s) selected in `BWF_BufLoad`.

Range: 0: Channel 0.
1: Channel 1.
2: Both channels 0 and 1.

checkMode selects whether to return complete or minimal status information and determines the value returned by **pointsOutput.**

Range: 0: Return minimal status information.
1: Return full status information.

Full status information requires interrupting the waveform generation, if DMA is being used, and can therefore affect waveform generation performance. Use minimal status information when possible for optimized performance.

status indicates whether waveform generation is in progress or has been stopped because the required number of cycles on the buffer loaded have been output; or an error has occurred; or `BWF_Stop` has been called.

0: The waveform generation is in progress.
1: The waveform generation has ended, that is, the required number of cycles of the buffer have been output, or an error occurred.
2: The waveform generation has stopped because `BWF_Stop` was called. The waveform operation can be resumed by calling `BWF_Resume`.
3: The waveform generation stopped because old data was about to be regenerated. This value of **status** is only possible when block update mode is enabled for the buffer being generated and **regenerationMode** was set to 1 in `BWF_BufLoad`.

bufferNumber indicates the buffer number that is currently being output if waveform generation is in progress or is the buffer number that was being output when waveform generation stopped. This will always return 1 for the NB-A2100.

cyclesOutput indicates the number of cycles of **bufferNumber** that have been output.

pointsOutput indicates the number of data buffer points that have been output. If **checkMode** is 0, **pointsOutput** will always be 0. If **checkMode** is 1, **pointsOutput** will return the number of the data buffer points that have been written to the D/A FIFO during the current cycle. The NB-A2100 has a 16-word (one word is 32 bits) FIFO on the board so the FIFO can contain up to 32 data points. Therefore, the **pointsOutput** indicates the index of the data point that will be output after 32 update intervals.

interval indicates the current update interval (that is, the amount of time to elapse between updates on the channel) being used for waveform generation. The actual update interval in seconds can be calculated as follows:

$$\frac{\textbf{interval}}{\text{timebase frequency}}$$

where the timebase frequency is defined by **timebase**, described as follows.

timebase indicates the frequency of the timer used to determine the update interval. On the NB-A2100, timebase has the following possible values:

1: 48 kHz.
2: 44.1 kHz.
3: 32 kHz.
4: 24 kHz.
5: 22.05 kHz.
6: 16 kHz.

BWF_Clear

Function

Stops the waveform generation in progress and deallocates the resources used for waveform generation.

Synopsis

C Syntax	`locus i32 BWF_Clear(u32 deviceNumber, u32 group, u32 channel);`
Pascal Syntax	`function BWF_Clear(deviceNumber : i32; group : i32; channel : i32) : i32;`
BASIC Syntax	`FN BWF_Clear(deviceNumber&, group&, channel&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group selects the analog output channel group number. **group** must be 0 for the NB-A2100.

channel is the analog output channel number(s) selected in `BWF_BufLoad`.

Range: 0: Channel 0.
1: Channel 1.
2: Both channels 0 and 1.

Call `BWF_Clear` before exiting to stop waveform generation in progress on **channel** and release any DMA channels used for waveform generation.

BWF_Rate

Function

Changes the current waveform generation rate at which the output channels are updated.

Synopsis

C Syntax	`locus i32 BWF_Rate(u32 deviceNumber, u32 group, u32 channel, u32 interval, u32 timebase, u32 mode);`
Pascal Syntax	`function BWF_Rate(deviceNumber : i32; group : i32; channel : i32; interval : i32; timebase : i32; mode : i32) : i32;`
BASIC Syntax	`FN BWF_Rate(deviceNumber&, group&, channel&, interval&, timebase&, mode&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is analog output channel group number. **group** must be 0 for NB-A2100 because the NB-A2100 does not allow configuration of channel groups.

channel is the analog output channel number(s) selected in `BWF_BufLoad`.

Range: 0: channel 0.
1: channel 1.
2: Both channels 0 and 1.

interval is the length of the update interval (that is, the amount of time to elapse between updates on the channel). The actual update interval in seconds can be calculated as follows:

interval / timebase frequency

where the timebase frequency is defined by **timebase**, described as follows.

On the NB-A2100, **interval** must be 1.

timebase is the frequency of the timer which determines the update interval. On the NB-A2100, timebase has the following possible values:

1: 48 kHz.
2: 44.1 kHz.
3: 32 kHz.
4: 24 kHz.
5: 22.05 kHz.
6: 16 kHz.

mode selects the mode to use for changing the waveform generation rate.

0: `BWF_Rate` immediately changes the current waveform generation rate.
1: `BWF_Rate` returns immediately, but synchronizes when the new waveform rate is changed. If waveform generation is in progress, the waveform generation rate is changed after the current waveform cycle is complete. If waveform generation is not in progress, the rate is changed when waveform generation is resumed on the current buffer or started on a new buffer whose rate was not specified in `BWF_BufLoad` (**interval** = 0).

`BWF_Rate` sets the default waveform generation rate for waveform buffers that have not specified a rate in `BWF_BufLoad`. If waveform generation has been started, `BWF_Rate` changes the current waveform generation rate of the buffer being output. However, if waveform generation is restarted through `BWF_Start`, the rate specified in `BWF_BufLoad` is used again.

BWF_Resume

Function

Resumes the waveform generation stopped by `BWF_Stop`.

Synopsis

C Syntax	`locus i32 BWF_Resume(u32 deviceNumber, u32 group, u32 channel);`
Pascal Syntax	`function BWF_Resume(deviceNumber : i32; group : i32; channel : i32) : i32;`
BASIC Syntax	`FN BWF_Resume(deviceNumber&, group&, channel&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is analog output channel group number. **group** must be 0 for the NB-A2100.

channel is the analog output channel number(s) selected in `BWF_BufLoad`.

Range: 0: Channel 0.
1: Channel 1.
2: Both channels 0 and 1.

BWF_Resume resumes waveform generation from the point where the generation was stopped by BWF_Stop. If the NB-A2100 was waiting for an external trigger when BWF_Stop was called, BWF_Resume resumes the waveform generation in the same trigger state. Otherwise, if the NB-A2100 was not waiting for a trigger when BWF_Stop is called, BWF_Resume provides a software trigger to resume waveform generation.

BWF_Start

Function

Starts waveform generation using the selected waveform buffer.

Synopsis

C Syntax	`locus i32 BWF_Start(u32 deviceNumber, u32 group, u32 channel, u32 *bufferNumbers, u32 startMode);`
Pascal Syntax	`function BWF_Start(deviceNumber : i32; group : i32; channel : i32; var bufferNumbers : i32; startMode : i32) : i32;`
BASIC Syntax	`FN WF_Start(deviceNumber&, channel&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is analog output channel group number. **group** must be 0 for NB-A2100 because the NB-A2100 does not allow configuration of channel groups.

channel is the analog output channel number(s) selected in BWF_BufLoad.

Range: 0: Channel 0.
1: Channel 1.
2: Both channels 0 and 1.

bufferNumber contains the buffer number for each channel in a group. Because the NB-A2100 does not allow configuration of channel groups, only the buffer number at index 0 of **bufferNumber** is used. Currently the buffer number in **bufferNumber** must be set to 1.

startMode selects the mode to use for starting the generation of data in the specified buffer number in **bufferNumber**. **startMode** must be set to 0 for the NB-A2100.

BWF_Stop

Function

Stops the waveform generation in progress.

Synopsis

C Syntax	`locus i32 BWF_Stop(u32 deviceNumber, u32 group, u32 channel, u32 stopMode);`
Pascal Syntax	`function BWF_Stop(deviceNumber : i32; group : i32; channel : i32; stopMode : i32) : i32;`
BASIC Syntax	`FN BWF_Stop(deviceNumber&, group&, channel&, stopMode&)`

Description

deviceNumber is the device number of the board you want NI-DAQ to use for this function. Please refer to the *Devices* section in Chapter 1, *Getting Started*, for more information.

group is analog output channel group number. **group** must be 0 for the NB-A2100.

channel is the analog output channel number(s) selected in `BWF_BufLoad`.

Range: 0: Channel 0.
1: Channel 1.
2: Both channels 0 and 1.

stopMode selects the mode for stopping waveform generation.

Range: 0: `BWF_Stop` stops waveform generation immediately.
1: `BWF_Stop` returns immediately. Waveform generation stops after the current cycle of the waveform is completed.

`BWF_Resume` resumes a waveform generation stopped by `BWF_Stop`.

Chapter 11
NI-DAQ for Macintosh Examples

This chapter describes the examples included in the NI-DAQ for Macintosh software. These examples show you how you can use various NI-DAQ for Macintosh functions in actual applications. Examples written in C are in the `C/C++ Examples` folder. The `Pascal Examples` folder includes examples written in Pascal. Refer to Chapter 1, *Getting Started*, for instructions on creating and running these example applications using the different language environments.

The example descriptions in this chapter apply to all languages. The corresponding source files can be found on disk with the example name followed by the appropriate language extension. Pascal source files use the `.pas` extension, C source files use the `.c` extension.

There are some example programs included on disk that are not described in this chapter. The comments in the source code of these programs should fully explain the operation and structure of the programs. In particular, there are examples describing the use of the SCXI functions that are included on disk, but are not described in this chapter.

NI-DAQ for Macintosh Examples

The following examples are included on the NI-DAQ for Macintosh diskettes. A list of boards that can be used with the example and the NI-DAQ function calls used follows each description:

- `OneShotScope(1ch)` — Uses NI-DAQ for Macintosh single-buffered Data Acquisition (`DAQ`) functions to perform single-channel data acquisition and graph the results.

  Boards supported: MIO boards, Lab and 1200 series, DAQCard-500, DAQCard-700

  NI-DAQ functions used: `DAQ_Start`, `DAQ_Check`, `DAQ_Scale`

- `OneShotScope(2ch)` — Uses NI-DAQ for Macintosh single-buffered Data Acquisition (`SCAN`) functions to perform multiple-channel scanned data acquisition and graph the results.

  Boards supported: MIO boards

  NI-DAQ functions used: `SCAN_Setup`, `SCAN_Start`, `SCAN_Check`, `SCAN_Demux`, `DAQ_Scale`, `DAQ_Clear`

- `Lab-OneShotScope(2ch)` — Uses NI-DAQ for Macintosh single-buffered Data Acquisition (`Lab_ISCAN`) functions to perform multiple-channel scanned data acquisition and graph the results.

  Boards supported: Lab and 1200 series, DAQCard-500, DAQCard-700

  NI-DAQ functions used: `Lab_ISCAN_Start`, `Lab_ISCAN_Check`, `SCAN_Demux`, `DAQ_Scale`, `DAQ_Clear`

- `Oscilloscope` — Uses NI-DAQ for Macintosh double-buffered Data Acquisition (`DAQ2`) functions to continuously acquire and graph the data from an A/D channel.

Boards supported: MIO boards and Lab and 1200 series

NI-DAQ functions used: `DAQ2Config`, `DAQ_Start`, `DAQ2TTap`, `DAQ_Scale`, `DAQ2Clear`

- `StreamToDisk(1ch)` — Uses NI-DAQ for Macintosh double-buffered Data Acquisition (`DAQ2`) functions to perform single-channel data acquisition and stream the data to disk as the data is acquired.

  Boards supported: MIO boards, Lab and 1200 series, DAQCard-500, DAQCard-700

  NI-DAQ functions used: `DAQ2Config`, `DAQ_Start`, `DAQ2TGet`, `DAQ2Clear`

- `StreamToDisk(4ch)` — Uses NI-DAQ for Macintosh double-buffered Data Acquisition (`DAQ2`) functions to perform multiple-channel data acquisition and stream the data to disk as the data is acquired. Individual files are created for each channel's data.

  Boards supported: MIO boards

  NI-DAQ functions used: `DAQ2Config`, `SCAN_Setup`, `SCAN_Start`, `DAQ2TGet`, `SCAN_Demux`, `DAQ2Clear`

- `AsyncFuncGenerator` — Uses NI-DAQ for Macintosh asynchronous Waveform Generation (`WF`) functions to produce a 400-Hz sine, square, or triangle waveform on an analog output channel.

  Boards supported: NB-MIO-16 and NB-AO-6; both boards require an NB-DMA-8-G or NB-DMA2800 in the system for waveform generation.

  NI-DAQ functions used: `WF_Setup`, `WF_Offset`, `WF_Load`, `WF_Start`, `WF_Stop`, `WF_Reset`

- `SyncFuncGenerator` — Uses NI-DAQ for Macintosh synchronous Waveform Generation (`WF_Grp`) functions to produce a 25-Hz sine waveform on analog output channel 0 and a 1.25-kHz square wave on analog output channel 1.

  Boards supported: NB-MIO-16X, E Series boards, Lab and 1200 series, and NB-AO-6; the NB-AO-6 requires an NB-DMA-8-G or NB-DMA2800 in the system for waveform generation.

  NI-DAQ functions used: `WF_Grp_Setup`, `WF_Load`, `WF_Grp_Start`, `WF_Grp_Stop`, `WF_Grp_Reset`

- `SampleAndGenerate` — Uses NI-DAQ for Macintosh asynchronous Waveform Generation (`WF`) functions together with double-buffered Data Acquisition (DAQ2) functions to continuously sample an input waveform and then generate the sample.

  Board supported: NB-MIO-16; this board requires an NB-DMA-8-G or NB-DMA2800 in the system for waveform generation.

  NI-DAQ functions used: `DAQ2Config`, `DAQ_Start`, `DAQ2TTap`, `DAQ2Clear`, `WF_Setup`, `WF_Offset`, `WF_Load`, `WF_Start`, `WF_Stop`, `WF_Reset`

- `PreTrig_Interval_Scan` Uses NI-DAQ for Macintosh (`SCAN_Int`) functions to acquire and graph data from an interval-scanning data acquisition using pretriggering. Interval scanning can be used to simulate *simultaneous sampling* of a group of channels.

  Boards supported: NB-MIO-16X and E Series devices

  NI-DAQ functions used: `DAQ_PreTrig`, `SCAN_Setup`, `SCAN_IntStart`, `SCAN_Check`, `SCAN_Demux`, `DAQ_Scale`, `DAQ_Clear`

- `Digital_Blk_Transfer` Uses NI-DAQ for Macintosh Digital Block (`DIG_Blk`) functions to transfer a buffer of digital data to and from two digital groups on a board using handshaking.

  Board supported: NB-DIO-32F

  NI-DAQ functions used: `DIG_Grp_Config`, `DIG_Blk_Start`, `DIG_Grp_Status`, `DIG_In_Group`, `DIG_Blk_Clear`

- `SqWaveGenerator` Uses NI-DAQ for Macintosh Interval Counter (`ICTR`) functions to produce a 50 percent duty-cycle square wave of any frequency between 32 Hz and 1 MHz.

  Boards supported: Lab and 1200 series, DAQCard-500, DAQCard-700

  NI-DAQ functions used: `ICTR_Setup`

- `MultiChannelDVM` Uses NI-DAQ for Macintosh Multiple-channel Analog Input (`MAI`) functions to collect four simultaneous readings from four channels of the NB-A2000 or two simultaneous readings from two channels of the NB-A2100 or NB-A2150.

  Boards supported: NB-A2000, NB-A2100, and NB-A2150

  NI-DAQ functions used: `MAI_Coupling`, `MAI_Setup`, `MAI_Read`, `MAI_Scale`, `MAI_Clear`

- `GetFramesAndGraph` Uses NI-DAQ for Macintosh double-buffered Multiple-channel Data Acquisition (`MDAQ`) functions to perform a high speed data acquisition and graph the results.

  Boards supported: NB-A2000, NB-A2100, and NB-A2150

  NI-DAQ functions used: `Board_ID`, `MAI_Coupling`, `MAI_Setup`, `MDAQ_Trig_Config`, `MDAQ_Setup`, `MDAQ_ScanRate`, `MDAQ_Start`, `MDAQ_Check`, `MDAQ_Get`, `MAI_Scale`, `MDAQ_Clear`

- `MDAQ_OpExample` Uses the example function `MDAQ_Op` to acquire one frame of data from the NB-A2000, the NB-A2100, or the NB-A2150 and then graphs the results.

  Boards supported: NB-A2000, NB-A2100, or NB-A2150

  NI-DAQ functions used: `MAI_Setup`, `MDAQ_ScanRate`, `MDAQ_Setup`, `MDAQ_Start`, `MDAQ_Check`, `MDAQ_Get`, `MDAQ_Clear`, `MAI_Scale`

- `MDAQ_Op` Designed to be used like an NI-DAQ for Macintosh function. Calls a series of double-buffered Multiple-Channel Data Acquisition (`MDAQ`) functions to acquire a single frame of data from the selected channels.

Boards supported: NB-A2000, NB-A2100, or NB-A2150

NI-DAQ functions used: `MAI_Setup`, `MDAQ_ScanRate`, `MDAQ_Setup`, `MDAQ_Start`, `MDAQ_Check`, `MDAQ_Get`, `MDAQ_Clear`

- `StreamToDisk(MDAQ)` — Uses NI-DAQ for Macintosh double-buffered Multiple-Channel Data Acquisition (`MDAQ`) functions to perform multiple-channel data acquisition and to stream the data to disk as the data is acquired.

  Boards supported: NB-A2000, NB-A2100, or NB-A2150

  NI-DAQ functions used: `Board_ID`, `MAI_Coupling`, `MAI_Setup`, `MDAQ_Trig_Config`, `MDAQ_Setup`, `MDAQ_ScanRate`, `MDAQ_Start`, `MDAQ_Get`, `MDAQ_Clear`

- `StreamFromDisk` — Uses NI-DAQ for Macintosh Buffered Waveform Generation (`BWF`) functions to generate the data from a disk file at the specified analog output channels of the NB-A2100. All or some of the data in the disk file can be generated.

  Board supported: NB-A2100

  NI-DAQ functions used: `BWF_BufLoad`, `BWF_Start`, `BWF_BlkLoad`, `BWF_Check`, `BWF_Clear`

- `PeriodMeasurement` — Uses NI-DAQ for Macintosh Counter (`CTR`) functions to measure period or pulse width.

  Boards supported: NB-MIO-16, NB-MIO-16X, or NB-TIO-10

  NI-DAQ functions used: `Board_ID`, `CTR_Config`, `CTR_Period`, `CTR_Square`, `CTR_EvRead`, `CTR_Reset`

Appendix A
NI-DAQ for Macintosh Function and Board Compatibility

This appendix contains a table of the National Instruments boards that work with NI-DAQ for Macintosh and the functions that work with each board, as well as a table of National Instruments SCXI chassis and modules that work with NI-DAQ for Macintosh and the functions that work with each element of SCXI hardware.

A √ signifies that the function operation is supported on the specified board. Notice that DMA support is automatically used for data acquisition, waveform generation, and buffered digital I/O on the NB-MIO-16, NB-MIO-16X, NB-AO-6, NB-DIO-32F, NB-A2100, and NB-A2000 if an NB-DMA-8-G or NB-DMA2800 is detected in the system.

Table A-1. NI-DAQ for Macintosh Function and Device Support

Functions	Device																				
	DAQCard-500	DAQCard-700	DAQCard-1200	DAQCard-AO-2DC	DAQCard-DIO-24	Lab-LC	Lab-NB	NB-A2000	NB-A2100	NB-A2150	NB-AO-6	NB-DIO-24	NB-DIO-32F	NB-DIO-96	NB-DMA2800	NB-DMA-8-G	NB-MIO-16	NB-MIO-16X	NB-PRL	NB-TIO-10	PCI-1200	PCI-MIO-16XE-50	PCI-DIO-96
A2000_Calibrate								√															
A2000_Config								√															
A2100_Calibrate									√	√													
A2100_Configure									√														
A2150_Config										√													
AI_Check	√	√	√			√	√										√	√			√	√	
AI_Clear	√	√	√			√	√										√	√			√	√	
AI_Configure	√	√	√			√	√										√	√			√	√	
AI_Mux_Config																	√	√				√	
AI_Read	√	√	√			√	√										√	√			√	√	
AI_Setup	√	√	√			√	√										√	√			√	√	
AI_VScale	√	√	√			√	√										√	√			√	√	
AO_Change_Parameter				√																			
AO_Setup			√	√		√	√				√						√	√			√	√	
AO_Update			√			√	√				√							√			√	√	

Table A-1. NI-DAQ for Macintosh Function and Device Support (Continued)

Functions	Device																					
	DAQCard-500	DAQCard-700	DAQCard-1200	DAQCard-AO-2DC	DAQCard-DIO-24	Lab-LC	Lab-NB	NB-A2000	NB-A2100	NB-A2150	NB-AO-6	NB-DIO-24	NB-DIO-32F	NB-DIO-96	NB-DMA2800	NB-DMA-8-G	NB-MIO-16	NB-MIO-16X	NB-PRL	NB-TIO-10	PCI-1200	PCI-MIO-16XE-50	PCI-DIO-96
AO_VScale			√	√		√	√		√		√						√	√			√	√	
AO_Write			√	√		√	√		√		√						√	√			√	√	
Board_ID	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
Board_Reset	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
BWF_BlkLoad									√														
BWF_BufLoad									√														
BWF_Check									√														
BWF_Clear									√														
BWF_Rate									√														
BWF_Resume									√														
BWF_Start									√														
BWF_Stop									√														
Calibrate_1200																					√		
Calibrate_E_Series																						√	
CTR_Config								√							√	√	√	√		√			
CTR_EvCount								√							√	√	√	√		√			
CTR_EvRead								√							√	√	√	√		√			
CTR_FOUT_Config															√	√	√	√		√			
CTR_Period								√							√	√	√	√		√			
CTR_Pulse								√							√	√	√	√		√			
CTR_Reset								√							√	√	√	√		√			
CTR_Restart								√							√	√	√	√		√			
CTR_Square								√							√	√	√	√		√			
CTR_State								√							√	√	√	√		√			
CTR_Stop								√							√	√	√	√		√			

Table A-1. NI-DAQ for Macintosh Function and Device Support (Continued)

Functions	Device																					
	DAQCard-500	DAQCard-700	DAQCard-1200	DAQCard-AO-2DC	DAQCard-DIO-24	Lab-LC	Lab-NB	NB-A2000	NB-A2100	NB-A2150	NB-AO-6	NB-DIO-24	NB-DIO-32F	NB-DIO-96	NB-DMA2800	NB-DMA-8-G	NB-MIO-16	NB-MIO-16X	NB-PRL	NB-TIO-10	PCI-1200	PCI-MIO-16XE-50	PCI-DIO-96
DAQ2Clear	√	√	√			√	√										√	√			√	√	
DAQ2Config	√	√	√			√	√										√	√			√	√	
DAQ2Get																	√	√				√	
DAQ2MemConfig	√	√	√			√	√										√	√			√		
DAQ2Tap																	√	√			√	√	
DAQ2TGet	√	√	√			√	√										√	√			√	√	
DAQ2TTap	√	√	√			√	√										√	√			√	√	
DAQ_Check	√	√	√			√	√										√	√			√	√	
DAQ_Clear	√	√	√			√	√										√	√			√	√	
DAQ_Config	√	√	√			√	√										√	√			√	√	
DAQ_PreTrig			√			√	√											√			√	√	
DAQ_Start	√	√	√			√	√										√	√			√	√	
DAQ_Trigger	√	√	√			√	√										√	√			√	√	
DAQ_VScale																					√	√	
DIG_Blk_Check			√		√	√	√					√	√	√					√		√		√
DIG_Blk_Clear			√		√	√	√					√	√	√					√		√		√
DIG_Blk_Start			√		√	√	√					√	√	√					√		√		√
DIG_Grp_Config													√										
DIG_Grp_Mode													√										
DIG_Grp_Status													√										
DIG_In_Group													√										
DIG_In_Line	√	√	√	√	√	√	√					√	√	√			√	√	√	√	√	√	√
DIG_In_Port	√	√	√	√	√	√	√					√	√	√			√	√	√	√	√	√	√
DIG_Line_Config																				√		√	
DIG_Out_Group													√										

Table A-1. NI-DAQ for Macintosh Function and Device Support (Continued)

Functions	Device																					
	DAQCard-500	DAQCard-700	DAQCard-1200	DAQCard-AO-2DC	DAQCard-DIO-24	Lab-LC	Lab-NB	NB-A2000	NB-A2100	NB-A2150	NB-AO-6	NB-DIO-24	NB-DIO-32F	NB-DIO-96	NB-DMA2800	NB-DMA-8-G	NB-MIO-16	NB-MIO-16X	NB-PRL	NB-TIO-10	PCI-1200	PCI-MIO-16XE-50	PCI-DIO-96
DIG_Out_Line	√	√	√	√	√	√	√					√	√	√			√	√	√	√	√	√	√
DIG_Out_Port	√	√	√	√	√	√	√					√	√	√			√	√	√	√	√	√	√
DIG_Prt_Config	√	√	√	√	√	√	√					√	√	√			√	√	√	√	√	√	√
DIG_Prt_Status			√		√	√	√					√		√					√		√		√
DIG_Scan_Setup			√		√	√	√					√		√					√		√		√
Get_DAQ_Device_Info	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
GPCTR_Change_Parameter																						√	
GPCTR_Config_Buffer																						√	
GPCTR_Control																						√	
GPCTR_Set_Application																						√	
GPCTR_Watch																						√	
ICTR_Read	√	√	√			√	√														√		
ICTR_Reset	√	√	√			√	√														√		
ICTR_Setup	√	√	√			√	√														√		
Lab_ISCAN_Check	√	√	√			√	√														√		
Lab_ISCAN_Start	√	√	√			√	√														√		
MAI_Arm								√	√	√													
MAI_Clear								√															
MAI_Coupling								√	√	√													
MAI_Read								√	√	√													
MAI_Scale								√	√	√													
MAI_Setup								√	√	√													
Master_Slave_Config								√		√													
MDAQ_Check								√	√	√													
MDAQ_Clear								√	√	√													

Table A-1. NI-DAQ for Macintosh Function and Device Support (Continued)

Functions	Device																					
	DAQCard-500	DAQCard-700	DAQCard-1200	DAQCard-AO-2DC	DAQCard-DIO-24	Lab-LC	Lab-NB	NB-A2000	NB-A2100	NB-A2150	NB-AO-6	NB-DIO-24	NB-DIO-32F	NB-DIO-96	NB-DMA2800	NB-DMA-8-G	NB-MIO-16	NB-MIO-16X	NB-PRL	NB-TIO-10	PCI-1200	PCI-MIO-16XE-50	PCI-DIO-96
MDAQ_Get								√	√	√													
MDAQ_ScanRate								√	√	√													
MDAQ_Setup								√	√	√													
MDAQ_Start								√	√	√													
MDAQ_Stop								√	√	√													
MDAQ_Trig_Config								√	√	√													
MDAQ_Trig_Delay								√	√	√													
MIO_16X_Config																		√					
MIO_Config			√																		√		
RTSI_Clear								√	√	√	√		√		√	√	√	√		√			
RTSI_Conn								√	√	√	√		√		√	√	√	√		√			
RTSI_DisConn								√	√	√	√		√		√	√	√	√		√			
SC_2040_Config																						√	
SCAN_Check																	√	√				√	
SCAN_Demux	√	√	√			√	√										√	√			√	√	
SCAN_IntStart																		√				√	
SCAN_Setup																	√	√				√	
SCAN_Start																	√	√				√	
Select_Signal																						√	
Set_DAQ_Device_Info	√	√	√			√	√		√	√	√		√				√	√			√	√	
WF_Check			√			√	√				√						√	√			√	√	
WF_DBLoad											√						√	√				√	
WF_Grp_Reset			√			√	√				√							√			√	√	
WF_Grp_Setup			√			√	√				√							√			√	√	
WF_Grp_Start			√			√	√				√							√			√	√	

Table A-1. NI-DAQ for Macintosh Function and Device Support (Continued)

Functions	Device																					
	DAQCard-500	DAQCard-700	DAQCard-1200	DAQCard-AO-2DC	DAQCard-DIO-24	Lab-LC	Lab-NB	NB-A2000	NB-A2100	NB-A2150	NB-AO-6	NB-DIO-24	NB-DIO-32F	NB-DIO-96	NB-DMA2800	NB-DMA-8-G	NB-MIO-16	NB-MIO-16X	NB-PRL	NB-TIO-10	PCI-1200	PCI-MIO-16XE-50	PCI-DIO-96
WF_Grp_Stop			√			√	√				√							√			√		
WF_Load			√			√	√				√						√	√			√	√	
WF_Offset																	√						
WF_Reset											√						√						
WF_Setup											√						√						
WF_Start											√						√						
WF_Stop											√						√						

Table A-2. SCXI Function and Hardware Support

Functions	Module												Device											
	SCXI-1100	SCXI-1102	SCXI-1120	SCXI-1121	SCXI-1122	SCXI-1124	SCXI-1140	SCXI-1141	SCXI-1160	SCXI-1161	SCXI-1162/62HV	SCXI-1163/63R	DAQCard-700	DAQCard-1200	DAQCard-DIO-24	Lab-NB	Lab-LC	NB-DIO-24	NB-DIO-32F	NB-DIO-96	NB-MIO-16	NB-MIO-16X	PCI-1200	PCI-MIO-16XE-50
SCXI_AO_Write						√																		
SCXI_Cal_Constants	√	√	√	√	√	√	√	√					√	√		√	√				√	√	√	√
SCXI_Calibrate_Setup	√			√	√			√																
SCXI_Change_Chan	√	√	√	√	√		√	√																
SCXI_Configure_Filter					√			√																
SCXI_Get_Chassis_Info																								
SCXI_Get_Module_Info	√	√	√	√	√	√	√	√	√	√	√	√												
SCXI_Get_State									√	√	√	√												
SCXI_Get_Status		√			√	√			√															
SCXI_Load_Config																								
SCXI_MuxCtr_Setup	√	√	√	√	√		√	√						√							√	√	√	√
SCXI_Reset	√	√	√	√	√	√	√	√	√	√	√	√												
SCXI_Scale	√	√	√	√	√		√	√					√	√		√	√				√	√	√	√
SCXI_Scan_Setup	√	√	√	√	√		√	√						√							√	√	√	√
SCXI_Set_Config	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
SCXI_Set_Gain	√	√			√			√																
SCXI_Set_Input_Mode					√																			
SCXI_Set_State									√	√		√												
SCXI_Single_Chan_Setup	√	√	√	√	√		√	√					√	√		√	√				√	√	√	√
SCXI_Track_Hold_Control							√						√	√		√	√				√	√	√	
SCXI_Track_Hold_Setup							√						√	√		√	√				√	√	√	√

Appendix B
Error Codes

This appendix lists the error codes NI-DAQ for Macintosh returns, including the error number, name, and description. Each function returns an error code that indicates whether the function was performed successfully.

All of the error code descriptions are also listed in the NI-DAQ control panel.

Table B-1. NI-DAQ Error Codes

Error Code	Name	Description
-10001	**syntaxError**	An error was detected in the input string; the arrangement or ordering of the characters in the string is not consistent with the expected ordering.
-10002	**semanticsError**	An error was detected in the input string; the syntax of the string is correct, but certain values specified in the string are inconsistent with other values specified in the string.
-10003	**invalidValueError**	The value of a numeric parameter is invalid.
-10004	**valueConflictError**	The value of a numeric parameter is inconsistent with another one, and therefore the combination is invalid.
-10005	**DSPbadDeviceError**	The device is invalid.
-10006	**badLineError**	The line is invalid.
-10007	**badChanError**	A channel is out of range for the board type or input configuration; or the combination of channels is not allowed; or the scan order must be reversed (0 last).
-10008	**badGroupError**	The group is invalid.
-10009	**badCounterError**	The counter is invalid.
-10010	**badCountError**	The count is too small or too large for the specified counter; or the given I/O transfer count is not appropriate for the current buffer or channel configuration.
-10011	**badIntervalError**	The analog input scan rate is too fast for the number of channels and the channel clock rate; or the given clock rate is not supported by the associated counter channel or I/O channel.
-10012	**badRangeError**	The analog input or analog output voltage range is invalid for the specified channel.
-10013	**badErrorCodeError**	The driver returned an unrecognized or unlisted error code.
-10014	**groupTooLargeError**	The group size is too large for the board.

Table B-1. NI-DAQ Error Codes (Continued)

Error Code	Name	Description
-10015	**badTimeLimitError**	The time limit is invalid.
-10016	**badReadCountError**	The read count is invalid.
-10017	**badReadModeError**	The read mode is invalid.
-10018	**badReadOffsetError**	The offset is unreachable.
-10019	**badClkFrequencyError**	The frequency is invalid.
-10020	**badTimebaseError**	The timebase is invalid.
-10021	**badLimitsError**	The limits are beyond the range of the board.
-10022	**badWriteCountError**	Your data array contains an incomplete update, or you are trying to write past the end of the internal buffer, or your output operation is continuous and the length of your array is not a multiple of one half of the internal buffer size.
-10023	**badWriteModeError**	The write mode is out of range or is disallowed.
-10024	**badWriteOffsetError**	Adding the write offset to the write mark places the write mark outside the internal buffer.
-10025	**limitsOutOfRange**	The requested input limits exceed the board's capability or configuration.
-10026	**badBufferSpec**	The requested number of buffers or the buffer size is not allowed; e.g., Lab-PC buffer limit is 64K samples, or the board does not support multiple buffers.
-10027	**badDAQEventError**	For DAQEvents 0 and 1 general value A must be greater than 0 and less than the internal buffer size.
-10028	**badFilterCutoffError**	The cutoff frequency specified is not valid for this device.
-10080	**badGainError**	The gain is invalid.
-10081	**badPretrigCountError**	The pretrigger sample count is invalid.
-10082	**badPosttrigCountError**	The posttrigger sample count is invalid.
-10083	**badTrigModeError**	The trigger mode is invalid.
-10084	**badTrigCountError**	The trigger count is invalid.
-10085	**badTrigRangeError**	The trigger range or trigger hysteresis window is invalid.
-10086	**badExtRefError**	The external reference is invalid.
-10087	**badTrigTypeError**	The trigger type is invalid.
-10088	**badTrigLevelError**	The trigger level is invalid.
-10089	**badTotalCountError**	The total count is inconsistent with the buffer size and pretrigger scan count or with the board type.

Table B-1. NI-DAQ Error Codes (Continued)

Error Code	Name	Description
-10090	**badRPGError**	The individual range, polarity, and gain settings are valid but the combination is not allowed.
-10091	**badIterationsError**	You have attempted to use an invalid setting for the iterations parameter. The iterations value must be 0 or greater. Your device may be limited to only two values, 0 and 1.
-10100	**badPortWidthError**	The requested digital port width is not a multiple of the hardware port width.
-10200	**EEPROMreadError**	Unable to read data from EEPROM.
-10201	**EEPROMwriteError**	Unable to write data to EEPROM.
-10240	**noDriverError**	The driver interface could not locate or open the driver.
-10241	**oldDriverError**	One of the driver files or the configuration utility is out of date.
-10242	**functionNotFoundError**	The specified function is not located in the driver.
-10243	**DSPconfigFileError**	The driver could not locate or open the configuration file, or the format of the configuration file is not compatible with the currently installed driver.
-10244	**deviceInitError**	The driver encountered a hardware-initialization error while attempting to configure the specified device.
-10245	**osInitError**	The driver encountered an operating-system error while attempting to perform an operation, or the operating system does not support an operation performed by the driver.
-10246	**communicationsError**	The driver is unable to communicate with the specified external device.
-10247	**DSPcmosConfigError**	The CMOS configuration-memory for the device is empty or invalid, or the configuration specified does not agree with the current configuration of the device, or the EISA system configuration is invalid.
-10248	**dupAddressError**	The base addresses for two or more devices are the same; consequently, the driver is unable to access the specified device.
-10249	**intConfigError**	The interrupt configuration is incorrect given the capabilities of the computer or device.
-10250	**dupIntError**	The interrupt levels for two or more devices are the same.
-10251	**dmaConfigError**	The DMA configuration is incorrect given the capabilities of the computer/DMA controller or device.
-10252	**dupDMAError**	The DMA channels for two or more devices are the same.

Table B-1. NI-DAQ Error Codes (Continued)

Error Code	Name	Description
-10253	**jumperlessBoardError**	Unable to find one or more jumperless boards you have configured using WDAQCONF.
-10254	**DAQCardConfError**	Cannot configure the DAQCard because 1) the correct version of the card and socket services software is not installed; 2) the card in the PCMCIA socket is not a DAQCard; or 3) the base address and/or interrupt level requested are not available according to the card and socket services resource manager.
-10340	**noConnectError**	No RTSI signal/line is connected, or the specified signal and the specified line are not connected.
-10341	**badConnectError**	The RTSI signal/line cannot be connected as specified.
-10342	**multConnectError**	The specified RTSI signal is already being driven by a RTSI line, or the specified RTSI line is already being driven by a RTSI signal.
-10343	**SCXIConfigError**	The specified SCXI configuration parameters are invalid, or the function cannot be executed with the current SCXI configuration.
-10360	**DSPInitError**	The DSP driver was unable to load the kernel for its operating system.
-10370	**badScanListError**	The scan list is invalid; for example, you are mixing AMUX-64T channels and onboard channels or are scanning SCXI channels out of order.
-10400	**userOwnedRsrcError**	The specified resource is owned by the user and cannot be accessed or modified by the driver.
-10401	**DSPunknownDeviceError**	The specified device is not a National Instruments product, or the driver does not support the device (e.g., the driver was released before the device was supported).
-10402	**deviceNotFoundError**	No device is located in the specified slot or at the specified address.
-10403	**DSPdeviceSupportError**	The specified device does not support the requested action (the driver recognizes the device, but the action is inappropriate for the device).
-10404	**noLineAvailError**	No line is available.
-10405	**noChanAvailError**	No channel is available.
-10406	**noGroupAvailError**	No group is available.
-10407	**lineBusyError**	The specified line is in use.
-10408	**chanBusyError**	The specified channel is in use.

Table B-1. NI-DAQ Error Codes (Continued)

Error Code	Name	Description
-10409	**groupBusyError**	The specified group is in use.
-10410	**relatedLCGBusyError**	A related line, channel, or group is in use; if the driver configures the specified line, channel, or group, the configuration, data, or handshaking lines for the related line, channel, or group will be disturbed.
-10411	**counterBusyError**	The specified counter is in use.
-10412	**noGroupAssignError**	No group is assigned, or the specified line or channel cannot be assigned to a group.
-10413	**groupAssignError**	A group is already assigned, or the specified line or channel is already assigned to a group.
-10414	**reservedPinError**	The selected signal requires a pin that is reserved and configured only by NI-DAQ.
-10415	**externalMuxSupporError**	This function does not support this device when an external multiplexer (such as AMUX-64T or SCXI) is connected to it.
-10440	**sysOwnedRsrcError**	The specified resource is owned by the driver and cannot be accessed or modified by the user.
-10441	**memConfigError**	No memory is configured to support the current data-transfer mode, or the configured memory does not support the current data-transfer mode.
-10442	**memDisabledError**	The specified memory is disabled or is unavailable given the current addressing mode.
-10443	**memAlignmentError**	The transfer buffer is not aligned properly for the current data-transfer mode.
-10444	**DSPmemFullError**	No more system memory is available on the heap, or no more memory is available on the device, or insufficient disk space is available.
-10445	**memLockError**	The transfer buffer cannot be locked into physical memory.
-10446	**memPageError**	The transfer buffer contains a page break; system resources may require reprogramming when the page break is encountered.
-10447	**memPageLockError**	The operating environment is unable to grant a page lock.
-10448	**stackMemError**	The driver is unable to continue parsing a string input due to stack limitations.
-10449	**cacheMemError**	A cache-related error occurred, or caching is not supported in the current mode.
-10450	**physicalMemError**	A hardware error occurred in physical memory, or no memory is located at the specified address.

Table B-1. NI-DAQ Error Codes (Continued)

Error Code	Name	Description
-10451	**virtualMemError**	The driver is unable to make the transfer buffer contiguous in virtual memory and therefore cannot lock it into physical memory; thus, the buffer cannot be used for DMA transfers.
-10452	**noIntAvailError**	No interrupt level is available for use.
-10453	**intInUseError**	The specified interrupt level is already in use by another device.
-10454	**noDMACError**	No DMA controller is available in the system.
-10455	**noDMAAvailError**	No DMA channel is available for use.
-10456	**DMAInUseError**	The specified DMA channel is already in use by another device.
-10457	**badDMAGroupError**	DMA cannot be configured for the specified group because it is too small, too large, or misaligned.
-10459	**DSPDLLInterfaceError**	The DLL could not be called due to an interface error.
-10460	**interfaceInteractionError**	You have mixed VIs from the DAQ library and the _DAQ compatibility library (LabVIEW 2.2 style VIs).
-10560	**invalidDSPhandleError**	The DSP handle input is not valid .
-10600	**noSetupError**	No setup operation has been performed for the specified resources.
-10601	**multSetupError**	The specified resources have already been configured by a setup operation.
-10602	**noWriteError**	No output data has been written into the transfer buffer.
-10603	**groupWriteError**	The output data associated with a group must be for a single channel or must be for consecutive channels.
-10604	**activeWriteError**	Once data generation has started, only the transfer buffers originally written to may be updated.
-10605	**endWriteError**	No data was written to the transfer buffer because the final data block has already been loaded.
-10606	**notArmedError**	The specified resource is not armed.
-10607	**armedError**	The specified resource is already armed.
-10608	**noTransferInProgError**	No transfer is in progress for the specified resource.
-10609	**transferInProgError**	A transfer is already in progress for the specified resource.
-10610	**transferPauseError**	A single output channel in a group may not be paused if the output data for the group is interleaved.
-10611	**badDirOnSomeLinesError**	Some of the lines in the specified channel are not configured for the transfer direction specified.

Table B-1. NI-DAQ Error Codes (Continued)

Error Code	Name	Description
-10612	**badLineDirError**	The specified line does not support the specified transfer direction.
-10613	**badChanDirError**	The specified channel does not support the specified transfer direction.
-10614	**badGroupDirError**	The specified group does not support the specified transfer direction.
-10615	**masterClkError**	The clock configuration for the clock master is invalid.
-10616	**slaveClkError**	The clock configuration for the clock slave is invalid.
-10617	**noClkSrcError**	No source signal has been assigned to the clock resource.
-10618	**badClkSrcError**	The specified source signal cannot be assigned to the clock resource.
-10619	**multClkSrcError**	A source signal has already been assigned to the clock resource.
-10620	**noTrigError**	No trigger signal has been assigned to the trigger resource.
-10621	**badTrigError**	The specified trigger signal cannot be assigned to the trigger resource.
-10622	**preTrigError**	The pretrigger mode is not supported or is not available in the current configuration, or no pretrigger source has been assigned.
-10623	**postTrigError**	No posttrigger source has been assigned.
-10624	**delayTrigError**	The delayed trigger mode is not supported or is not available in the current configuration, or no delay source has been assigned.
-10625	**masterTrigError**	The trigger configuration for the trigger master is invalid.
-10626	**slaveTrigError**	The trigger configuration for the trigger slave is invalid.
-10627	**noTrigDrvError**	No signal has been assigned to the trigger resource.
-10628	**multTrigDrvError**	A signal has already been assigned to the trigger resource.
-10629	**invalidOpModeError**	The specified operating mode is invalid, or the resources have not been configured for the specified operating mode.
-10630	**invalidReadError**	An attempt was made to read 0 bytes from the transfer buffer, or an attempt was made to read past the end of the transfer buffer.
-10631	**noInfiniteModeError**	Continuous input or output transfers are not allowed in the current operating mode.

Table B-1. NI-DAQ Error Codes (Continued)

Error Code	Name	Description
-10632	**someInputsIgnoredError**	Certain inputs were ignored because they are not relevant in the current operating mode.
-10633	**invalidRegenModeError**	The specified analog output regeneration mode is not allowed for this board.
-10680	**badChanGainError**	All channels of this board must have the same gain.
-10681	**badChanRangeError**	All channels of this board must have the same range.
-10682	**badChanPolarityError**	All channels of this board must be the same polarity.
-10683	**badChanCouplingError**	All channels of this board must have the same coupling.
-10684	**badChanInputModeError**	All channels of this board must have the same input mode.
-10685	**clkExceedsBrdsMaxConvRate**	The clock rate exceeds the board's recommended maximum rate.
-10686	**scanListInvalidError**	A configuration change has invalidated the scan list.
-10687	**bufferInvalidError**	A configuration change has invalidated the allocated buffer.
-10688	**noTrigEnabledError**	The number of total scans and pretrigger scans implies that a triggered start is intended, but triggering is not enabled.
-10689	**digitalTrigBError**	Digital trigger B is illegal for the number of total scans and pretrigger scans specified.
-10690	**digitalTrigAandBError**	This board does not allow digital triggers A and B to be enabled at the same time.
-10691	**extConvRestrictionError**	This board does not allow an external sample clock with an external scan clock, start trigger, or stop trigger.
-10692	**chanClockDisabledError**	The acquisition cannot be started because the channel clock is disabled.
-10693	**extScanClockError**	You cannot use an external scan clock when doing a single scan of a single channel.
-10694	**unsafeSamplingFreqError**	The sample frequency exceeds the safe maximum rate for the hardware, gains, and filters used.
-10695	**DMAnotAllowedError**	You have set up an operation that requires the use of interrupts.
-10696	**multiRateModeError**	Multi-rate scanning cannot be used with the AMUX-64, SCXI, or pretriggered acquisitions.
-10697	**rateNotSupportedError**	Unable to convert your timebase/interval pair to match the actual hardware capabilities of this board.
-10698	**timebaseConflictError**	You cannot use this combination of scan and sample clock timebases for this board.

Table B-1. NI-DAQ Error Codes (Continued)

Error Code	Name	Description
-10699	**polarityConflictError**	You cannot use this combination of scan and sample clock source polarities for this operation and board.
-10700	**signalConflictError**	You cannot use this combination of scan and convert clock signal sources for this operation and board.
-10740	**SCXITrackHoldError**	A signal has already been assigned to the SCXI track-and-hold trigger line, or a control call was inappropriate because the specified module is not configured for one-channel operation.
-10780	**sc2040InputModeError**	When you have an SC2040 attached to your device, all analog input channels must be configured for differential input mode.
-10781	**outputTypeMustBeVoltageError**	The polarity of the output channel cannot be bipolar when outputting currents.
-10800	**timeOutError**	The operation could not complete within the time limit.
-10801	**calibrationError**	An error occurred during the calibration process.
-10802	**dataNotAvailError**	The requested amount of data has not yet been acquired.
-10803	**transferStoppedError**	The transfer has been stopped to prevent regeneration of output data.
-10804	**earlyStopError**	The transfer stopped prior to reaching the end of the transfer buffer.
-10805	**overRunError**	The clock source for the input task is faster than the maximum clock rate the device supports.
-10806	**noTrigFoundError**	No trigger value was found in the input transfer buffer.
-10807	**earlyTrigError**	The trigger occurred before sufficient pretrigger data was acquired.
-10808	**LPTCommunicationError**	An error occurred in the parallel port communication with the DAQ device.
-10809	**gateSignalError**	Attempted to start a pulse width measurement with the pulse in the phase to be measured (e.g., high phase for high-level gating).
-10840	**softwareError**	An unexpected error occurred inside the driver when performing the given operation.
-10841	**firmwareError**	The firmware does not support the specified operation, or the firmware operation could not complete due to a data-integrity problem.

Table B-1. NI-DAQ Error Codes (Continued)

Error Code	Name	Description
-10842	**hardwareError**	The hardware is not responding to the specified operation, or the response from the hardware is not consistent with the functionality of the hardware.
-10843	**underFlowError**	Because of system limitations, the driver could not write data to the device fast enough to keep up with the device throughput.
-10844	**underWriteError**	New data was not written to the output transfer buffer before the driver attempted to transfer the data to the device.
-10845	**overFlowError**	Because of system limitations, the driver could not read data from the device fast enough to keep up with the device throughput; the onboard device memory reported an overflow error.
-10846	**overWriteError**	The driver wrote new data into the input transfer buffer before the previously acquired data was read.
-10847	**dmaChainingError**	New buffer information was not available at the time of the DMA chaining interrupt; DMA transfers will terminate at the end of the currently active transfer buffer.
-10848	**noDMACountAvailError**	The driver could not obtain a valid reading from the transfer-count register in the DMA controller.
-10849	**openFileError**	The configuration file or DSP kernel file could not be opened.
-10850	**closeFileError**	Unable to close a file.
-10851	**fileSeekError**	Unable to seek within a file.
-10852	**readFileError**	Unable to read from a file.
-10853	**writeFileError**	Unable to write to a file.
-10854	**miscFileError**	An error occurred accessing a file.
-10855	**osUnsupportedError**	NI-DAQ does not support the current operation on this version of the operating system.
-10856	**osError**	An unexpected error occurred from the operating system while performing the given operation.
-10880	**updateRateChangeError**	A change to the update rate is not possible at this time because 1) when waveform generation is in progress, you cannot change the interval timebase or 2) when you make several changes in a row, you must give each change enough time to take effect before requesting further changes.
-10881	**partialTransferCompleteError**	You cannot do another transfer after a successful partial transfer.

Table B-1. NI-DAQ Error Codes (Continued)

Error Code	Name	Description
-10920	**gpctrDataLossError**	One or more data points may have been lost during buffered GPCTR operations due to speed limitations of your system.

Additional error codes can be returned to signify Macintosh system error codes. These errors may be due to uninstalled or corrupted drivers.

Appendix C
Using an External Multiplexer

This appendix contains information on using the AMUX-64T.

An external multiplexer board (AMUX-64T) can be used to expand the number of analog input signals that can be measured with an MIO board. The AMUX-64T has 16 separate four-to-one analog multiplexer circuits. One AMUX-64T board can multiplex up to 64 analog input signals. Four AMUX-64T boards can be cascaded to permit up to 256 single-ended (128 differential) signals to be multiplexed by one MIO board.

The following table shows the number of channels available on an MIO board with an external multiplexer.

Table C-1. Analog Input Channel Range

Number of External Multiplexer (AMUX-64T) Boards	Channel Range, Single-Ended	Channel Range, Differential
0	0 – 15	0 – 7
1	0 – 63	0 – 31
2	0 – 127	0 – 31, 64 – 95
4	0 – 255	0 – 31, 64 – 95, 128 – 159, 192 – 223

`AI_Mux_Config` configures the number of multiplexer boards connected to the MIO board. Input channels are then referenced in subsequent data acquisition calls with respect to the external AMUX-64T analog input channel numbers rather than the MIO board onboard channel numbers. For example, with one external board, **channel** can have a value of 0 through 63 (single-ended), or 0 through 31 (differential). With two or four AMUX-64T boards, the second board's channel numbering can be from 64 through 127 (single-ended), or from 64 through 95 (differential). Therefore, single-ended and differential channels always begin at the same number on each board.

When more than one AMUX-64T board is used, the channels on the different boards are addressed as follows:

Table C-2. External Multiplexer Channels

AMUX-64T Board	Channel Number	
	Single-Ended	Differential
Board A	0 - 63	0 - 31
Board B	64 - 127	64 - 95
Board C	128 - 191	128 - 159
Board D	192 - 255	192 - 223

The channel address on each AMUX-64T depends on the switch setting on each board. (See the *AMUX-64T User Manual* for more information on the external multiplexer board.)

Scanning Order Using the AMUX-64T

The scanning counters on the AMUX-64T and on an MIO board perform automatic scanning of the AMUX-64T analog input channels. When a multiple-channel scan data acquisition is performed with an AMUX-64T, one of the counter/timers on the MIO board normally available to the user, Counter 1, is used for switching the MIO board onboard multiplexers.

Scanning is a simple operation for one AMUX-64T board, but becomes more complex for multiple AMUX-64T boards. The following paragraphs explain in detail how channels are scanned from the AMUX-64T. You must know this scanning order so that you can determine from which analog input channel the data was scanned during a data acquisition operation. When a single AMUX-64T board is connected to the MIO board, four AMUX-64T input channels must be scanned for every MIO board channel. If two AMUX-64T boards are attached to the MIO board, then eight AMUX-64T channels must be scanned for every MIO board input channel. For example, channels 0 through 3 on AMUX-64T board A and channels 64 through 67 on AMUX-64T board B are multiplexed together into MIO board channel 0. Notice that the first four channels on board A are scanned, followed by the first four channels on board B.

If four AMUX-64T boards are attached to the MIO board, then 16 AMUX-64T channels must be scanned for every MIO board input channel. For example, channels 0 through 3 on AMUX-64T board A, channels 64 through 67 on AMUX-64T board B, channels 128 through 131 on AMUX-64T board C, and channels 192 through 195 on board D are multiplexed together into MIO board channel 0. Notice that the first four channels on board A are scanned, followed by the first four channels on board B, the first four channels on board C, and, finally, the first four channels on board D.

The order in which channels are scanned depends on the scan channel sequence specified in `SCAN_Setup`. This scan sequence is an array of MIO board onboard channel numbers that specifies the order in which the MIO board onboard channels are scanned. The scanning order on the AMUX-64T, however, is fixed. Table B-3 shows the order in which the AMUX-64T channels are scanned for every MIO board input channel for different AMUX-64T configurations.

Table C-3. AMUX-64T Scanning Order for Each MIO Board Input Channel

MIO Board Channel	AMUX-64T Channels						
	One Board	Two Boards		Four Boards			
	Board A	Board A	Board B	Board A	Board B	Board C	Board D
0	0-3	0-3	64-67	0-3	64-67	128-131	192-195
1	4-7	4-7	68-71	4-7	68-71	132-135	196-199
2	8-11	8-11	72-75	8-11	72-75	136-139	200-203
3	12-15	12-15	76-79	12-15	76-79	140-143	204-207
4	16-19	16-19	80-83	16-19	80-83	144-147	208-211
5	20-23	20-23	84-87	20-23	84-87	148-151	212-215
6	24-27	24-27	88-91	24-27	88-91	152-155	216-219
7	28-31	28-31	92-95	28-31	92-95	156-159	220-223
8	32-35	32-35	96-99	32-35	96-99	160-163	224-227
9	36-39	36-39	100-103	36-39	100-103	164-167	228-231
10	40-43	40-43	104-107	40-43	104-107	168-171	232-235
11	44-47	44-47	108-111	44-47	108-111	172-175	236-239
12	48-51	48-51	112-115	48-51	112-115	176-179	240-243
13	52-55	52-55	116-119	52-55	116-119	180-183	244-247
14	56-59	56-59	120-123	56-59	120-123	184-187	248-251
15	60-63	60-63	124-127	60-63	124-127	188-191	252-255

For example, if one AMUX-64T board is used, then whenever channel 0 on the MIO board is selected in the scan sequence, channels 0 through 3 on the AMUX-64T are automatically scanned. If two AMUX-64T boards are used, channels 0 through 3 (board A) and channels 64 through 67 (board B) are automatically scanned whenever channel 0 is selected in the scan sequence. If four AMUX-64T boards are used, channels 0 through 3 (board A), channels 64 through 67 (board B), channels 128 through 131 (board C), and channels 192 through 195 (board D) are automatically scanned whenever channel 0 is selected in the scan sequence.

If the MIO board is programmed with a sequential channel scan sequence of 0 through 7 or 0 through 15, then the AMUX-64T channels are scanned from top to bottom in the order given in Table C-3. Notice that if differential input configuration is used, only MIO board channels 0 through 7 should be entered in the scan sequence in `SCAN_Setup`, in which case only the information pertaining to those channels in Table C-3 applies. See the *AMUX-64T User Manual* for more information on the external multiplexer board.

Appendix D
Transducer Conversion Routines

This appendix describes the transducer conversion routines included in NI-DAQ for Macintosh. You can use these routines to convert analog input voltages read from thermocouples, RTDs, and strain gauges into units of temperature or strain. There is a folder for each language that contains source files for each conversion routine. You can cut and paste, include, or merge these conversion routines into your application source files so that the routines may be called in your application. One of the SCXI example programs includes the thermocouple conversion routine, so you can refer to that program to see how the conversion is incorporated into an application.

The conversion routines were included in NI-DAQ for Macintosh as source files rather than driver function calls so that you have complete access to the conversion formulas that are used. You can edit the conversion formulas or replace them with your own to meet the specific accuracy requirements of your application. Comments in the conversion source code are provided to facilitate any editing you feel is necessary.

There is a header file for each language that contains constant definitions that are used in the conversion routines. This header file should also be included or merged into your application program.

The conversion routine descriptions in this appendix apply to all languages.

NI-DAQ for Macintosh includes the following routines:

- `Thermocouple_Convert` — Both single-voltage and voltage-buffer routines are supplied that convert voltages read from E-, J-, K-, R-, S-, or T-type thermocouples into temperature in Celsius, Fahrenheit, Kelvin, or Rankine.
- `RTD_Convert` — Both single-voltage and voltage-buffer routines are supplied that convert voltages read from an RTD into resistance and then into temperature in Celsius, Fahrenheit, Kelvin, or Rankine.
- `Strain_Convert` — Both single-voltage and voltage-buffer routines are supplied that convert voltages read from a strain gauge into measured strain using the formula appropriate to the strain gauge bridge configuration used.
- `Thermistor_Convert` — Both single-voltage and voltage-buffer routines are supplied that convert voltages read from a thermistor into temperature in Celsius, Fahrenheit, Kelvin, or Rankine.

Thermocouple_Convert
Thermocouple_Buf_Convert

Function

Converts a voltage or voltage buffer that was read from a thermocouple into temperature.

Parameters

TCType is an integer indicating what type of thermocouple was used to read the temperature. Constant definitions for each thermocouple type are given in the conversion header file. You can use the constants that have been defined, or you can pass integer values to the routine.

E = 1
J = 2
K = 3
R = 4
S = 5
T = 6.

CJCTemp is the temperature in Celsius that will be used for cold-junction compensation of the thermocouple temperature. If you are using SCXI, this will most likely be the temperature that was read from the temperature sensor on the SCXI terminal block. The AMUX-64T also has a temperature sensor that can be used for this purpose.

TempScale is an integer indicating in which temperature unit you want your return values to be. Constant definitions for each temperature scale are given in the conversion header file.

Celsius = 1
Fahrenheit = 2
Kelvin = 3
Rankine = 4

The `Thermocouple_Convert` routine has two remaining parameters: **TCVolts** is the voltage that was read from the thermocouple, and **TCTemp** is the return temperature value.

The `Thermocouple_Buf_Convert` routine has three remaining parameters: **numPts** is the number of voltage points to convert, **TCVoltBuf** is the array that contains the voltages that were read from the thermocouple, and **TCTempBuf** is the return array that will contain the temperatures.

Description

These conversions routines plug **TCVolts** (or each element of **TCVoltBuf**) into a polynomial that is appropriate for the given **TCType**. The resulting temperature (in Celsius) is then added to the given **CJCTemp** for cold-junction compensation. That result is then converted to the desired temperature scale specified by **TempScale**.

The valid temperature ranges and accuracies for the polynomials used for each thermocouple type are given in Table D-1. The errors listed in the table refer to the polynomials only; they do not take into consideration the accuracy of the thermocouple itself, the SCXI modules, or the DAQ board that is used to take the voltage reading.

Table D-1. Valid Thermocouple Temperature Ranges and Accuracies

Thermocouple Type	Temperature Range	Error
E	-100° C to 1000° C	±0.5° C
J	0° C to 760° C	±0.1° C
K	0° C to 1370° C	±0.7° C
R	0° C to 1000° C	±0.5° C
S	0° C to 1750° C	±1° C
T	-160° C to 400° C	±0.5° C

The method of cold-junction compensation used in these routines is not the most accurate method. A more accurate method would use the following algorithm:

1. Convert the given **CJCTemp** to a thermocouple voltage using a polynomial that converts temperature to voltage.
2. Add that voltage to **TCVolts** (or each element of **TCVoltBuf**).
3. Plug the resulting voltage(s) into the provided polynomial to obtain temperature.

The weakness of this algorithm is speed, because there are two polynomial expressions that need to be evaluated. In the routines provided, we have sacrificed some accuracy in favor of speed. If your application requires more accuracy, you can modify the routines to use the method outlined above.

RTD_Convert
RTD_Buf_Convert

Function

Converts a voltage or voltage buffer that was read from an RTD into temperature.

Parameters

convType is an integer that indicates whether to use the given conversion formula, or to use a user-defined formula that you have put into the routine.

convType = 0 use the given conversion formula.
convType = -1 use a user-defined formula that has been added to the routine.

Iex is the excitation current that was used with the RTD. If a zero is passed in **Iex,** a default excitation current of 0.15 mA is assumed.

Ro is the RTD resistance at 0° C.

A and **B** are the coefficients of the Callendar Van-Dusen equation that fit your RTD.

TempScale is an integer indicating in which temperature unit you want your return values to be. Constant definitions for each temperature scale are given in the conversion header file.

Celsius = 1
Fahrenheit = 2
Kelvin = 3
Rankine = 4

The `RTD_Convert` routine has two remaining parameters: **RTDVolts** is the voltage that was read from the RTD, and **RTDTemp** is the return temperature value.

The `RTD_Buf_Convert` routine has three remaining parameters: **numPts** is the number of voltage points to convert, **RTDVoltBuf** is the array that contains the voltages that were read from the RTD, and **RTDTempBuf** is the return array that will contain the temperatures.

Description

The conversion routines first find the RTD resistance by dividing **RTDVolts** (or each element of **RTDVoltBuf**) by **Iex**. That resistance is converted to temperature using a solution to the Callendar Van-Dusen equation for RTDs:

$$Rt = Ro[1 + At + Bt^2 + C(t-100)t^3]$$

For temperatures above 0° C, the C coefficient is zero and the above equation reduces to a quadratic equation for which we have found the appropriate root. Thus, these conversion routines are only accurate for temperatures above 0° C.

Your RTD documentation should give you **Ro** and the **A** and **B** coefficients for the Callendar Van-Dusen equation. The most common RTDs are 100-Ω platinum RTDs that either follow the European temperature curve (also known as the DIN 43760 standard) or the American curve. Those values for **A** and **B** are given below:

European Curve (DIN 43760):
A = 3.90802e-03
B = -5.80195e-07
(α = 0.00385; ∂ = 1.492)
American Curve:
A = 3.9784e-03
B = -5.8408e-07
(α = 0.00392; ∂ = 1.492)

Some RTD documentation will give you values for α, β, and ∂, from which you can calculate **A**, **B**, and **C** using the following equations:

$$\mathbf{A} = \alpha(1 + \partial/100)$$
$$\mathbf{B} = -\alpha\partial/100^2$$
$$\mathbf{C} = -\alpha\beta/100^4 \text{ for } < 0° \text{ C}$$

Strain_Convert
Strain_Buf_Convert

Description

Converts a voltage or voltage buffer that was read from a strain gauge to units of strain.

Parameters

bridgeConfig is an integer indicating in what type of bridge configuration the strain gauge is mounted. Figure D-1 shows all the different bridge configurations and the corresponding values that should be passed in **bridgeConfig**.

Vex is the excitation voltage that was used. If the value of **Vex** is 0, a default excitation voltage of 3.333 V is assumed. The SCXI-1121 module provides for excitation voltages of 10 V and 3.333 V.

GF is the gauge factor of the strain gauge.

v is Poisson's Ratio (only needed in certain bridge configurations).

Rg is the strain gauge nominal value.

RL is the lead resistance. In many cases, the lead resistance will be negligible and you can pass a value of 0 for **RL** to the routine. Otherwise you can measure **RL** to be more accurate.

Vinit is the unstrained voltage of the strain gauge after it has been mounted in its bridge configuration. You should read this voltage at the beginning of your application and save it to pass to the strain gauge conversion routines.

The `Strain_Convert` routine has two remaining parameters: **strainVolts** is the voltage that was read from the strain gauge, and **strainVal** is the return strain value.

The `Strain_Buf_Convert` routine has three remaining parameters: **numPts** is the number of voltage points to convert, **strainVoltBuf** is the array that contains the voltages that were read from the strain gauge, and **strainValBuf** is the return array that will contain the strain values.

Description

The conversion formula used is based solely on the bridge configuration. The seven bridge configurations supported and the corresponding formulas are given in Figure D-1. For all bridge configurations, the following formula is used to obtain Vr:

Vr = (**strainVolts** - **Vinit**) / **Vex**

In the circuit diagrams, VOUT is the voltage that is measured and passed to the conversion routines in **strainVolts** or **strainVoltBuf**. In the quarter-bridge and half-bridge configurations, R_1 and R_2 are dummy resistors that are not directly incorporated into the conversion formula. The SCXI-1121 provides R_1 and R_2 for a bridge-completion network, if needed. Please refer to the *SCXI-1121 User Manual* for more information on bridge-completion networks and voltage excitation.

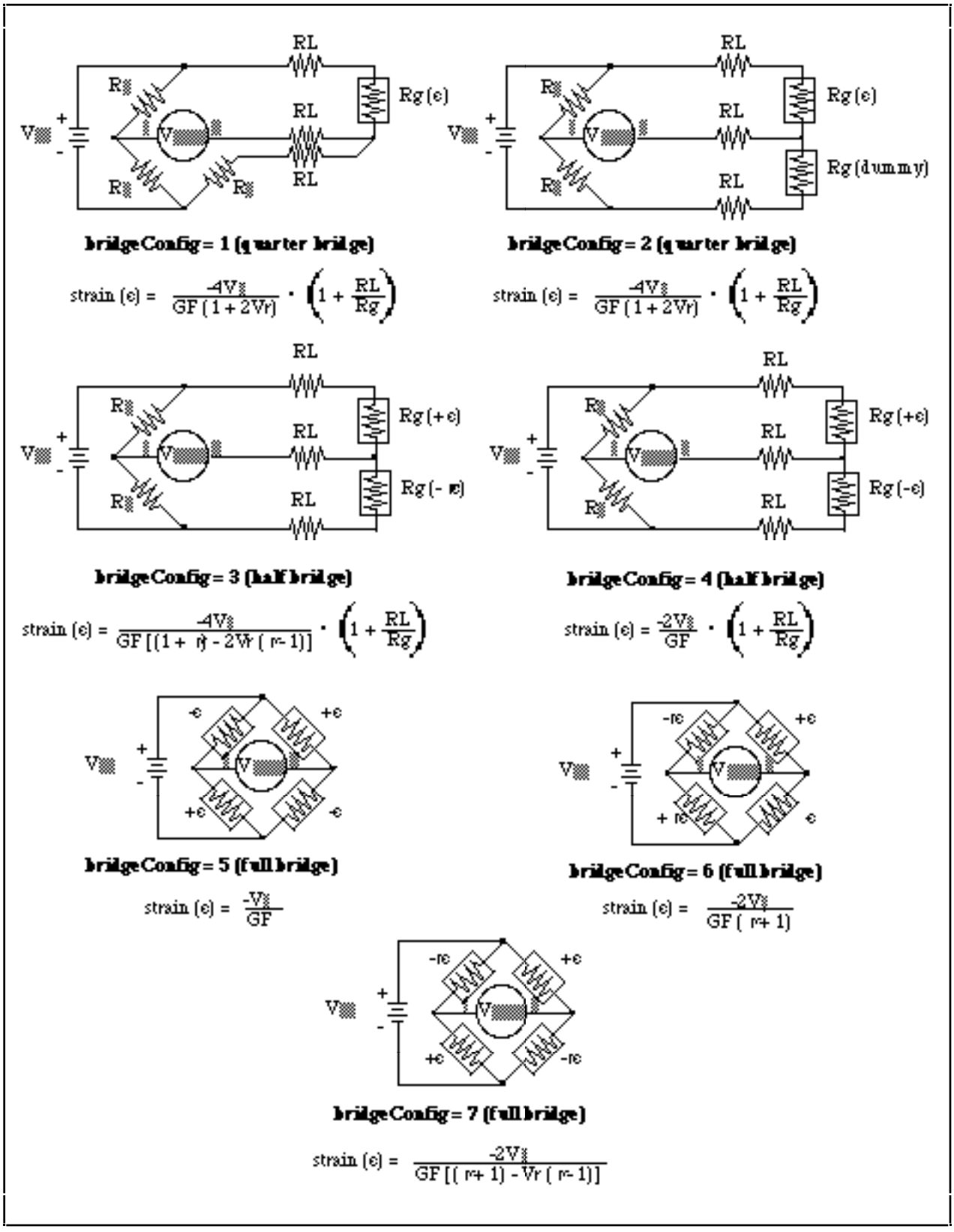

Figure D-1. Strain Gauge Bridge Configurations

Thermistor_Convert
Thermistor_Buf_Convert

Function

Converts a voltage or voltage buffer that was read from a thermistor to temperature. Some SCXI terminal blocks have onboard thermistors that you can use to do cold-junction compensation.

Parameters

Vref is the voltage reference you apply across the thermistor circuit (see Figure D-2). The thermistor on the SCXI terminal blocks has a **Vref** of 2.5 volts.

R1 is the value expressed in Ohms of the resistor in series with your thermistor (see Figure D-2). The thermistor on the SCXI terminal blocks has an **R1** value of 5000 Ohms.

TempScale is an integer indicating in which temperature unit you want your return values to be. Constant definitions for each temperature scale are given in the conversion header file.

Celsius = 1
Fahrenheit = 2
Kelvin = 3
Rankine = 4

The `Thermistor_Convert` function has two remaining parameters–**Volts** is the voltage that you read from the thermistor, and **Temperature** is the return temperature value given in units determined by **TempScale**.

The `Thermistor_Buf_Convert` function has three remaining parameters–**numPts** is the number of voltage points to convert, **VoltBuf** is the array of voltages that you read from the thermistor, and **TempBuf** is the return array of temperature values given in units determined by **TempScale**.

Description

The thermistor conversion routines use the following equation, which expresses the relationship between **Volts** and Rt, the thermistor resistance (see Figure D-2).

Volts = **Vref** (Rt / (**R1** + Rt))

Solving the previous equation for Rt:

Rt = **R1** (**Volts** / (**Vref** + **Volts**))

Once the routine calculates Rt, it uses the following equation to convert Rt, the thermistor resistance, to temperature in Kelvin. Then it converts the temperature to the desired temperature scale if necessary.

T = 1 / (a + b(ln Rt) + c(ln Rt)^3)

The following values used for a, b, and c are correct for the thermistors provided on the SCXI terminal blocks. If you are using a thermistor with different values for a, b, and c (consult you thermistor data sheet), you can edit the thermistor conversion routine to use your own a, b, and c values.

a = 1.295361E-3
b = 2.343159E-4
c = 1.018703E-7

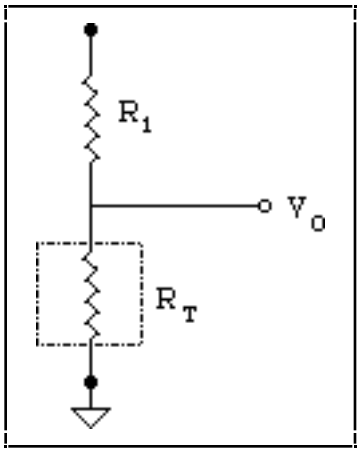

Figure D-2. Circuit Diagram of a Thermistor in a Voltage Divider

Appendix E
Analog Input Channel and Gain Settings and Voltage Calculation

This appendix lists the valid channel and gain settings for DAQ boards, describes how NI-DAQ calculates voltage, and describes the measurement of offset and gain adjustment.

DAQ Device Analog Input Channel Settings

Table E-1 lists the valid analog input (ADC) channel settings. If you have one or more AMUX-64T boards and an MIO board, see Appendix C, *Using an External Multiplexer*, for more information.

Table E-1. Valid Analog Input Channel Settings

Device	Settings	
	Single-ended configuration	**Differential configuration**
NB-MIO-16, NB-MIO-16X	0–15	0–7
Lab-LC, DAQCard-1200, PCI-1200	0–7	0, 2, 4, 6
DAQCard-700	0–15	0–7
DAQCard-500, Lab-NB, Lab-LC	0–7	N/A
PCI-MIO-16XE-50	0–15	0–7

†`ND_CJ_TEMP` is a constant that is defined in a header file. The header file you should use depends on the language you are using:

- C programmers—`NIDAQCNS.H`
- Pascal programmers—`NIDAQCNS.PAS`

Voltage Calculation

Table E-2 lists the valid return values for different devices:

Table E-2. Valid Return Values

Board(s)	Unipolar Mode	Bipolar Mode
NB-MIO-16, Lab and 1200 Series	0 to 4,095	-2,048 to 2,047
NB-MIO-16X, PCI-MIO-16XE-50	0 to 65,535	-32,768 to 32,767
DAQCard-500, DAQCard-700	—	-2,048 to 2,047

`AI_VScale` and `DAQ_VScale` calculate voltage from **reading** as follows:

$$\text{voltage} = \left(\frac{\textbf{reading} - \textbf{offset}}{\textbf{maxReading}}\right) * \left(\frac{\textbf{maxVolt}}{\textbf{gain} * \textbf{gainAdjust}}\right)$$

where:

maxReading is the maximum binary reading for the given board, channel, range, and polarity.

maxVolt is the maximum voltage the board can measure at a gain of 1 in the given range and polarity.

Table E-3 lists the values of **maxReading** and **maxVolt** for different boards.

Table E-3. The Values of maxReading and maxVolt

Board(s)	Unipolar Mode		Bipolar Mode	
	maxReading	**maxVolt**	**maxReading**	**maxVolt**
NB-MIO-16	4,096	*	2,048	*
NB-MIO-16X, PCI-MIO-16XE-50	65,536	10 V	32,768	10 V
Lab and 1200 Series	4,096	10 V	2,048	5 V
DAQCard-500, DAQCard-700	4,096	*	2,048	*
* The value of **maxVolt** depends on **inputRange**, as discussed in `AI_Configure`.				

For the DAQCard-1200, **gain** is ignored, and the following formula is used:

$$\text{voltage} = \left(\frac{\textbf{reading} - \textbf{offset}}{\textbf{maxReading}}\right) * (\textbf{maxVolt})$$

Offset and Gain Measurement

Measurement of Offset

To determine the **offset** parameter used in the `AI_VScale` and `DAQ_VScale` functions, follow this procedure:

1. Ground analog input channel *i*, where *i* can be any valid input channel.
2. Call the `AI_Read function` with **gain** set to the gain that will be used in your real acquisition (*g*). The reading given by the `AI_Read function is the value of` **offset**. The offset is only valid for the gain setting at which it was measured. Remember that the data type of **offset** in the `AI_VScale` and `DAQ_VScale` functions is floating point, so if you use `AI_Read` to get the offset, you will have to typecast it before passing it to the scale function.

Note: ***Another way to read the offset is to perform multiple readings using a DAQ function call and average them to be more accurate and reduce the effects of noise.***

Measurement of Gain Adjustment

To determine the **gainAdjust** parameter used in the `AI_VScale` and `DAQ_VScale` functions, follow this procedure:

1. Connect the known voltage V_{in} to channel *i*.
2. Call the `AI_Read` function with gain equal to *g*. Use the reading returned by `AI_Read` with the offset value determined above to calculate the real gain.

Note: ***You can use the DAQ functions to take many readings and average them instead of using the*** `AI_Read` ***function.***

The real gain is computed as follows:

$$G_R = \left(\frac{\textbf{reading} - \textbf{offset}}{\textbf{maxReading}}\right) * \left(\frac{\textbf{maxVolt}}{V_{in}}\right)$$

The gain adjustment is computed as follows:

$$\textbf{gainAdjust} = \left[1 - \frac{(g - G_R)}{g}\right]$$

Appendix F
Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

Corporate Headquarters
(512) 794-0100
Technical support fax: (512) 794-5678

Branch Offices	**Phone Number**	**Fax Number**
Australia	03 9 879 9422	03 9 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	519 622 9310	
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 71 11
Finland	90 527 2321	90 502 2930
France	1 48 14 24 24	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	95 800 010 0793	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___)_____ Phone (___)_____

Computer brand _____ Model _____ Processor _____

Operating system _____

Speed _____ MHz RAM _____ MB Display adapter _____

Mouse _____yes _____ no Other adapters installed _____

Hard disk capacity _____MB Brand _____

Instruments used _____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is _____

_____

_____

_____

_____

List any error messages _____

_____

_____

_____

The following steps will reproduce the problem _____

_____

_____

_____

NI-DAQ for Macintosh Hardware and Software Configuration Form

---

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

- Data Acquisition Hardware _____
- Interrupt Level of Hardware _____
- DMA Channels of Hardware _____
- Base I/O Address of Hardware _____
- NI-DAQ Version _____
- LabVIEW Version _____

Other Products

- Computer Make and Model _____
- Microprocessor _____
- Clock Frequency _____
- Type of Video Board Installed _____
- System Version _____
- Programming Language _____
- Programming Language Version _____
- Other Boards in System _____
- Base I/O Address of Other Boards _____
- DMA Channels of Other Boards _____
- Interrupt Level of Other Boards _____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: **NI-DAQ® Software Reference Manual for Macintosh, Version 4.8**

Edition Date: **February 1996**

Part Number: **371345A-01**

Please comment on the completeness, clarity, and organization of the manual.

If you find errors in the manual, please record the page numbers and describe the errors.

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

Phone (_____)_____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678

Glossary

Prefix	Meaning	Value
n-	nano-	10^{-9}
µ-	micro-	10^{-6}
m-	milli-	10^{-3}
k-	kilo-	10^{3}
M-	mega-	10^{6}

°	degrees
A	amperes
A/D	analog-to-digital
AC	alternating current
ADC	A/D converter
BCD	binary-coded decimal
C	Celsius
CPU	central processing unit
D/A	digital-to-analog
DAC	D/A converter
DC	direct current
DMA	direct memory access
DSP	digital signal processing
EEPROM	electrically erasable programmable read-only memory
FIFO	first-in-first-out
GPIB	General Purpose Interface Bus
hex	hexadecimal
Hz	hertz
I/O	input/output
LSB	least significant bit
MB	megabytes of memory
min	minutes
NB	NuBus
PIA	peripheral interface adapter
PPI	programmable peripheral interface
pt	points
rms	root mean square
ROM	read-only memory
RTD	resistance temperature detector
RTSI	Real-Time System Integration
s	seconds
SCXI	Signal Conditioning eXtensions for Instrumentation
SPDT	single-pole double-throw
TC	terminal count
V	volts
Vrms	volts, root mean square

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z

