NATIONAL INSTRUMENTS™
LabVIEW™

Measurements Manual

NATIONAL INSTRUMENTS™

July 2000 Edition
Part Number 322661A-01

Worldwide Technical Support and Product Information

www.ni.com

National Instruments Corporate Headquarters

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

Worldwide Offices

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 284 5011,
Canada (Calgary) 403 274 9391, Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521,
China 0755 3904939, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24,
Germany 089 741 31 30, Greece 30 1 42 96 427, Hong Kong 2645 3186, India 91805275406,
Israel 03 6120092, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico (D.F.) 5 280 7625,
Mexico (Monterrey) 8 357 7695, Netherlands 0348 433466, New Zealand 09 914 0488, Norway 32 27 73 00,
Poland 0 22 528 94 06, Portugal 351 1 726 9011, Singapore 2265886, Spain 91 640 0085,
Sweden 08 587 895 00, Switzerland 056 200 51 51, Taiwan 02 2528 7227, United Kingdom 01635 523545

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

Important Information

Warranty

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

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

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

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

Copyright

Trademarks

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

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

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

Contents

Chapter 4
Example Measurements

Part II
DAQ Basics

Chapter 5
Introduction to Data Acquisition in LabVIEW

Chapter 6
Analog Input

Chapter 7
Analog Output

Chapter 8
Digital I/O

Chapter 9
SCXI—Signal Conditioning

Chapter 10
High-Precision Timing (Counters/Timers)

Part III
Measurement Analysis in LabVIEW

Chapter 11
Introduction to Measurement Analysis in LabVIEW

Chapter 12
DC/RMS Measurements

Chapter 13
Frequency Analysis

Chapter 14
Distortion Measurements

Chapter 15
Limit Testing

Chapter 16
Digital Filtering

Chapter 17
Signal Generation

PART IV
Instrument Control in LabVIEW

Chapter 18
Using LabVIEW to Control Instruments

Chapter 19
Instrument Drivers in LabVIEW

Chapter 20
VISA in LabVIEW

Tables

About This Manual

The *LabVIEW Measurements Manual* contains information you need to take and analyze measurement data in LabVIEW. You should have a basic knowledge of LabVIEW before you try to read this manual. If you have never worked with LabVIEW, please read through *Getting Started with LabVIEW* before you begin.

This manual supplements the *LabVIEW User Manual*, and assumes that you are familiar with that material. You also should be familiar with the operation of LabVIEW, your computer, your computer's operating system, and your data acquisition (DAQ) device.

Conventions

The following conventions appear in this manual:

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

This icon denotes a tip, which alerts you to advisory information.

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

bold Bold text denotes items that you must select or click on in the software, such as menu items, dialog box options, and palettes. Bold text also denotes controls and buttons on the front panel and parameter names on the block diagram.

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

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

monospace bold	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
monospace italic	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.
Platform	Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

Related Documentation

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

- *Getting Started with LabVIEW*
- *LabVIEW User Manual*
- *LabVIEW Help*, available by selecting **Help»Contents and Index**
- The user manuals for your data acquisition devices
- Various Application Notes, available on the National Instruments Web site at `http://zone.ni.com/appnotes.nsf/`

Part I

Introduction to Measurement

This part contains information you should know before taking measurements in LabVIEW.

Part I, *Introduction to Measurement*, contains the following chapters:

- Chapter 1, *What Is Measurement and Virtual Instrumentation?*, introduces the concepts of measurement and virtual instrumentation.
- Chapter 2, *Comparing DAQ Devices and Special-Purpose Instruments for Data Acquisition*, describes your options for hardware and software systems.
- Chapter 3, *Installing and Configuring Your Measurement Hardware*, explains how to set up your system to use data acquisition with LabVIEW and your DAQ hardware.
- Chapter 4, *Example Measurements*, explains several examples of common measurements using LabVIEW.

1

What Is Measurement and Virtual Instrumentation?

You take measurements with instruments. Instrumentation helps science and technology progress. Scientists and engineers around the world use instruments to observe, control, and understand the physical universe. Our quality of life depends on the future of instrumentation—from basic research in life sciences and medicine to design, test and manufacturing of electronics, to machine and process control in countless industries.

History of Instrumentation

As a first step in understanding how instruments are built, consider the history of instrumentation. Instruments have always made use of widely available technology. In the 19th century, the jeweled movement of the clock was first used to build analog meters. In the 1930s, the variable capacitor, the variable resistor, and the vacuum tube from radios were used to build the first electronic instruments. Display technology from the television has contributed to modern oscilloscopes and analyzers. And finally, modern personal computers contribute high-performance computation and display capabilities at an ever-improving performance-to-price ratio.

What Is Virtual Instrumentation?

Virtual instrumentation is defined as combining hardware and software with industry-standard computer technologies to create user-defined instrumentation solutions. National Instruments specializes in developing plug-in hardware and driver software for data acquisition (DAQ), IEEE 488 (GPIB), VXI, serial, and industrial communications. The driver software is the programming interface to the hardware and is consistent across a wide range of platforms. Application software such as LabVIEW, LabWindows/CVI, ComponentWorks, and Measure deliver sophisticated display and analysis capabilities required for virtual instrumentation.

You can use virtual instrumentation to create a customized system for test, measurement, and industrial automation by combining different hardware and software components. If the system changes, you often can reuse the virtual instrument components without purchasing additional hardware or software.

System Components for Taking Measurements with Virtual Instruments

Different hardware and software components can make up your virtual instrumentation system. Many of these options are described in more detail throughout this manual. There is a wide variety of hardware components you can use to monitor or control a process or test a device. As long as you can connect the hardware to the computer and understand how it makes measurements, you can incorporate it into your system.

2

Comparing DAQ Devices and Special-Purpose Instruments for Data Acquisition

Measurement devices, such as general-purpose data acquisition (DAQ) devices and special-purpose instruments, are concerned with the acquisition, analysis, and presentation of measurements and other data you acquire.

Acquisition is the means by which physical signals, such as voltage, current, pressure, and temperature, are converted into digital formats and brought into the computer. Popular methods for acquiring data include plug-in DAQ and instrument devices, GPIB instruments, VXI instruments, and RS-232 instruments.

Data analysis transforms raw data into meaningful information. This can involve such things as curve fitting, statistical analysis, frequency response, or other numerical operations.

Data presentation is the means for communicating with your system in an intuitive, meaningful format.

Building a computer-based measurement system can be a daunting task. There is a wide variety of hardware components you can use to monitor or control a process or test a device. Should you build on traditional rack-and-stack IEEE 488 equipment or look to modular VXI-based solutions? Or maybe you should consider a PC-based plug-in board approach. Which type of hardware meets your needs today and will be around for the long run? What are the differences between all the choices? This chapter will describe several types of hardware solutions to help you answer these questions.

DAQ Devices versus Special-Purpose Instruments

The fundamental task of all measurement systems is the measurement and/or generation of real-world physical signals. The primary difference between the various hardware options is the method of communication between the measuring hardware and the computer. In this chapter we will separate the discussion into two categories: general purpose DAQ devices and special purpose instruments.

General purpose DAQ devices are devices that connect to the computer allowing the user to retrieve digitized data values. These devices typically connect directly to the computer's internal bus through a plug-in slot. Some DAQ devices are external and connect to the computer via serial, GPIB, or ethernet ports. The primary distinction of a test system that utilizes general purpose DAQ devices is where measurements are performed. With DAQ devices, the hardware only converts the incoming signal into a digital signal that is sent to the computer. The DAQ device does not compute or calculate the final measurement. That task is left to the software that resides in the computer. The same device can perform a multitude of measurements by simply changing the software application that is reading the data. So, in addition to controlling, measuring, and displaying the data, the user application for a computer-based DAQ system also plays the role of the firmware—the built-in software required to process the data and calculate the result—that would exist inside a special purpose instrument. While this flexibility allows the user to have one hardware device for many types of tests, the user must spend more time developing the different applications for the different types of tests. Fortunately, LabVIEW comes with many acquisition and analysis functions to make this easy.

Instruments are like the general purpose DAQ device in that they digitize data. However, they have a special purpose or a specific type of measurement capability. The software, or firmware, required to process the data and calculate the result is usually built in and cannot be modified. For example, a multi-meter can not read data the way an oscilloscope can because the program that is inside the multi-meter is permanently stored and cannot be changed dynamically. Most instruments are external to the computer and can be operated alone, or they may be controlled and monitored through a connection to the computer. The instrument has a specific protocol that the computer must use in order to communicate with the instrument. The connection to the computer could be Ethernet, Serial, GPIB, or VXI. There are some instruments that can be installed into the computer like the general purpose DAQ devices. These devices are called computer-based instruments.

The following sections discuss the communication between computers and measurement hardware.

How Do Computers Talk to DAQ Devices?

Before a computer-based system can measure a physical signal, a sensor or transducer must convert the physical signal into an electrical one, such as voltage or current. The plug-in DAQ device is often considered to be the entire DAQ system, although it is actually only one system component. Unlike most stand-alone instruments, you cannot always directly connect signals to a plug-in DAQ device. In these cases, you must use accessories to condition the signals before the plug-in DAQ device converts them to digital information. The software controls the DAQ system by acquiring the raw data, analyzing the data, and presenting the results.

Figure 2-1 shows two options for a DAQ system. In Option A, the plug-in DAQ device resides in the computer. In Option B, the DAQ device is external. With an external board, you can build DAQ systems using computers without available plug-in slots, such as some laptops. The computer and DAQ module communicate through various buses such as the parallel port, serial port, and Ethernet. These systems are practical for remote data acquisition and control applications.

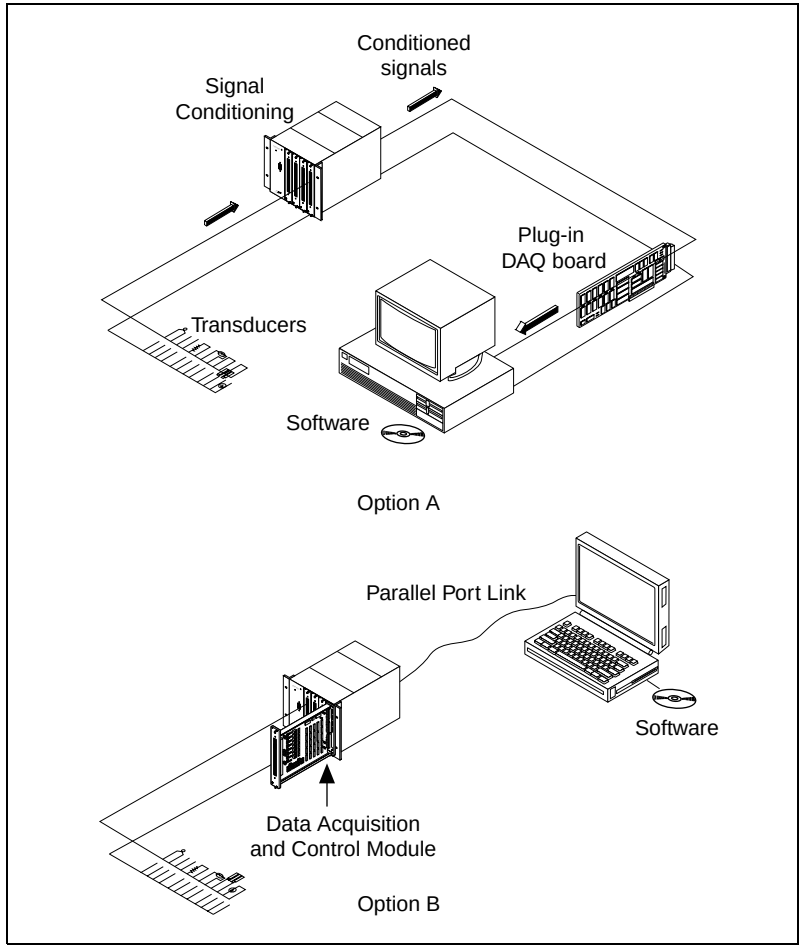


Figure 2-1. DAQ System Components

A third option, not shown in Figure 2-1, uses the PCMCIA bus found on some laptops. A PCMCIA DAQ device plugs into the computer, and signals are connected to the board just as they are in Option A. This allows for a portable, compact DAQ system.

Role of Software

The computer receives raw data. Software takes the raw data and presents it in a form the user can understand. Software manipulates the data so it can appear in a graph or chart or in a file for report. The software also controls the DAQ system, telling the DAQ device when to acquire data, as well as from which channels to acquire data.

Typically, DAQ software includes drivers and application software. Drivers are unique to the device or type of device and include the set of commands the device accepts. Application software (such as LabVIEW) sends the commands to the drivers, such as acquire a thermocouple reading and return the reading, then displays and analyzes the data acquired.

LabVIEW includes a set of VIs that let you configure, acquire data from, and send data to DAQ devices. This saves you the trouble of having to write those programs yourself. LabVIEW DAQ VIs make calls to the NI-DAQ Application Program Interface (API). The NI-DAQ API contains the tools and basic functions that interface to DAQ hardware.

How Do Computers Talk to Special-Purpose Instruments?

The fundamental task of an instrument is to measure some natural phenomenon. Unlike data acquisition, the signal the computer ultimately receives requires no conditioning. How the computer controls the instrument and acquires data from the instrument depends on how the instrument is built. Common types of instruments include the following:

- GPIB
- Serial Port
- VXI
- PXI
- Computer-based instruments

These instrument types are discussed in more detail in Appendix A, *Types of Instruments*.

All external instruments communicate with the computer through some type of bus where a communication protocol has been defined. The instrument has a set of commands that it understands. The user writes an application that sends commands to and receives data from an instrument. As the test system designer, you have to be concerned with the software connection with the instrument. That is, you have to understand how your application and your instrument communicate with each other. Additionally, you have to be concerned with the type of hardware connection to your instrument.

How Do Programs Talk to Instruments?

As test developers over the years have discovered, instrument drivers are a key factor in test development. An instrument driver is a collection of functions that implement the commands necessary to perform the instrument's operations. LabVIEW instrument drivers simplify instrument programming to high-level commands, so you do not need to learn the low-level instrument-specific syntax needed to control your instruments. Instrument drivers are not necessary to use your instrument. They are merely time savers to help you develop your project so you do not need to study the instrument manual before writing a program.

Instrument drivers create the instrument commands and communicate with the instrument over the serial, GPIB, or VXI bus. In addition, instrument drivers receive, parse, and scale the response strings from instruments into scaled data that can be used in your test programs. With all of this work already done for you in the driver, instrument drivers can significantly reduce development time.

Instrument drivers can help make test programs more maintainable in the long-run because instrument drivers contain all of the I/O for an instrument within one library, separate from your other code. You are protected against hardware changes and upgrades because it is much easier to upgrade your test code when all of the code specific to that particular instrument is self-contained within the instrument driver.

LabVIEW provides more than 700 LabVIEW instrument drivers from more than 50 vendors. A list is available on the National Instruments Developer Zone, `zone.ni.com/idnet` You can use these instrument drivers to build complete systems quickly. Instrument drivers drastically reduce software development costs because developers do not need to spend time programming their instruments. You can reuse the drivers in a variety of systems and configurations.

Deciding what kind of instrument to use depends on the tests and measurements you are taking. Appendix A, *Types of Instruments*, describes several traditional forms of instrumentation hardware that you are likely to encounter when developing a measurement system.

3

Installing and Configuring Your Measurement Hardware

This chapter explains how to set up your system to take measurements with LabVIEW and your data acquisition hardware. This chapter contains hardware installation and configuration and software configuration instructions and some general information and techniques.

Overview

NI-DAQ, the driver software for National Instruments DAQ devices, provides LabVIEW with a high-level interface to measurement devices. National Instruments also supplies driver software for communicating with special purpose instruments, including NI-488.2, NI-VISA, and IVI.

Figure 3-1 shows the relationship between LabVIEW, driver software, and measurement hardware. The LabVIEW VIs call into the driver software which communicates with the measurement hardware.

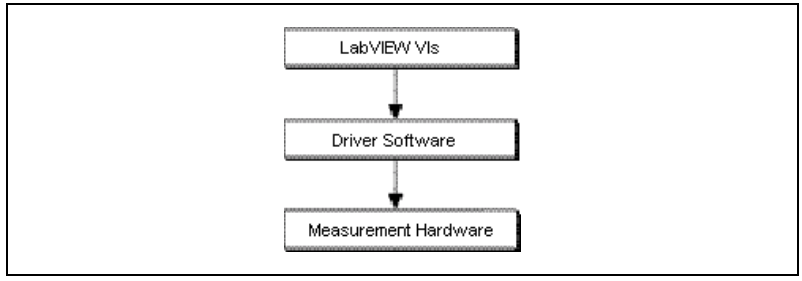


Figure 3-1. Relationship between LabVIEW, Driver Software, and Measurement Hardware

Installing and Configuring Your Hardware

Before you begin your measurement application development, you must install and configure your measurement hardware. The software drivers need the hardware configuration information to program your hardware properly.

As mentioned earlier, each system architecture is different. Some systems might use general purpose plug-in DAQ devices or a PCMCIA card, others might use a parallel port to control external devices. Still others might use special purpose instruments controlled through serial, Ethernet, GPIB, or VXI ports. Your system requires a unique configuration procedure to ensure your measurement devices work properly and coexist with other peripherals. However, in most cases, you complete the following general steps to install your DAQ device.

1. Install LabVIEW and your driver software. The LabVIEW installer installs National Instruments driver software if the version included with LabVIEW is newer than any previously installed version of the drivers.
2. Power off your computer.
3. Install your measurement hardware.
4. Power on your computer.
5. Configure your measurement hardware using Measurement & Automation Explorer (**Windows**) or the Configuration Utility (**Macintosh**).

Note For Windows 2000/NT, be sure you log on as an administrator when installing the LabVIEW and driver software and when configuring your measurement hardware.

Before installing your measurement hardware, consult your hardware user manual to see if you need to change any hardware-selectable options. For example, some hardware have jumpers to select analog input polarity, input mode, analog output reference, and so on. Make a note of which options you change so you can notify the driver software either by entering the information in one of the configuration utilities or using function calls in your application.

Refer to Measurement & Automation Explorer Help or the Troubleshooting Wizards, available at `www.ni.com`, for more specific information about installing and configuring your hardware.

The following sections discuss the utilities available to you to configure your hardware on different operating systems.

Measurement & Automation Explorer (Windows)

Measurement & Automation Explorer is a Windows-based application installed during your National Instruments driver software installation. Double-click the Measurement & Automation icon on your desktop to configure your National Instruments software and hardware, execute system diagnostics, add new channels and interfaces to your system, and view the devices and instruments you have connected.

NI-DAQ Configuration Utility (Macintosh)

The NI-DAQ Configuration Utility configures the parameters for the DAQ devices installed in your Macintosh computer. The Macintosh OS automatically recognizes DAQ devices. After you install your DAQ device in your system, you must use the Configuration Utility to assign a device number to the device. The configuration utility saves the device number and the configuration parameters for your DAQ device and SCXI system. After you configure your system, you do not need to run the Configuration Utility again unless you change the system parameters. A shortcut to the NI-DAQ Configuration Utility is installed in the LabVIEW folder.

NI-488.2 Configuration Utility (Macintosh)

The NI-488.2 Configuration Utility, available at **Start»Settings»Control Panel**, configures the parameters for the GPIB devices installed in your Macintosh computer. The Macintosh OS automatically recognizes GPIB devices. You can view or modify the default configuration settings using this utility.

Configuring Your DAQ Channels

After you install and configure your DAQ device, you can configure your DAQ channels. LabVIEW NI-DAQ software includes the DAQ Channel Wizard, which you can use to configure the analog and digital channels on your DAQ device—DAQ plug-in boards, stand-alone DAQ products, or SCXI modules. On Windows, access the DAQ Channel Wizard through the **Data Neighborhood** in Measurement & Automation Explorer. On Macintosh, access the DAQ Channel Wizard by selecting **Tools»Data Acquisition»DAQ Channel Wizard** in LabVIEW. The DAQ Channel Wizard helps you define the physical quantities you measure or generate on each DAQ hardware channel. You can configure information about the

physical quantity you are measuring, the sensor or actuator you are using, and the associated DAQ hardware.

As you configure channels in the DAQ Channel Wizard, you give each channel configuration a unique name that is used when addressing your channels in LabVIEW. The channel configurations you define are saved in a file that instructs the NI-DAQ driver how to scale and process each DAQ channel by its name. You can simplify the programming required to measure your signal by using the DAQ Channel Wizard to configure your channels.

Assigning VISA Aliases and IVI Logical Names

On Windows, you can assign meaningful VISA Aliases and IVI Logical Names to your instruments that you control using VISA and IVI. Assign VISA Aliases in the **Devices and Interfaces** section in Measurement & Automation Explorer. Configure IVI Logical Names in the **IVI** section in Measurement & Automation Explorer. The aliases and logical names can be used in your LabVIEW application development to refer to your instrument. For example, you can assign the alias `scope` to the port that has a scope connected to it. Refer to Part IV, *Instrument Control in LabVIEW*, for more information about communicating with special purpose instruments.

On UNIX, you can set VISA Aliases to make VISA resource names easier to remember by running `visaconf`.

Configuring Serial Ports on Macintosh

Launch the VISA Find Resource function, available on the **Functions» Instrument I/O»VISA»VISA Advanced** palette. When you launch this function, new ports are automatically detected and assigned VISA resource names.

Configuring Serial Ports on UNIX

Run visaconf. Click the **Add Static Resource** button and create a new resource name, such as `ASRL99::INSTR`. Then fill in the remaining fields.

4

Example Measurements

This chapter describes how to take common measurements using LabVIEW. Many of the examples show how to take these measurements using Multifunction Input Output (MIO) type DAQ devices. These examples use the DAQ VIs. Other examples explain how to take these measurements using an instrument. An instrument can be a stand-alone device connected to a GPIB or Serial bus or a dedicated plug-in instrument board. The instrument examples use the IVI class driver VIs but are similar to how you can build an application with any kind of instrument driver.

Example DMM Measurements

This section describes how to take measurements typical of a digital multimeter (DMM).

Use the DAQ Channel Wizard to name and configure analog and digital channels on your DAQ device. Refer to Chapter 3, *Installing and Configuring Your Measurement Hardware*, for more information about the DAQ Channel Wizard.

How to Measure DC Voltage

Direct Current (DC) signals are analog signals that slowly vary with time. Common DC signals include voltage, temperature, pressure, and strain. Since DAQ devices read voltage, most of these measurements require a transducer. A transducer is a device that converts a physical phenomenon into an electrical signal.

With DC signals you are most interested in how accurately you can measure the amplitude of a signal at a given point in time. To improve the accuracy of most measurements, use signal conditioning. Signal conditioning involves manipulating the signal using hardware and software. Common software signal conditioning includes averaging, filtering, and linearization. In this manual, you primarily use software signal conditioning. Common hardware signal conditioning includes amplification, cold junction compensation (for thermocouples), excitation, bridge completion, and filtering. Refer to Chapter 9, *SCXI—Signal Conditioning*, for more information about software signal conditioning.

Refer to Application Note 048, *Signal Conditioning Fundamentals for PC-Based Data Acquisition Systems*, available on the National Instruments Web site at `zone.ni.com/appnotes.nsf/`, for more information about hardware signal conditioning.

The examples that follow show the data acquisition system, the typical signal being acquired, a sketch of how to physically connect the transducer involved, and LabVIEW diagrams of how to acquire the signal.

Single-Point Acquisition Example

Figure 4-1 shows a simple data acquisition system for DC measurements using an anemometer to measure wind speed.

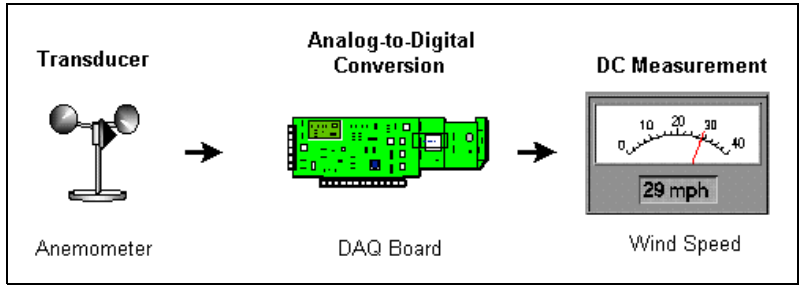


Figure 4-1. Simple Data Acquisition System

In this example, you take a single wind-speed measurement. In the *Averaging a Scan Example* section, you apply some simple software signal conditioning to improve our measurement.

Figure 4-2 shows what the actual wind speed might be at a given time.

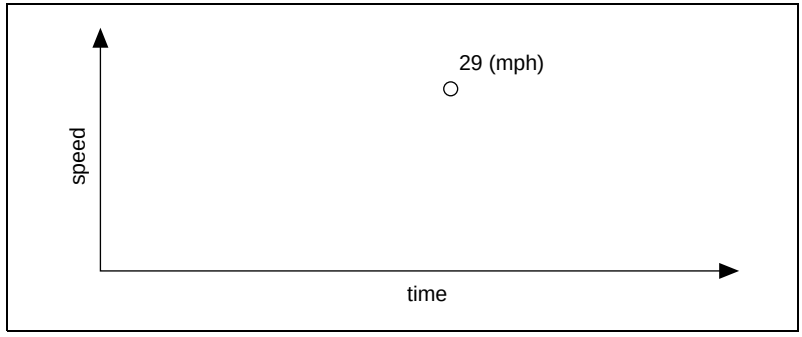


Figure 4-2. Wind Speed

Figure 4-3 shows a typical wiring diagram for an anemometer with an output range of 0 to 10 V, which corresponds to wind speed from 0 to 200 mph. This means that in software, you will need to scale the data using the following formula:

anemometer reading (V) × 20 (mph/V) = wind speed (mph) (4-1)

Notice the use of a resistor, R, because an anemometer is usually not a grounded signal source. If the anemometer transducer were already grounded, using R would cause a ground loop and result in erroneous readings. Refer to Chapter 6, *Analog Input*, for more information about grounded and floating signal sources.

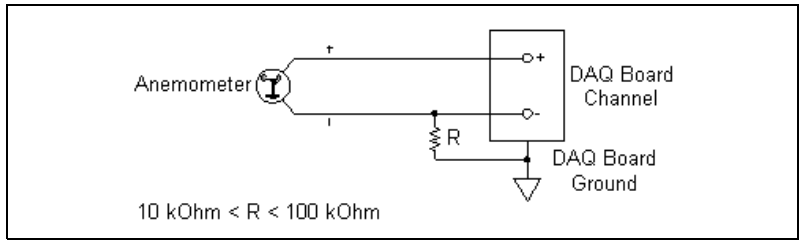


Figure 4-3. Anemometer Wiring

Figure 4-4 shows the block diagram needed to measure the wind speed. In this diagram, **device** is the number assigned to the plug-in DAQ device during configuration. **Channel** is the analog input channel the anemometer is wired to. The **high limit** and **low limit** values show the expected voltage range. This range determines the amount of gain the DAQ device will apply. AI Sample Channel is the DAQ subVI that acquires a single value, in this case raw voltage. The scaling value of 20 mph/V is used to scale the input voltage range of 0 to 10 V to the wind speed range of 0 to 200 mph according to Equation 4-1.

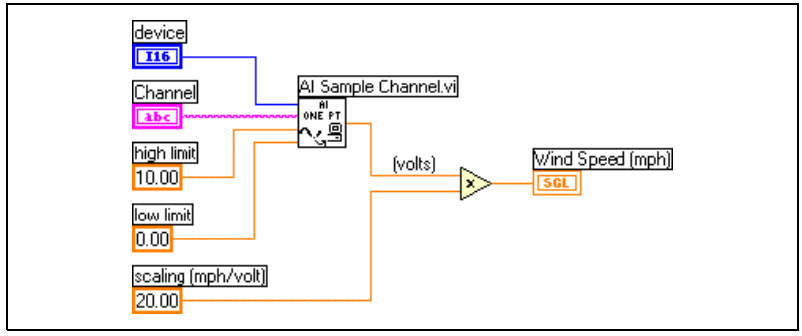


Figure 4-4. Measuring Voltage and Scaling to Wind Speed

You can simplify this block diagram by using DAQ Named Channels. Refer to Chapter 3, *Installing and Configuring Your Measurement Hardware*, for more information about DAQ Named Channels.

Figure 4-5 shows the LabVIEW diagram needed to measure the wind speed using DAQ Named Channels. This simplifies the block diagram, because the DAQ Named Channel remembers information about the device, channel, gains, and the scaling equation. Again, AI Sample Channel acquires a single value, but in this case, it returns the wind speed.

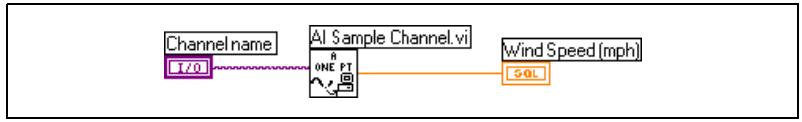


Figure 4-5. Measuring Wind Speed Using DAQ Named Channels

Averaging a Scan Example

One of the most useful and easy-to-use forms of signal conditioning is averaging data in software. Averaging can yield a more useful reading if a signal is rapidly changing or if there is noise on the line. Refer to Chapter 12, *DC/RMS Measurements*, for more information about averaging to improve your measurements.

Figure 4-6 shows the data acquisition system for measuring wind speed with the addition of software averaging.

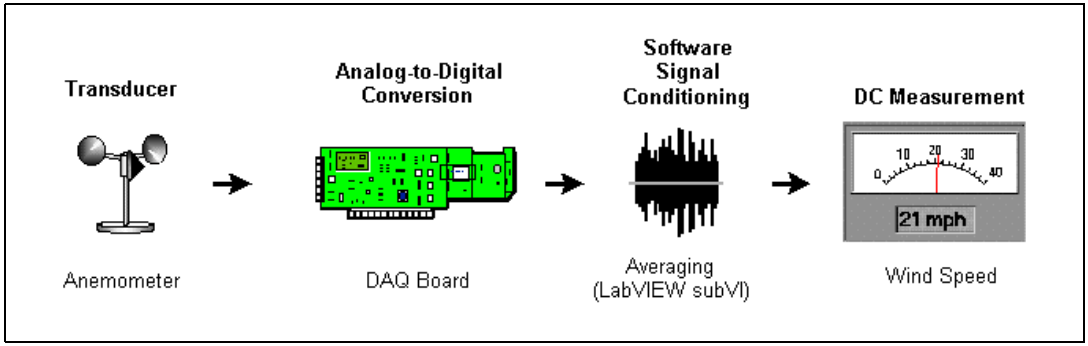


Figure 4-6. DAQ System for Measuring Wind Speed with Averaging

Figure 4-7 shows what the actual wind speed might look like over time. Due to gusting winds, the speed values look noisy. Notice that our earlier wind speed reading of 29 mph is a peak speed, but may give the impression

that the wind is holding at 29 mph. A better representation might be to take the average speed over a short period of time.

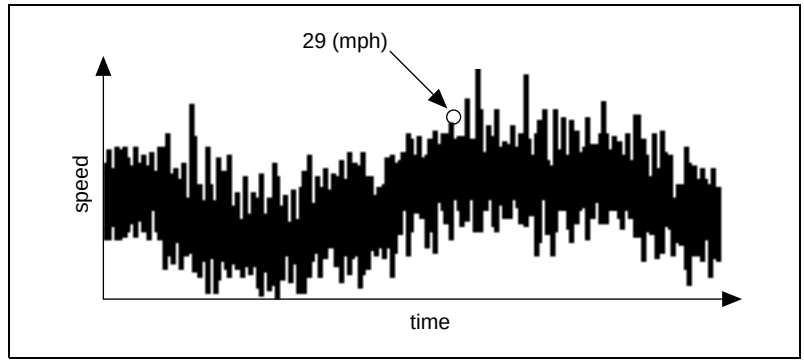


Figure 4-7. Wind Speed

Because you average in software, the hardware wiring for this approach does not change. It is the same as in Figure 4-3. The block diagram in Figure 4-8 shows the software to measure an average wind speed if you are using DAQ Named Channels. Again, the DAQ Named Channel remembers information about the device, channel, gains, and scaling. Notice that the DAQ subVI of this example differs from the *Single-Point Acquisition Example* in that it acquires a waveform instead of a single value. The **number of samples** and **sample rate** inputs define the waveform of data acquired. For example, if you set the **number of samples** to 1000 and the **sample rate** to 500 (samples/sec), it takes two seconds to acquire the 1000 points. The waveform of data from AI Acquire Waveform is then wired to the Mean subVI. The Mean subVI returns the average wind speed for two seconds of time.

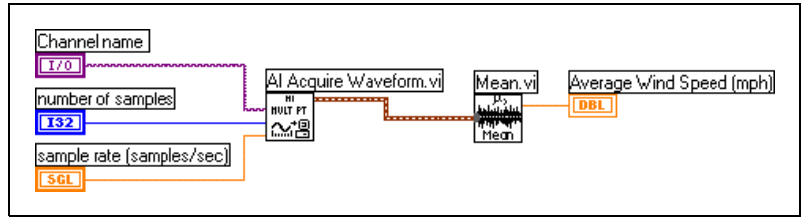


Figure 4-8. Average Wind Speed Using DAQ Named Channels

One common reason for averaging is to eliminate 50 or 60 Hz powerline noise. The oscillating magnetic field around powerlines can induce noise voltages on unshielded transducer wiring. Because powerline noise is sinusoidal, the average over one period is zero. If you use a scan rate that is

an integer multiple of the noise and average data for an integer multiple of periods, the line noise is eliminated. One example that works for both 50 and 60 Hz is to scan at 300 scans per second and then average 30 points. Notice that 300 is an integer multiple of both 50 and 60. One period of the 50 Hz noise is 300/50=6 points. One period of the 60 Hz noise is 300/60=5 points. Averaging 30 points is an integer multiple of both periods, so you can ensure that you average whole periods.

How to Measure AC Voltage

The early days of high-voltage electricity were dominated by DC applications. The constant nature of DC made it easy to measure voltage, current, and power. The power formulas developed for DC are the following:

$$P = I^2 \cdot R$$

and

$$P = \frac{V^2}{R}$$

where P is power (watts), I is current (amps), R is resistance (ohms), and V is voltage (Volts DC).

Today most power lines deliver alternating current (AC) for home, lab, and industrial applications. Alternating waveforms continuously increase, decrease, and reverse polarity on a repetitive basis. This means the voltage, current, and power are not constant values. However, it is useful to measure voltage, current, and power such that a load connected to a 120 VAC source develops the same amount of power as that same load connected to a 120 VDC source. For this reason, V_{rms} (root mean square) was developed. With RMS, the power laws shown above work for AC. For sinusoidal waveforms:

$$V_{rms} = \frac{V_{peak}}{\sqrt{2}}$$

Since voltmeters read V_{rms}, the 120 VAC of a typical U.S. wall outlet actually has a peak value of about 170 V.

LabVIEW makes it easy to measure V_{rms}. Figure 4-9 shows the data acquisition system for measuring V_{rms}.

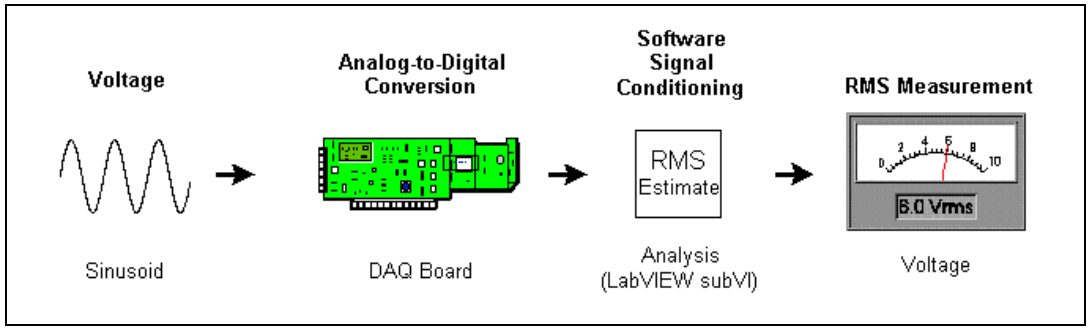


Figure 4-9. Data Acquisition System for V_{rms}

Figure 4-10 shows what the actual sinusoid signal might look like.

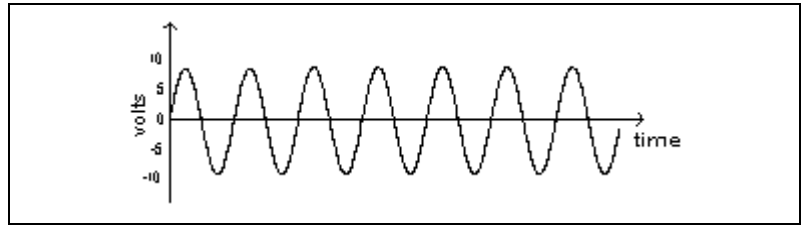


Figure 4-10. Sinusoidal Voltage

The block diagram in Figure 4-11 shows the software to measure V_{rms} if you are using DAQ Named Channels.

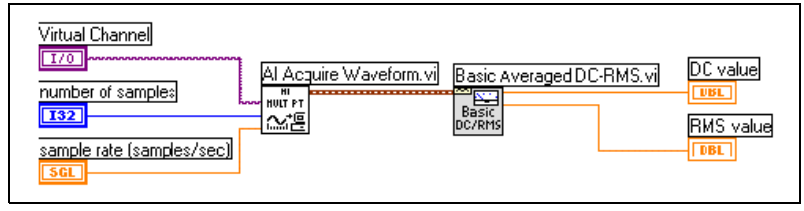


Figure 4-11. V_{rms} Using DAQ Named Channels

The DAQ subVI AI Acquire Waveform acquires a waveform. The **number of samples** and **sample rate** define the waveform. The Basic Averaged DC-RMS VI takes the waveform and estimates the RMS and DC components. For a sinusoidal waveform centered about zero, this subVI returns V_{rms}. For a sinusoidal waveform offset from zero, the **DC value**

returns the DC shift and the **RMS value** returns V_{rms} as if the waveform were centered about zero. One advantage of using the Basic Averaged DC-RMS VI is that it can make good estimations with the least amount of data. According to the Nyquist Theorem you must acquire at a rate at least twice as fast as the signal being acquired in order to get reliable frequency data. However, V_{rms} is not concerned with frequency data. It is related to the shape of the waveform. Typically, to get a good idea of a waveform shape, you must acquire at five to ten times the rate of the waveform. The advantage of the Basic Averaged DC-RMS VI is that it makes a good estimation even when acquiring at only three times the frequency of the waveform.

This same AC voltage measurement can be made using an instrument. Figure 4-12 shows the acquisition system for this measurement. In this case, a stand-alone instrument is shown. However, this could also be an instrument board that plugs directly into a PC.

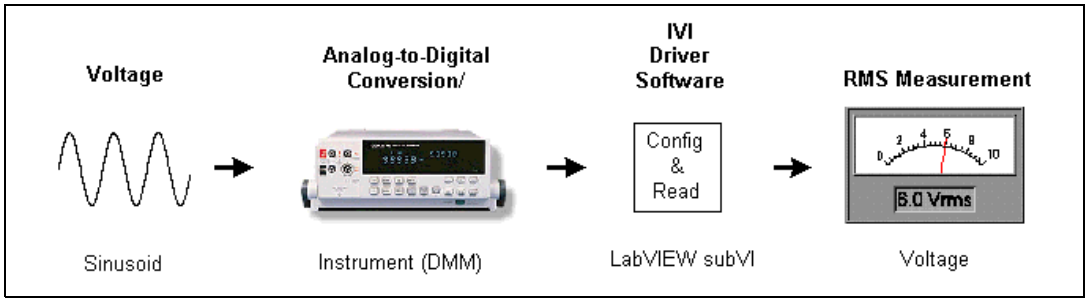


Figure 4-12. Instrument Control System for V_{rms}

Figure 4-13 shows the block diagram to measure V_{rms} using the IVI class driver VIs. In this example, the instrument is first initialized using a logical name to create a session. Next the instrument is configured for the desired measurement, in this case AC Volts. After configuration, the measurement reading is taken. Finally the session is closed.

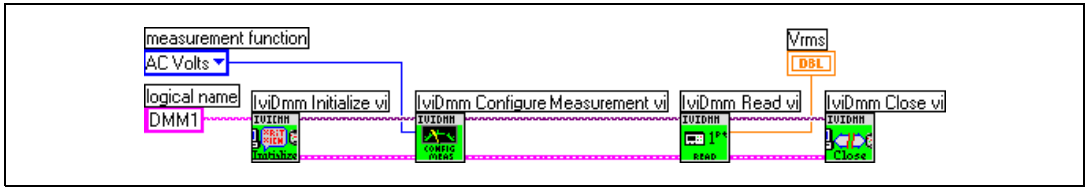


Figure 4-13. V_{rms} Using an Instrument

How to Measure Current

The 4-20 mA loop has been an industry standard for many years. It is popular because it couples a wide dynamic range with a live zero of 4 mA for open circuit detection in a system that does not produce sparks. Other advantages include a variety of compatible hardware, a long distance up to 2000 feet, and low cost. The 4-20 mA loop has a variety of uses including digital communications, control applications, and reading remote sensors. This section describes how to measure current in order to read a remote sensor.

In this example, you measure current in order to read the fluid level in a tank. Figure 4-14 shows a data acquisition system that could be used to do this.

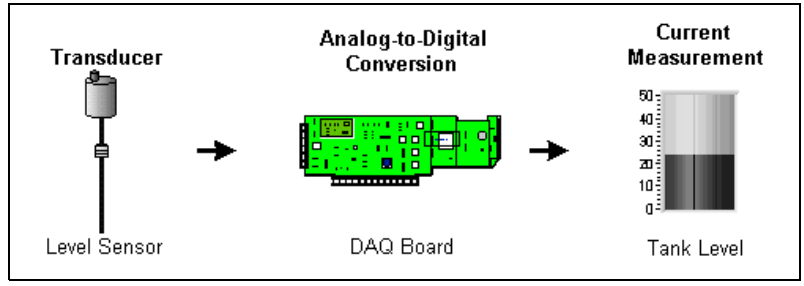


Figure 4-14. Data Acquisition System for Current

Since MIO-type DAQ devices cannot directly measure current, the voltage is read across a precision resistor used in series with the current loop circuit. Figure 4-15 shows the current loop wiring diagram.

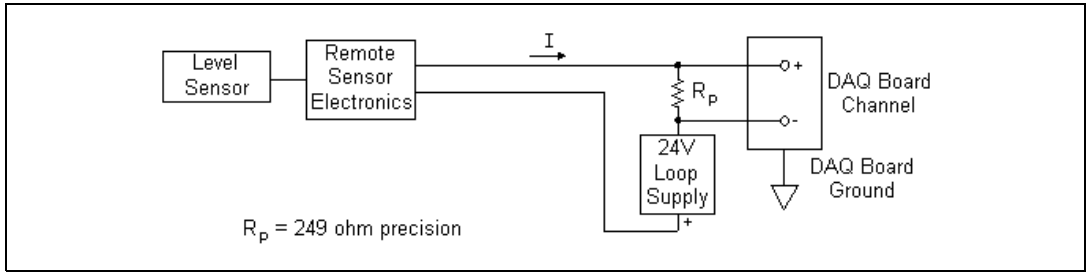


Figure 4-15. Current Loop Wiring

The purpose of this 4-20 mA current loop is for the sensor to transmit a signal in the form of a current. In the diagram, the Level Sensor and Remote Sensor Electronics are typically built into a single unit. An external 24 VDC supply powers the sensor. The current is regulated by the sensor and represents the value of whatever parameter the sensor might measure, in this case the fluid level in a tank. The DAQ device reads the voltage drop across the 249Ω resistor R_p. Then Ohm's Law is used to derive the current:

$$I_{(mA)} = \frac{V_{(Volts)}}{R_{p(Kohms)}}$$

Because the current is 4-20 mA, and R_p is 249Ω, V ranges from 0.996 V to 4.98 V. This is within the range that DAQ devices can read. While the above equation is useful for calculating the current, the current typically represents a physical quantity you want to measure. In this example, the level sensor measures 0 to 50 feet. This means 4 mA represents 0 feet and 20 mA represents 50 feet. Assuming this to be a linear relationship, it can be described by the graph and equation shown in Figure 4-16 where L is the tank level and I is the current.

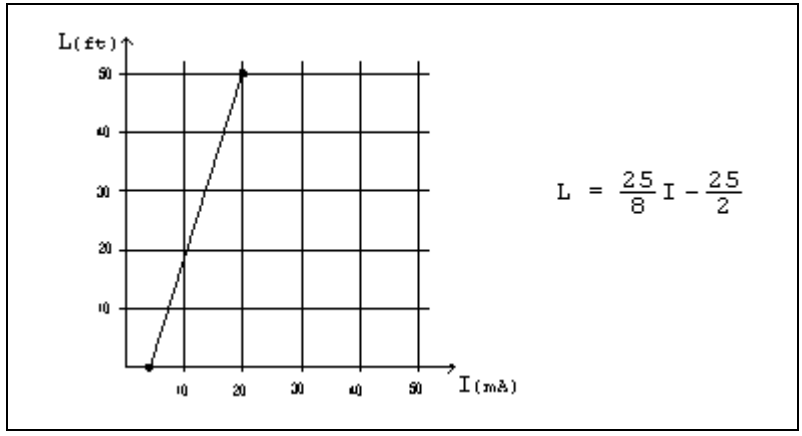


Figure 4-16. Linear Relationship between Tank Level and Current

Using the Ohm's Law equation and substituting 0.249 for the value of R_p, you can derive L in terms of our measured voltage:

$$L = \frac{25 \cdot V}{8 \cdot 0.249} - \frac{25}{2}$$

The above equation can be implemented on the block diagram as shown in Figure 4-17.

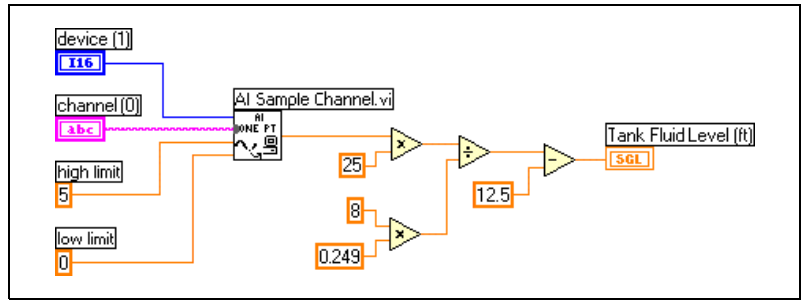


Figure 4-17. Measuring Fluid Level Without DAQ Named Channels

Alternatively, a DAQ Named Channel configured in the DAQ Channel Wizard can handle this scaling. In this case, the LabVIEW diagram is simplified to that of Figure 4-18.

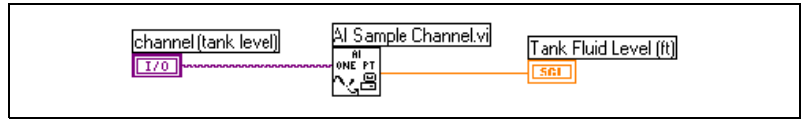


Figure 4-18. Measuring Fluid Level Using DAQ Named Channels

How to Measure Resistance

It is simple to use either the NI 4050 or NI 4060 DMM to measure resistance. Figure 4-19 shows an instrument control system to measure resistance.

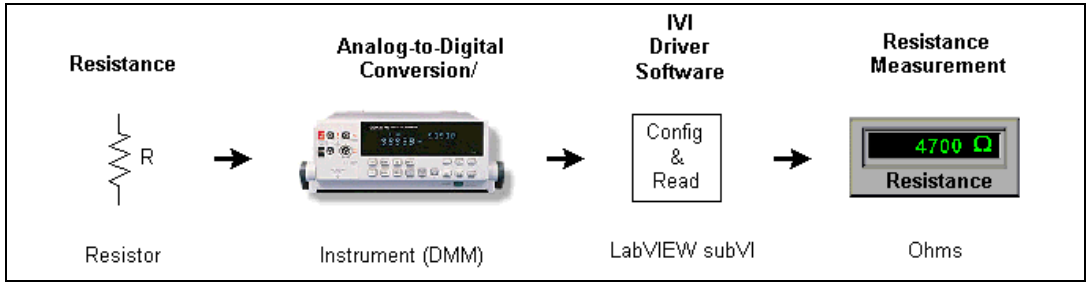


Figure 4-19. Instrument Control System for Resistance

Figure 4-20 shows the LabVIEW diagram to measure resistance using the IVI class driver VIs. Notice that this diagram is similar to Figure 4-13. The difference is the measurement function has been changed to 2-wire resistance.

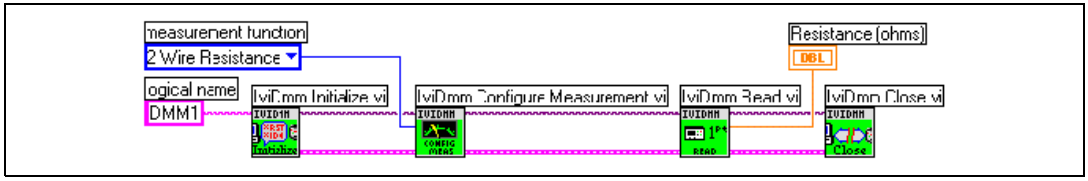


Figure 4-20. Measuring Resistance Using an Instrument

How to Measure Temperature

A thermocouple is formed when two dissimilar metals come in contact with each other, and a temperature related voltage is produced. Because they are inexpensive, easy to use, and easy to obtain, thermocouples are commonly used in science and industry. This section examines a simple approach to measuring temperature using a thermocouple. Refer to Application Note 043, *Measuring Temperature with Thermocouples – a Tutorial,* for more information about measuring temperature using a thermocouple. This application note can be found on our web site at `zone.ni.com/appnotes.nsf/`

In this example, you will learn how to measure a single temperature value following the diagram of Figure 4-21.

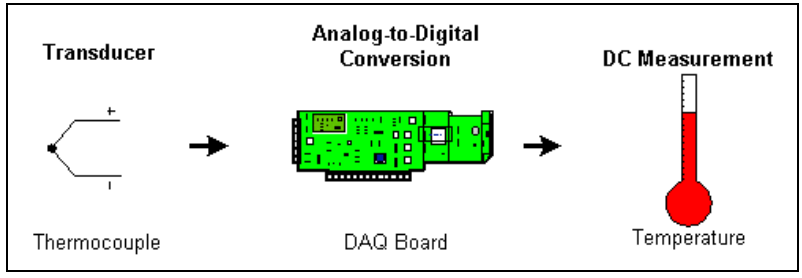


Figure 4-21. Simple Temperature System

Figure 4-22 shows a typical wiring diagram for a thermocouple. Notice that the resistor, R, is only used if the thermocouple is not grounded at any other point. If, for example, the thermocouple tip were already grounded, using R would cause a ground loop and result in erroneous readings.

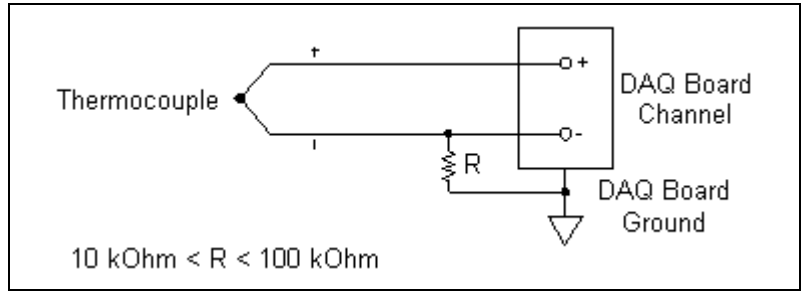


Figure 4-22. Thermocouple Wiring

Figure 4-23 shows the block diagram needed to measure the temperature if you are using DAQ Named Channels. In this case, the DAQ Named Channel handles all gain, linearization, and cold-junction compensation.

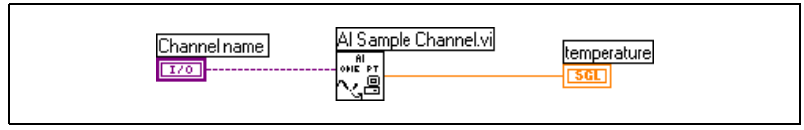


Figure 4-23. Measuring Temperature Using DAQ Named Channels

If you do not want to use DAQ Named Channels to measure temperature, you must write a VI that determines the gain needed for your temperature range, read the thermocouple voltage, read the cold-junction voltage, and convert all this information into a temperature. Refer to the Single Point Thermocouple Measurement VI located in `examples\daq\solution\transduc.llb` for an example of how to do this. Refer to the Single Point RTD Measurement VI located in `examples\daq\solution\transduc.llb` for an example of how to measure temperature using an RTD.

Example Oscilloscope Measurements

This section discusses how to take measurements that are typical of an oscilloscope. The examples show how to use an MIO-type DAQ device or an instrument to take these measurements.

How to Measure Maximum, Minimum, and Peak-to-Peak Voltage

This example assumes you have some type of signal that changes over time. Figure 4-24 shows what your measurement system might look like.

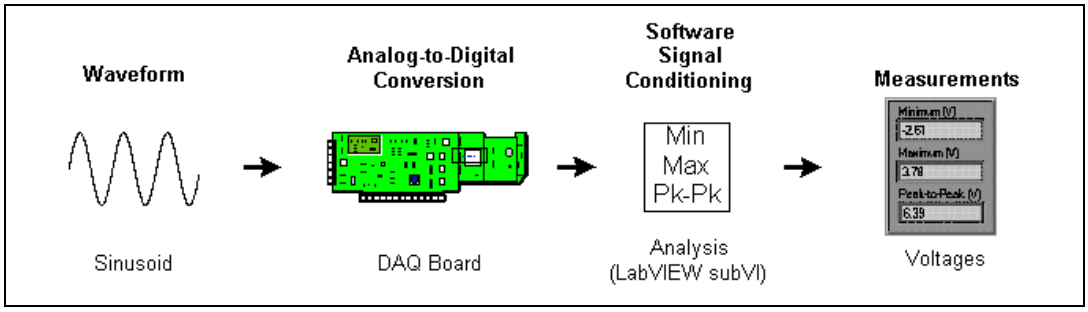


Figure 4-24. Data Acquisition System for Minimum, Maximum, Peak-to-Peak

For this measurement, your signal might typically be repetitive, but does not have to be in order to read the maximum, minimum, and peak-to-peak values. The peak-to-peak value is the maximum voltage swing (maximum – minimum). Figure 4-25 shows the LabVIEW diagram to take these measurements.

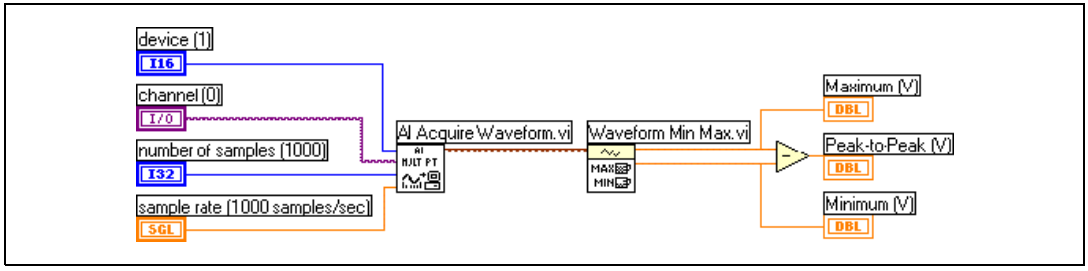


Figure 4-25. Measuring Minimum, Maximum, and Peak-to-Peak Voltages

AI Acquire Waveform VI is called to scan data from one channel of the DAQ device. The acquired waveform is passed to Waveform Min Max VI, which returns the minimum and maximum values of the waveform. The difference of these values is the peak-to-peak voltage.

A peak-to-peak voltage measurement can also be made using an instrument. Figure 4-26 shows the acquisition system for this measurement. In this case, a stand-alone oscilloscope is shown. However, this could also be an instrument board that plugs directly into a PC.

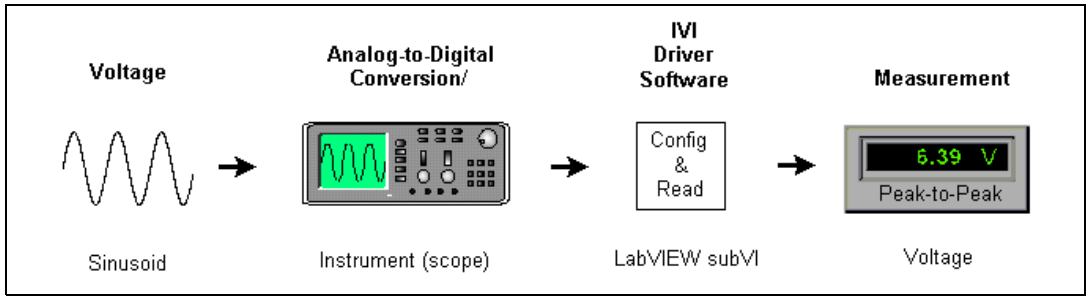


Figure 4-26. Instrument Control System for Peak-to-Peak Voltage

Figure 4-27 shows the LabVIEW diagram to measure peak-to-peak voltage using the IVI class driver VIs.

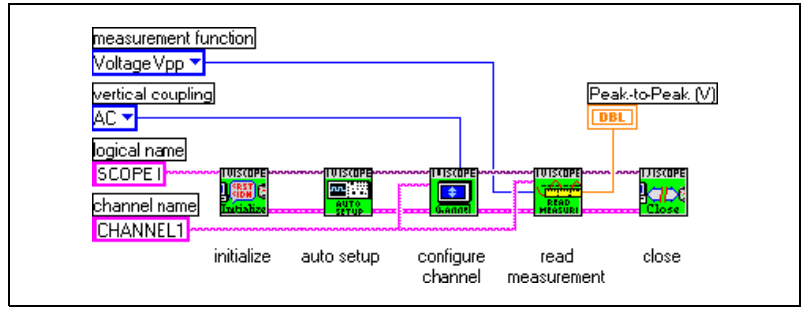


Figure 4-27. Measuring Peak-to-Peak Voltage Using an Instrument

The example shown in Figure 4-27 uses the following VIs in order:

1. The IviScope Initialize VI initializes the scope and creates a session.
2. The IviScope Auto Setup [AS] VI senses the input signal and automatically configures many instrument settings.
3. The IviScope Configure Channel VI sets the coupling to AC. This removes the DC component of the signal.
4. The IviScope Read Waveform Measurement [WM] VI reads the peak-to-peak voltage.
5. The IviScope Close VI closes the session and releases resources.

How to Measure Frequency and Period of a Repetitive Signal

Measuring Frequency and Period Example

For this example, you need to have a repetitive signal. Our measurement system is similar to that of Figure 4-24, except that the analysis is to measure frequency. To get reasonable results, be aware of the Nyquist Theorem, which states that the highest frequency that can be accurately represented is half the sampling rate. This means that if you want to measure the frequency of a 100 Hz signal, you will need a sampling rate of at least 200 S/s. In practice, sampling rates of five to ten times the expected frequencies are used. Figure 4-28 shows the block diagram to take this measurement. Once the frequency has been determined, the period of the signal is simply the inverse of the frequency.

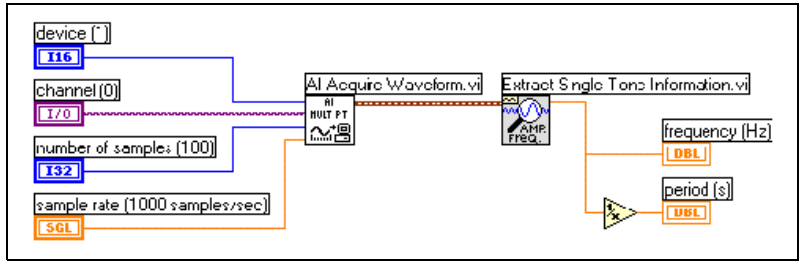


Figure 4-28. Measuring Frequency and Period

In addition to sample rate, you need to determine the number of samples to acquire. In general, more is better, but there are two things to consider. First, a minimum of three cycles of the signal must be sampled. That means in the 100 Hz example, if the sample rate is 500 S/s, you would need to collect at least 15 points. This is because you are sampling about five times faster than our signal frequency. That means you sample about 5 points per cycle of the signal. Because you need data from 3 cycles you get $5 \times 3 = 15$ samples. Second, the number of points you collect determines the number of frequency "bins" your data will fall into. With more bins, the frequency you measure might fit into one bin rather than several bins. The size of each bin is the sampling rate divided by the number of points collected. If you sample at 500 S/s and collect 100 points, you have bins at 5 Hz intervals. The Extract Single Tone Information VI used in this example uses data from the three dominant bins to determine the frequency. One rule of thumb is to sample 5 to 10 times faster than your expected signal, and to acquire 10 or more cycles.

Frequency also can be measured using an instrument. The instrument control system setup is the same as Figure 4-26. Figure 4-29 shows the block diagram for this measurement. Notice that this is like Figure 4-27 except the measurement function and output are frequency. Because frequency measurement is inherent to the instrument, the frequency value is not calculated in LabVIEW. Rather, it is simply returned by the instrument.

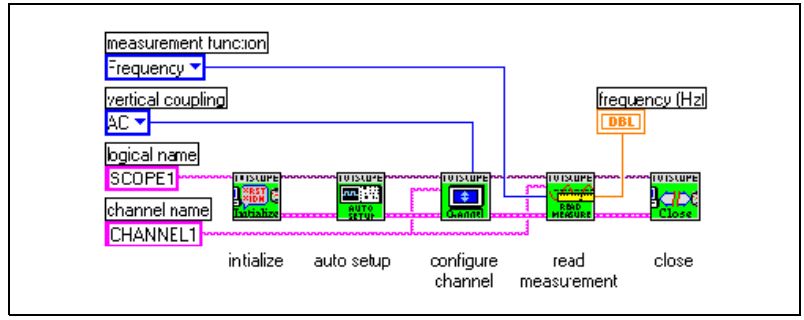


Figure 4-29. Measuring Frequency Using an Instrument

Measuring Frequency and Period with Filtering Example

As shown in the *Measuring Frequency and Period Example*, the Nyquist frequency is the bandwidth of the sampled signal and is equal to half the sampling frequency. But what happens to other frequency components that might be mixed in with the signal you are trying to measure? Frequency components below the Nyquist frequency simply appear as they are. A frequency component above the Nyquist frequency appears aliased between 0 and the Nyquist frequency. The aliased component is the absolute value of the difference between the actual component and the closest integer multiple of the sampling rate. For example, if you have a signal with a component at 800 Hz, and you sample at 500 S/s, that component appears aliased at

$$|800 - (2 \cdot 500)| = 200Hz$$

One way to eliminate aliased components is to use an analog hardware filter prior to digitizing and analyzing for frequency information. Refer to Chapter 16, *Digital Filtering*, for more information about hardware filtering. If you want to do all of your filtering in software, you must first sample at a rate fast enough to correctly represent the highest frequency component contained in your signal. For this example, with the highest component at 800 Hz, the minimum sample rate is 1600 Hz. In practice, a

sampling rate of five to ten times faster than 800 Hz should be used. Suppose now that the frequency you are trying to measure is around 100 Hz. You can use a lowpass Butterworth filter with a cutoff frequency (f_c) set to 250 Hz. This filters out frequencies above 250 Hz and pass frequencies below 250 Hz. Figure 4-30 shows a lowpass filter.

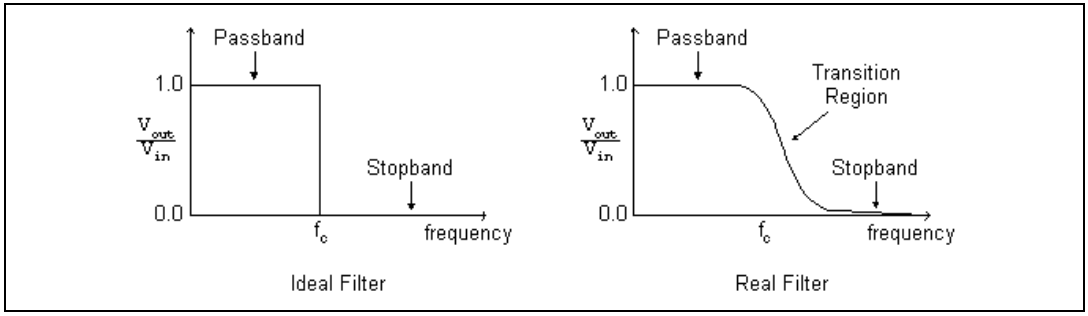


Figure 4-30. Lowpass Filter

The **Ideal Filter** is what you want. All frequencies above the Nyquist are rejected. The **Real Filter** is what you might actually be able to accomplish with a Butterworth filter. The pass band is where V_{out}/V_{in} is close to 1. The stop band is where V_{out}/V_{in} is close to 0. In between is the transition region, where frequencies are gradually attenuated.

Figure 4-31 shows the block diagram to filter before measuring frequency. Notice the Digital IIR Filter VI and the IIR filter specifications.

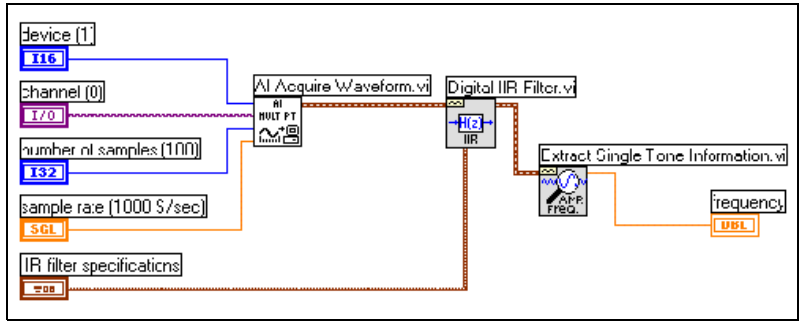


Figure 4-31. Measuring Frequency after Filtering

Figure 4-32 shows what the IIR filter specifications look like on the front panel. This is where you choose the design parameters for your filter.

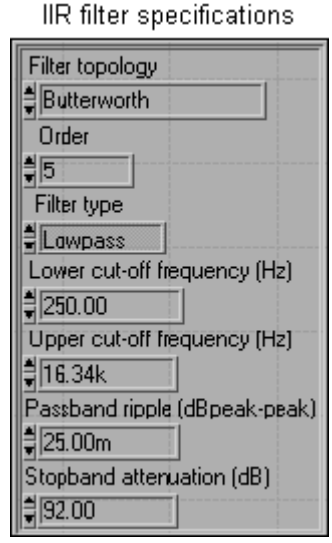


Figure 4-32. Front Panel IIR Filter Specifications

In this example, the fifth order lowpass Butterworth filter is used with a cutoff frequency of 250 Hz. The order determines how steep the transition region will be. A higher order yields a steeper transition. However, a lower order decreases both computation time and error. In the case of our chosen filter, the **Upper cut-off frequency**, **Passband ripple**, and **Stopband attenuation** inputs are ignored. Refer to Chapter 16, *Digital Filtering*, for more information about software filtering.

Frequency with software filtering can also be measured using an instrument. The instrument control system setup is the same as Figure 4-26. Figure 4-33 shows the block diagram for this measurement. Notice that the IVI subVIs called are like those of Figure 4-27. The only difference is that IviScope Read Waveform Measurement [WM] VI has been replaced with IviScope Read Waveform VI in order to read an array of data. The outputs of this subVI are built into a waveform data type. The Digital IIR Filter VI and Extract Single Tone Information VI are used as discussed earlier to determine the frequency.

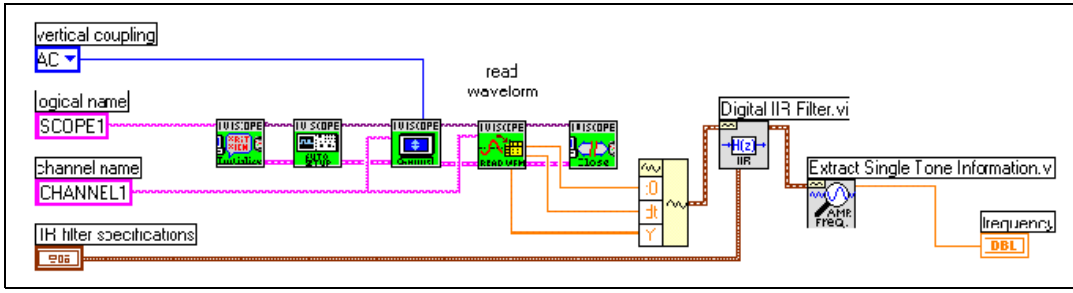


Figure 4-33. Measuring Frequency after Filtering Using an Instrument

Part II

DAQ Basics

This part contains information you need to use LabVIEW for data acquisition.

Part II, *DAQ Basics*, contains the following chapters:

- Chapter 5, *Introduction to Data Acquisition in LabVIEW*, contains all the information you should know before you start learning about data acquisition with LabVIEW.
- Chapter 6, *Analog Input*, contains basic information about acquiring data with LabVIEW, including acquiring a single point or multiple points, triggering your acquisition, and using outside sources to control acquisition rates.
- Chapter 7, *Analog Output*, contains basic information about generating data with LabVIEW, including generating a single point or multiple points.
- Chapter 8, *Digital I/O*, describes basic concepts about how to use digital signals with data acquisition in LabVIEW, including immediate, handshaked, and timed digital I/O.
- Chapter 9, *SCXI—Signal Conditioning*, contains basic information about setting up and using SCXI modules with your data acquisition application, special programming considerations, common SCXI applications, and calibration information.
- Chapter 10, *High-Precision Timing (Counters/Timers)*, describes the different ways you can use counters with your data acquisition application, including generating a pulse or pulses; measuring pulse width, frequency, and period; counting events and time; and dividing frequencies for precision timing.

5

Introduction to Data Acquisition in LabVIEW

This chapter explains background information about data acquisition using National Instruments DAQ hardware and software.

Basic LabVIEW Data Acquisition Concepts

This section explains how data acquisition works with LabVIEW. Before you start building your data acquisition (DAQ) application, you should know some of the following basic LabVIEW DAQ concepts:

- Where to find common DAQ examples
- Where to find the DAQ VIs in LabVIEW
- How the DAQ VIs are organized
- Polymorphic DAQ VIs
- VI parameter conventions
- Default and current value conventions
- The Waveform Control
- Channel, port, and counter addressing
- Limit settings
- Other DAQ VI parameters
- How DAQ VIs handle errors
- Organization of analog data

Finding Common DAQ Examples

Refer to the examples in `examples\daq` for examples of many common applications involving data acquisition in LabVIEW.

There are two ways to locate specific DAQ examples. One way is to run the DAQ Solution Wizard. The other way is to run the *Search Examples Help*. You can launch either tool directly from the **LabVIEW** dialog box.

Finding the Data Acquisition VIs in LabVIEW

The **Functions»Data Acquisition** palette contains six subpalettes that contain the different classes of DAQ VIs. The DAQ VIs are classified as follows:

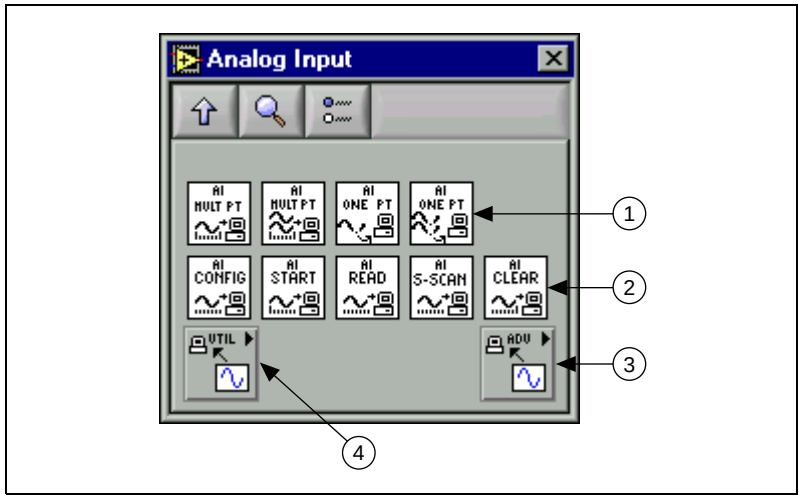
- Analog Input VIs
- Analog Output VIs
- Digital I/O VIs
- Counter VIs
- Calibration and Configuration VIs
- Signal Conditioning VIs

DAQ VI Organization

Most of the DAQ VI subpalettes arrange the VIs in different levels according to their functionality. You can find some of the following four levels of DAQ VIs within the DAQ VI subpalettes:

- Easy VIs
- Intermediate VIs
- Utility VIs
- Advanced VIs

Figure 5-1 shows an example of a DAQ subpalette that contains all of the available levels of DAQ VIs.



1	Easy Analog Input VIs	3	Advanced Analog Input VIs
2	Intermediate Analog Input VIs	4	Analog Input Utility VIs

Figure 5-1. Analog Input VI Palette Organization

Easy VIs

The Easy VIs perform simple DAQ operations and typically reside in the first row of VIs in the DAQ palettes. You can run these VIs from the front panel or use them as subVIs in basic applications.

You need only one Easy VI to perform each basic DAQ operation. Unlike intermediate- and advanced-level VIs, Easy VIs automatically alert you to errors with a dialog box that allows you to stop the execution of the VI or to ignore the error.

The Easy VIs usually are composed of Intermediate VIs, which are in turn composed of Advanced VIs. The Easy VIs provide a basic interface with only the most commonly used inputs and outputs. For more complex applications, use the intermediate- or advanced-level VIs to achieve more functionality and better performance.

Intermediate VIs

The Intermediate VIs have more hardware functionality and efficiency in developing applications than the Easy VIs. The Intermediate VIs contain groups of Advanced VIs, but they use fewer parameters and do not have some of the more advanced capabilities.

Intermediate VIs give you more control over error-handling than the Easy VIs. With each VI, you can check for errors or pass the error cluster on to other VIs.

Note Most LabVIEW data acquisition examples shown in this manual are based on the Intermediate VIs.

Utility VIs

The Utility VIs, found in many of the LabVIEW DAQ palettes, are also intermediate-level VIs and thus have more hardware functionality and efficiency in developing your application than the Easy VIs.

Advanced VIs

The Advanced VIs are the lowest-level interface to the DAQ driver. Very few applications require the use of the Advanced VIs. Advanced VIs return the greatest amount of status information from the DAQ driver. Use the Advanced VIs when the Easy or Intermediate VIs do not have the inputs necessary to control an uncommon DAQ function.

Polymorphic DAQ VIs

Some of the DAQ VIs are polymorphic. This means that they accept or return data of various types. For example, the Easy Analog Input VIs can return data as either a waveform or an array of scaled values. By default the Polymorphic Analog Input VIs return data as a waveform. To change the return type, right-click on the VI icon and choose **Select Type** from the shortcut menu, as shown in Figure 5-2. The easy analog output VIs can accept data as either a waveform or an array of scaled values. The polymorphic analog output VIs adapt to the type of data that is connected to them.

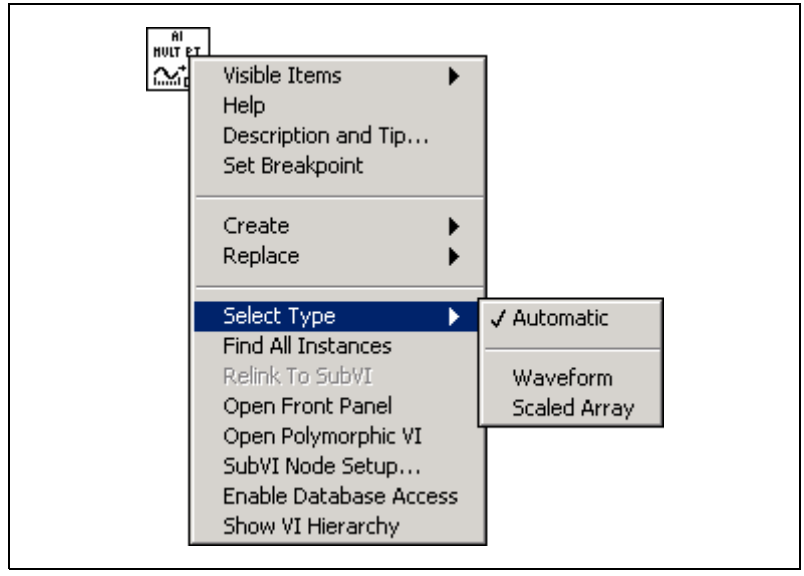


Figure 5-2. Polymorphic DAQ VI Shortcut Menu

VI Parameter Conventions

In each LabVIEW DAQ VI front panel or **Context Help** window, the appearance of the control and indicator labels denotes the importance of that parameter. Control and indicator names shown in bold are required and must be wired to a node on the block diagram for your application to run. Parameter names that appear in plain text are optional and are not necessary for your program to run. You rarely need to use the parameters with labels in square brackets ([ ]). Remember that these conventions apply only to the information in the **Context Help** window and on the front panel. Default input values appear in parentheses to the right of the parameter names.

Figure 5-3 illustrates these **Context Help** window parameter conventions for the AI Read One Scan VI. As the window text for this VI indicates, you must wire the **device** (if you are not using channel names), **channels**, **error in**, and **iteration** input parameters and the **waveform data** and **error out** output parameters. To pass error information from one VI to another, connect the **error out** cluster of the current VI to the **error in** cluster of the next VI. The **coupling & input config**, **input limits**, **output units**, and **number of AMUX boards** input parameters are optional.

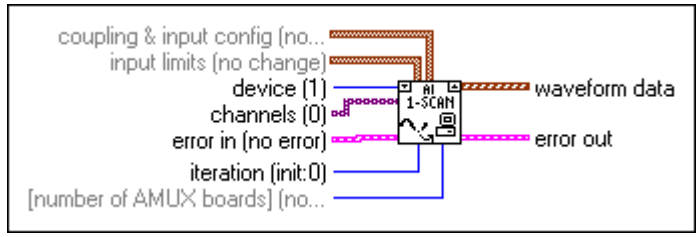


Figure 5-3. LabVIEW Context Help Window Conventions

Default and Current Value Conventions

To use the DAQ VIs, you need to know the difference between a default input, a default setting, and a current setting. A default input is the default value of a front panel control. When you do not wire an input to a terminal of a VI, the default input for the control associated with that terminal passes to the driver. The **Help** window shows the default inputs inside parentheses beside the parameter names. A default setting is a default parameter value recorded in the driver. The current setting is the value of a control at any given moment. The current setting of a control is the default setting until you change the value of the control.

In many cases, a control input defaults to a certain value (most often 0), which means you can use the current setting. For example, the default input for a parameter can be **do not change the current setting**, and the current setting can be **no AMUX-64T boards**. If you change the value of such a parameter, the new value becomes the current setting.

The Waveform Control

LabVIEW represents a waveform with the waveform control parameter by default. A 1D array of waveform controls represents multiple waveforms. The VIs, functions, and front panel objects you use to build VIs that acquire, analyze, and display analog measurements accept or return waveform data by default.

The waveform control contains data associated with a single waveform, including data values and timing information.

The waveform control passes the waveform components to the VIs and functions you use to build measurement applications. Use the waveform VIs and functions to extract and edit the components of the waveform.

The waveform control can be customized to have many appearances. The DAQ VIs use two common appearances, one to reflect single-point waveforms and one to reflect multi-point waveforms. Figure 5-4 shows these appearances of the waveform control.

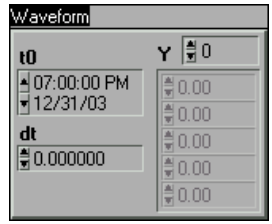


Figure 5-4. Waveform Control

Waveform Control Components

The waveform control is a special cluster of components that includes time-domain, uniformly sampled waveform information only. Use the waveform functions to access and manipulate individual components.

Start Time (t_0)

The start time is the start time of the first point in the waveform. Use the start time to synchronize plots on a multi-plot waveform graph or to determine delays between waveforms. This value is not used by the Analog Output VIs.

Delta t (dt)

Delta *t* is the time between successive data points in the waveform. This value is not used by the Analog Output VIs.

Waveform Data (Y)

The waveform data is a 1D array of double-precision numbers that represents the waveform. Generally, the number of data values in the array corresponds directly to the number of scans taken from a data acquisition device. Refer to the *Using the Waveform Control* section for more information about acquiring and generating waveform data.

Attributes

The attribute component may be used to contain other information about the waveform. You can set attributes with the Set Waveform Attribute function and read attributes with the Get Waveform Attribute function.

Using the Waveform Control

There are a number of LabVIEW VIs and primitives that accept, operate on, and/or return waveforms. In addition, you can connect the waveform control wires directly to many LabVIEW controls, including the graph, chart, numeric, and numeric array controls.

The block diagram in Figure 5-5 acquires a waveform from a channel on a data acquisition device, sends it through a Butterworth filter, and plots the resulting waveform on a graph.

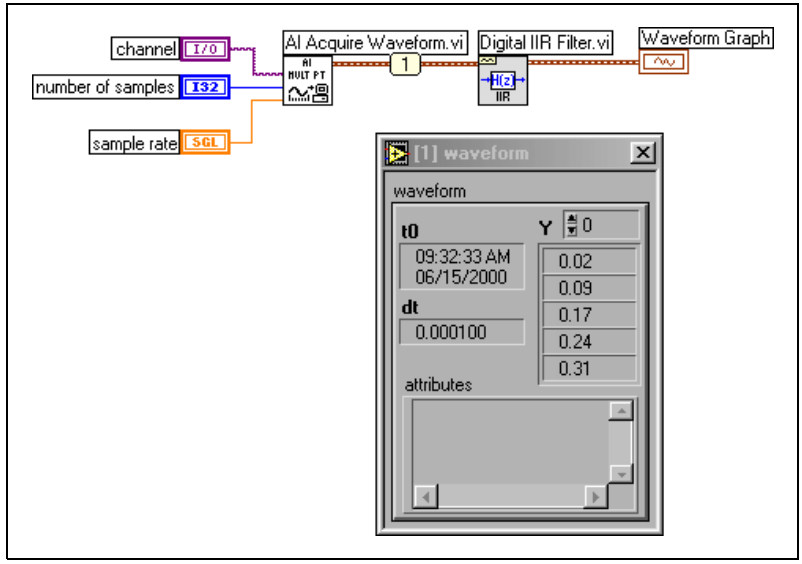


Figure 5-5. Using the Waveform Data Type

The AI Acquire Waveform VI takes a number of samples from a channel at a specified scan rate at a particular time. The VI returns a waveform. The probe displays the components of the waveform, which includes the time the acquisition began (**t0**), the time between successive data points (**dt**), and the data of a waveform acquired with each scan (**Y**). The Waveform FIR Filter VI accepts the array of waveforms and automatically filters the data (**Y**) of each waveform. The waveform graph then plots and displays the waveform.

The waveform control can also be used with single point acquisitions as shown in Figure 5-6. The AI Sample Channel VI takes a single sample from a channel and returns a single-point waveform. The waveform contains the value read from the channel and the time the channel was read. The chart and the temperature controls accept the waveform and display its data. The

Get Waveform Components function is used to extract the start time from the waveform.

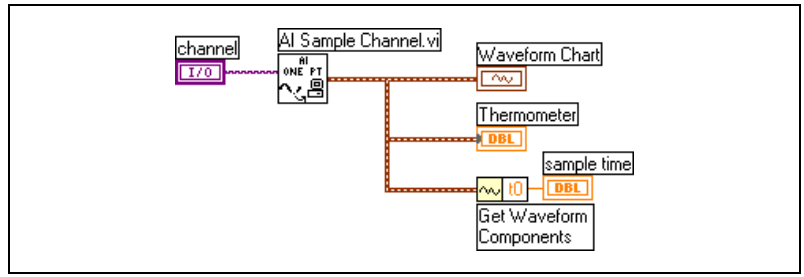


Figure 5-6. Single-Point Example

The waveform control can be used with analog output as shown in Figure 5-7. The Sine Waveform VI generates a sine waveform. The AO Generate Waveform VI sends the waveform to the device.

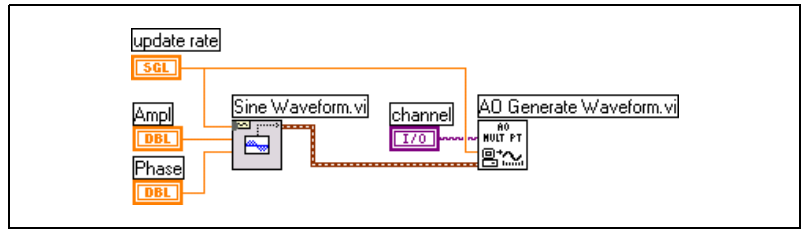


Figure 5-7. Using the Waveform Control with Analog Output

Extracting Waveform Components

Use the Get Waveform Components function to extract and manipulate the components of a waveform you generate. The VI in Figure 5-8 uses the Get Waveform Components function to extract the waveform data. The Negate function negates the waveform data and plots the results to a graph.

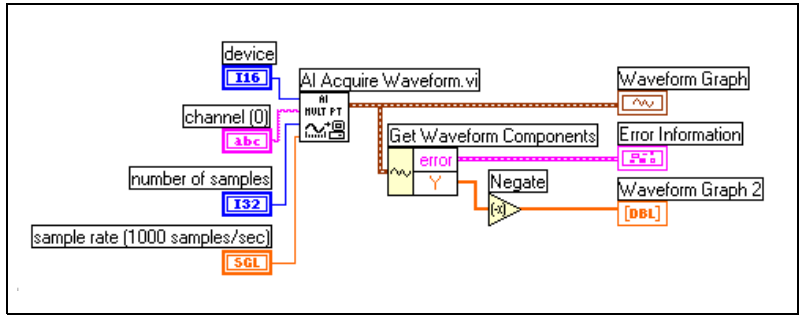


Figure 5-8. Extracting Waveform Components

Waveform Data on the Front Panel

On the front panel, use the **Waveform** control, available on the **Controls»I/O** palette, or a **Waveform Graph**, available on the **Controls»Graph** palette, to represent waveform data.

Use the **Waveform** control to manipulate the **t0**, **dt**, and **Y** components of the waveform or display those components as an indicator. Refer to the *Waveform Control Components* section for more information about each component.

When you wire a waveform to a graph, the **t0** component is the initial value on the x-axis. The number of scans acquired and the dt component determine the subsequent values on the x-axis. The data elements in the Y component comprise the points on the plot of the graph.

If you want to let a user control a certain component, such as the **dt** component, create a front panel control and wire it to the appropriate component in the Build Waveform function.

The VI in Figure 5-9 continuously acquires 10,000 scans from a data acquisition device at a scan rate of 1,000 scans per second, which began at 7:00 p.m. You can control the Δt of the waveform.

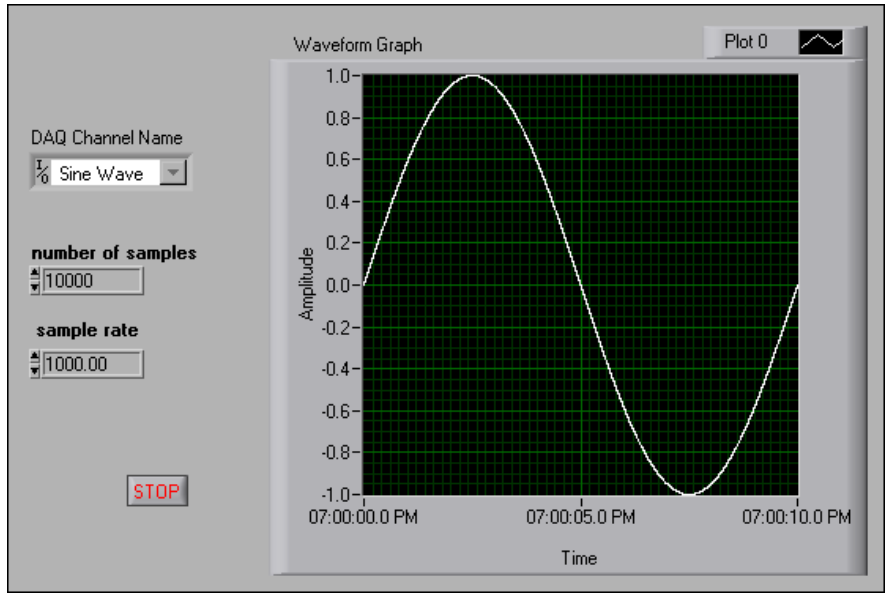


Figure 5-9. Waveform Graph

The waveform data (**Y**) is plotted on the graph. The trigger time (**t0**) is 7:00:00 P.M. and is the first point on the x-axis. The Δt of the waveform (**dt)** is 1.00 millisecond, so the 10,000 scans are distributed over 10 seconds with the last data value plotted at 7:00:10 P.M.

Channel, Port, and Counter Addressing

The Analog Input and Analog Output VIs have a **channels** parameter where you can specify the channels from which the VIs read or to which they write. The Digital Input and Output VIs have a similar parameter, called **digital channel list**, and the equivalent value is called **counter list** for the Counter VIs.

Note To simplify the explanation of channel addressing concepts, the **channels**, **digital channels**, and **counter list** parameters are all referred to as **channels** in this section.

Each channel you specify in the **channels** parameter becomes a member of a group. For each group, you can acquire or generate data on the channels listed in the group. VIs scan (during acquisition) or update (during generation) the channels in the same order they are listed. To erase a group, pass an empty **channels** parameter and the group number to the VI or assign a new **channels** parameter to the group. You can change groups only at the Advanced VI level.

DAQ Channel Name Control

The **channels** parameter in the Analog and Digital VIs is a DAQ Channel Name control. If you configured your channels in the DAQ Channel Wizard, the controls menu lists the channel names of your configured channels. Select a channel name from the menu or type a channel name or number into the string area of the control.

Channel Name Addressing

If you use the DAQ Channel Wizard to configure your analog and digital channels, you can address your channels by name in the **channels** parameter in LabVIEW. **channels** can be an array of strings or, as with the Easy VIs, a scalar string control, as shown in Figure 5-10. If you have a **channels** array, you can use one channel entry per array element, specify the entire list in a single element, or use any combination of these two methods. If you enter multiple channel names in **channels**, you must configure all of the channels in the list for the same DAQ device. If you configure channels with names of `temperature` and `pressure`, both of which are measured by the same DAQ device, you can specify a list of channels in a single element by separating them by commas—for example, `temperature,pressure`. If you configure channels with names of `temp1`, `temp2`, and `temp3`, you can specify a range of channels by separating them with a colon, for example, `temp1:temp3`. In specifying channel names, spelling and spaces are important, but case is not.

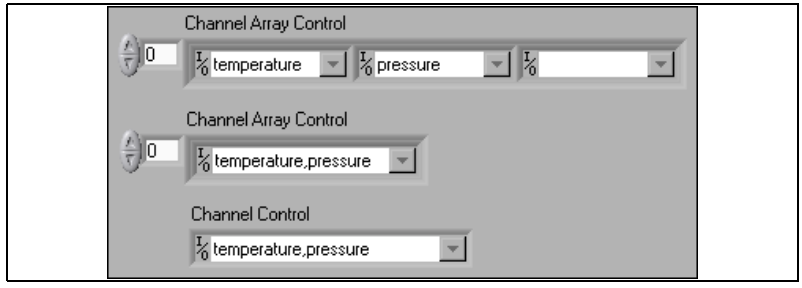


Figure 5-10. Channel Controls

When using channel names, you do not need to wire the **device**, **input limits**, or **input config** input parameters. LabVIEW always ignores the **device** input when using channel names. LabVIEW configures your hardware in terms of your channel configuration.

In addition, LabVIEW orders and pads the channels specified in **channels** as needed according to any special device requirements.

Channel Number Addressing

If you are not using channel names to address your channels, you can address your channels by channel numbers in the **channels** parameter. The **channels** can be an array of strings or, as with the Easy VIs, a scalar string control. If you have a **channels** array, you can use one channel entry per array element, specify the entire list in a single element, or use any combination of these two methods. For instance, if your channels are 0, 1, and 2, you can specify a list of channels in a single element by separating the individual channels by commas—for example, `0, 1, 2`. Or you can specify the range by separating the first and last channels with a colon—for example, `0:2`. Figure 5-11 shows several ways you can address channels `0`, `1`, and `2`.

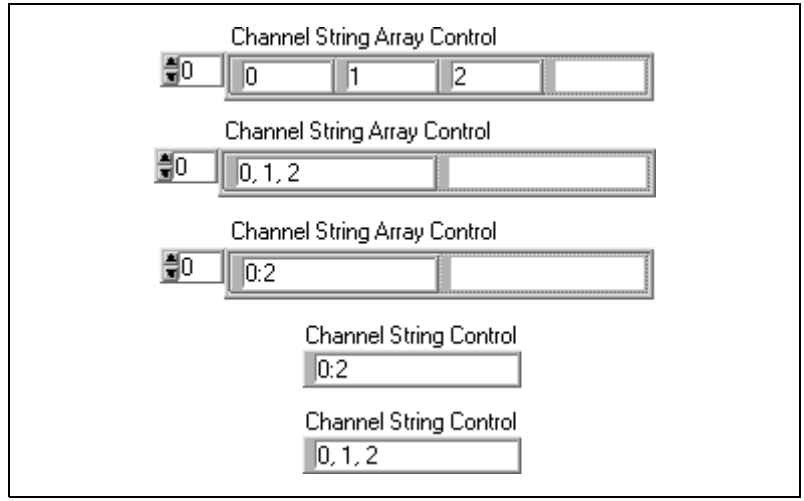


Figure 5-11. Channel String Array Controls

Some Easy and Advanced Digital VIs and Intermediate Counter VIs allow only one port or counter to be specified.

Limit Settings

Limit settings are the maximum and minimum values of the analog signal(s) you are measuring or generating. The pair of limit setting values can be unique for each analog input or output channel. For analog input applications, the limit setting values must be within the range for the device.

Each pair of limit setting values forms a cluster. Analog output limits have a third member, the reference source. For simplicity, LabVIEW refers to

limit settings as a pair of values. LabVIEW uses an array of these clusters to assign limits to the channels in your **channel** string array.

If you use the DAQ Channel Wizard to configure your analog input channels, the physical unit you specified for a particular channel name is applied to the limit settings. For example, if you configured a channel in the DAQ Channel Wizard to have physical units of **Deg C**, the limit settings are treated as limits in degrees Celsius. LabVIEW configures your hardware to make the measurement in terms of your channel name configuration. Unless you need to overwrite your channel name configuration, do not wire this input. Allow LabVIEW to set it up for you.

If you are not using the DAQ Channel Wizard, the default unit applied to the limit settings is usually volts, although the unit applied to the limit settings can be volts, current, resistance, or frequency, depending on the capability and configuration of your device.

The default range of the device, set in the configuration utility or by LabVIEW according to the channel name configuration in the DAQ Channel Wizard, is used whenever you leave the limit settings terminal unwired or you enter `0` for your upper and lower limits.

As explained in the *Channel, Port, and Counter Addressing* section, LabVIEW uses an array of strings to specify which channels belong to a group. Also, remember LabVIEW lists one channel to as many as all of the device channels in a single array element in the **channels** array. LabVIEW also assigns all the channels listed in a **channels** array element the same settings in the corresponding **limit settings** cluster array element. Figure 5-12 illustrates one such case.

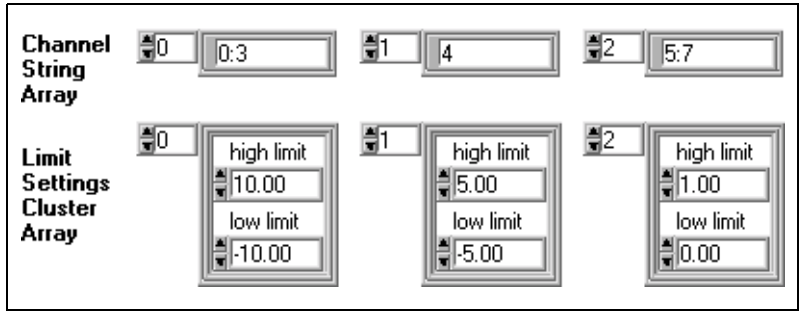


Figure 5-12. Limit Settings, Case 1

In this example, channels 0, 1, 2, and 3 are assigned limits of 10.00 to –10.00. Channel 4 has limits of 5.00 to –5.00. Channels 5, 6, and 7 have limit settings of 1.00 to 0.00.

If the **limit settings** cluster array has fewer elements than the **channel** string array, LabVIEW assigns any remaining channels the limit settings contained in the last entry of the **limit settings** cluster array. Figure 5-13 illustrates this case.

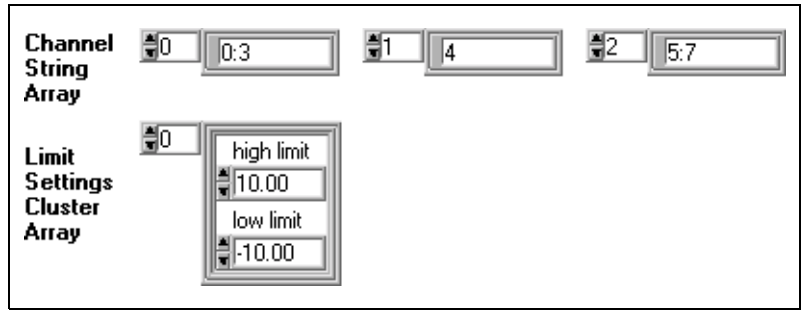


Figure 5-13. Limit Settings, Case 2

In this example, channels 0, 1, 2, and 3 have limits of 10.00 to –10.00. There are more channels left, but the **limit settings** cluster array is exhausted. Therefore, the remaining channels (4, 5, 6, and 7) are also assigned limits of 10.00 to –10.00.

The Easy Analog Input VIs have only one pair of input limits. This pair forms a single cluster element. If you specify the default limit settings, all channels scanned with these VIs have identical limit settings. The Easy Analog Output VIs do not have limit settings. All the Intermediate VIs, both analog input and output, have the **channels** array and the **limit settings** (or **input limits**) cluster array on the same VI. Assignment of limits to channels works exactly as described above.

In analog applications, you not only specify the range of the signal, you also must specify the range and the polarity of the device. A unipolar range is a range containing either positive or negative values, but never both. A bipolar range is a range that has both positive and negative values. When a device uses jumpers or DIP switches to select its range and polarity, you must enter the correct jumper setting in the configuration utility.

In DAQ hardware manuals and in the configuration utility, you may find reference to the concept of gain. Gain is the amplification or attenuation of a signal. Most National Instruments DAQ devices have programmable gains (no jumpers), but some SCXI modules require the use of jumpers or DIP switches. Limit settings determine the gain for all DAQ devices used

with LabVIEW. However, for some SCXI modules, you must enter the gain in the configuration utility.

Other DAQ VI Parameters

The **device** input on analog I/O, digital I/O, and counter VIs specifies the number the DAQ configuration software assigned to your DAQ device. Your software assigns a unique number to each DAQ device. The **device** parameter usually appears as an input to the configuration VIs. Another common configuration VI parameter, **task ID**, assigns your specific I/O operation and device a unique number that identifies it throughout your program flow.

Some DAQ VIs perform either the device configuration or the I/O operation, while other DAQ VIs perform both configuration and the operation. Some of the VIs that handle both functions have an **iteration** input. When your VI has the **iteration** set to 0, LabVIEW configures the DAQ device and then performs the specific I/O operation. For iteration values greater than 0, LabVIEW uses the existing configuration to perform the I/O operation. You can improve the performance of your application by not configuring the DAQ device every time an I/O operation occurs. Typically, you should wire the **iteration** input to an iteration terminal in a loop as shown in Figure 5-14.

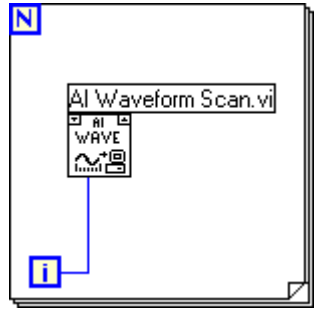


Figure 5-14. Wiring the **iteration** Input

Wiring the **iteration** input this way means the device is configured only on the first I/O operation. Subsequent I/O operations use the existing configuration.

Error Handling

Each Easy VI contains an error handling VI. A dialog box appears immediately if an error occurs in an Easy VI.

Each Intermediate and Advanced VI contains an **error in** input cluster and an **error out** output cluster, as shown in Figure 5-15. The clusters contain a Boolean indicator that indicates whether an error occurred, the **code** for the error, and **source** or the name of the VI that returned the error. If **error in** indicates an error, the VI passes the error information to **error out** and does not execute any DAQ functions.

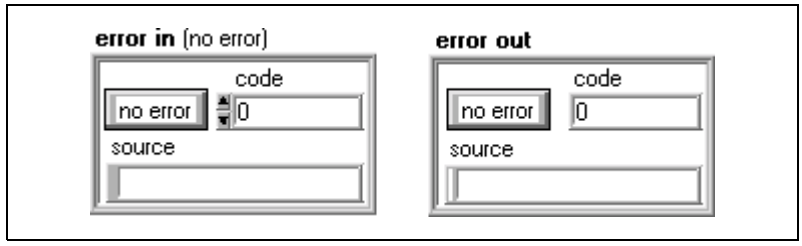


Figure 5-15. LabVIEW Error In and Error Out Error Clusters

Organization of Analog Data

If you acquire data from more than one channel multiple times, the data may be returned as an array of waveforms. Each waveform represents a separate channel in the waveform array. Refer to the *The Waveform Control* section for more information about waveforms.

The data also may be returned as a two-dimensional (2D) array. This section explains the organization of analog data as a 2D array.

If you were to create a 2D array and label the index selectors on a LabVIEW front panel, the array might look like Figure 5-16.

Figure 5-16. Example of a Basic 2D Array

The two vertically arranged boxes on the left are the row and column index selectors for the array. The top index selects a row, and the bottom index selects a column.

The Analog Input VIs organize their data by columns. Each column holds data from one channel, so selecting a column selects a channel. Selecting a row selects a scan of data. This ordering method is often called column major order. If you were to label your index selectors for a column major 2D array, the array might look like Figure 5-17.

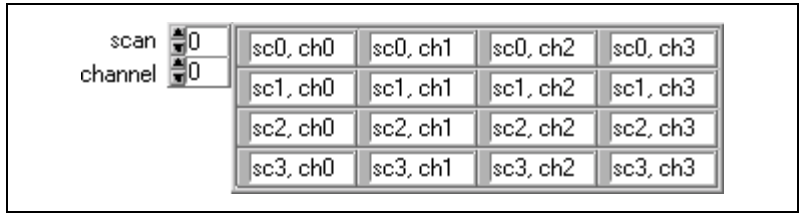


Figure 5-17. 2D Array in Column Major Order

To graph a column major order 2D array, you must configure the waveform chart or graph to treat the data as transposed by turning on this option in the graph shortcut menu.

Note This option is dimmed until you wire the 2D array to a graph. To convert the data to row major order, use the Transpose 2D Array function, available on the **Functions» Array** palette. You also can transpose the array data from the graph by right-clicking on the graph and selecting **Transpose Array** from the shortcut menu.

To extract a single channel from a column major 2D array, use the Index Array function, available on the **Functions»Array** palette. You select a column (or channel) by wiring your selection to the bottom left index input, the Index Array function produces the entire column of data as a 1D array, as shown in Figure 5-18.

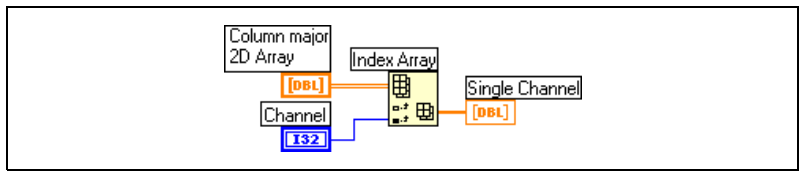


Figure 5-18. Extracting a Single Channel from a Column Major 2D Array

Analog output buffers that contain data for more than one channel are also column major 2D arrays. To create such an array, first make the data from each output channel a 1D array. Then select the Build Array function on the **Functions»Array & Cluster** palette. Add as many input terminals (rows) to the Build Array terminal as you have channels of data. Wire each 1D array to the Build Array terminal to combine these arrays into a single row

major 2D array. Then use the Transpose 2D Array function to convert the array to a column major array.

The finished array is ready for the AO Write VI, as shown in Figure 5-19.

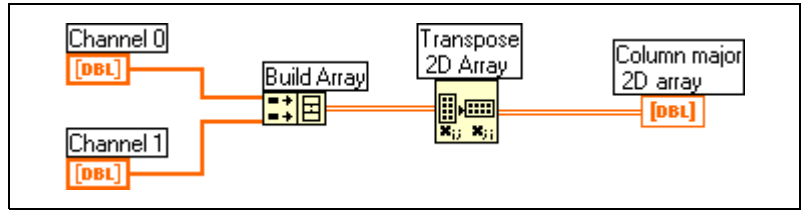


Figure 5-19. Analog Output Buffer 2D Array

Where You Should Go Next

This section directs you to the chapter in this manual best suited to answer questions about your data acquisition application. You answer a series of questions that help determine the purpose of your application. At first, the questions are general and then become more focused until you reach a reference to a specific section in the manual dealing with your type of application. The questions will guide you to the relevant sections in the manual for your particular application.

1. **What kind of Measuring Device do I use—DAQ Device or SCXI Module?**

   Are you working in an environment with a lot of EMI? If you are, you may have SCXI modules connected to your DAQ device or the parallel port of your computer. SCXI modules can filter and isolate noise from signals. They also can amplify low signals. SCXI modules expand the number of channels to acquire or generate data.

   DAQ devices are primarily used alone when extra signal conditioning is not necessary.

   If you are using a DAQ device, read question 2. If you are using SCXI, go to Chapter 9, *SCXI—Signal Conditioning*.

2. **Analog or Digital Signal Analysis?**

   Does your signal have two discrete values that are TTL signals? If so, you have a digital signal. Otherwise, you have an analog signal. The type of information you would need to know from an analog signal is the level (discrete value), shape, and frequency content.

3. Analog or Digital Signal Acquisition or Generation?

If you want to measure and analyze signals from a source outside the computer, you want to *acquire* signals. If you want to send signals to an outside instrument to control its operation, you want to *generate* signals.

If you want to acquire analog signals, go to question 5. If you want to generate analog signals, refer to question 6. If you want to acquire and generate analog signals, refer to the*Single-Point Acquisition* section in Chapter 6, *Analog Input*.

If you want to acquire or generate *digital* signals, read question 4.

4. Digital or Counter Interfacing?

Digital I/O interfaces primarily with binary operations, such as turning external equipment on or off, or sense logic states, such as the on/off position of the switch. Counters generate individual digital pulses or waves or count digital events, like how many times a digital signal rises or falls in value.

If you are performing digital I/O, refer to question 9. If you need to use counters, read question 10.

5. Single-Point or Multiple-Point Acquisition?

Do you want to acquire a signal value(s) at one time or over a period of time at a certain rate? If you measure a signal at a given instant of time, you are performing *single-point acquisition*. If you measure signals over a period of time at a certain rate, you are performing *multiple-point* or *waveform acquisition*.

If you want single-point acquisition, refer to the*Single-Point Acquisition* section in Chapter 6, *Analog Input*. If you want multiple-point acquisition, read question 7.

6. Single-Point or Multiple-Point Generation?

Are you outputting a steady (DC) signal or are you generating a changing signal at a certain rate? A constant or slowly changing signal output is called *single-point generation.* The output of a changing signal at a certain rate is called *multiple-point* or *waveform generation*.

If you want to perform single-point generation, refer to the *Single-Point Generation* section in Chapter 7, *Analog Output*. If you want multiple-point generation, refer to the *Waveform Generation (Buffered Analog Output)* section in Chapter 7, *Analog Output*.

7. Triggering a Signal or Using a Clock?

You can start an analog acquisition when a certain analog or digital value occurs by *triggering* the acquisition.

If you want to trigger an analog acquisition, refer to the *Controlling Your Acquisition with Triggers* section in Chapter 6, *Analog Input*.

8. Multiple-Point Acquisition with an Internal or External Clock?

Multiple-point or waveform acquisition can be done at a rate set by an internal DAQ device clock or an external clock. The external clock is a TTL signal produced at a certain rate.

If you want to acquire a waveform at the rate of an external signal, refer to the *Letting an Outside Source Control Your Acquisition Rate* section in Chapter 6, *Analog Input*. If not, read the *Buffered Waveform Acquisition* section in Chapter 6, *Analog Input*.

9. Immediate, Handshaked, or Timed Digital I/O?

If you want your program to read the latest digital input or immediately write a new digital output value, use non-latched (immediate) digital I/O. When a DAQ device accepts or transfers data after a digital pulse is received, it is called latched (handshaked) digital I/O. With latched digital I/O, you can store the values you want to transfer in a buffer. Only one value is transferred after each handshaked pulse. If you want to read or write digital data (patterns) at a fixed rate using a clock source, use timed digital I/O.

If you want to use non-latched (immediate) digital I/O, refer to the *Immediate Digital I/O* section in Chapter 8, *Digital I/O*. If you want to perform latched (handshaked) digital I/O, refer to the *Handshaking* section in Chapter 8, *Digital I/O*. If you want to perform timed digital I/O, refer to the *Immediate Digital I/O* section in Chapter 8, *Digital I/O*.

10. Counters—Counting or Generating Digital Pulses?

If you want to generate digital pulses from a counter at a certain rate, read the *Generating a Square Pulse or Pulse Trains* section in Chapter 10, *High-Precision Timing (Counters/Timers)*. If you want to measure the width of a digital pulse, refer to the *Measuring Pulse Width* section in Chapter 10, *High-Precision Timing (Counters/Timers)*. If you want to measure the frequency or period of a digital signal, refer to the *Measuring Frequency and Period* section in Chapter 10, *High-Precision Timing (Counters/Timers)*. If you just

want to count how many times a digital signal rises or falls, refer to the *Counting Signal Highs and Lows* section in Chapter 10, *High-Precision Timing (Counters/Timers)*. To learn how to slow the frequency of a digital signal, refer to the *Dividing Frequencies* section in Chapter 10, *High-Precision Timing (Counters/Timers)*.

6

Analog Input

This chapter explains analog input for data acquisition.

Things You Should Know about Analog Input

Engineers and scientists use data acquisition to acquire the information they need. This section describes the terms, tools, and techniques for successfully acquiring analog data.

Defining Your Signal

Analog signals can be grouped into three categories: DC, time domain, and frequency domain. Figure 6-1 illustrates which signals correspond to certain types of signal information.

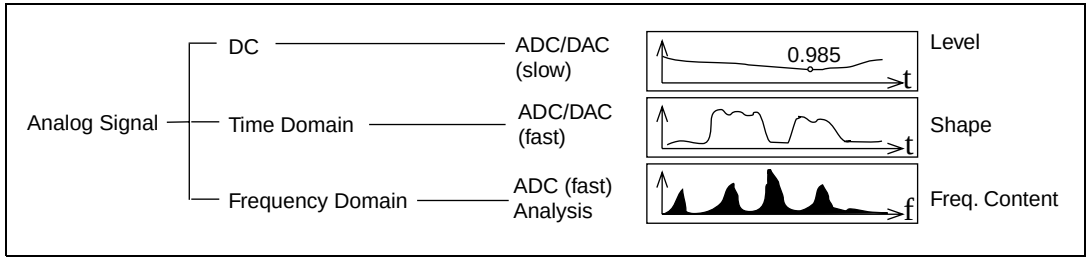


Figure 6-1. Types of Analog Signals

You must define a few more signal characteristics before you can begin measuring. For example, to what is your signal referenced? How fast does the signal vary over time?

You can treat a DC signal as a form of time domain signal. With a slowly-varying signal, you often can acquire a single point for your measurement. However, some DC signals might have noise, which varies quickly. Refer to Chapter 4, *Example Measurements*, for more information about handling noise in a DC signal by treating it as a time domain signal.

For time and frequency domain signals, you acquire several points of data at a fast scan rate. The rate you sample determines how often the analog-to-digital conversions take place. A fast sampling rate acquires more points in a given time and, therefore, can often form a better representation of the original signal than a slow sampling rate.

The sampling rate you should use depends on the types of features you are trying to find in your waveform. For example, if you are trying to detect a quick pulse in the time domain, you must sample fast enough that you do not miss the pulse. The time between successive scans must be smaller than the pulse period. If you are interested in measuring the rise time of a pulse, you must sample at an even faster rate, which depends on how quickly the pulse rises.

If you are measuring frequency characteristics of a waveform, you often do not need to sample as fast as you do for time domain measurements. According to the Nyquist Theorem, you must sample at a rate greater than twice the maximum frequency component in a signal to get accurate frequency information about that signal. This is usually not a fast enough rate to recreate the shape of the signal in the time domain, but it does record the frequency information. The frequency at one half the sampling frequency is referred to as the Nyquist frequency. Refer to the *Measuring Frequency and Period with Filtering Example* section in Chapter 4, *Example Measurements*, and the *Data Sampling* section in Chapter 11, *Introduction to Measurement Analysis in LabVIEW*, for more information about the Nyquist Theorem and the Nyquist frequency.

Signals come in two forms: *referenced* and *non-referenced* signal sources. More often, referenced sources are said to be *grounded* signals, and non-referenced sources are called *floating* signals.

Grounded Signal Sources

Grounded signal sources have voltage signals that are referenced to a system ground, such as earth or a building ground. Devices that plug into a building ground through wall outlets, such as signal generators and power supplies, are the most common examples of grounded signal sources, as shown in Figure 6-2.

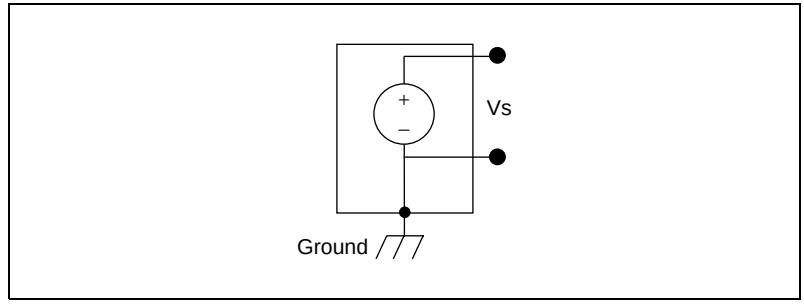


Figure 6-2. Grounded Signal Sources

Floating Signal Sources

Floating signal sources contain a signal, such as a voltage, that is not connected to an absolute reference, such as earth or a building ground. Some common examples of floating signals are batteries, battery-powered sources, thermocouples, transformers, isolation amplifiers, and any instrument that explicitly floats its output signal. Notice that in Figure 6-3 neither terminal of the floating source is connected to the electrical outlet ground.

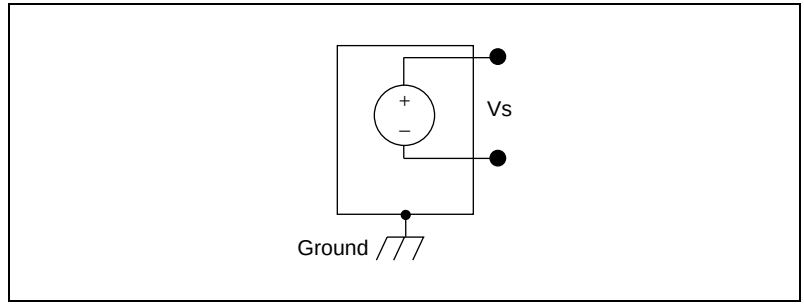


Figure 6-3. Floating Signal Sources

Now that you know how your signal is referenced, read on to learn about the different systems available to acquire these signals.

Choosing Your Measurement System

Now that you have defined your signal, you must choose a measurement system. You have an analog signal, so you must convert the signal with an analog to digital converter (ADC) measurement system, which converts your signal into information the computer can understand. Some of the issues you must resolve before choosing a measurement system are your ADC bit resolution, device range, and signal range.

Resolution

The number of bits used to represent an analog signal determines the *resolution* of the ADC. You can compare the resolution on a DAQ device to the marks on a ruler. The more marks you have, the more precise your measurements. Similarly, the higher the resolution, the higher the number of divisions into which your system can break down the ADC range, and therefore, the smaller the detectable change. A 3-bit ADC divides the range into 2^3 or 8 divisions. A binary or digital code between 000 and 111 represents each division. The ADC translates each measurement of the analog signal to one of the digital divisions. Figure 6-4 shows a sine wave digital image as obtained by a 3-bit ADC. Clearly, the digital signal does not represent the original signal adequately, because the converter has too few digital divisions to represent the varying voltages of the analog signal. By increasing the resolution to 16 bits, however, the ADC's number of divisions increases from 8 to 65,536 (2^{16}). The ADC now can obtain an extremely accurate representation of the analog signal.

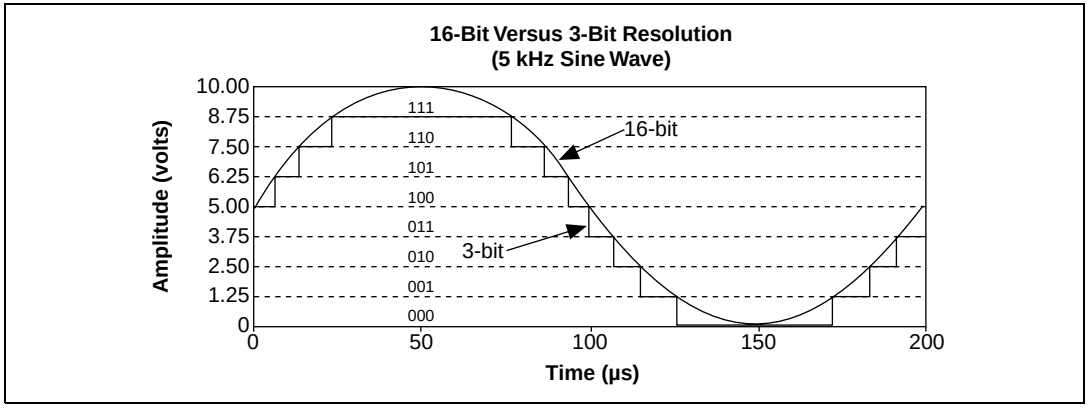


Figure 6-4. The Effects of Resolution on ADC Precision

Device Range

Range refers to the minimum and maximum analog signal levels that the ADC can digitize. Many DAQ devices feature selectable ranges, so you can match the ADC range to that of the signal to take best advantage of the available resolution. For example, in Figure 6-5, the 3-bit ADC, as shown in the left chart, has eight digital divisions in the range from 0 to 10 V. If you select a range of –10.00 to 10.00 V, as shown in the right chart, the same ADC now separates a 20 V range into eight divisions. The smallest detectable voltage increases from 1.25 to 2.50 V, and you now have a much less accurate representation of the signal.

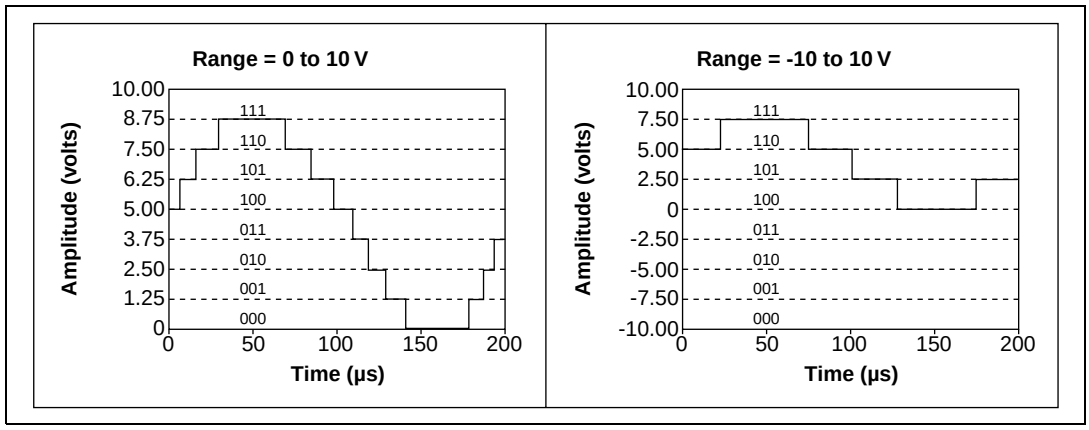


Figure 6-5. The Effects of Range on ADC Precision

Signal Limit Settings

Limit settings are the maximum and minimum values of the signal you are measuring. A more precise limit setting allows the ADC to use more digital divisions to represent the signal. Figure 6-6 shows an example of this theory. Using a 3-bit ADC and a device range setting of 0.00 to 10.00 V, Figure 6-6 shows the effects of a limit setting between 0 and 5 V and 0 and 10 V. With a limit setting of 0 to 10 V, the ADC uses only four of the eight divisions in the conversion. But using a limit setting of 0 to 5 V, the ADC now has access to all eight digital divisions. This makes the digital representation of the signal more accurate.

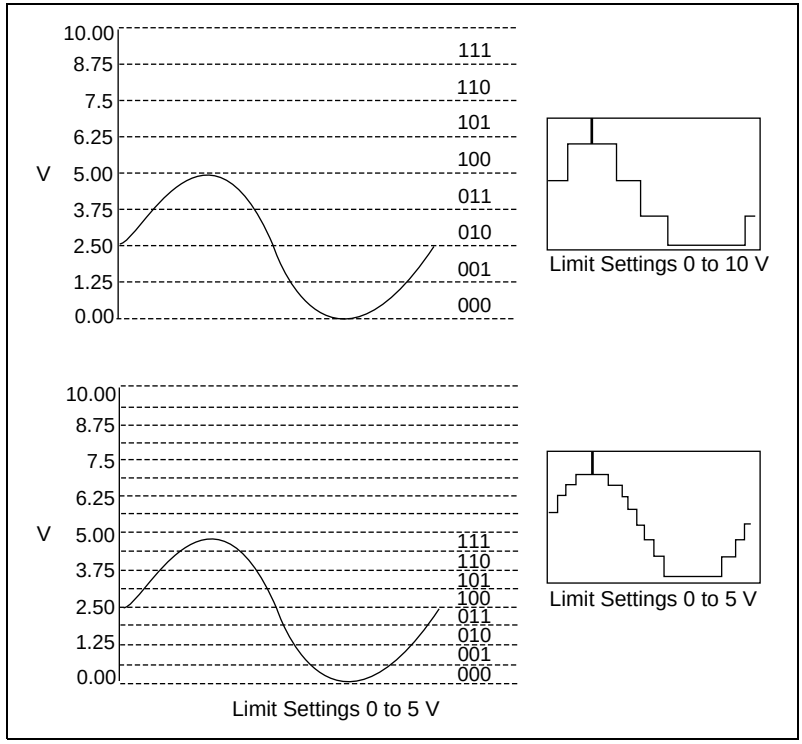


Figure 6-6. The Effects of Limit Settings on ADC Precision

Considerations for Selecting Analog Input Settings

The resolution and device range of a DAQ device determine the smallest detectable change in the input signal. You can calculate the smallest detectable change, called the *code width*, using the following formula.

$$code\ width = \frac{device\ range}{2^{resolution}}$$

For example, a 12-bit DAQ device with a 0 to 10 V input range detects a 2.4 mV change, while the same device with a –10 to 10 V input range detects only a change of 4.8 mV.

$$\frac{device\ range}{2^{resolution}} = \frac{10}{2^{12}} = 2.4\ mV$$

$$\frac{device\ range}{2^{resolution}} = \frac{20}{2^{12}} = 4.8\ mV$$

A high-resolution A/D converter provides a smaller code width given the device voltage ranges shown above.

$$\frac{device\ range}{2^{resolution}} = \frac{10}{2^{16}} = .15\ mV$$

$$\frac{device\ range}{2^{resolution}} = \frac{20}{2^{16}} = .3\ mV$$

The smaller your code width, the more accurate your measurements will be.

There are times you must know whether your signals are unipolar or bipolar. Unipolar signals are signals that range from 0 value to a positive value (for example, 0 to 5 V). Bipolar signals are signals that range from a negative to a positive value (for example, –5 to 5 V). To achieve a smaller code width if your signal is unipolar, specify that the device range is unipolar, as shown previously. If your signal range is smaller than the device range, set your limit settings to values that more accurately reflect your signal range. Table 6-1 shows how the code width of the 12-bit DAQ devices varies with device ranges and limit settings, because your limit settings automatically adjust the gain on your device.

Table 6-1. Measurement Precision for Various Device Ranges and Limit Settings (12-Bit A/D Converter)

Device Voltage Range	Limit Settings	Precision[1]
0 to 10 V	0 to 10 V	2.44 mV
	0 to 5 V	1.22 mV
	0 to 2.5 V	610 µV
	0 to 1.25 V	305 µV
	0 to 1 V	244 µV
	0 to 0.1 V	24.4 µV
	0 to 20 mV	4.88 µV
–5 to 5 V	–5 to 5 V	2.44 mV
	–2.5 to 2.5 V	1.22 mV
	–1.25 to 1.25 V	610 µV
	–0.625 to 0.625 V	305 µV
	–0.5 to 0.5 V	244 µV
	–50 to 50 mV	24.4 µV
	–10 to 10 mV	4.88 µV
–10 to 10 V	–10 to 10 V	4.88 mV
	–5 to 5 V	2.44 mV
	–2.5 to 2.5 V	1.22 mV
	–1.25 to 1.25 V	610 µV
	–1 to 1 V	488 µV
	–0.1 to 0.1 V	48.8 µV
	–20 to 20 mV	9.76 µV

[1] The value of 1 Least Significant Bit (LSB) of the 12-bit ADC. In other words, the voltage increment corresponding to a change of 1 count in the ADC 12-bit count.

Now that you know which kind of ADC to use and what settings to use for your signal, you can connect your signals to be measured. On most DAQ devices, there are three different ways to configure your device to read the signals: differential, referenced single-ended (RSE), and nonreferenced single-ended (NRSE).

Differential Measurement System

In a differential measurement system, you do not need to connect either input to a fixed reference, such as earth or a building ground. DAQ devices with instrumentation amplifiers can be configured as differential measurement systems. Figure 6-7 depicts the 8-channel differential measurement system used in the MIO series devices. Analog multiplexers increase the number of measurement channels while still using a single

instrumentation amplifier. For this device, the pin labeled AIGND (the analog input ground) is the measurement system ground.

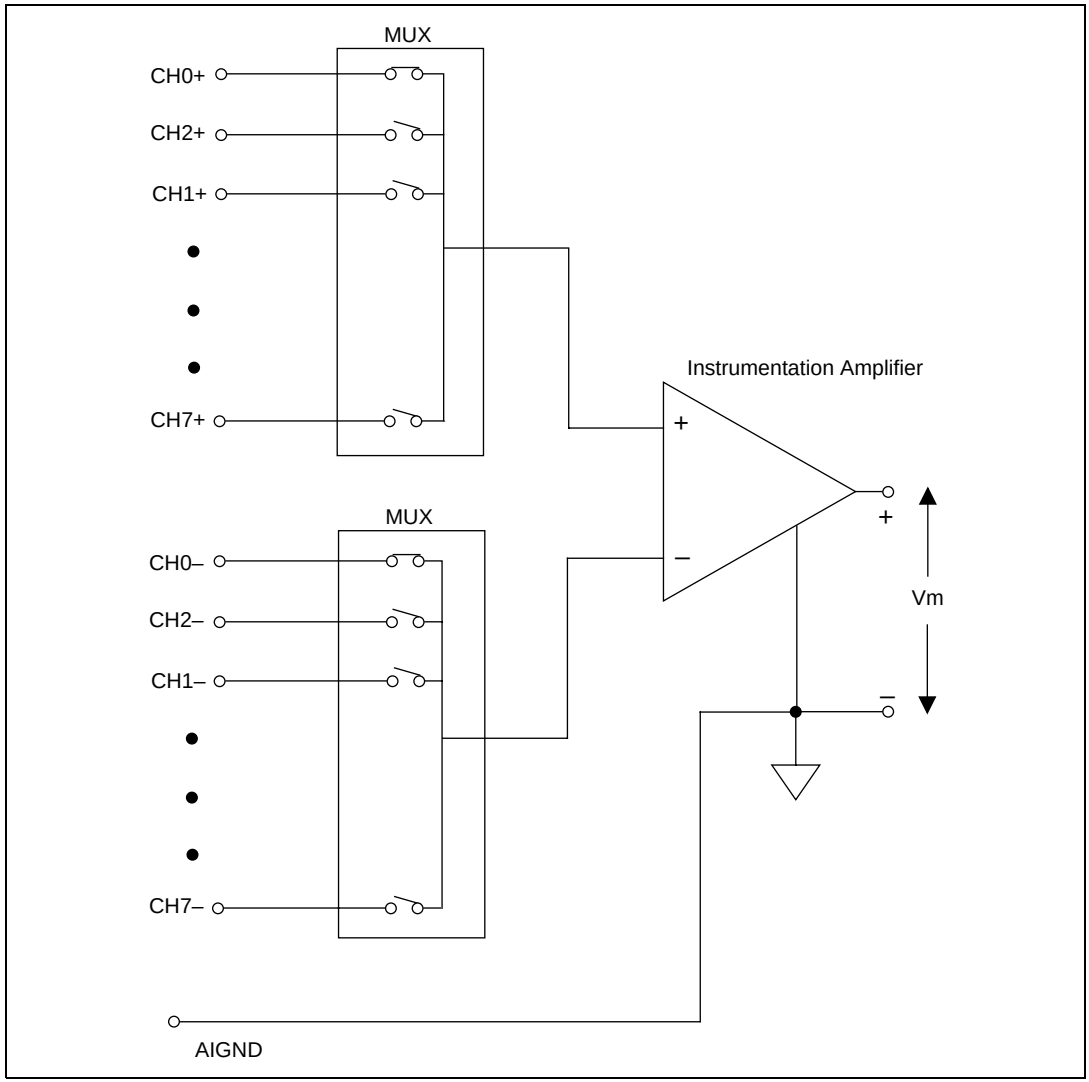


Figure 6-7. 8-Channel Differential Measurement System

In general, a differential measurement system is preferable because it rejects not only ground loop-induced errors, but also the noise picked up in the environment to a certain degree. Use differential measurement systems when all input signals meet the following criteria:

- Low-level signals (for example, less than 1 V)
- Long or non-shielded cabling/wiring traveling through a noisy environment
- Any of the input signals require a separate ground-reference point or return signal

An ideal differential measurement system reads only the potential difference between its two terminals—the positive (+) and negative (–) inputs. Any voltage present at the instrumentation amplifier inputs with respect to the amplifier ground is called a *common-mode voltage*. An ideal differential measurement system completely rejects (does not measure) common-mode voltage, as shown in Figure 6-8.

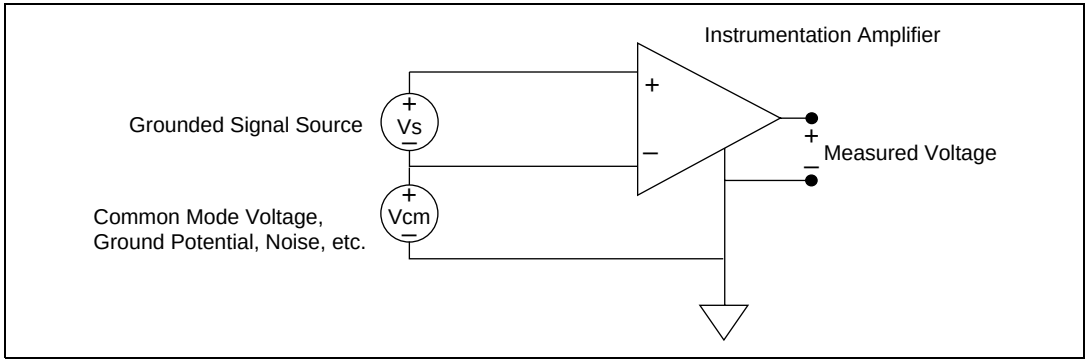


Figure 6-8. Common-Mode Voltage

Referenced Single-Ended Measurement System

An RSE measurement system is used to measure a floating signal, because it grounds the signal with respect to building ground. Figure 6-9 depicts a 16-channel RSE measurement system. You should use this measurement system only when you need a single-ended system and your device does not work with NRSE measurement.

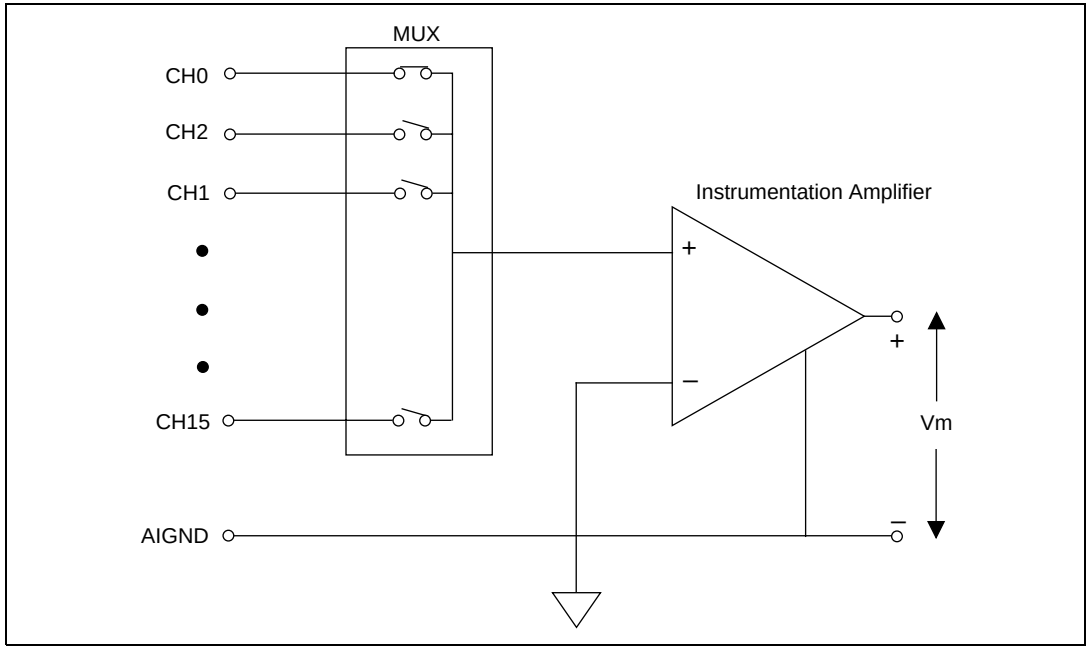


Figure 6-9. 16-Channel RSE Measurement System

Nonreferenced Single-Ended Measurement System

DAQ devices often use a variant of the RSE measurement technique, known as the NRSE measurement system. In an NRSE measurement system, all measurements are made with respect to a common reference, because all of the input signals are already grounded. Figure 6-10 depicts an NRSE measurement system where AISENSE is the common reference for taking measurements and AIGND is the system ground. All signals must share a common reference at AISENSE.

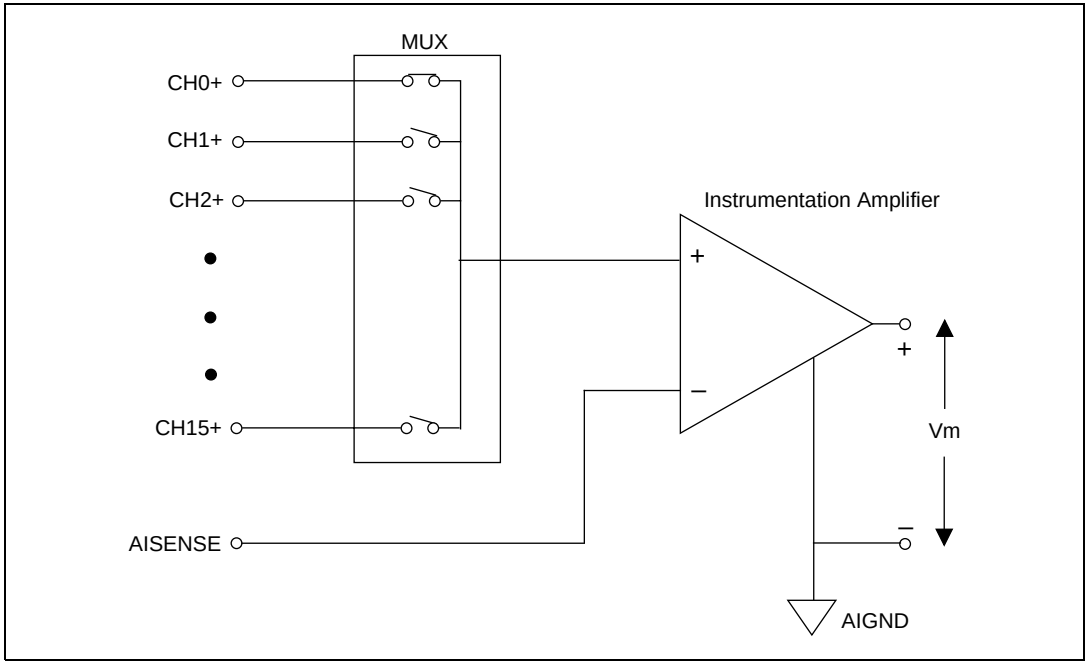


Figure 6-10. 16-Channel NRSE Measurement System

In general, a differential measurement system is preferable because it rejects not only ground loop-induced errors, but also the noise picked up in the environment to a certain degree. On the other hand, the single-ended configuration allows for twice as many measurement channels and is acceptable when the magnitude of the induced errors is smaller than the required accuracy of the data.

You can use single-ended measurement systems when all input signals meet the following criteria:

- High-level signals (normally, greater than 1 V)
- Short or properly-shielded cabling/wiring traveling through a noise-free environment (normally, less than 15 ft)
- All signals can share a common reference signal at the source

Use differential connections when your system violates any of the above criteria.

Channel Addressing with the AMUX-64T

An AMUX-64T external multiplexer accessory expands the number of analog input signals a DAQ device can measure. You can attach 1, 2, or 4 AMUX-64T accessories to a DAQ device. The number of AMUX accessories used is set in the device configuration of Measurement & Automation Explorer (**Windows**) or the NI-DAQ Configuration Utility (**Macintosh**). Every four channels on the AMUX accessory are multiplexed to one channel on the DAQ device. In LabVIEW, each onboard channel corresponds to four AMUX channels on each AMUX. For example, with one AMUX-64T, the channel string `0:1` acquires data from AMUX channels 0 through 7, and so on. With two AMUX-64T accessories, the channel string `0:1` acquires data from both AMUX accessories channels 0 through 7.

You also can acquire data from a single AMUX-64T channel by using the channel string `AMy!x`, where *x* defines the channel number and *y* defines the number of the desired AMUX (*y* = 1 if you have just one AMUX configured). For example, `AM3!8` returns channel 8 on the third configured AMUX-64T. Refer to the Acquire 1 Pt, 1 Ch via AMUX-64T VI in the `examples\daq\anlogin\anlogin.llb` for an example of acquiring data from a single AMUX-64T channel.

Important Terms You Should Know

The following are some definitions of common terms and parameters that you should remember when acquiring your data:

- A *scan* is one acquisition or reading from each channel in your channel string.
- **Number of scans to acquire** refers to the number of data acquisitions or readings to acquire from each channel in the channel string. **Number of samples** is the number of data points you want to sample from each channel.

- The **scan rate** determines how many times per second LabVIEW acquires data from channels. **scan rate** enables *interval scanning* (a longer interval between scans than between individual channels comprising a scan) on devices that support this feature. **channel clock rate** defines the time between the acquisition of consecutive channels in your channel string. Refer to the *Letting an Outside Source Control Your Acquisition Rate* section later in this chapter for more information about scan and channel clock rates.

Single-Point Acquisition

This section shows how you can acquire one data point from a single channel and then one data point from each of several channels using LabVIEW.

Single-Channel, Single-Point Analog Input

A single-channel, single-point analog input is an immediate, non-buffered operation. In other words, the software reads one value from an input channel and immediately returns the value. This operation does not require any buffering or timing. Use single-channel, single-point analog input when you need one data point from one channel. An example of this would be if you periodically needed to monitor the fluid level in a tank. You can connect the transducer that produces a voltage representing the fluid level to a single channel on your DAQ device and initiate a single-channel, single-point acquisition whenever you want to know the fluid level.

For most basic operations, use the AI Sample Channel VI, available on the **Functions»Data Acquisition»Analog Input** palette. The Easy Analog Input VI, AI Sample Channel, measures the signal attached to the channel you specify on your DAQ device and returns the scaled value.

Note If you configured your channel in the DAQ Channel Wizard, you do not need to enter the device or input limits. Instead, enter a channel name in the channel input, and the value returned is relative to the physical units you specified for that channel in the DAQ Channel Wizard. If you specify the input limits, they are treated as being relative to the physical units of the channel. LabVIEW ignores the device input when channel names are used. This principle applies throughout this manual.

Refer to the Acquire 1 Point from 1 Channel VI in the `examples\daq\anlogin\anlogin.llb` for an example of how to use the AI Sample Channel VI to acquire data. Open and examine its block diagram.

The Acquire 1 Point from 1 Channel VI initiates an A/D conversion on the DAQ device and returns the scaled value as an output. The **high limit** is the highest expected level of the signals you want to measure. The **low limit** is the lowest expected level of the signals you want to measure. Refer to the *Buffered Waveform Acquisition* section later in this chapter for more information about acquiring multiple points from a single channel.

Single-channel acquisition makes acquiring one channel very basic, but what do you do if you need to take more than one channel sample? For example, you might need to monitor the temperature of the fluid as well as the fluid level of the tank. In this case, two transducers must be monitored. You can monitor both transducers using a multiple-channel, single-point acquisition in LabVIEW.

Multiple-Channel, Single-Point Analog Input

With a multiple-channel, single-point read (or scan), LabVIEW returns the value on several channels at once. Use this type of operation when you have multiple transducers to monitor and you want to retrieve data from each transducer at the same time. Your DAQ device executes a scan across each of the specified channels and returns the values when finished.

The Easy I/O VI, AI Sample Channels, acquires single values from multiple channels. The AI Sample Channels VI performs a single A/D conversion on the specified channels and returns the scaled values in a waveform. The expected range for all the signals, specified by **high limit** and **low limit** inputs, applies to all the channels. Figure 6-11 shows how to acquire a signal from multiple channels using this VI.

Note Remember to use commas to delimit individual channels in the channel string. Use a colon to indicate an inclusive list of channels.

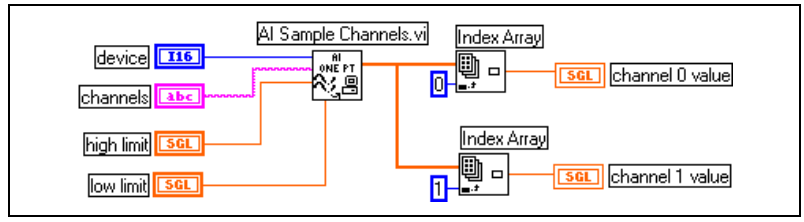


Figure 6-11. Acquiring a Voltage from Multiple Channels with the AI Sample Channels VI

The Easy Analog Input VIs have several benefits. You need only one icon in your block diagram to perform the task. Easy VIs require only a few basic inputs, and they have built-in error checking. However, these VIs have limited programming flexibility. Because Easy VIs have only a few inputs, you cannot implement some of the more detailed features of DAQ devices, such as triggering or interval scanning. In addition, these VIs always reconfigure at start-up, which can slow down processing time.

When you need more speed and efficiency, use the Intermediate VIs, which configure an acquisition only once and then continually acquire data without re-configuring. The Intermediate VIs also offer more error handling control, more hardware functionality, and more efficiency in developing your application than the Easy VIs. You typically use the Intermediate VIs to perform buffered acquisitions.

The AI Single Scan VI returns one scan of data. You also can use this VI to read only one point if you specify one channel. Use this VI only in conjunction with the AI Config VI.

Figure 6-12 shows a simplified block diagram for non-buffered applications. LabVIEW calls the AI Config VI, which configures the channels, selects the **input limits** (the **high limit** and **low limit** inputs in the Easy VIs), and generates a **taskID**. The program passes the **taskID** and the error cluster to the AI Single Scan VI, which returns the data in an array (one point for each channel specified).

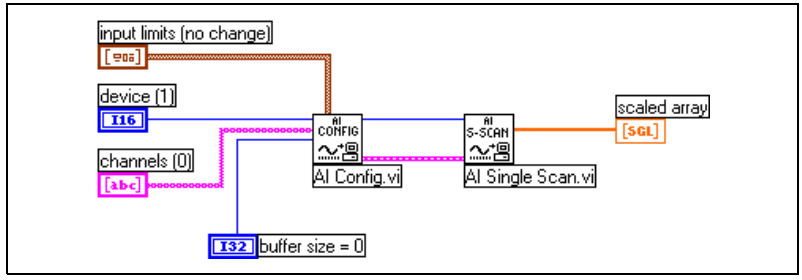


Figure 6-12. Using the Intermediate VIs for a Basic Non-Buffered Application

Refer to the Cont Acq&Chart (immediate) VI in the `examples\daq\anlogin\anlogin.llb` for an example of how to program the AI Config and AI Single Scan VIs to perform a series of single scans by using software timing (a While Loop) and processing each scan. Open this VI and examine its block diagram.

The advantage of using the intermediate-level VIs is that you do not have to configure the channels every time you want to acquire data as you do

when using the Easy VIs. To call the AI Config VI only once, put it outside of the While Loop in your program. The AI Config VI configures channels, selects a high/low limit, and generates a **taskID**. Then, the AI Config VI passes the **taskID** and error cluster into the While Loop. LabVIEW calls the AI Single Scan VI to retrieve a scan and passes the returned data to the My Single-Scan Processing VI. With this VI, you can program any processing needs your application calls for, such as looking for a limit to be exceeded. The VI then passes the data through the Build Array function to a waveform chart for display on the front panel. The Wait Until Next ms Multiple (metronome) function controls the loop timing. After you enter a scan rate, the application converts the value into milliseconds and passes the converted value to the Wait Until Next ms Multiple function. The loop then executes at the rate of scanning. The loop ends when you press the stop button or an when error occurs. After the loop finishes, the Simple Error Handler VI displays any errors that occurred.

The previous examples use software-timed acquisition. With this type of acquisition, the CPU system clock controls the rate at which you acquire data. The system clock can be interrupted by user interaction, so if you do not need a precise acquisition rate, use software-timed analog input.

Using Analog Input/Output Control Loops

When you want to output analog data after receiving some analog input data, use analog input/output (I/O) control loops. With control loops, this process is repeated over and over again.

The single-point analog input and output VIs support several analog I/O control loops at once because you can acquire analog inputs from several different channels in one scan and write all the analog output values with one update. You perform a single analog input call, process the analog output values for each channel, and then perform a single analog output call to update all the output channels.

The following sections describe the two different types of analog I/O control loop techniques: *software-timed* and *hardware-timed analog I/O*.

Using Software-Timed Analog I/O Control Loops

With software-timed analog control loops the analog acquisition rate and subsequent control loop rate are controlled by a software timer such as the Wait Until Next ms Multiple timer. The acquisition is performed during each loop iteration when the AI Single Scan VI is called and the control loop is executed once for each time interval. Your loop timing can be interrupted by any user interaction, which means your acquisition rate is

not as consistent as that which can be achieved through hardware-timed control loops. Generally, if you do not need a precise acquisition rate for your control loop, software timing is appropriate.

In addition to user interaction, a large number or large-sized front panel indicators, like charts and graphs, affect control loop rates. Refreshing the monitor screen interrupts the system clock, which controls loop rates. Therefore, keep the number of charts and graphs to a minimum when you are using software-timed control loops.

Refer to the Analog IO Control Loop (immed) VI in the `examples\daq\anlog_io\anlog_io.llb` for an example of software-timed control loops. Open this example VI to see how it performs software-timed analog I/O using the AI Read One Scan and AO Write One Update VIs.

The AI Read One Scan VI configures your DAQ device to acquire data from analog input channels 0 and 1. Once your program acquires a data point from channels 0 and 1, it performs calculations on the data and outputs the results through analog output channels 0 and 1. Because the iteration count is connected to the AI Read One Scan and AO Write One Update VIs, the application configures the DAQ device for analog input and output only on the first iteration of the loop. The loop rate as well as the acquisition rate is specified by **loop rate**. The reason why the **actual loop period** is important is because user interaction affects the loop and acquisition rate. For example, pressing the mouse button interrupts the system clock, which controls the loop rate. If your analog acquisition rate for control loops does not need to be consistent, use software-timed control loops.

Refer to the examples in the `examples\daq\solution\control.llb` for more control examples.

Using Hardware-Timed Analog I/O Control Loops

For more precise timing of your control loops and more precise analog input scan rate, use hardware-timed control loops.

Refer to the Analog IO Control Loop (hw timed) VI in the `examples\daq\anlog_io\anlog_io.llb` for an example of hardware-timed, non-buffered control loops. Open and examine its block diagram.

With hardware-timed control loops, your acquisition is not interrupted by user interaction. Hardware-timed analog input automatically places the data in your DAQ device FIFO buffer at an interval determined by the analog input scan rate. You can synchronize your control loop diagram to

this precise analog input scan rate by repeatedly calling the AI Single Scan VI to read the oldest data in the FIFO buffer.

The AI Single Scan VI returns as soon as the next scan has been acquired by the DAQ device. If more than one scan is stored in the DAQ device FIFO buffer when the AI Single Scan VI is called, then LabVIEW was not able to keep up with the acquisition rate. You can detect this by monitoring the data remaining output of the AI Single Scan VI. In other words, you have missed at least one control loop interval. This indicates that your software overhead is preventing you from keeping up with your hardware-timed loop rate. In the Analog IO Control Loop (hw timed) VI, the **loop too slow** Boolean indicator is set to TRUE whenever this occurs.

In this block diagram, the AI Config VI configures the device to acquire data on channels 0 and 1. The application does not use a buffer created in CPU memory, but instead uses the DAQ device FIFO buffer. **Input limits** (also known as *limit settings*) affects the expected range of the input signals. The AI Start VI begins the analog acquisition at the **loop rate** (scan rate) parameter. On the first iteration of the loop, the AI Single Scan VI reads the newest data in the FIFO buffer. Some data may have been acquired between the execution of the AI Start and the AI Single Scan VIs. On the first iteration of the loop, the application reads the latest data acquired between the AI Start and the AI Single Scan VIs. On every subsequent iteration of the loop, the application reads the oldest data in the FIFO buffer, which is the next acquired point in the FIFO buffer.

If more than one value was stored in the DAQ device FIFO buffer when you read it, your application was not able to keep up with the control loop acquisition and you have not responded with one control loop interval. This eventually leads to an error condition, which makes the loops complete. After the application completes analog acquisition and generation, the AI Clear VI clears the analog input task.

The block diagram of the Analog IO Control Loop (hw timed) VI also includes a waveform chart in the control loop. This reduces your maximum loop rate. You can speed up the maximum rate of the control loop by removing this graph indicator.

You easily can add other processing to your analog I/O control loop by putting the analog input, control loop calculations, and analog output in the first frame of a sequence inside the loop, and additional processing in subsequent frames of the sequence. Keep in mind that this additional processing must be less than your control loop interval. Otherwise, you will not be able to keep up with your control loop rate.

Improving Control Loop Performance

There are some performance issues you should take into account if you plan to have other VIs or loops run in parallel with your hardware-timed control loop. When you call the AI Single Scan VI in a hardware-timed control loop, the VI waits until the next scan is acquired before returning, which means that the CPU is waiting inside the NI-DAQ driver until the scan is acquired. Consequently, if you try to run other LabVIEW VIs or while loops in the same block diagram in parallel with your hardware-timed control loop, they may run more slowly or intermittently. You can reduce this problem by putting a software delay, with the Wait (ms) VI, at the end of your loop after you write your analog output values. Your other LabVIEW VIs and loops can then execute during this time.

Another good technique is to poll for your analog input without waiting in the driver. You can set the AI Single Scan VI **time limit in sec** to 0. Then, the VI reads the DAQ Device FIFO buffer and returns immediately, regardless of whether the next scan was acquired. The AI Single Scan VI **scaled data** output array is empty if the scan was not yet acquired. Poll for your analog input by using a Wait (ms) or Wait Until Next ms Multiple function together with the AI Single Scan VI in a While Loop within your control loop diagram. Set the wait time smaller than your control loop interval (at least half as small). If the **scaled data** output array is not empty, exit the polling loop passing out the **scaled data** array and execute the rest of your control loop diagram. This method does not return data as soon as the scan has been acquired, as in the example described previously, but provides ample time for other VIs and loops to execute. This method is a good technique for balancing the CPU load between several loops and VIs running in parallel.

Refer to the examples in the `examples\daq\solution\control.llb` for more control examples.

Buffered Waveform Acquisition

One way to acquire multiple data points for one or more channels is to use the non-buffered methods described earlier in this chapter in a repetitive manner. However, acquiring a single data point from one or more channels over and over is very inefficient and time consuming. Also, with this method of acquisition, you do not have accurate control over the time between each sample or channel. You can use a data buffer in computer memory to acquire data more efficiently.

If you want to take more than one reading on one or more channels, acquire your data as waveforms. There are two buffered waveform acquisition techniques you can use depending on what you want to do with the data after you acquire it: simple-buffered acquisition and circular-buffered acquisition. This section explains buffered waveform acquisition and shows these two techniques. Throughout the chapter are some basic examples of some common DAQ applications that use these two methods.

Using Simple Buffers to Acquire Waveforms with the Data Acquisition Input VIs

With buffered I/O, LabVIEW transfers data taken at timed intervals from a DAQ device to a data buffer in memory. In your VI, you must specify the number of samples to be taken and the number of channels from which LabVIEW will take the samples. From this information, LabVIEW allocates a buffer in memory to hold a number of data points equal to the number of samples per channel multiplied by the number of channels. As the data acquisition continues, the buffer fills with the data. However, the data may not actually be accessible until LabVIEW acquires all the samples. Once the data acquisition is complete, the data in the buffer can be analyzed, stored to disk, or displayed on the screen by your VI.

Acquiring a Single Waveform

The easiest way to acquire a single waveform from a single channel is to use the AI Acquire Waveform VI, as shown in Figure 6-13. Using this VI requires you to specify a device and/or channel, the number of samples you want to acquire from the channel, and the sample rate (measured in samples per second). The information you enter in these parameters are included in the waveform data.

You can programmatically set the **gain** by setting the **high limit** and the **low limit**. Using only the minimal set of inputs makes programming the VI easier, but the VI lacks more advanced capabilities, such as triggering.

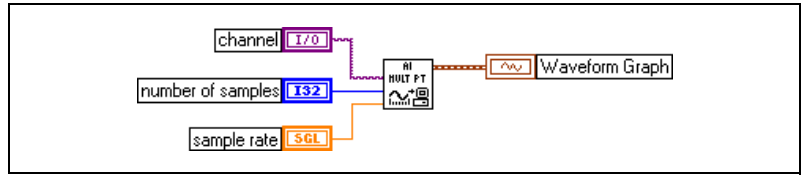


Figure 6-13. Acquiring and Graphing a Single Waveform

Acquiring Multiple Waveforms

You can acquire more than one waveform at a time with another of the Easy Analog Input VIs, AI Acquire Waveforms. This VI also has a minimal set of inputs, but it allows inputs of more than one channel to read and returns an array of waveforms from all channels it reads.

To access or control an individual waveform, index the array of waveforms with the Index Array function or use input indexing on a For or While Loop.

The VI in Figure 6-14 acquires waveforms from multiple channels and plots the waveforms on a graph. In addition, the Index Array function accesses the first waveform in the array and sends it to a filter, which sends the waveform to another graph.

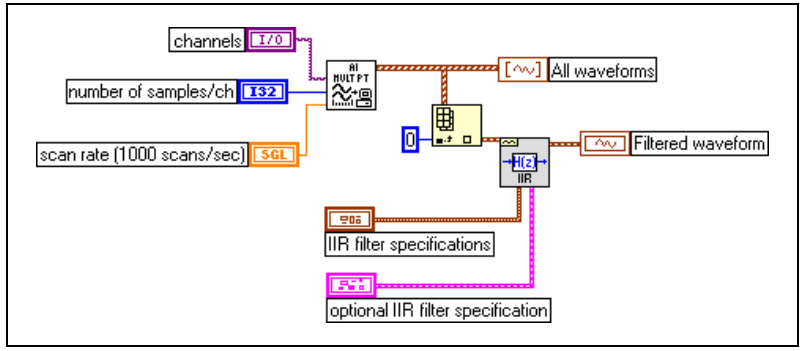


Figure 6-14. Acquiring and Graphing Multiple Waveforms and Filtering a Single Waveform

The **channels** input for the AI Acquire Waveforms VI has a pull down menu where you can select a channel from a list of configured named channels. You also can type a list of channels into this input. You can set the **high limit** and **low limit** inputs for all the channels to the same value. Like the other Easy VIs, you cannot use any advanced programming features with the AI Acquire Waveforms VI.

You also can acquire multiple waveforms using the Intermediate VIs. The Intermediate VIs provide more control over your data acquisition processes, like being able to read any part of the buffer. An example similar to Figure 6-15 is the Acquire N Scans VI, located in `labview\examples\daq\anlogin\anlogin.llb`. With the Intermediate Analog Input VIs, you must wire a **taskID** to identify the DAQ operation to make sure the VIs execute in the correct order.

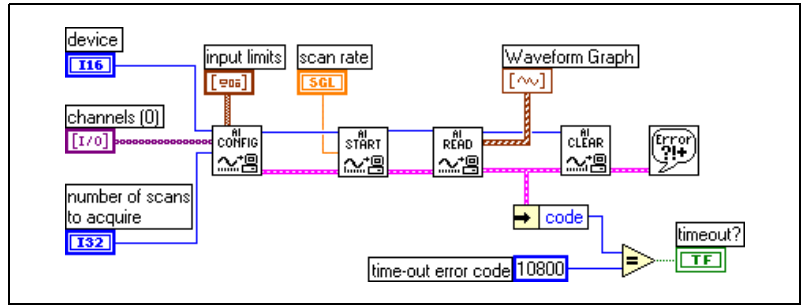


Figure 6-15. Using the Intermediate VIs to Acquire Multiple Waveforms

With these VIs, not only can you configure triggering, coupling, acquisition timing, retrieval, and additional hardware, but you also can control when each step of the data acquisition process occurs. With the AI Config VI, you can configure the different parameters of the acquisition, such as the channels to be read and the size of the buffer to use. In the AI Start VI, you specify parameters used in your program to start the acquisition, such as the number of scans to acquire, the rate at which your VI takes the data, and the trigger settings. In the AI Read VI, you specify parameters to retrieve the data from the data acquisition buffer. Then, the application calls the AI Clear VI to deallocate all buffers and other resources used for the acquisition by invalidating the **taskID**. If an error occurs in any of these VIs, your program passes the error through the remaining VIs to the Simple Error Handler VI, which notifies you of the error.

For many DAQ devices, the same ADC samples many channels instead of only one. The maximum sampling rate per channel is the maximum sampling rate of the device divided by the number of channels.

The scan rate input in all the VIs described above is the same as the sampling rate per channel. To figure out your maximum scan rate, you must divide the maximum sampling rate by the number of channels.

Simple-Buffered Analog Input Examples

This section contains several different examples of simple-buffered analog input.

Simple-Buffered Analog Input with Graphing

Figure 6-16 shows how you can use the AI Acquire Waveforms VI to acquire two waveforms from channels 0 and 1 and then display the waveforms on separate graphs. This type of VI is useful for comparing two or more waveforms or for analyzing how a signal looks before and after

going through a system. In this illustration, 1,000 scans of channels 0 and 1 are taken at the rate of 5,000 scans per second. The **actual scan period** output displays in the actual timebase on the X-axis of the graphs.

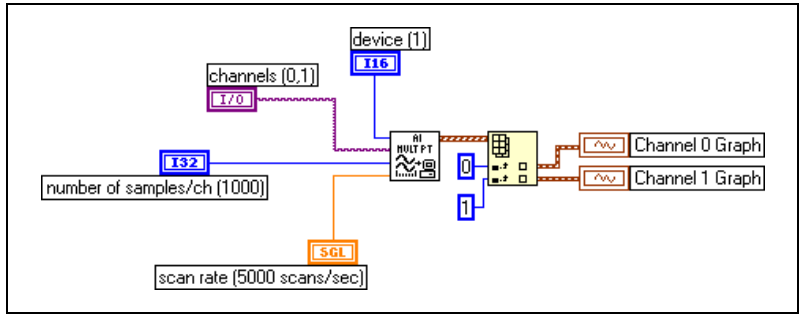


Figure 6-16. Simple Buffered Analog Input Example

Refer to the Acquire N Scans example VI in the `examples\daq\anlogin\anlogin.llb` for an example of a simple buffered input application that uses graphing.

Simple-Buffered Analog Input with Multiple Starts

In some cases, you might not want to acquire contiguous data, such as in an oscilloscope application. In this case, you want to take only a specified number of samples as a snapshot of what the input looks like periodically. Refer to the Acquire N-Multi-Start VI in the `examples\daq\anlogin\anlogin.llb` for an example using the Intermediate VIs similar to the Acquire N Scans example, except the acquisition only occurs each time the start button on the front panel is pressed.

This example is similar to the standard simple buffered analog input VI, but now both the AI Start and AI Read VIs are in a While Loop, which means the program takes a number of samples every time the While Loop iterates.

Note The AI Read VI returns 1,000 samples, taken at 5,000 samples per second, every time the While Loop iterates. However, the duration of the iterations of the While Loop can vary greatly. This means that, with this VI, you can control the rate at which samples are taken, but you may not be able to designate exactly when your application starts acquiring each set of data.

Simple-Buffered Analog Input with a Write to Spreadsheet File

If you want to write the acquired data to a file, there are many file formats in which you can store the data. The spreadsheet file format is used most often because you can read it using most spreadsheet applications for later data graphing and analysis. In LabVIEW, you can use VIs to send data to a file in spreadsheet format or read back data from such a file. You can find these VIs on the **Functions»File I/O** palette and on the **Functions» Waveform»Waveform File I/O** palette. The VI used in this example is the Export Waveforms to Spreadsheet File VI, shown in Figure 6-17. In this exercise, the Intermediate analog input VIs acquire an array of waveform data, graph the data, and create a spreadsheet file containing the data.

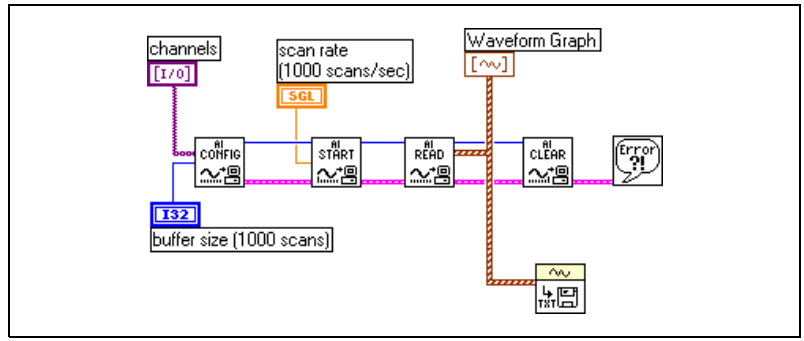


Figure 6-17. Writing to a Spreadsheet File after Acquisition

Using Circular Buffers to Access Your Data during Acquisition

You can apply the simple buffering techniques in many DAQ applications, but there are some applications where these techniques are not appropriate. If you want to view, process, or log portions of your data as it is being acquired, do not use these simple-buffered techniques. For these types of applications, you should set up a circular buffer to store acquired data in memory. Figure 6-18 shows how a circular buffer works. Portions of data are read from the buffer while the buffer is being filled. Using a circular buffer, you can set up your device to continuously acquire data in the background while LabVIEW retrieves the acquired data.

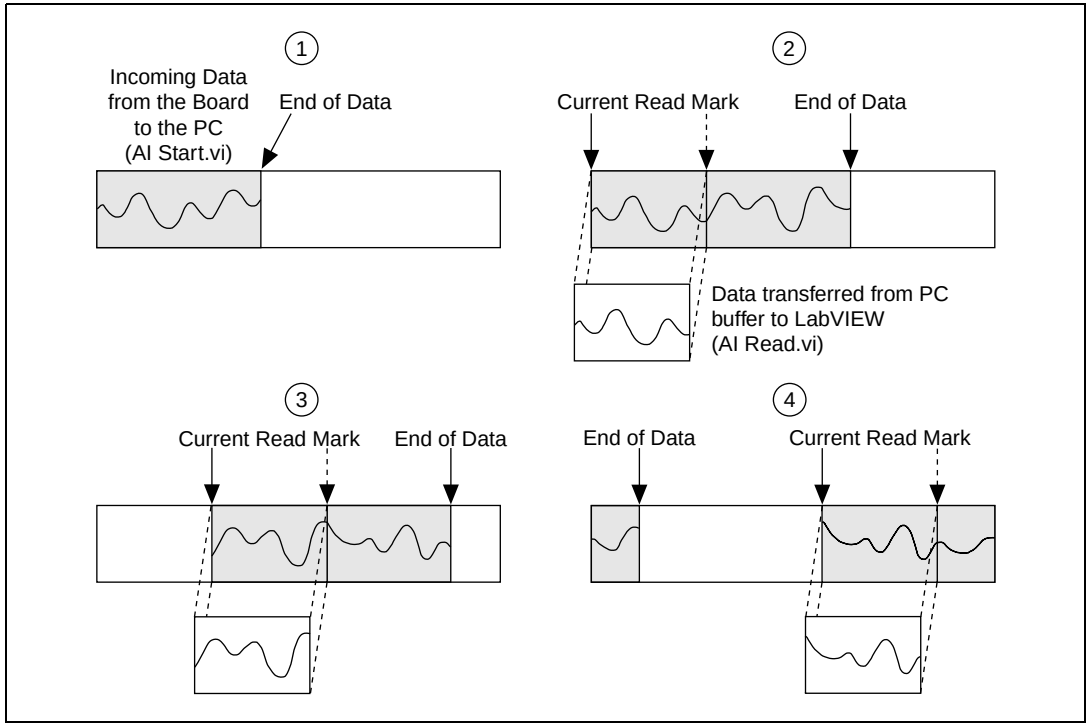


Figure 6-18. How a Circular Buffer Works

A circular buffer differs from a simple buffer only in how LabVIEW places the data into it and retrieves data from it. A circular buffer is filled with data, just as a simple buffer. However, when it gets to the end of the buffer, it returns to the beginning and fills up the same buffer again. This means data can read continuously into computer memory, but only a defined amount of memory can be used. Your VI must retrieve data in blocks, from one location in the buffer, while the data enters the circular buffer at a different location, so that unread data is not overwritten by newer data. Because of the buffer maintenance, you can use only the Intermediate or Advanced VIs with this type of data acquisition.

While a circular buffer works well in many applications, there are two possible problems that can occur with this type of acquisition: Your VI could try to retrieve data from the buffer faster than data is placed into it, or your VI might not retrieve data from the buffer fast enough before LabVIEW overwrites the data into the buffer. When your VI tries to read data from the buffer that has not yet been collected, LabVIEW waits for the data your VI requested to be acquired and then returns the data. If your VI

does not read the data from the circular buffer fast enough, the VI sends back an error, advising you that some data has been overwritten and lost.

Continuously Acquiring Data from Multiple Channels

You can acquire time-sampled data continuously from one or more channels with the Intermediate VIs. Refer to the Acquire & Process N Scans VI in the `examples\daq\anlogin\anlogin.llb` for an example using these VIs. Open this VI and examine its block diagram.

There are inputs for setting the channels, size of the circular buffer, scan rate, and the number of samples to retrieve from the circular buffer each time. This VI defaults to an **input buffer size** of 2,000 samples and 1,000 **number of scans to read at a time**, which means the VI reads in half of the buffer's data while the VI fills the second half of the buffer with new data.

Note The **number of scans to read** can be any number less than the input buffer size.

If you do not retrieve data from the circular buffer fast enough, your unread data will be overwritten by newer data. You can resolve this problem by adjusting one of these three parameters: the **input buffer size**, the **scan rate**, or the **number of scans to read at a time**. If your program overwrites data in the buffer, then data is coming into the buffer faster than your VI can read all of the previous buffer data, and LabVIEW returns the error code **`-10846 overWriteError`**. If you increase the size of the buffer so that it takes longer to fill up, your VI has more time to read data from it. If you slow down the **scan rate**, you reduce the speed at which the buffer fills up, which also gives your program more time to retrieve data. You also can increase the **number of scans to read at a time**. This retrieves more data out of the buffer each time and effectively reduces the number of times to access the buffer before it becomes full. Check the output **scan backlog** to see how many data values remain in the circular buffer after the read.

Because this uses Intermediate VIs, you also can control other parameters such as triggering, coupling, and additional hardware.

Asynchronous Continuous Acquisition Using DAQ Occurrences

The main advantage of acquiring data as described in the previous section is that you are free to manipulate your data between calls to the AI Read VI. One limitation, however, is that the acquisition is synchronous. This means that once you call the AI Read VI, you cannot perform any other tasks until

the AI Read VI returns your acquired data. If your DAQ device is still busy collecting data, you will have to sit idle until it finishes. On multithreaded platforms like Windows, this limitation can be worked around by allocating additional threads or by changing the preferred execution system of parts of your application.

Another alternative is to use asynchronous acquisition. You can acquire asynchronous continuous data from multiple channels using the same intermediate DAQ VIs by adding DAQ Occurrences. Refer to the Cont Acq&Chart (Async Occurrence) VI in the `examples\daq\anlogin\anlogin.llb` for an example of asynchronous acquisition. Notice that it is very similar to the example described previously, the Acquire & Process N Scans VI.

The difference is that this example uses the DAQ Occurrence Config VI and the Wait on Occurrence function to control the reads. The first DAQ Occurrence Config VI sets the DAQ Event. In this example the **DAQ Event** is to set the occurrence every time a number of scans is acquired equal to the value of general value *A*, where general value *A* is the **number of scans to read at a time**. Inside the While Loop, the Wait on Occurrence function sleeps in the background until the chosen **DAQ Event** takes place. Notice that the **timed out** output from the Wait on Occurrence function is wired to the selection terminal of the Case structure that encloses the AI Read VI. This means that AI Read is not called until the **number of scans to read at a time** have been acquired. The result is that the While Loop is effectively put to sleep, because you do not try to read the data until you know it has been acquired. This frees up the execution thread to do other tasks while you are waiting for the DAQ Event. If the DAQ Occurrence times out, the timed-out output value would be TRUE, and AI Read would never be called. When your acquisition is complete, DAQ Occurrence is called again to clear all occurrences.

Circular-Buffered Analog Input Examples

The only differences between the simple-buffered applications and circular-buffered applications in the block diagram is the setting of the **number of scans to acquire** input of the AI Start VI, and you must call the AI Read VI repeatedly to retrieve your data. These changes can be applied to many of the examples in the previous section on simple buffered analog input. However, this section reviews the basic circular-buffered analog input VI here and describes some other example VIs that are included with LabVIEW.

Basic Circular-Buffered Analog Input

Figure 6-19 shows an example VI that brings data from a channel at a rate of 1,000 samples/s into a buffer that can hold 4,000 samples. This type of example might be handy if you want to watch the data from a channel over a long period of time, but you cannot store all the data in memory at once. The AI Config VI sets up the channel specification and buffer size, then the AI Start VI initiates the background data acquisition and specifies the rate. Inside the While Loop, the AI Read VI repeatedly reads blocks of data from the buffer of a size equal to either 1,000 scans or the size of the **scan backlog**—whichever one is larger. The VI does this by using the Max & Min function to determine the larger of the two values. You do not have to use the Max & Min function in this way for the application to work, but this function helps control the size of the **scan backlog**, which is how many samples are left over in the buffer. This VI continuously reads and displays the data from channel 0 until an error occurs or until you click the **Stop** button.

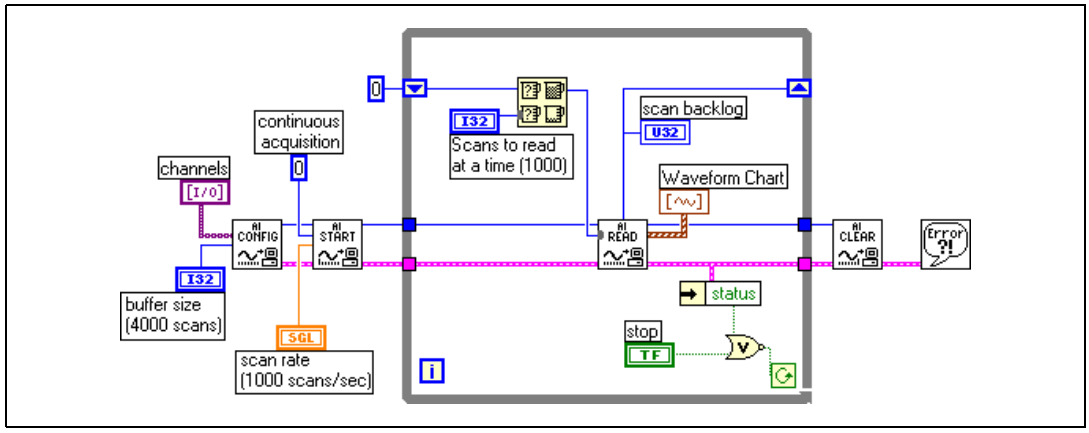


Figure 6-19. Basic Circular-Buffered Analog Input Using the Intermediate VIs

Other Circular-Buffered Analog Input Examples

Refer to the VIs in the `examples\daq\anlogin\anlogin.llb` and the rest of the example VIs in `examples\daq\anlogin\strmdisk.llb` for many other circular-buffered analog input VIs that are included with your LabVIEW application. The following list describes some of these VIs:

- **Cont Acq & Chart (buffered) VI**—Demonstrates circular-buffered analog input similarly to the previous example, but this VI includes other front panel inputs.

- **Cont Acq & Graph (buffered) VI**—Is similar to the Cont Acq & Chart (buffered) VI, except this VI displays data in a waveform graph.
- **Cont Acq to File (binary) VI**—Acquires data through circular-buffered analog input and stores it in a specified file as binary data. This process is more commonly called streaming to disk.
- **Cont Acq to File (scaled) VI**— Is similar to the previous binary VI, with the exception that this VI writes the acquired data to a file as scaled voltage readings rather than binary values.
- **Cont Acq to Spreadsheet File VI**—Continuously reads data that LabVIEW acquires in the circular buffer and stores this data to a specified file in spreadsheet format. You can view the data stored in a spreadsheet file by this VI in any spreadsheet application.

Simultaneous Buffered Waveform Acquisition and Waveform Generation

You might discover that along with your analog input acquisition, you also want to output analog data. If so, refer to the *Simultaneous Buffered Waveform Acquisition and Generation* section in Chapter 7, *Analog Output*, for more information about simultaneous buffered waveform acquisition and generation.

Controlling Your Acquisition with Triggers

The single-point and waveform acquisitions described in the previous chapters start at random times relative to the data. However, there are times when you need to be able to set your analog acquisition to start at a certain time. One example is if you want to test the response of a circuit board to a pulse input. The pulse input also can be used to tell the DAQ device to start acquiring data. Without this input, you must start acquiring before applying the test pulse. This is an inefficient use of computer memory and disk space because you must allocate and use more than is necessary. Sometimes the data you need might be closer to the front of the buffer and other times it might be closer to the end of the buffer.

You can start an acquisition based on the condition or state of an analog or digital signal using a technique called *triggering*. Generally, a *trigger* is any event that starts data capture. There are two basic types of triggering—hardware and software triggering. In LabVIEW, you can use software triggering to start acquisitions or use it with an external device to perform hardware triggering.

Hardware Triggering

Hardware triggering lets you set the start time of an acquisition and gather data at a known position in time relative to a trigger signal. External devices produce hardware trigger signals. In LabVIEW, you specify the triggering conditions that must be reached before acquisition begins. When the conditions are met, the acquisition begins. You also can analyze the data before the trigger.

There are two types of hardware triggers: digital and analog. In the following two sections, you will learn about the necessary conditions to start an acquisition with a digital or an analog signal.

Digital Triggering

A *digital trigger* is usually a transistor-transistor logic (TTL) signal having two discrete levels: a high and a low level. When moving from high to low or low to high, a digital edge is created. There are two types of edges: rising and falling. You can set your analog acquisition to start as a result of the rising or falling edge of your digital trigger signal.

In Figure 6-20, the acquisition begins after the falling edge of the digital trigger signal. Usually, digital trigger signals are connected to STARTTRIG*, EXTTRIG*, DTRIG, EXT TRIG IN, or PFI pins on your DAQ device. If you want to know which pin your device has, check your hardware manual. The STARTTRIG* and EXTTRIG* pins, which have an asterisk after their names, regard a falling edge signal as a trigger. Make sure you account for this when specifying your triggering conditions.

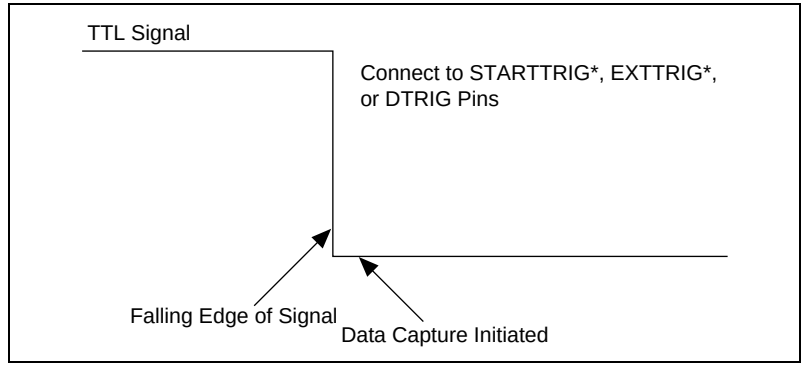


Figure 6-20. Diagram of a Digital Trigger

Figure 6-21 shows a timeline of how digital triggering works for post-triggered data acquisition. In this example, an external device sends a trigger, or TTL signal, to your DAQ device. As soon as your DAQ device receives the signal and your trigger conditions are met, your device begins acquiring data.

With NI 406*X* hardware, start trigger pulses can be generated externally or internally. The following start trigger pulse sources apply:

- Software start trigger
- External trigger

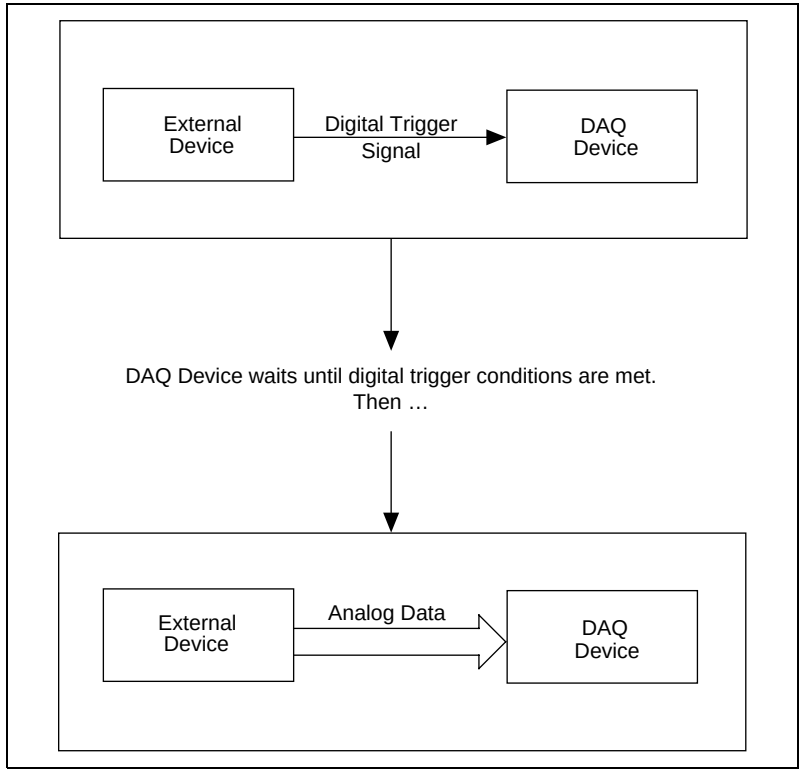


Figure 6-21. Digital Triggering with Your DAQ Device

Digital Triggering Examples

Refer to the Acquire N Scans Digital Trig VI in the `examples\daq\anlogin\anlogin.llb` for an example of digital triggering. Open this VI and examine its block diagram. This VI uses the Intermediate VIs to perform a buffered acquisition, where LabVIEW stores data in a memory

buffer during acquisition. After the acquisition completes, the VI retrieves all the data from the memory buffer and displays it.

You must tell your device the conditions on which to start acquiring data.

For this example, the **choose trigger type** Boolean should be set to **START OR STOP TRIGGER**. Select **START & STOP TRIGGER** only when you have two triggers: start and stop. In addition, if you use a DAQ device with PFI lines (for example, E Series devices), you can specify the trigger signal condition in the **trigger channel** control in the **analog chan & level** cluster.

You can acquire data both before and after a digital trigger signal. If **pretrigger scans** is greater than 0, your device acquires data before the triggering conditions are met. It then subtracts the **pretrigger scans** value from the **number of scans to acquire** value to determine the number of scans to collect after the triggering conditions are met. If **pretrigger scans** is 0, you acquire the **number of scans to acquire** after the triggering conditions are met.

Before you start acquiring data, you must specify in the **trigger edge** input whether the acquisition is triggered on the rising or falling edge of the digital trigger signal. You also can specify a value for the **time limit**, the maximum amount of time the VI waits for the trigger and requested data.

The Acquire N Scans Digital Trig VI example holds the data in a memory buffer until your device completes the acquisition. The number of data points you need to acquire must be small enough to fit in memory. This VI views and processes the information only after the acquisition. Refer to the Acquire & Proc N Scans-Trig VI in the `examples\daq\anlogin\anlogin.llb` for viewing and processing information during the acquisition. Refer to the Acquire N-Multi-Digital Trig VI in the `examples\daq\anlogin\anlogin.llb` if you expect multiple digital trigger signals that start multiple acquisitions.

Analog Triggering

You connect analog trigger signals to the analog input channels—the same channels where you connect analog data. Your DAQ device monitors the analog trigger channel until trigger conditions are met. You configure the DAQ device to wait for a certain condition of the analog input signal, such as the signal level or slope (either rising or falling). Once the device identifies the trigger conditions, it starts an acquisition.

Note If you are using channel names configured in the DAQ Channel Wizard, the signal level is treated as being relative to the physical units specified for the channel. For example, if you configure a channel called `temperature` to have a physical unit of `Deg. C`, the value you specify for the trigger signal level is relative to `Deg. C`. If you are not using channel names, the signal level is treated as volts.

In Figure 6-22, the analog trigger is set to start the data acquisition on the rising slope of the signal, when the signal reaches 3.2.

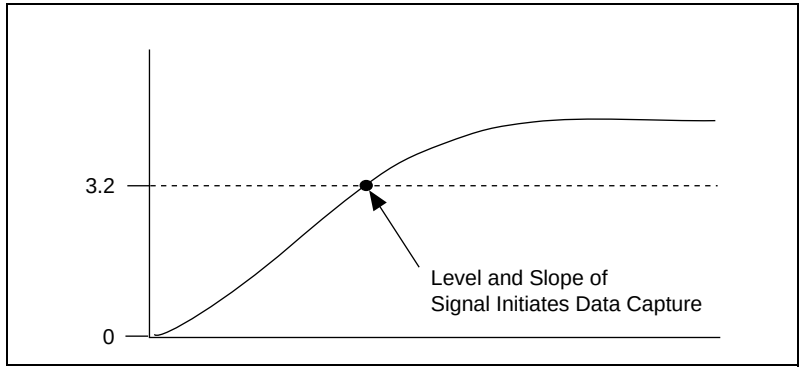


Figure 6-22. Diagram of an Analog Trigger

Figure 6-23 illustrates analog triggering for post-triggered data acquisition using a timeline. You configure your DAQ hardware in LabVIEW to begin taking data when the incoming signal is on the rising slope and when the amplitude reaches 3.2. Your DAQ device begins capturing data when the specified analog trigger conditions are met.

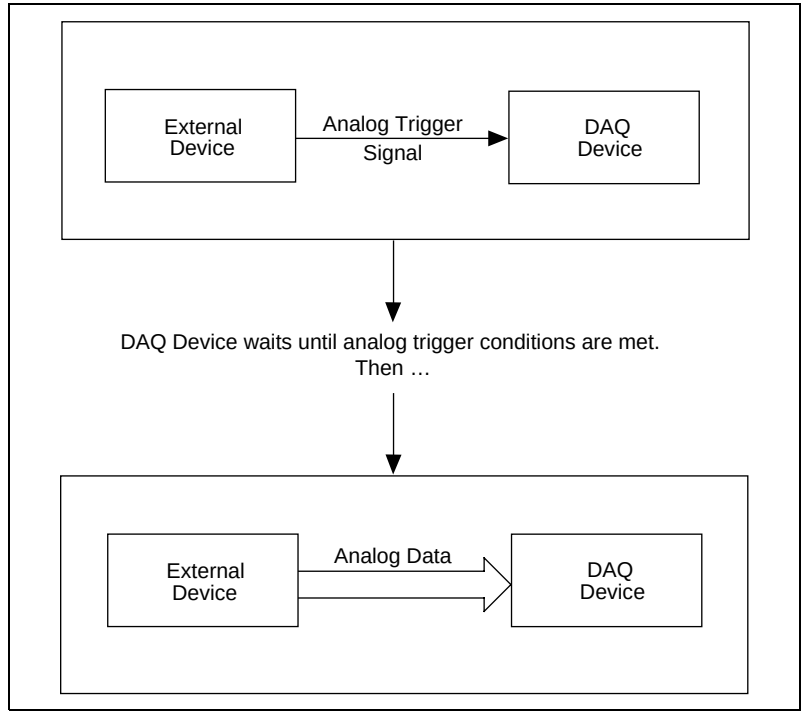


Figure 6-23. Analog Triggering with Your DAQ Device

Analog Triggering Examples

Refer to the Acquire N Scans Analog Hardware Trig VI in the `examples\daq\anlogin\anlogin.llb` for a common example of analog triggering in LabVIEW. This VI uses the Intermediate VIs to perform buffered acquisition, where data is stored in a memory buffer during acquisition. After the acquisition completes, the VI retrieves all the data from the memory buffer and displays it.

You must tell your device the conditions on which to start acquiring data.

In LabVIEW, you can acquire data both before and after an analog trigger signal. If the **pretrigger scans** is greater than 0, your device acquires data before the triggering conditions. It then subtracts the **pretrigger scans** value from the **number of scans to acquire** value to determine the number of scans to collect after the triggering conditions are met. If **pretrigger scans** is 0, then the **number of scans to acquire** is acquired after the triggering conditions are met.

Complete the following steps before you start acquiring data.

1. Specify in the **trigger slope** input whether to trigger the acquisition on the rising or falling edge of the analog trigger signal.
2. Enter the **trigger channel** to use for connecting the analog triggering signal.
3. Specify the **trigger level** on the triggering signal needed to begin acquisition.

After you specify the channel of the triggering signal, LabVIEW waits until the slope and trigger level conditions are met before starting a buffered acquisition. If you use channel names configured in the DAQ Channel Wizard, **trigger level** is treated as being relative to the physical units specified for the channel in the DAQ Channel Wizard. Otherwise, **trigger level** is treated as volts.

The Acquire N Scans Analog Hardware Trig VI example, available in the `examples\daq\anlogin\anlogin.llb`, holds the data in a memory buffer until the device completes data acquisition. The number of data points you want to acquire must be small enough to fit in memory. This VI views and processes the information only after the acquisition. Refer to the Acquire & Proc N Scans-Trig VI in the `examples\daq\anlogin\anlogin.llb` if you need to view and process information during the acquisition. Refer to the example Acquire N-Multi-Analog Hardware Trig VI in the `examples\daq\anlogin\anlogin.llb` if you expect multiple analog trigger signals that will start multiple acquisitions.

Software Triggering

With software triggering, you can simulate an analog trigger using software. This form of triggering is often used in situations where hardware triggers are not available. Another name for software triggering signals, specifically analog signals, is *conditional retrieval*. With conditional retrieval, you set up your DAQ device to collect data, but the device does not return any data to LabVIEW unless the data meets your retrieval conditions. LabVIEW scans the input data and performs a comparison with the conditions, but does not store the data until it meets your specifications. Figure 6-24 shows a timeline of events that typically occur when you perform conditional retrieval.

The read/search position pointer traverses the buffer until it finds the scan location where the data has met the retrieval conditions. Offset indicates the scan location from which the VI begins reading data relative to the read/search position. A negative offset indicates that you need pretrigger

data (data prior to the retrieval conditions). If offset is greater than 0, you need posttrigger data (data after retrieval conditions).

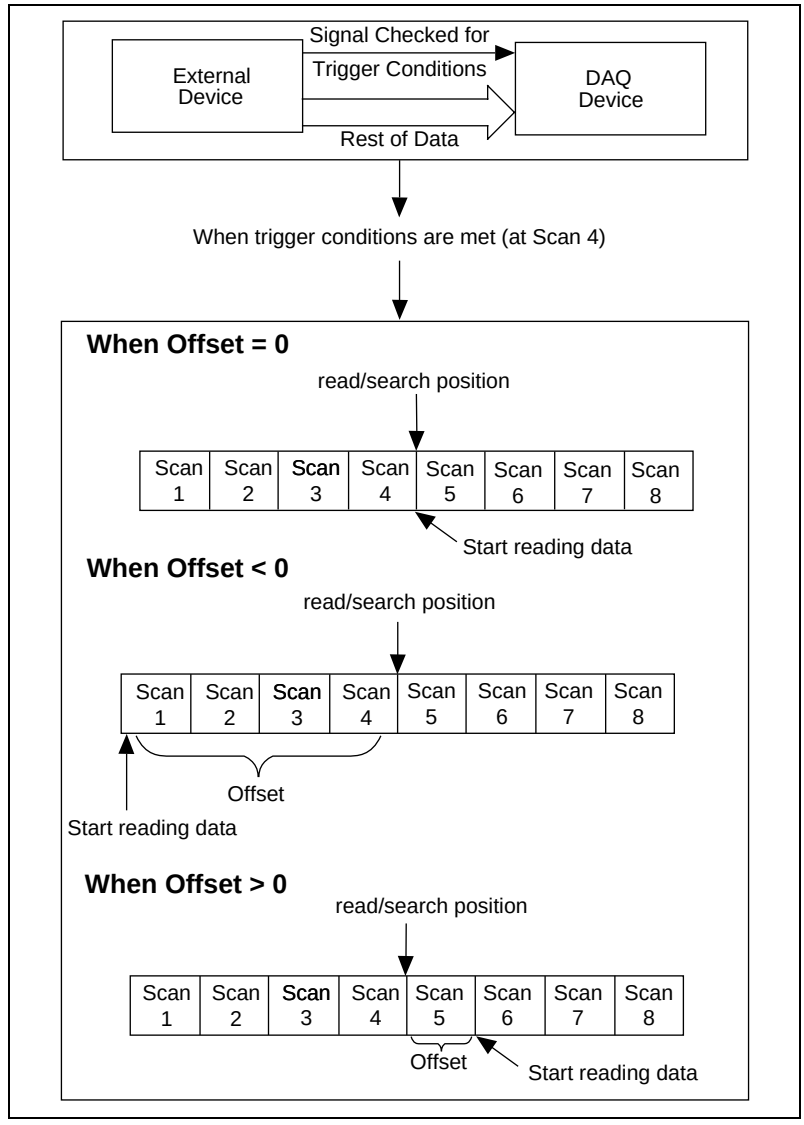


Figure 6-24. Timeline of Conditional Retrieval

The **conditional retrieval** cluster of the AI Read VI specifies the analog signal conditions of retrieval, as shown in Figure 6-25.

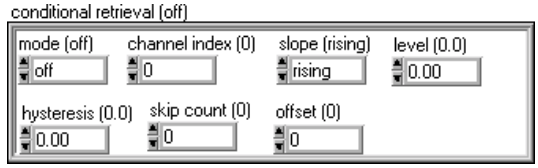


Figure 6-25. The AI Read VI Conditional Retrieval Cluster

Tip The actual data acquisition is started by running your VI. The conditional retrieval just controls how data already being acquired is returned.

When acquiring data with conditional retrieval, you typically store the data in a memory buffer, similar to hardware triggering applications. After you start running the VI, the data is placed in the buffer. Once the retrieval conditions have been met, the AI Read VI searches the buffer for the desired information. As with hardware analog triggering, you specify the analog channel of the triggering signal by specifying its **channel index,** an index number corresponding to the relative order of a single channel in a channel list. You also specify the **slope** (rising or falling) and the **level** of the trigger signal.

Note The channel index might not be equal to the channel value. You can use the Channel to Index VI to get the channel index for a channel. You can find this VI on the **Data Acquisition»Calibration and Configuration** palette.

The AI Read VI begins searching for the retrieval conditions in the buffer at the read/search position, another input of the AI Read VI. The **offset**, a value of the **conditional retrieval** input cluster, is where you specify the scan locations from which the VI begins reading data relative to the read/search position. A negative **offset** indicates data prior to the retrieval condition pretrigger data, and a positive **offset** indicates data after the retrieval condition posttrigger data. The **skip count** input allows you to specify the number of times the trigger conditions are met and skipped before data is returned. The **hysteresis** input controls the range used to meet the retrieval conditions. It is useful when your signal has noise that might inadvertently trigger your acquisition. Once the **slope** and **level** conditions on **channel index** have been found, the read/search position indicates the location where the retrieval conditions were met.

If you are using channel names configured in the DAQ Channel Wizard, **level** and **hysteresis** are treated as being relative to the physical units specified for the channel. If you are not using channel names, these inputs are treated as volts.

Conditional Retrieval Examples

The Acquire N Scans Analog Software Trig VI example, available in the `examples\daq\anlogin\anlogin.llb`, uses the Intermediate VIs. Open this VI and examine its block diagram.

The main difference between this software triggering example and hardware triggering is the use of the **conditional retrieval** input for the AI Read VI. You set up the **trigger channel**, **trigger slope**, and **trigger level** the same way for both triggering methods. The **pretrigger scans** value is negated and connected to the **offset** value in the **conditional retrieval** cluster of the AI Read VI. When the trigger conditions are met, the VI returns the requested number of scans.

Letting an Outside Source Control Your Acquisition Rate

Typically, a DAQ device uses internal counters to determine the rate to acquire data, but sometimes you might need to capture your data at the rate of particular signals in your system. For example, you also can read temperature channels every time a pulse occurs, which represents pressure rising above a certain level. In this case, internal counters are inefficient for your needs. You must control your acquisition rate by some other, external source.

You can compare a scan of your channels to taking a snapshot of the voltages on your analog input channels. If you set your scan rate to 10 scans per second, you are taking 10 snapshots each second of all the channels in your channel list. In this case, an internal clock within your device (the scan clock) sets the scan rate, which controls the time interval between scans.

Also, remember that most DAQ devices (those that do not sample simultaneously) proceed from one channel to the next or from one sample to the next, depending on the channel clock rate. Therefore, the channel clock is the clock controlling the time interval between individual channel samples within a scan, which means the channel clock proceeds at a faster rate than the scan clock.

The faster the channel clock rate, the more closely in time your system samples the channels within each scan, as shown in Figure 6-26.

Note For devices with both a scan and channel clock, lowering the scan rate does not change the channel clock rate.

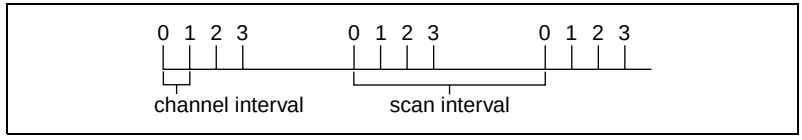


Figure 6-26. Channel and Scan Intervals Using the Channel Clock

Some DAQ devices do not have scan clocks, but rather use *round-robin scanning*. Figure 6-27 shows an example of round-robin scanning.

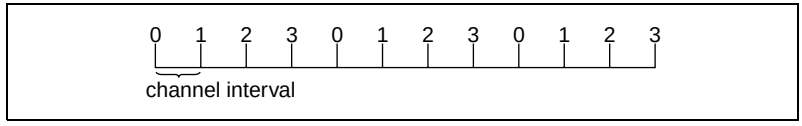


Figure 6-27. Round-Robin Scanning Using the Channel Clock

The devices that always perform round-robin scanning include, but are not limited to, the following:

- PC-LPM-16
- PC-LPM-16PnP
- PC-516
- DAQCard-500
- DAQCard-516
- DAQCard-700
- Lab-LC
- NI 4060

With no scan clock, the channel clock is used to switch between each channel at an equal interval. The same delay exists among all channel samples, as well as between the last channel of a scan and the first channel of the next scan. For boards with scan and channel clocks, round-robin scanning occurs when you disable the scan clock by setting the scan rate to 0 and using the **interchannel delay** of the AI Config VI to control your acquisition rate.

LabVIEW is *scan-clock oriented*. In other words, when you select a scan rate, LabVIEW automatically selects the channel clock rate for you. LabVIEW selects the fastest channel clock rate that allows adequate settling time for the ADC.

LabVIEW adds an extra 10 µs to the interchannel delay to compensate for any unaccounted factors. However, LabVIEW does not consider this additional delay for purposes of warnings. If you have specified a scan rate that is adequate for acquisition but too fast for LabVIEW to apply the 10-µs delay, it configures the acquisition but does not return a warning.

You can set your channel clock rate with the **interchannel delay** input of the AI Config VI, which calls the Advanced AI Clock Config VI to actually configure the channel clock. The simplest method to select an interchannel delay is to gradually increase the delay, or clock period, until the data appears consistent with data from the previous delay setting.

Refer to your hardware manuals for the required setting time for your channel clock. You also can find the interchannel delay by running the low-level AI Clock Config VI for the channel clock with no frequency specified.

Externally Controlling Your Channel Clock

There are times when you might need to control the channel clock externally. The channel clock rate is the same rate at which analog conversions occur. For instance, suppose you need to know the strain value at an input, every time an infrared sensor sends a pulse. Most DAQ devices have an EXTCONV* pin or a PFI pin on the I/O connector for providing your own channel clock. For NI 406*x* Series devices, use the EXTRIG input pin. This external signal must be a TTL level signal. The asterisk on the signal name indicates that the actual conversion occurs on the falling edge of the signal, as shown in Figure 6-28. For devices with PFI lines and for the NI 406*X* Series devices, you can select either the rising edge or falling edge using LabVIEW. With devices that have a RTSI connector, you can get your channel clock from other National Instruments DAQ devices.

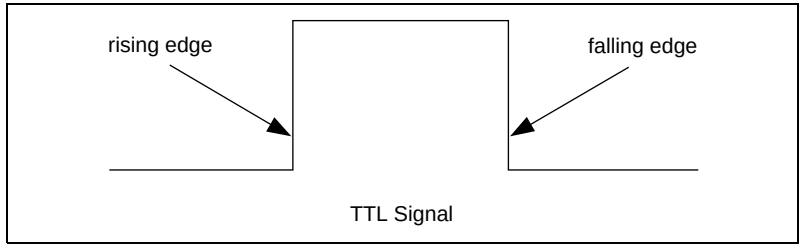


Figure 6-28. Example of a TTL Signal

Refer to the Acquire N Scans-ExtChanClk VI in the `examples\daq\anlogin\anlogin.llb` for an example of how to set up your acquisition for an externally controlled channel clock. The VI includes the AI Clock Config VI and the clock source connected to the I/O connector.

You can enable external conversions by calling the advanced-level AI Clock Config VI. Remember that the AI Clock Config VI, which is called by the AI Config VI, normally sets internal channel delay automatically or manually with the **interchannel delay** control. However, calling the AI Clock Config VI after the AI Config VI resets the channel clock so that it comes from an external source for external conversion. Also, notice that the scan clock is set to 0 to disable it, allowing the channel clock to control the acquisition rate.

Note The 5102 devices do not support external channel clock pulses, because there is no channel clock on the device.

On most devices, external conversions occur on the falling edge of the EXTCONV* line. Consult your hardware reference manual for timing diagrams. On devices with PFI lines (such as E Series devices), you can set the **Clock Source Code** input of AI Clock Config VI to the PFI pin with either falling or rising edge or use the default PFI2/Convert* pin where the conversions occur on the falling edge.

Because LabVIEW determines the length of time before the AI Read VI times out based on the **interchannel delay** and **scan clock rate**, you may need to force a time limit for the AI Read VI, as illustrated in the Acquire N Scans-ExtChanClk VI described previously.

Note On the Lab-PC+ and 1200 devices, the first clock pulse on the EXTCONV* pin configures the acquisition but does not cause a conversion. However, all subsequent pulses cause conversions.

Figure 6-29 shows an example of using an external scan clock to perform a buffered acquisition.

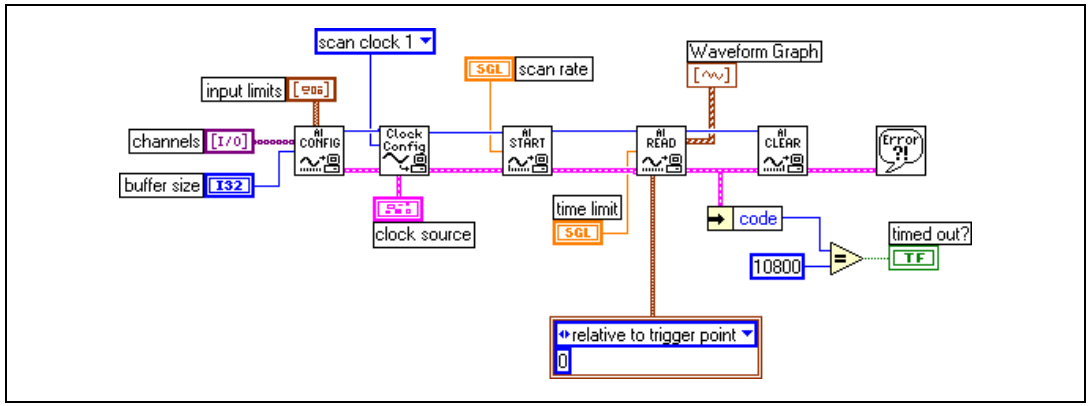


Figure 6-29. Acquiring Data with an External Scan Clock

Externally Controlling Your Scan Clock

External scan clock control might be more useful than external channel clock control if you are sampling multiple channels, but might not be as obvious to find because it does not have the input on the I/O connector labeled `ExtScanClock`, the way the EXTCONV* pin does.

Note Some MIO devices have an output on the I/O connector labeled `SCANCLK`, which is used for external multiplexing and is not the analog input scan clock. This cannot be used as an input.

The appropriate pin to input your external scan clock can be found in Table 6-2.

Table 6-2. External Scan Clock Input Pins

Device	External Scan Clock Input Pin
All E Series Devices	Any PFI Pin (Default: PF17/STARTSCAN)
Lab-PC+ 1200 devices	OUT B1

Note Some devices do not have internal scan clocks and therefore do not support external scan clocks. These devices include but are not limited to the following: PC-LPM-16,

PC-LPM-16PnP, PC-516, DAQCard-500, DAQCard-516, DAQCard-700, NI 4060, and Lab-LC.

After connecting your external scan clock to the correct pin, set up the external scan clock in software. Refer to the Acquire N Scans-ExtScanClk VI in the `examples\daq\anlogin\anlogin.llb` for an example of how to set up the external clock in software. Two Advanced VIs, AI Clock Config and AI Control, are used in place of the Intermediate AI Start VI. This allows access to the **clock source** input. This is necessary because it allows access to the **clock source string**, which is used to identify the PFI pin to be used for the scan clock for E Series boards. The **clock source** also includes the **clock source code** (on the front panel), which is set to the I/O connector. The `0.0` wired to the Clock Config VI disables the internal clock.

Remember that the **which clock** input of the AI Clock Config VI should be set to **scan clock (1)**.

Note You must divide the timebase by some number between 2 and 65,535 or you will get a bad input value error.

Because LabVIEW determines the length of time before AI Read times out based on the interchannel delay and scan clock rate, you might need to force a time limit into AI Read. In the example Acquire N Scans-ExtScanClk VI, the time limit is 5 seconds.

Externally Controlling the Scan and Channel Clocks

You can control the scan and channel clocks simultaneously. However, make sure that you follow the proper timing. Figure 6-30 demonstrates how you can set up your application to control both clocks.

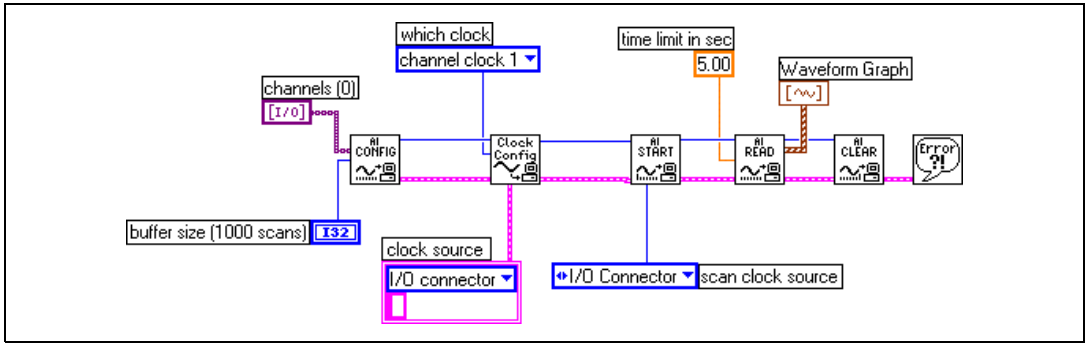


Figure 6-30. Controlling the Scan and Channel Clock Simultaneously

7

Analog Output

This chapter explains analog output for data acquisition.

Things You Should Know about Analog Output

Some measuring systems require that analog signals be generated by a DAQ device. Each of these analog signals can be a steady or slowly changing signal, or a continuously changing waveform. This section describes how to use LabVIEW to produce all of these different types of signals.

Single-Point Output

When the signal level at the output is more important than the rate at which the output value changes, you need to generate a steady DC value. You can use the single-point analog output VIs to produce this type of output. With single-point analog output, any time you want to change the value on an analog output channel, you must call one of the VIs that produces a single update (a single value change). Therefore, you can change the output value only as fast as LabVIEW calls the VIs. This technique is called software timing. You should use software timing if you do not need high-speed generation or the most accurate timing. Refer to the *Single-Point Generation* section, later in this chapter, for more information about single-point output.

Buffered Analog Output

Sometimes in performing analog output, the rate that your updates occur is just as important as the signal level. This is called waveform generation, or buffered analog output. For example, you might want your DAQ device to act as a function generator. You can do this by storing one cycle of sine wave data in an array and programming the DAQ device to generate the values continuously in the array one point at a time at a specified rate. This is known as single-buffered waveform generation. But what if you want to generate a continually changing waveform? For example, you might have a large file stored on disk that contains data you want to output. Because LabVIEW cannot store the entire waveform in a single buffer, you must continually load new data into the buffer during the generation. This

process requires the use of circular-buffered analog output in LabVIEW. Refer to the *Waveform Generation (Buffered Analog Output)* section, later in this chapter, for more information about single or circular buffering.

Single-Point Generation

When the signal level at the output is more important than the rate at which the output value changes, you need to generate a steady DC value. You can use the single-point analog output VIs to produce this type of output. With single-point analog output, any time you want to change the value on an analog output channel, you must call one of the VIs that produces a single update (a single value change). Therefore, you can change the output value only as fast as LabVIEW calls the VIs. This technique is called *software timing*. You should use software timing if you do not need high-speed generation or the most accurate timing.

Single-Immediate Updates

The simplest way to program single-point updates in LabVIEW is by using the Easy Analog Output VI, AO Update Channels. This VI writes values to one or more output channels on the output DAQ device.

An array of values is passed as an input to the VI. The first element in the array corresponds to the first entry in the channel string, and the second array element corresponds to the second channel entry. If you use channel names configured in the DAQ Channel Wizard in your channel string, **values** is relative to the physical units you specify in the DAQ Channel Wizard. Otherwise, **values** is relative to volts. Remember that Easy VIs already have built-in error handling.

Refer to the Generate 1 Point on 1 Channel VI in the `examples\daq\anlogout\anlogout.llb` for an example of writing values for multiple channels. This VI generates one value for one channel.

If you want more control over the limit settings for each channel, you also can program a single-point update using the Intermediate Analog Output VI, AO Write One Update.

In this VI, your program passes the error information to the Simple Error Handler VI. The **iteration** input optimizes the execution of this VI if you place it in a loop. With Intermediate VIs, you gain more control over when you can check for errors.

Multiple-Immediate Updates

Refer to the Write N Updates example VI in the `examples\daq\anlogout\anlogout.llb` for an example of a VI that performs multiple updates. Its block diagram resembles the one shown for the AO Write One Update VI described previously, except that the While Loop runs the subVI repeatedly until either the error status or the stop Boolean is TRUE. You can use the Easy Analog Output VI, AO Write One Update VI, in a loop, but this is inefficient because the Easy I/O VIs configure the device every time they execute. The AO Write One Update VI configures the device only when the value of the **iteration** input is set to 0.

The Write N Updates example VI illustrates an immediate, software-timed analog output VI application. This means that software timing in a loop controls the update rate. One good reason to use immediate, software-timed output is that your application calculates or processes output values one at a time. However, remember that software timing is not as accurate as hardware-timed analog output.

Waveform Generation (Buffered Analog Output)

This section shows you which VIs to use in LabVIEW to perform buffered analog updates.

Buffered Analog Output

You can program single-buffered analog output in LabVIEW using an Easy Analog Output VI, AO Generate Waveforms VI. This VI writes an array of output values to the analog output channels at a rate specified by **update rate**. For example, if **channels** consists of two channels and the **waveforms** array consists of waveform data for the two channels, LabVIEW writes values from the waveform array to the corresponding channels at every update interval. After LabVIEW writes all the values in the array to the channels, the VI stops. The signal level on the output channels maintains the value of the final value row in the waveform array until another value is generated. If you use channel names configured in the DAQ Channel Wizard in **channels**, **waveforms** is relative to the units specified in the DAQ Channel Wizard. Otherwise, **waveforms** is relative to volts.

Easy VIs contain error handling. If an error occurs in the AO Generate Waveforms VI, a dialog box appears displaying the error number and description, and the VI stops running.

As with single-point analog output, you can use the Analog Output Utility VI, AO Waveform Gen VI, for most of your programming needs. This VI has several inputs and outputs that the Easy I/O VI does not have. You have the option of having the data array generated once, several times, or continuously through the **generation count** input. Figure 7-1 shows an example block diagram of how to program this VI.

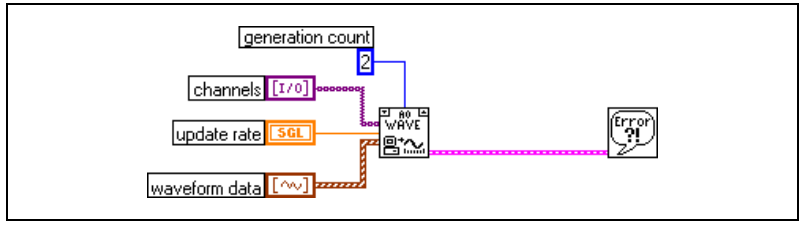


Figure 7-1. Waveform Generation Using the AO Waveform Gen VI

In this example, LabVIEW generates the data in the array two times before stopping.

The Generate N Updates example VI, available in `examples\daq\anlogout\anlogout.llb`, uses the AO Waveform Gen VI. Placing this VI in a loop and wiring the iteration terminal of the loop to the iteration input on the VI optimizes the execution of this VI. When iteration is 0, LabVIEW configures the analog output channels appropriately. If the iteration is greater than 0, LabVIEW uses the existing configuration, which improves performance. With the AO Waveform Gen VI, you also can specify the limit settings input for each analog output channel.

If you want even more control over your analog output application, use the Intermediate DAQ VIs, as shown in Figure 7-2.

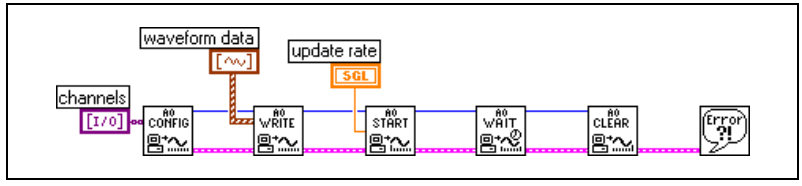


Figure 7-2. Waveform Generation Using Intermediate VIs

With these VIs, you can set up an alternate update clock source (such as an external clock or a clock signal coming from another device) or return the update rate. The AO Config VI sets up the channels you specify for analog output. The AO Write VI places the data in the buffer, the AO Start VI begins the actual generation at the **update rate**, and the AO Wait VI waits

until the waveform generation completes. Then, the AO Clear VI clears the analog channels.

The Generate Continuous Sinewave VI, available in `examples\daq\anlogout\anlogout.llb`, is similar in structure to Figure 7-2. This example VI continually outputs a sine waveform through the channel you specify.

Changing the Waveform during Generation: Circular-Buffered Output

When the waveform data is too large to fit in a memory buffer or is constantly changing, use a *circular buffer* to output the data. You also can use the Easy Analog Output VIs in a loop to create a circular-buffered output; but this sacrifices efficiency because Easy VIs configure, allocate, and deallocate a buffer every time they execute, which causes time gaps between the data output.

Open the AO Continuous Gen VI to see one way to perform circular-buffered analog output using the Intermediate VIs. This VI is more efficient than the Easy Analog Output VIs in that it configures and allocates a buffer when its **iteration** input is 0 and deallocates the buffer when the **clear generation** input is TRUE.

With the AO Continuous Gen VI, you can configure the size of the data buffer and the limit settings of each channel. Refer to the *Basic LabVIEW Data Acquisition Concepts* section in Chapter 5, *Introduction to Data Acquisition in LabVIEW* for more information about how to set limit settings.

Refer to the Continuous Generation VI in the `examples\daq\anlogout\anlogout.llb` for an example of using the AO Continuous Gen VI. In this example, the data completely fills the buffer on the first iteration. On subsequent iterations, new data is written into one half of the buffer while the other half continues to output data.

To gain more control over your analog output application, use the Intermediate VIs shown in Figure 7-3. With these VIs, you can set up an alternate update clock source and you can monitor the update rate the VI actually uses. The AO Config VI sets up the channels you specify for analog output. The AO Write VI places the data in a buffer. The AO Start VI begins the actual generation at the **update rate**. The AO Write VI in the While Loop writes new data to the buffer until you click the **Stop** button. The AO Clear VI clears the analog channels.

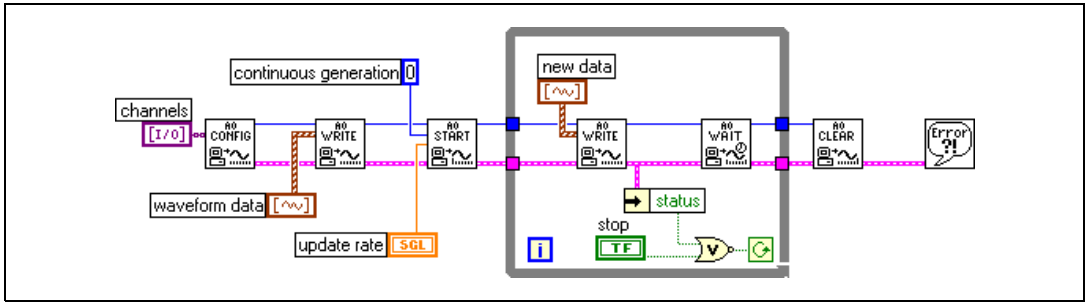


Figure 7-3. Circular Buffered Waveform Generation Using Intermediate VIs

Refer to the Function Generator VI in the `examples\daq\anlogout\anlogout.llb` for a more advanced example than the one shown in Figure 7-3. This VI changes the output waveform on-the-fly, responding to changing signal types (sine or square), amplitude, offset, update rate, and phase settings on the front panel.

Eliminating Errors from Your Circular-Buffered Application

If you get an error, **`-10843 underFlowError`**, while performing circular-buffered output, it means your program cannot write data fast enough to the buffer to output the data at the update rate. To solve this problem, decrease the speed of the update rate or increase the buffer size.

Circular-Buffered Analog Output Examples

Another example VI in this library you might find helpful is the Display and Output Acq'd File (scaled) VI.

You can use this VI in conjunction with the Cont Acq to File (scaled) VI, available in `examples\daq\anlogin\anolgin.llb`. After running the Cont Acq to File (scaled) VI and saving your acquired data to disk, you can run the Display and Output Acq'd File (scaled) VI to generate your data from the file you created. This example uses circular-buffered output. To generate data at the same rate at which it was acquired, you must know the rate at which your data was acquired, and use that as the **update rate**.

Sometimes in performing analog output, the rate that your updates occur is just as important as the signal level. This is called waveform generation, or buffered analog output. For example, you might want your DAQ device to act as a function generator. You can do this by storing one cycle of sine wave data in an array and programming the DAQ device to generate the

values continuously in the array one point at a time at a specified rate. This is known as single-buffered waveform generation. But what if you want to generate a continually changing waveform? For example, you might have a large file stored on disk that contains data you want to output. Because LabVIEW cannot store the entire waveform in a single buffer, you must continually load new data into the buffer during the generation. This process requires the use of circular-buffered analog output in LabVIEW.

Letting an Outside Source Control Your Update Rate

DAQ devices use internal counters and timers to determine the rate of data generation. However, you might encounter times when you need to generate data in sync with other signals in your system. For example, you might need to output data to a test circuit every time that test circuit emits a pulse. In this case, internal counter/timers are inefficient for your needs. You need to control the update rate with your own external source of pulses.

Externally Controlling Your Update Clock

This section explains how to use these Intermediate VIs to generate data using an external update clock.

The update clock controls the rate at which digital-to-analog conversions occur. To control your data generation externally, you must supply this clock signal to the appropriate pin on the I/O connector of your DAQ device. The clock source you supply must be a TTL signal. Refer to the Generate N Updates-ExtUpdateClk VI in the `examples\daq\anlogout\anlogout.llb` for an example of this process.

To use an external update clock, you must set the **clock source** of the AO Start VI to **I/O connector**. When you connect your external clock, you find that different DAQ devices use different pins for this input. These input pins are described in Table 7-1.

Table 7-1. External Update Clock Input Pins

Device	External Update Clock Input Pin
All E Series Devices with analog output	Any PFI pin (Default: PFI5/UPDATE*)
Lab-PC+ 1200 devices AT-AO-6/10	EXTUPDATE*

For waveform generation, you must supply an array of waveform data. The example Generate N Updates-ExtUpdateClk VI described previously uses data created in the Compute Waveform VI. When you run the example VI, the data is output on channel 0 (the DAC0OUT pin) of your DAQ device.

Supplying an External Test Clock from Your DAQ Device

To use an external update clock when you do not have an external clock available, create an external test clock using outputs from a counter/timer on your DAQ device, and then wire the output to your external update clock source.

If your DAQ device has an FOUT or FREQ_OUT pin, you can generate a 50% duty-cycle TTL pulse train using the Generate Pulse Train on FOUT or FREQ_OUT VI, available in `examples\daq\counter\DAQ-STC.llb`. The advantage of this VI is that it does not use one of the available counters, which you might need for other reasons.

You also can use the Pulse Train VIs to create an external test clock. These VIs are located in `examples\daq\counter\DAQ-STC.llb`, `examples\daq\counter\NI-TIO.llb`, `examples\daq\counter\Am9513.llb`, and `examples\daq\counter\8253.llb`.

Simultaneous Buffered Waveform Acquisition and Generation

This section describes how to perform buffered waveform acquisition and generation simultaneously on the same DAQ device.

Using E Series MIO Boards

E series devices, such as the PCI-MIO-16E-1, have separate counters dedicated to analog input and analog output timing. For this reason, they are the best choice for simultaneous input/output.

Software Triggered

Open the Simul AI/AO Buffered (E Series MIO) VI, available in `examples\daq\anlog_io\anlog_io.llb`, and examine its block diagram.

This example VI uses Intermediate DAQ VIs. By following the **error** cluster wire, which enters each DAQ VI on the bottom left and exits on the bottom right, you can see that because of data dependency, the waveform generation starts before the waveform acquisition, and each task is configured to run continuously. This example VI is software-triggered, because it starts via software when you click the **Run** button.

Once you call the AO Start and AI Start VIs, the While Loop executes. Inside the While Loop, the AI Read VI returns acquired data from the analog input buffer. There is not a call to the AO Write VI inside the While Loop because it is not needed if the same data from the first AO Write VI is regenerated continuously. To generate new data each time the While Loop iterates, add an AO Write VI inside the While Loop. The While Loop stops when an error occurs or you click the **Stop** button. Your DAQ device resources are cleared by calling the AI Clear and AO Clear VIs after the loop stops.

Hardware Triggered

Open the Simul AI/AO Buffered Trigger (E Series MIO) VI, available in `examples\daq\anlog_io\anlog_io.llb`, and examine its block diagram.

Although this VI is similar to the Simul AI/AO Buffered (E Series MIO) VI described previously, it is more advanced because it uses a hardware trigger. The waveform acquisition trigger is set up with the **trigger type** input to the AI Start VI set to **digital A** (start), and by default this trigger is expected on the PFI0 pin. Hardware triggering for waveform generation requires an additional VI. The AO Trigger and Gate Config VI is an advanced analog output VI for E Series boards only. The trigger parameters are set using three inputs. The **trigger or gate source** is used to choose the source of your trigger, such as a PFI pin or a RTSI pin. The **trigger or gate source specification** is used in conjunction with the **trigger or gate source** to choose which PFI or RTSI pin number to use, such as 0 through 9 for a PFI pin. The **trigger or gate condition** is used to select a rising or falling trigger edge. The default analog output trigger for this example is a rising edge on PFI0. Because this is the same pin as the analog input trigger, the waveform acquisition and generation start simultaneously. However, they are not controlled by independent counter/timers, so you can run them at different rates.

Using Lab/1200 Boards

Lab/1200 boards, such as the Lab-PC-1200 or the DAQCard-1200, also can perform simultaneous waveform acquisition and generation. The approach is similar to the previous descriptions. Refer to the examples Simul AI/AO Buffered (Lab/1200) VI and Simul AI/AO Buffered Trigger (Lab/1200) VI located in `examples\daq\anlog_io\anlog_io.llb` for examples of how this acquisition and generation is performed.

8

Digital I/O

This chapter explains digital I/O for data acquisition.

Things You Should Know about Digital I/O

Digital I/O interfaces often are used to control processes, generate patterns for testing, and communicate with peripheral equipment such as heaters, motors, and lights. Digital I/O components on DAQ devices and SCXI modules consist of hardware parts that generate or accept binary on/off signals, where on is typically 5 V and off is 0 V.

As shown in Figure 8-1, all digital lines are grouped into ports on DAQ devices and banks on SCXI modules. The number of digital lines per port or bank is specific to the particular device or module used, but most ports or banks consist of four or eight lines. Except for the 653*X* (DIO-32HS) in unstrobed mode, TIO-10, and E Series devices, the lines within the same port or bank must all be of the same direction (must either be all input or all output), as shown in Figure 8-1. By writing to or reading from a port, you can simultaneously set or retrieve the states of multiple digital lines.

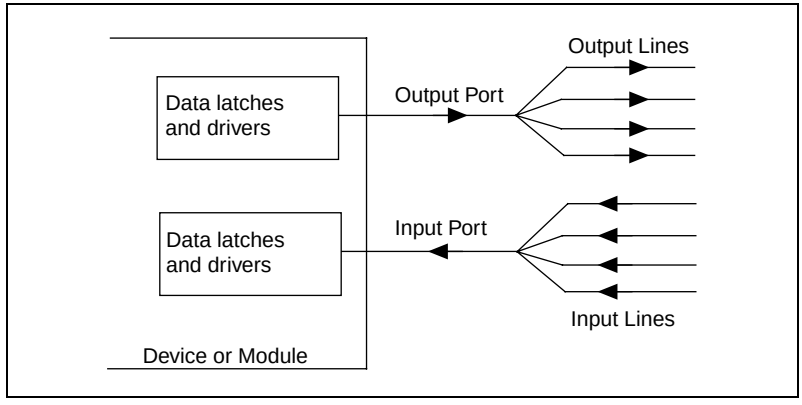


Figure 8-1. Digital Ports and Lines

Types of Digital Acquisition/Generation

There are several types of digital acquisition/generation*: unstrobed* (or *static*) and *strobed* (or *handshaked*), which includes *pattern* I/O. With unstrobed digital I/O, your system updates the digital lines immediately through software calls. With handshaked digital I/O, a device or module accepts or transfers data after a digital pulse has been received. With pattern I/O, data, or patterns, are written or read at fixed rate. The 653*X* family of boards can perform pattern I/O.

Handshaking can be either nonbuffered or buffered. Not all devices and modules support handshaking.

Knowing Your Digital I/O Chip

The digital I/O chips in most National Instruments DAQ devices belong to one of three families: 653*X*, E Series, or 8255.

- The 653*X* family includes devices such as the PXI-6533 and PCI-DIO-32HS.
- The E Series family typically has one 8-bit digital I/O port as a part of the DAQ-STC.
- The 8255 family includes low-cost Lab/1200 devices and 650*x* type boards (DIO-24 and DIO-96) which use the 8255 DIO chip.

Note Some E Series boards have more than eight digital lines. These boards typically have an additional 8255 chip. This means that with regards to digital I/O, they belong to both the E Series and 8255 families.

653*X* Family

The 653*X* family of digital devices uses the National Instruments DAQ-DIO ASIC, a 32-bit general-purpose digital I/O interface specifically designed for high performance. The 653*X* devices can perform unstrobed I/O, pattern I/O, and high-speed data transfer using a wide range of handshaked protocols. These devices also are equipped with sophisticated trigger circuitry to start or stop the digital data transfer based on several different types of events. These devices also feature the RTSI bus (PCI and AT version) and PXI trigger bus (PXI) to synchronize digital and timing signals with other devices.

E Series Family

E Series boards have one port of eight digital I/O lines. The direction of each line is software programmable on a per-line basis. The digital input circuitry has an 8-bit register that can read back outgoing digital signals and read incoming signals. These boards perform unstrobed I/O. Some E Series boards have an additional 24 digital I/O lines provided through an 8255 PPI. For discussions in this document, these boards can be thought of as belonging to both the E Series and 8255 families.

8255 Family

Many digital I/O boards use the 8255 programmable peripheral interface (PPI). This PPI controls 24 bits of digital I/O and has three 8-bit ports: A, B, and C. Each port can be programmed as input or output. Ports A and B always are used for digital I/O, while port C can be used for I/O or handshaking. Most boards have from 3 to 12 8-bit ports. A port width must be a multiple of 8 bits with a maximum of 32 bits. These boards perform unstrobed or handshaked digital I/O.

Immediate Digital I/O

This section focuses on transferring data across a single port. The most common way to use digital lines is with unstrobed (immediate) digital I/O. All DAQ devices and SCXI modules with digital components support this mode.

When your program calls a function (subVI) in unstrobed digital I/O mode, LabVIEW immediately writes or reads digital data. If the digital direction is set to output, LabVIEW updates the digital line or port output state. If the digital direction is set to input, LabVIEW returns the current digital line or port value. For each function called, LabVIEW inputs or outputs only one value on each digital line in this mode. You can completely configure port (and for some devices, line) direction in software, and you can switch directions repeatedly in a program.

Application examples of unstrobed digital I/O include controlling relays and monitoring alarm states. You also can use multiple ports or groups of ports to perform digital I/O functions. If you want to group digital ports, you must use Intermediate or Advanced I/O VIs in LabVIEW.

Using Channel Names

If you have configured channels using the DAQ Channel Wizard, **digital channel** can consist of a digital channel name. The channel name may refer to either a port or a line in a port. You do not need to specify **device**, **line**, or **port width** because LabVIEW does not use these inputs if a channel name is specified in **digital channel**.

As an alternative, **digital channel** can consist of a port number. The port number specifies the port of digital lines that you will use during your digital operation. In this case, to further define your digital operation, you also must specify **device**, **line**, and **port width** where applicable.
The **device** input identifies the DAQ device you are using. The **line** input is an individual port bit or line in the port specified by **digital channel**. The **port width** input specifies the number of lines that are in the port you are using.

If you are using an SCXI module for nonlatched digital I/O and you are not using channel names, refer to the *SCXI Channel Addressing* section in Chapter 9, *SCXI—Signal Conditioning*, for instructions on how to specify port numbers.

The **pattern** or **line state** is the value(s) you want to read from or write to a device. You can display pattern values in decimal (default), hexadecimal, octal, or binary form.

Immediate I/O Using the Easy Digital VIs

You can use the Easy Digital VIs for nonlatched digital I/O. There are four Easy I/O subVIs you can use to immediately read data from or write data to a single digital line or to an entire port. These subVIs are available on the **Functions»Data Acquisition»Digital I/O** palette. Use the Easy Digital VIs for most digital testing purposes. All of the Easy Digital VIs have error reporting. The following sections identify the examples you can access for further information.

653*X* Family

The examples Read from 1 Dig Line(653*X*) VI and Write to 1 Dig Line(653*X*) VI show how to use the Easy Digital I/O VIs to read from or write to one digital line. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to use the Easy Digital I/O VIs. You can write similar examples to read from or write to a single port.

E Series Family

The examples Read 1 Pt from Dig Line(E) VI and Write 1 Pt to Dig Line(E) VI show how to use the Easy Digital I/O VIs to read from or write to one digital line. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to use the Easy Digital I/O VIs. You can write similar examples to read from or write to a single port.

8255 Family

The examples Read from 1 Dig Line(8255) VI and Write to 1 Dig Line(8255) VI show how to use the Easy Digital I/O VIs to read from or write to one digital line. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to use the Easy Digital I/O VIs. You can write similar examples to read from or write to a single port.

Immediate I/O Using the Advanced Digital VIs

The Advanced Digital I/O subVIs give you more programming flexibility. Only the following three subVIs are needed to create the examples in this section: DIO Port Config VI, DIO Port Read VI, and DIO Port Write VI.

653*X* Family

The examples Read from 1 Dig Port(653X) VI and Write to 1 Dig Port(653X) VI show how to use the Advanced Digital I/O VIs to read from or write to one digital 8-bit port. The examples Read from 2 Dig Ports(653X) VI and Write to 2 Dig Ports(653X) VI show how to use the Advanced Digital I/O VIs to read from or write to two separate 8-bit ports. The examples Read from Digital Port(653X) VI and Write to Digital Port(653X) VI show how to use the Advanced Digital I/O VIs to read from or write to one digital port where the port width can be 8, 16, 24, or 32 bits. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to use the Advanced Digital I/O VIs.

E Series Family

The examples Read from 1 Dig Port(E) VI and Write to 1 Dig Port(E) VI show how to use the Advanced Digital I/O VIs to read from or write to one digital 8-bit port. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to use the Advanced Digital I/O VIs.

8255 Family

The examples Read from 1 Dig Port(8255) VI and Write to 1 Dig Port(8255) VI show how to use the Advanced Digital I/O VIs to read from or write to one digital 8-bit port. The examples Read from 2 Dig Ports(8255) VI and Write to 2 Dig Ports(8255) VI show how to use the Advanced Digital I/O VIs to read from or write to two separate 8-bit ports. The examples Read from Digital Port(8255) VI and Write to Digital Port(8255) VI show how to use the Advanced Digital I/O VIs to read from or write to one digital port where the port width can be 8, 16, 24, or 32 bits. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to use the Advanced Digital I/O VIs.

Handshaking

If you want to pass a digital pattern after receiving a digital pulse, you should use strobed (handshaked) digital I/O. Handshaking allows you to synchronize digital data transfer between your DAQ device and instrument. For example, you might want to acquire an image from a scanner. The process involves the following steps:

1. The scanner sends a pulse to your DAQ device after the image has been scanned and it is ready to transfer the data.
2. Your DAQ device reads an 8-, 16-, or 32-bit digital pattern.
3. Your DAQ device then sends a pulse to the scanner to let it know the digital pattern has been read.
4. The scanner sends out another pulse when it is ready to send another digital pattern.
5. After your DAQ device receives this digital pulse, it reads the data.
6. This process repeats until all the data is transferred.

Many DAQ devices support digital handshaking, including the following:

- 653*X* family devices
  - 6533 (DIO-32HS) devices
  - DIO-32F
- 6534 devices
- 8255 family devices
  - 6503 (DIO-24) devices
  - 6507/6508 (DIO-96) devices
  - Lab/1200 Series devices

- E Series family devices
  - MIO-16DE-10
  - 6025E devices

Note Only E Series boards with more than eight digital lines—those that have an additional 8255 chip onboard—support handshaking. These boards also are part of the 8255 family.

Note You cannot use channel names that were configured in the DAQ Channel Wizard with handshaking.

Handshaking Lines

653*X* Family

The names and functions of handshaking signals vary. The DIO-32 devices have two main handshaking lines—the REQ (request) line and the ACK (acknowledge) line. REQ is an input indicating the external device is ready. ACK is an output indicating the DIO-32 device is ready. Burst mode on a 6533 device also uses a third handshaking signal, PCLK (peripheral clock).

8255 Family

For 8255-based DAQ devices that perform handshaking, there are four handshaking signals:

- Strobe Input (STB)
- Input Buffer Full (IBF)
- Output Buffer Full (OBF)
- Acknowledge Input (ACK)

You use the STB and IBF signals for digital input operations and the OBF and ACK signals for digital output operations. When the STB line is low, LabVIEW loads data into the DAQ device. After the data has been loaded, IBF is high, which tells the external device that the data has been read. For digital output, OBF is low while LabVIEW sends the data to an external device. After the external device receives the data, it sends a low pulse back on the ACK line. Refer to your hardware manual to determine which digital ports you can configure for handshaking signals.

Digital Data on Multiple Ports

653*X* Family

You can group multiple ports together so you can send more digital values out at a time. For DIO-32 devices, the ports in the group determine which handshaking lines are used. If the group includes port 0 or 1, handshaking occurs on the group 1 handshaking lines. Otherwise, if the group consists only of a combination of ports 2 and 3, handshaking occurs on the group 2 handshaking lines. In either case, the LabVIEW group number does not affect which handshaking lines are used.

8255 Family

For 8255 devices, the ports in the group and the order of the ports both affect which handshaking lines are used. If you want to group ports 0 and 1 and you list the ports in the order of **0:1**, you should use the handshaking lines associated with port 1.

In other words, always use the handshaking lines associated with the last port in the list. So, if the ports are listed **1:0**, use the handshaking lines associated with port 0.

For 8255-based devices that perform handshaking, such as the DIO-24 or DIO-96, connect all the STB lines together if you are grouping ports for digital input, as shown in Figure 8-2. Connect only the IBF line of the last port in the port list to the other device. No connection is needed for the IBF signals for the other ports in the port list.

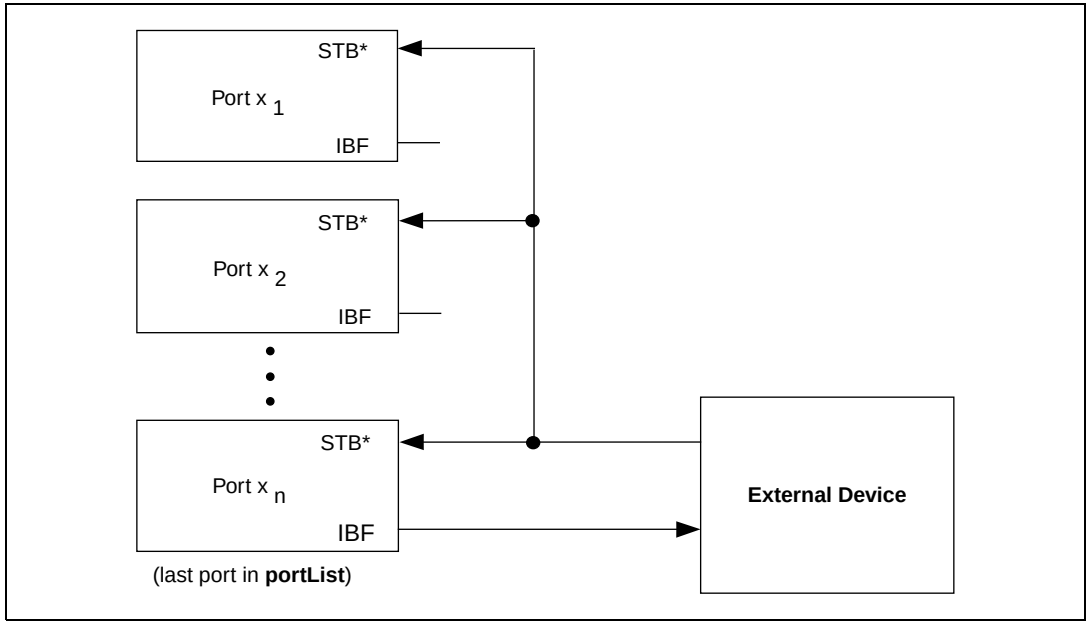


Figure 8-2. Connecting Signal Lines for Digital Input

If you group ports for digital output on an 8255-based device, connect only the handshaking signals of the last port in the port list, as shown in Figure 8-3.

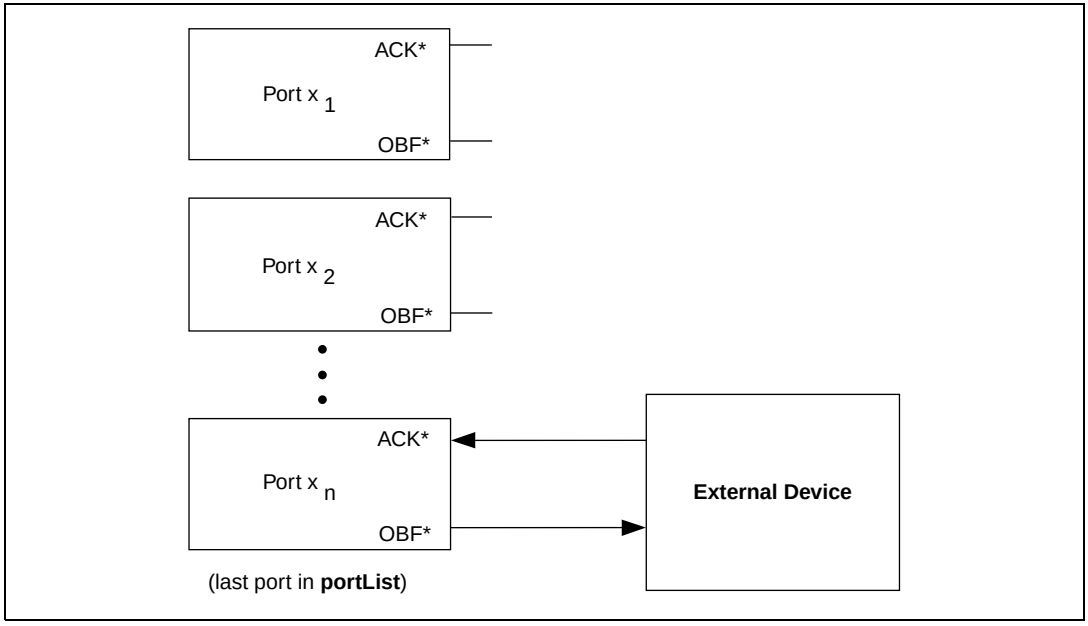


Figure 8-3. Connecting Signal Lines for Digital Output

Types of Handshaking

Digital handshaking can be either nonbuffered or buffered. Nonbuffered handshaking is similar to unstrobed digital I/O because LabVIEW updates the digital lines immediately after every digital or handshaked pulse.

Note For the 6533 devices, LabVIEW returns immediately after storing data in its FIFO buffer.

With buffered handshaking, LabVIEW stores digital values in memory to be transferred after every handshaked pulse. Both nonbuffered and buffered handshaking transfer one digital value after each handshaked pulse. Use nonbuffered handshaking for basic digital applications. Use buffered handshaking when your application requires multiple handshaking pulses or high speeds. By using a buffer with multiple handshaking pulses, the software spends less time reading or writing data, leaving more time for other operations.

Note On an AT-DIO-32F device with nonbuffered handshaking, you can group 1, 2, or 4 ports together. For buffered handshaking on the AT-DIO-32F, you can group only 2 or 4 ports together.

You can use only Intermediate or Advanced Digital I/O VIs for digital handshaking in LabVIEW. The Intermediate I/O VIs work for most nonbuffered and buffered digital handshaking applications. However, for some DAQ devices, you might need a combination of Intermediate and Advanced I/O VIs.

Nonbuffered Handshaking

Nonbuffered handshaking occurs when your program transfers one digital value after receiving a digital pulse on the handshaking lines. LabVIEW does not store these digital values in computer memory. You should use nonbuffered handshaking when you expect only a few digital handshaking pulses. For multiple-pulsed or high-speed applications, you should use buffered handshaking.

653*X* Family

The example Dig Word Handshake In(653X) VI shows how to read nonbuffered data using handshaking. The example Dig Word Handshake Out(653X) VI shows how to write nonbuffered data using handshaking. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to read nonbuffered data using handshaking.

8255 Family

The example Dig Word Handshake In(8255) VI shows how to read nonbuffered data using handshaking. The example Dig Word Handshake Out(8255) VI shows how to write nonbuffered data using handshaking. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to read nonbuffered data using handshaking.

Buffered Handshaking

With buffered handshaking, you can store multiple points in computer memory. You also can access data as it is being acquired, through the **read location** and **scan backlog** terminals of the DIO Read subVI. Use this technique if multiple pulses are expected on the handshaking lines. Buffered handshaking comes in several forms: simple, iterative, and circular. You can use simple and iterative buffered handshaking on all DAQ devices that support handshaking. You can perform circular-buffered handshaking only on 6533 devices.

Simple-Buffered Handshaking

You can think of a simple buffer as a storage place in computer memory, where **buffer size** equals the number of updates multiplied by the number of ports. With simple-buffered handshaking, one or more ports can be used to read or write data. All of the data is handshaked into the buffer before it is read into LabVIEW.

653*X* Family

The examples Buff Handshake Input VI and Buff Handshake Output VI show how to read or write data, respectively, using buffered handshaking. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to read or write data using buffered handshaking.

The examples Burst Mode Input VI and Burst Mode Output VI demonstrate the use of the burst-mode protocol for maximum device throughput. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to use the burst-mode protocol.

8255 Family

The example Dig Buf Handshake In(8255) VI shows how to read buffered data using handshaking. The example Dig Buf Handshake Out(8255) VI shows how to write buffered data using handshaking. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to read buffered data using handshaking.

Iterative-Buffered Handshaking

Iterative-buffered handshaking sets up a buffer the same way as simple-buffered handshaking. With iterative-buffered handshaking, one or more ports can be used to read or write data. What differs between iterative-buffered handshaking and simple-buffered handshaking is that with iterative handshaking, data is read from the buffer before the buffer has been filled. This allows you to see data as it is acquired as opposed to waiting until all the data has been handshaked into the buffer. It also may free up processor time spent waiting for the buffer to fill.

653*X* Family

The example Dig Buf Hand Iterative(653X) VI shows how to read data as it is being handshaked into a buffer.

8255 Family

The example Dig Buf Hand Iterative(8255) VI shows how to read data as it is being handshaked into a buffer. Another example, Dig Buf Hand Occur(8255) VI, also uses iterative reads. This example, which works only on the Windows platform, also shows how to use DAQ Occurrences to search for specific bit patterns in a port. Refer to the VIs in the `examples\daq\digio.llb` for examples of reading data as it is being handshaked into a buffer.

Circular-Buffered Handshaking

A circular buffer differs from a simple buffer only in the way your program places the data into it and retrieves data from it. A circular buffer fills with data the same as a simple buffer, but when it reaches the end of the buffer LabVIEW returns to the beginning of the buffer and fills up the same buffer again. Use simple-buffered handshaking when you have a predetermined number of values to acquire or generate. Use circular-buffered handshaking when you want to acquire or generate data continuously.

Circular-buffered handshaking is similar to simple-buffered handshaking in that both types of handshaking place data in a buffer. However, a circular-buffer application returns to the beginning of the buffer when it reaches the end and fills the same buffer again.

Note Circular-buffered handshaking works only on 653*X* devices.

The examples Cont Handshake Input VI and Cont Handshake Output VI show how to read or write data respectively, using a circular buffer to implement continuous-buffered handshaking. Refer to the VIs in the `examples\daq\digio.llb` for examples of reading and writing data using a circular buffer.

Pattern I/O

This section describes pattern I/O, which is also known as pattern generation. Pattern I/O implies reading or writing digital data (or patterns) at a fixed rate. This mode is especially useful when you want to synchronize digital I/O with other events. For example, you might want to synchronize or time stamp digital data with analog data being acquired by another device.

Digital I/O timing can be controlled by one of the following methods:

- Onboard clock on 653*X* (DIO-32HS) devices
- User-supplied clock
- Change detection mode (input only), where a pattern is acquired whenever there is a state transition on one of the data lines

There are two general categories of timed digital I/O:

- With Finite Timed Digital I/O, a predetermined number of patterns are acquired or generated at a rate controlled by one of the timing sources discussed above.
- With Continuous Timed Digital I/O, digital data is continuously acquired or generated until the user stops the process. The rate can be controlled by one of the timing sources discussed above.

Finite Pattern I/O

In finite pattern I/O mode, LabVIEW allocates a single buffer of computer memory large enough to hold all the patterns. Optionally, you can use triggering in this mode of digital I/O.

Finite Pattern I/O without Triggering

In this mode, the start and/or stop of the digital I/O process is not controlled by an external trigger event. Instead, it initiates as soon as the VI is run.

The Buffered Pattern Input VI and Buffered Pattern Output VI show how to perform finite pattern I/O. In these examples, the 653*X* onboard clock is programmed to the **clock frequency** value when the **clock source** parameter on the VI front panel is set to **internal**. A user-supplied clock is used when the **clock source** is set to **external**. Refer to the VIs in `examples\daq\digio.llb` for examples of how to perform finite pattern I/O.

The Change Detection Input VI reads in a fixed number of patterns, where each pattern is read in when there is a state transition on one of the data lines. Further, the **line mask** parameter can be used to selectively monitor transitions only on certain lines. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to read in a fixed number of patterns.

The Multi Board Synchronization VI shows how to synchronize two 653*X* devices to acquire data simultaneously. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to synchronize two 6533 devices.

Finite Pattern I/O with Triggering

You can use triggering to control the start and/or stop of the digital I/O process. The trigger event can be one of the following:

- Rising edge of a digital pulse
- Falling edge of a digital pulse
- Pattern match trigger, where the trigger event occurs when data on the lines being monitored matches a specified pattern
- Pattern not match trigger, where the trigger event occurs when data on the lines being monitored differs from a specified pattern

The Buffered Pattern Input-Trig VI, Buffered Pattern Output-Trig VI, and Start & Stop Trig VI show how to use the triggering features on the 653*X* devices. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to use the triggering features.

Continuous Pattern I/O

In this mode, digital data is continuously acquired or generated until the user stops the process. The data is stored in a circular buffer in memory in order to reuse finite computer memory resources.

Circular buffering for input works in the following manner. While the buffer is being filled by data acquired from the board, LabVIEW reads data out of the buffer for processing, for example, to save to disk or update screen graphics. When the buffer is filled, the operation resumes at the beginning of the buffer. In this manner, continuous data collection and processing can be sustained indefinitely, assuming your application can retrieve and process data faster than the buffer is being filled. A similar buffering technique is used when generating digital output data continuously.

The Cont Pattern Input VI, Cont Change Detection Input VI, and Cont Pattern Output VI show how to continuously acquire or generate digital data. Refer to the VIs in the `examples\daq\digio.llb` for examples of how to continuously acquire or generate digital data.

9

SCXI—Signal Conditioning

SCXI is a highly expandable signal conditioning system. This chapter describes the basic concepts of signal conditioning, the setup procedure for SCXI hardware, the hardware operating modes, the procedure for software installation and configuration, the special programming considerations for SCXI in LabVIEW, and some common SCXI applications.

Things You Should Know about SCXI

What Is Signal Conditioning?

Transducers can generate electrical signals to measure physical phenomena, such as temperature, force, sound, or light. Table 9-1 lists some common transducers.

Table 9-1. Phenomena and Transducers

Phenomena	Transducer
Temperature	Thermocouples Resistance temperature detectors (RTDs) Thermistors Integrated circuit sensor
Light	Vacuum tube photosensors Photoconductive cells
Sound	Microphone
Force and pressure	Strain gauges Piezoelectric transducers Load cells
Position (displacement)	Potentiometers Linear voltage differential transformer (LVDT) Optical encoder

Table 9-1. Phenomena and Transducers (Continued)

Phenomena	Transducer
Fluid flow	Head meters Rotational flowmeters Ultrasonic flowmeters
pH	pH electrodes

To measure signals from transducers, you must convert them into a form that a DAQ device can accept. For example, the output voltage of most thermocouples is very small and susceptible to noise. Therefore, you may need to amplify and/or filter the thermocouple output before digitizing it. The manipulation of signals to prepare them for digitizing is called signal conditioning. Common types of signal conditioning include the following:

- Amplification
- Linearization
- Transducer excitation
- Isolation
- Filtering

Figure 9-1 shows some common types of transducers/signals and the required signal conditioning for each.

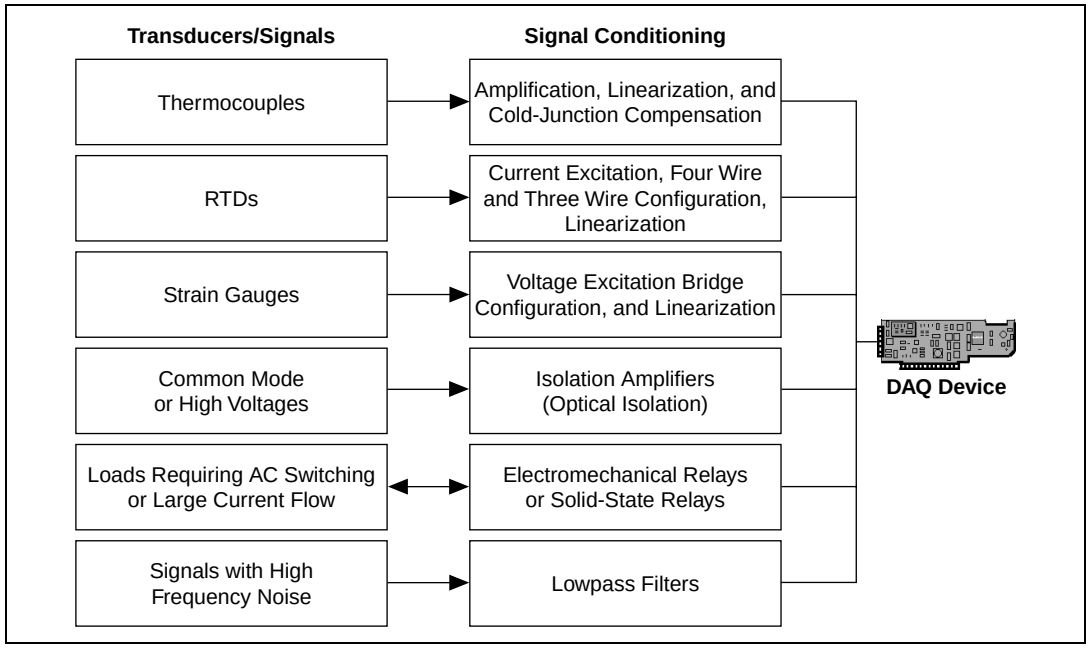


Figure 9-1. Common Types of Transducers/Signals and Signal Conditioning

Amplification

The most common type of signal conditioning is *amplification*. Amplify electrical signals to improve the digitized signal accuracy and to reduce noise.

For the highest possible accuracy, amplify the signal so the maximum voltage swing equals the maximum input range of the ADC, or digitizer. Your system should amplify low-level signals at the DAQ device or at the SCXI module located nearest to the signal source, as shown in Figure 9-2.

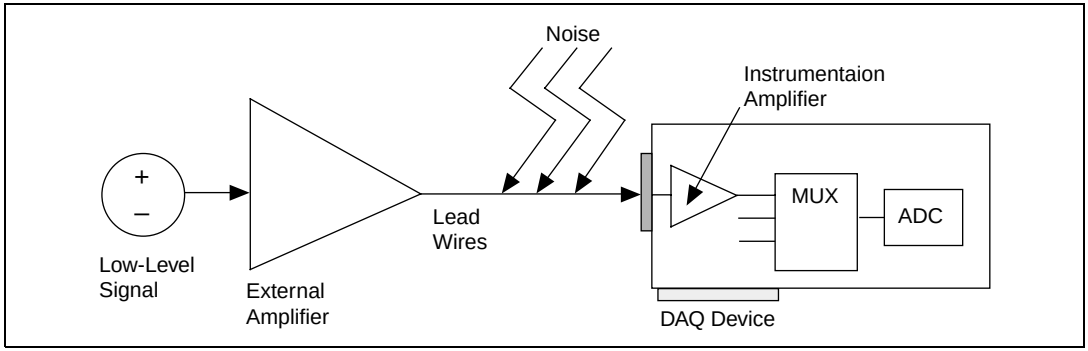


Figure 9-2. Amplifying Signals near the Source to Increase Signal-to-Noise Ratio (SNR)

Tip Use shielded cables or a twisted pair of cables. By minimizing wire length, you can minimize noise the lead wires pick up. Also, keep signal wires away from AC power cables and monitors to reduce 50 or 60 Hz noise.

If you amplify the signal at the DAQ device, the signal is measured and digitized with noise that may have entered the lead wires. However, if you amplify the signal close to the signal source with an SCXI module, noise has a less destructive effect on the signal. In other words, the digitized representation is a better reflection of the original low-level signal. Refer to *Application Note 025, Field Wiring and Noise Considerations for Analog Signals* for more information about analog sin gals. You can access this note from the National Instruments Developer Zone, `zone.ni.com`.

Linearization

Many transducers, such as thermocouples, have a nonlinear response to changes in the physical phenomena being measured. LabVIEW can linearize the voltage levels from transducers, so the voltages can be scaled to the measured phenomena. LabVIEW provides simple scaling functions to convert voltages from strain gauges, RTDs, thermocouples, and thermistors.

Transducer Excitation

Signal conditioning systems can generate excitation for some transducers. Strain gauges and RTDs require external voltage and currents, respectively, to excite their circuitry into measuring physical phenomena. This type of excitation is similar to a radio that needs power to receive and decode audio signals. Several plug-in DAQ devices and SCXI modules, including the

SCXI-1121 and SCXI-1122 modules, provide the necessary excitation for transducers.

Isolation

Another common way to use SCXI is to isolate the transducer signals from the computer for safety purposes. When the signal being monitored contains large voltage spikes that could damage the computer or harm the operator, do not connect the signal directly to a DAQ device without some type of isolation.

You also can use isolation to ensure that measurements from the DAQ device are not affected by differences in ground potentials. When the DAQ device and the signal are not referenced to the same ground potential, a ground loop can occur. Ground loops can cause an inaccurate representation of the measured signal. If the potential difference between the signal ground and the DAQ device ground is large, damage can even occur to the measuring system. Using isolated SCXI modules eliminates the ground loop and ensures that the signals are accurately measured.

Filtering

Signal conditioning systems can filter unwanted signals or noise from the signal you are trying to measure. You can use a noise filter on low-rate (or slowly changing) signals, such as temperature, to eliminate higher-frequency signals that can reduce signal accuracy. A common use of a filter is to eliminate the noise from a 50 or 60 Hz AC power line. A lowpass filter of 4 Hz, which exists on several SCXI modules, is suitable for removing the 50 or 60 Hz AC noise from signals sampled at low rates. A lowpass filter eliminates all signal frequency components above the cutoff frequency. Many SCXI modules have lowpass filters that have software-selectable cutoff frequencies from 10 Hz to 25 kHz.

Hardware and Software Setup for Your SCXI System

SCXI hardware conditions signals close to the signal source and increases the number of analog and digital signals that a DAQ device can analyze. With PC-compatible computers, you can configure SCXI in two ways—a front-end signal conditioning system for plug-in DAQ devices, or an external data acquisition and control system. Furthermore, when SCXI is configured as an external data acquisition and control system, you can connect it to the parallel port of the computer using an SCXI-1200, or the serial port of the computer using either an SCXI-2000 remote chassis or an SCXI-2400 remote communications module in an SCXI-100*X* chassis.

For Macintosh computers, you can use SCXI hardware only as a front-end signal conditioning system for plug-in DAQ devices. Figure 9-3 demonstrates these configurations.

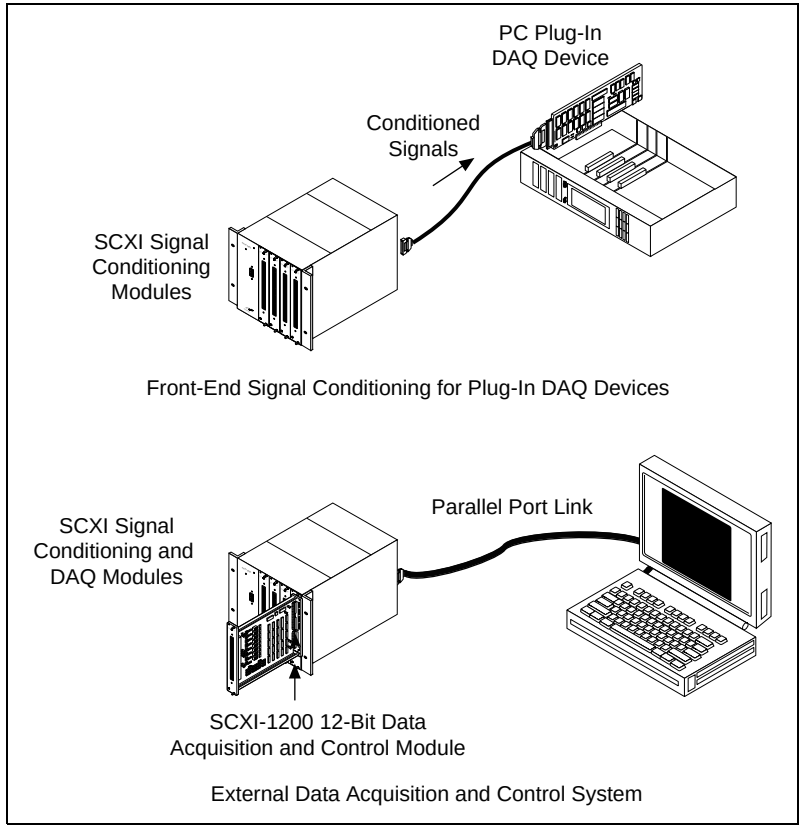


Figure 9-3. SCXI System

Figure 9-4 shows the components of an SCXI system. An SCXI system consists of an SCXI chassis that houses signal conditioning modules, terminal blocks that plug directly into the front of the modules, and a cable assembly that connects the SCXI system to a plug-in DAQ device or the parallel or serial port of a computer. If you are using SCXI as an external DAQ system where there are no plug-in DAQ devices, you can use the SCXI-1200 module, which is a multifunction analog, digital, and timing I/O (counters) module. The SCXI-1200 can control several SCXI signal conditioning modules installed in the same chassis. The functionality of the SCXI-1200 module is similar to the plug-in 1200 series devices.

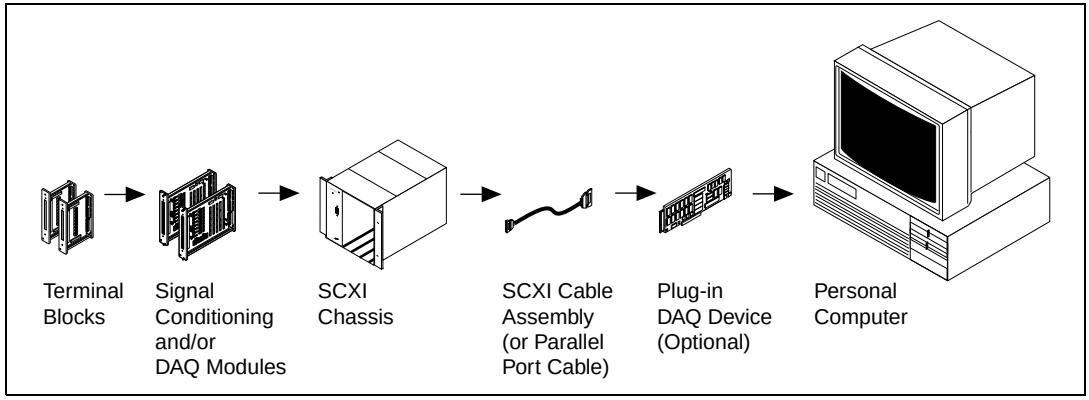


Figure 9-4. Components of an SCXI System

Note For information on how to set up each module and transducer, consult your hardware user manuals and the *Getting Started with SCXI* manual.

How do you transfer data from the SCXI chassis to the DAQ device or parallel or serial port? Figure 9-5 shows a diagram of an SCXI chassis. When you use SCXI as a front-end signal conditioning system, the analog and digital bus backplane, also known as the SCXIbus, transfers analog and/or digital data to the DAQ device. Some of the analog and digital lines on the DAQ device are reserved for SCXI chassis communication.

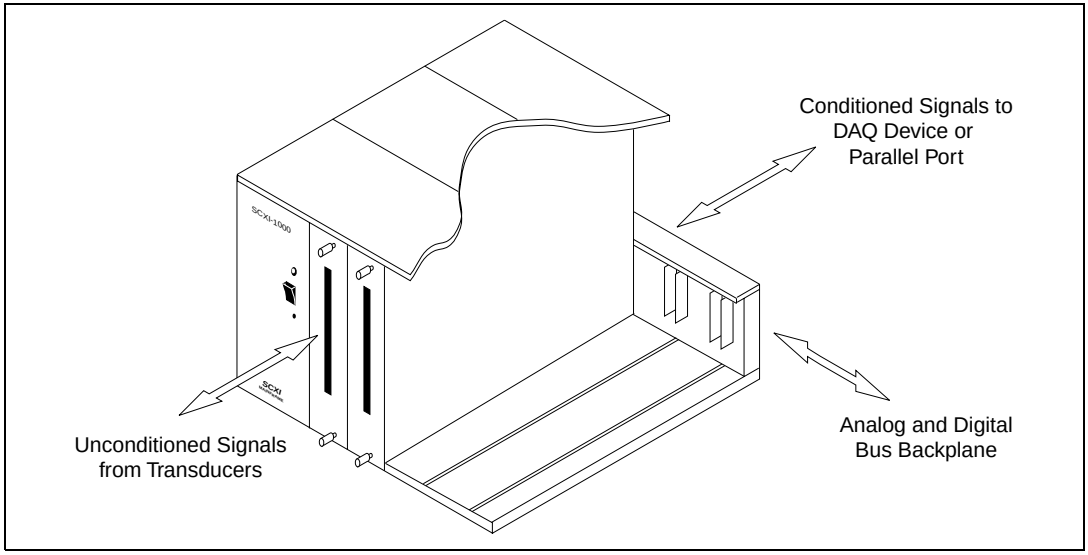


Figure 9-5. SCXI Chassis

When you use SCXI as an external DAQ system, only some of the digital I/O lines of the DAQ device are reserved for SCXI chassis communication when other modules are present. The DAQ device digitizes any analog input data and transfers it back to the computer through the parallel or serial port.

Note When using remote SCXI, be aware of the sampling rate limitations from the data being sent over the serial port. To reduce delays in serial port communication, National Instruments recommends that you use the fastest baud rate possible for the serial port of your computer. If you have a 16550 or compatible universal asynchronous receiver-transceiver (UART), you can use baud rates up to 57,600 baud. If you have an 8250 or compatible UART, you can use only up to 19,200 baud.

SCXI Operating Modes

The SCXI operating mode determines the way that DAQ devices access signals. There are two basic operating modes for SCXI modules, multiplexed and parallel. You designate the mode in the **operating mode** input in Measurement & Automation Explorer or the configuration utility. Also, you may have to set up jumpers on the module for the correct operating mode. Check your SCXI module user manual for more information.

Note National Instruments recommends that you use the multiplexed mode for most purposes, because this allows all channels on all modules to be accessed with a single DAQ device. Parallel mode can offer faster I/O for some modules, particularly the digital modules, but you would need a DAQ device for each module in the chassis.

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 device to a multiplexed analog input module, the DAQ device has access to multiplexed output of that module, as well as all other modules in the chassis through the SCXIbus. The analog input VIs route the multiplexed analog signals on the SCXIbus for you transparently. So, if you operate all modules in the chassis in multiplexed mode, you need to cable only one of the modules directly to the DAQ device.

Note MIO/AI devices, and Lab-PC+ and 1200 devices support multiple-channel and multiple-scan acquisitions in multiplexed mode. The Lab-LC, LPM devices, and DAQCard-700 support only single-channel or single-scan acquisitions in multiplexed mode.

When you connect a DAQ device 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 device by default.

Multiplexed Mode for the SCXI-1200 (Windows)

In multiplexed mode, the SCXI-1200 can access the analog signals on the SCXIbus. The DAQ VIs can multiplex the channels of analog input modules and send them on the SCXIbus. This means that if you configure the SCXI-1200 for multiplexed mode, you can read the multiplexed output from other SCXI analog input modules in the chassis.

Note The SCXI-1200 reads analog input module channels configured only in multiplexed mode, not in parallel mode.

Make sure that you change the jumper in the SCXI-1200 to the ground position to connect the SCXI-1200 and SCXIbus grounds together. Refer to your SCXI user manual for more information about connecting the SCXI-1200 and SCXIbus grounds.

Multiplexed Mode for Analog Output Modules

Because LabVIEW communicates with the multiplexed modules over the SCXIbus backplane, you need to cable only one multiplexed module in each chassis to a DAQ device to communicate with any multiplexed modules in the chassis.

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 mode, LabVIEW communicates the module channel states serially over the SCXIbus backplane.

Parallel Mode for Analog Input Modules

When an analog input module operates in parallel mode, the module sends each of its channels directly to a separate analog input channel of the DAQ device cabled to the module. You *cannot* multiplex parallel outputs of a module on the SCXIbus. You must cable a DAQ device directly to a module in parallel mode to access its input channels. In this configuration, the number of channels available on the DAQ device limits the total number of analog input channels. In some cases, however, you can cable more than one DAQ device to separate modules in an SCXI chassis. For example, you can use two AT-MIO-16E-2 devices operating in parallel mode and cable each one to a separate SCXI-1120 module in the chassis.

By default, when a module operates in parallel mode, the module sends its channel 0 output to differential analog input channel 0 of the DAQ device, the channel 1 output to analog input channel 1 of the DAQ device, and so on.

When you use the analog input VIs, specify the correct onboard channel for each parallel SCXI channel. If you are using a range of SCXI channels, LabVIEW assumes the onboard channel numbers match the SCXI channel numbers.

Parallel Mode for the SCXI-1200 (Windows)

In parallel mode, the SCXI-1200 reads only its own analog input channels. The SCXI-1200 does not have access to the analog bus on the SCXI backplane in parallel mode. You should use parallel mode if you are not using other SCXI analog input modules in the chassis with the SCXI-1200.

Parallel Mode for Digital Modules

When you operate a digital module in parallel mode, the digital lines on your DAQ device directly drive the individual digital channels on your SCXI module. You must cable a DAQ device directly to every module operated in parallel mode.

You may want 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 appropriate onboard port numbers instead of the SCXI channel string syntax in the digital VIs.

Note If you are using a 6507/6508 (DIO-96) or an AT-MIO-16DE-10 device, you also can operate a digital module in parallel mode using the digital ports on the second half of the NB5 or R1005050 ribbon cable (lines 51–100). Therefore, the DIO-96 can operate two digital modules in parallel mode, one module using the first half of the ribbon cable (lines 1–50), and another module using the second half of the ribbon cable (lines 51–100).

SCXI Software Installation and Configuration

After you assemble your SCXI system, you must run Measurement & Automation Explorer or the configuration utility to enter your SCXI configuration. LabVIEW needs the configuration information to program your SCXI system correctly.

Special Programming Considerations for SCXI

When you want LabVIEW to acquire data from SCXI analog input channels, you use the analog input VIs in the same way that you acquire data from onboard channels. You also read and write to your SCXI relays and digital channels using the digital VIs in the same way that you read and write to onboard digital channels. You can write voltages to your SCXI analog output channels using the analog output VIs. The following sections describe special programming considerations for SCXI in LabVIEW, including channel addressing, gains (limit settings), and settling time.

Note This section does not apply if you use the DAQ Channel Wizard to configure your channels. On Windows, the DAQ Channel Wizard is part of Measurement & Automation Explorer. If you use the DAQ Channel Wizard, you address SCXI channels the same way you address onboard channels—by specifying the channel name(s). LabVIEW configures your hardware by selecting the best input limits and gain for the named channel based on the channel configuration.

SCXI Channel Addressing

If you operate a module in parallel mode, you can specify an SCXI channel either by specifying the corresponding onboard channels or by using the SCXI channel syntax described in this section. If you operate the modules in multiplexed mode, you must use the SCXI channel syntax.

An SCXI channel number has four parts: the onboard channel (optional), the chassis ID, the module number, and the module channel.

In the following table of examples, `x` is any chassis ID, `y` is any module number, `a` is any module channel, and `b` is any module channel greater than `a`. `z` is the onboard channel from which the conditioned data is retrieved. If you operate in multiplexed mode, analog input channel 0 reads the data from the first cabled chassis. If you use VXI-SC submodules, LabVIEW ignores the onboard channel, because VXI-DAQ provides a special channel for retrieving data from submodules.

Channel List Element	Channel Specified
`obz!scx!mdy!a`	Channel `a` on the module in slot `y` of the chassis with ID `x` is multiplexed into onboard channel `z`.
`obz!scx!mdy!a:b`	Channels `a` through `b` inclusive on the module in slot `y` of the chassis with ID `x` are multiplexed into onboard channel `z`.
`obz!scx!mdy!(a,b,c)`	Channels `a`, `b`, and `c` (nonconsecutive) on the module in slot `y` of the chassis with ID `x` are multiplexed into onboard channel `z`. (Only supported on certain SCXI modules such as the SCXI-1125.)

The **channel** input for DAQ VIs is either a string (with the Easy I/O VIs) or an array of strings. Each string value can list the channels for only one module. With the array structure for channel values, you can list the channels for several modules. Therefore, for one scanning operation, you can scan several modules. You can scan an arbitrary number of channels for each module, but you must scan the channels of each module in consecutive, ascending order.

Note You do not need the SCXI channel string syntax to access channels on the SCXI-1200 module. Use `0` for channel 0, `1` for channel 1, and so on. The SCXI-1200 module is identified by its logical device number.

Note When you connect any type of SCXI module to a DAQ device, certain analog input and digital lines on the DAQ device are reserved for SCXI control. On MIO Series devices, lines 0, 1, and 2 are unavailable for general-purpose digital I/O. On MIO E Series devices, lines 0, 1, 2, and 4 are unavailable for general-purpose digital I/O.

For the fastest performance in parallel mode on digital modules, use the appropriate onboard port numbers instead of the SCXI channel string syntax in the digital VIs.

SCXI Gains

SCXI modules provide higher analog input gains than those available on most DAQ plug-in devices.

Enter the gain jumper settings in Measurement & Automation Explorer or the NI-DAQ Configuration utility for each channel on each module with jumpered gains. LabVIEW stores these gain settings and uses them to scale the input data. When you use the **input limits** control of the analog input VIs, LabVIEW chooses onboard gains that complement the jumpered SCXI gains to achieve the given input limits as closely as possible.

For analog input modules with programmable gains, LabVIEW uses the gain setting you enter in Measurement & Automation Explorer or the NI-DAQ Configuration utility for each module as the *default gain* for that module. LabVIEW uses the default gain for the module whenever you leave the **input limits** terminal to the analog input VIs unwired, or if you enter `0` for your upper and lower input limits.

When you use the **input limits** to specify non-zero limits for a module with programmable gains, LabVIEW chooses the most appropriate SCXI gain for the given limits. LabVIEW selects the highest SCXI gain possible for the given limits, and then selects additional DAQ device gains if necessary.

If your module has programmable gains and only one gain for all channels and you are using an MIO/AI DAQ device, you can specify different input limits for channels on the same module by splitting up your channel range over multiple elements of the channel array, and using a different set of input limits for each element. LabVIEW selects one module gain suitable for all of the input limits for that module, then chooses different MIO/AI gains to achieve the different input limits. The last three examples in Table 9-2 illustrate this method. The last example shows a channel list with two modules.

Table 9-2. SCXI-1100 Channel Arrays, Input Limits Arrays, and Gains

Array Index	SCXI-1100 Channel List Array	Input Limits Array	LabVIEW Selected SCXI Gain	LabVIEW Selected MIO/AI Gain
0	ob0!sc1!md1!0:7	–0.01 to 0.01	1000	1
0	ob0!sc1!md1!0:7	–0.001 to 0.001	2000	5[1]
0	sc1!md1!0:7	–0.001 to 0.001	2000	1
0 1	ob0!sc1!md1!0:3 ob0!sc1!md1!4:15	–0.1 to 0.1 –0.01 to 0.01	100 100	1 10
0 1	ob0!sc1!md1!0:15 ob0!sc1!md1!16:31	–0.01 to 0.01 –1.0 to 1.0	10 10	100[2] 1
0 1 2	ob0!sc1!md1!0:3 ob0!sc1!md1!4:15 ob0!sc1!md2!0:7	–1.0 to 1.0 –0.1 to 0.1 –0.01 to 0.01	10 10 1000	1 10 1

[1] Applies if the MIO/AI device supports a gain of 5 (some MIO/AI devices do not).

[2] This case forces a smaller gain at the SCXI module than at the MIO/AI device, because the input limits for the next channel range on the module require a small SCXI gain. This type of gain distribution is not recommended because it defeats the purpose of providing amplification for small signals at the SCXI module. The small input signals are only amplified by a factor of 10 before they are sent over the ribbon cable, where they are very susceptible to noise. To use the optimum gain distribution for each set of input signals, do not mix very small input signals with larger input signals on the same SCXI-1100 module unless you are sampling them at different times.

You can open the AI Hardware Config VI to see the gain selection. After running this VI, the **group channel settings** cluster array at the right side of the panel shows the settings for each channel. The **gain** indicator displays the total gain for the channel, which is the product of the SCXI gain and the DAQ device gain, and the actual limit settings. The **group channel settings** cluster array also shows the input limits for each channel.

LabVIEW scales the input data as you specified, unless you select binary data only. Therefore, the gains are transparent to the application. You can specify the input signal limits and let LabVIEW do the rest.

SCXI Settling Time

The filter and gain settings of your SCXI modules affect the settling time of the SCXI amplifiers and multiplexers. Always enter your jumpered filter settings and your jumpered gain settings (if applicable) in the Measurement & Automation Explorer or the configuration utility. LabVIEW uses the gain and filter settings to determine a safe interchannel delay that allows

the SCXI amplifiers and multiplexers to settle between channel switching before sampling the next channel.

LabVIEW calculates the delay for you. If you set a scan rate that is too fast to allow for the default interchannel delay, LabVIEW shrinks the interchannel delay and returns a warning from the AI Start or AI Control VIs. Refer to your hardware manuals for SCXI settling times.

You can open the advanced-level AI Clock Config VI to retrieve the channel clock selection. Set the **which clock** control to **channel clock 1**, and set the **clock frequency** to **–1.00 (no change)**. Now run the VI. The **actual clock rate specification** cluster is on the right side of the panel.

Note When using NI 406*X* devices with SCXI, you cannot use the external triggering feature of the NI 406*X* device.

Common SCXI Applications

Now that you have your SCXI system set up and you are aware of the special SCXI programming considerations, you should learn about some common SCXI applications. This section covers example VIs for analog input, analog output, and digital modules. For analog input, you will learn how to measure temperature (with thermocouples and RTDs) and strain (with strain gauges) using the SCXI-1100, SCXI-1101, SCXI-1102, SCXI-1112, SCXI-1121, SCXI-1122, SCXI-1125, SCXI-1141, SCXI-1142, SCXI-1143, and SCXI-1520 modules. If you are not measuring temperature or pressure, you still can gain basic conceptual information on how to measure voltages with an analog input module.

Four other analog input modules, the SCXI-1140, SCXI-1520, SCXI-1530, and SCXI-1531, are simultaneous sampling modules. All the channels acquire voltages at the same time, which means you can preserve interchannel phase relationships. After all channel voltages are sampled by going into hold mode, the software reads one channel at a time. When a scan of channels is done, the module returns to track mode until the next scan period. Both of these operations are performed by the analog input VIs. You can use any of the DAQ VIs, available in the `examples\daq\anlogin\anlogin.llb`, or the Getting Started Analog Input VI, available in the `examples\daq\run_me.llb`, to acquire data from the SCXI-1140, SCXI-1520, SCXI-1530, or SCXI-1531 module.

For analog output, you will learn how to output voltage or current values using the SCXI-1124 module. For digital I/O, you will learn how to input values on the SCXI-1162/1162HV modules and output values on the

SCXI-1160, SCXI-1161, and SCXI-1163/1163R modules. For switching, you will learn how to open, close, or make other channel connections on the SCXI-1127, SCXI-1128, SCXI-1150, and SCXI-1151.

Analog Input Applications for Measuring Temperature and Pressure

Two common transducers for measuring temperature are thermocouples and RTDs. A common transducer for measuring pressure is strain gauges. Read the following sections on special measuring considerations needed for each transducer.

If you use the DAQ Channel Wizard to configure your analog input channels, you can simplify the programming needed to measure your channels. This section describes methods of measuring data using named channels configured in the DAQ Channel Wizard and using the conventional method.

Measuring Temperature with Thermocouples

If you want to measure the temperature of the environment, you can use the temperature sensors in the terminal blocks. If you want to measure the temperature of an object away from the SCXI chassis, you must use a transducer, such as a thermocouple. A *thermocouple* is a junction of two dissimilar metals that gives varying voltages based on the temperature. However, when using thermocouples, you need to compensate for the thermocouple voltages produced at the screw terminal because the junction with the screw terminals itself forms another thermocouple. You can use the resulting voltage from the temperature sensor on the terminal block for cold-junction compensation (CJC). The CJC voltage is used when linearizing voltage readings from thermocouples into temperature values.

The SCXI modules used to measure temperature in this section are the SCXI-1100, SCXI-1101, SCXI-1102, SCXI-1112, SCXI-1120, SCXI-1120D, SCXI-1121, SCXI-1122, SCXI-1125, SCXI-1127, SCXI-1141, SCXI-1142, and SCXI-1143. Most of the terminal blocks used with these modules have temperature sensors that you can use for CJC.

In addition, the SCXI-1100, SCXI-1112, SCXI-1122, SCXI-1125, SCXI-1141, SCXI-1142, and SCXI-1143 offer a way for you to ground the module amplifier inputs so you can read the amplifier offset. You can subtract the amplifier offset value to determine the actual voltages.

Temperature Sensors for Cold-Junction Compensation

The temperature sensors in the terminal blocks for the analog input modules can be used for CJC. If you are operating your SCXI modules in multiplexed mode, leave the cold-junction sensor jumper on the terminal block in the `mtemp` (factory default) position. If you are using parallel mode, you can use the `dtemp` jumper setting.

Note The SCXI-1101, SCXI-1102, SCXI-1112, and SCXI-1127 use the `cjtemp` string only in multiplexed mode. The SCXI-1125 also uses the `cjtemp` string when the temperature sensor is configured in `mtemp` mode.

To read the temperature sensor, use the standard SCXI string syntax in the **channels** array with `mtemp` substituted for the channel number, as shown in the following table.

Channel List Element	Channel Specified
`ob0!scx!mdy!mtemp`	The temperature sensor configured in `mtemp` mode on the multiplexed module in slot `y` of the chassis with ID `x`.
`ob0!scx!mdy!cjtemp`	The temperature sensor configured in `cjtemp` mode on the multiplexed SCXI-1102 module in slot `y` of the chassis with ID `x`, or the temperature sensor configured in `mtemp` mode on the multiplexed SCXI-1125 module in slot `y` of the chassis with ID `x`.
`ob0!scx!mdy!cjtempz`	The temperature sensor configured in `cjtemp` mode for the analog channel `z` on the multiplexed SCXI-1112 module in slot `y` of the chassis with ID `x`.

If you want to read the cold-junction temperature sensor in `dtemp` mode, you can read the following onboard channels for these modules.

Modules	Channel
SCXI-1100	1
SCXI-1120/SCXI-1120D	15 (use referenced single-ended mode)
SCXI-1121	4
SCXI-1122	1

For example, you can run the Getting Started Analog Input VI, available in the `examples\daq\run_me.llb`, with the channel string `ob0!sc1!md1!mtemp` to read the temperature sensor on the terminal block connected to the module in slot 1 of SCXI chassis 1.

SCXI terminal blocks have two different kinds of sensors: an Integrated Circuit (IC) sensor or a thermistor. For terminal blocks that have IC sensors, such as the SCXI-1300 and the SCXI-1320, you can multiply the voltage read from the IC sensor by 100 to get the ambient temperature in degrees Centigrade at the terminal block. For terminal blocks that have thermistors, such as the SCXI-1303, SCXI-1322, SCXI-1327, and SCXI-1328, use the Convert Thermistor VI, available on the **Functions»Data Acquisition»Signal Conditioning** palette, to convert the raw voltage data into units of temperature.

You cannot sample other SCXI channels from the same module while you are sampling the `mtemp` sensor. However, if you are in parallel mode, you can sample the `dtemp` sensor along with other channels on the same module at the same time because you are not performing any multiplexing on the SCXI module. You also can sample the `cjtemp` sensor along with other channels on the SCXI-1101, SCXI-1102, SCXI-1112, SCXI-1125, and SCXI-1127. For the SCXI-1102, `cjtemp` must be the first channel in the channel list. The SCXI-1112 and SCXI-1125 can sample `cjtemp` channels in any order in the channel list.

For greater accuracy, take several readings from the temperature sensor and average those readings to yield one value. If you do not want to average several readings, take a single reading using the Easy Analog Input VI, AI Sample Channel.

Refer to the SCXI Thermocouple VIs in the appropriate library in `examples\daq\scxi` for examples of using the `cjtemp` or `mtemp` string to read the temperature sensor and using the reading for thermocouple cold-junction compensation.

Amplifier Offset

The SCXI-1100, SCXI-1101, SCXI-1112, SCXI-1122, SCXI-1125, SCXI-1141, SCXI-1142, and SCXI-1143 have a special calibration feature that enables LabVIEW to ground the module amplifier inputs so that you can read the amplifier offset. For the other SCXI analog input modules, you must physically wire your terminals to ground. The measured amplifier offset is for the entire signal path including the SCXI module and the DAQ device.

To read the grounded amplifier on the SCXI-1100, SCXI-1101, or SCXI-1122, use the standard SCXI string syntax in the **channels** array with `calgnd` substituted for the channel number, as shown in the following table.

Channel List Element	Channel Specified
`ob0!scx!mdy!calgnd`	(SCXI-1100 and SCXI-1122 only) The grounded amplifier of the module in slot `y` of the chassis with ID `x`.
`ob0!scx!mdy!calgndz`	Where `z` is the appropriate SCXI channel needing shunting for the SCXI-1112, SCXI-1125, SCXI-1141, SCXI-1142, or SCXI-1143.

For example, you can run the Getting Started Analog Input VI, available in the `examples\daq\run_me.llb`, with the channel string `ob0!sc1!md1!calgnd` to read the grounded amplifier of the module in slot 1 of SCXI chassis 1. The voltage reading should be very close to 0 V. The AI Start VI grounds the amplifier before starting the acquisition, and the AI Clear VI removes the grounds from the amplifier after the acquisition completes.

The SCXI-1112, SCXI-1125, SCXI-1141, SCXI-1142, and SCXI-1143 have separate amplifiers for each channel, so you must specify the channel number when you ground the amplifier. To specify the channel number, attach the channel number to the end of the string `calgnd`. For example, `calgnd2` grounds the amplifier inputs for channel 2 and reads the offset.

You also can specify a range of channels. The string `calgnd0:7` grounds the amplifier inputs for channels 0 through 7 and reads the offset for each amplifier.

Use the Scaling Constant Tuner VI, available on the **Functions»Data Acquisition»Signal Conditioning** palette, to modify the scaling constants so LabVIEW automatically compensates for the amplifier offset when scaling binary data to voltage. Refer to the SCXI-1100 Voltage VI in the `examples\daq\scxi\scxi1100.llb` for an example of how to use the Scaling Constant Tuner VI.

VI Examples

If you use the DAQ Channel Wizard to configure your channels, you can simplify the programming needed to measure your signal. LabVIEW configures the hardware with the appropriate input limits and gain, and performs CJC, amplifier offsets, and scaling for you. You can use the Easy VIs or the Continuous Transducer VI, available in the `examples\daq\solution\transduc.llb`, to measure a channel using a channel name. Enter the name of your configured channel in the **channels** input. The device input value is not used by LabVIEW when you use channel names. The acquired data is in the physical units you specified in the DAQ Channel Wizard.

The remainder of this section describes how to measure temperature with the SCXI-1100 and SCXI-112*X* modules using thermocouples when you do not use the DAQ Channel Wizard. The temperature examples below use both cold-junction measurements and amplifier offsets. In SCXI analog input examples, you cannot set the scaling constants with the Easy VIs (determined by the amplifier offset). With the Intermediate VIs, you can change the scaling constants before acquisition begins, and the Advanced VIs include functions that are not necessary to accurately measure temperature with SCXI modules. The examples described in this section use Intermediate VIs along with transducer-specific VIs.

First, you should learn how to measure temperature using the SCXI-1100 with thermocouples. You can use the example SCXI-1100 Thermocouple VI, available in the `examples\daq\scxi\scxi1100.llb`. Open the VI and continue reading this section.

To reduce the noise on the slowly varying signals produced by thermocouples, you can average the data and then linearize it. For greater accuracy, you can measure the amplifier offset, which helps scale the data and lets you eliminate the offset error from your measurement. Figure 9-6 shows how you can program the Acquire and Average VI, available in the

`vi.lib\daq\zdaqutil.llb`, to measure the amplifier offset. This VI acquires 100 measurements from the amplifier offset, designated in the **offset channel** input by `calgnd`, and then averages the measurements. When you determine the amplifier offset, you must always use the same input limits and clock rates that you will be using in the acquisition. The Acquire and Average VI can measure the amplifier offset of many modules at once, but in Figure 9-6, it measures only one module.

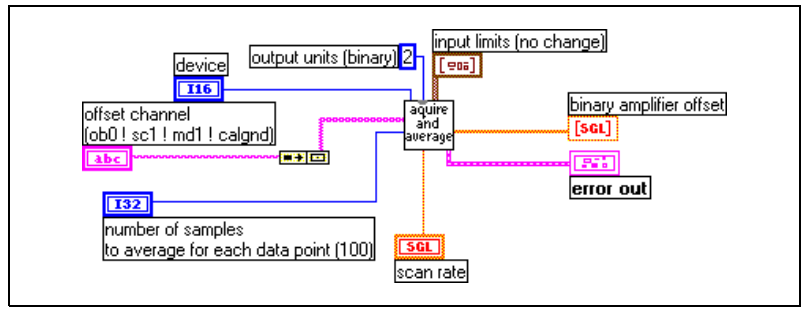


Figure 9-6. Measuring a Single Module with the Acquire and Average VI

After measuring the amplifier offset, measure the temperature sensor for CJC. Both the amplifier offset and cold-junction measurements should be taken before any thermocouple measurements are taken. Use the Acquire and Average VI to measure temperature sensors, as shown in Figure 9-7. The main differences between the amplifier offset measurement and temperature sensor measurement are the channel string and the input limits. If you set the temperature sensor in `mtemp` mode (the most common mode), you access the temperature by using `mtemp`. If you set the temperature sensor in `dtemp` mode, you read the corresponding DAQ device onboard channel. Make sure you use the temperature sensor input limits, which are different from your acquisition input limits. To read from a temperature sensor based on an IC sensor or a thermistor, set the input limit range from +2 to –2 V.

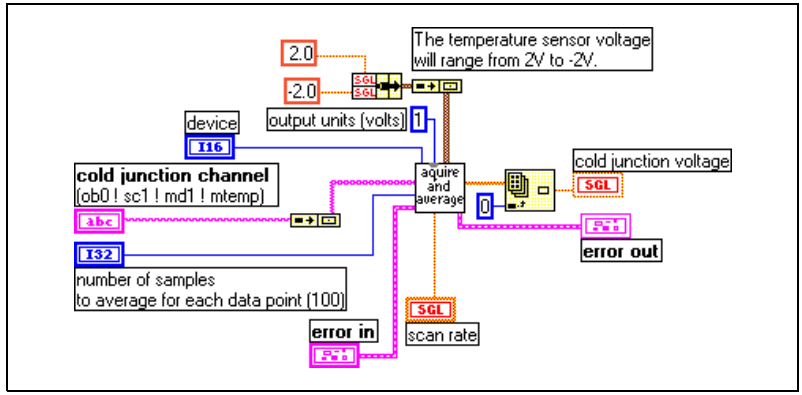


Figure 9-7. Measuring Temperature Sensors Using the Acquire and Average VI

After determining the average amplifier offset and cold-junction compensation, you can acquire data using the Intermediate VIs as shown in Figure 9-8. This example continually acquires data until an error occurs or the user stops the execution of the VI. For continuous, hardware-timed acquisition, you need to set up a buffer. In this case, the buffer is 10 times the number of points acquired for each channel. Before you initiate the acquisition with the AI Start VI, you need to set up the binary-to-voltage scaling constants by using the Scaling Constant Tuner VI. This VI, available on the **Functions»Data Acquisition»Signal Conditioning** palette, passes the amplifier offset to the DAQ driver so that LabVIEW accounts for the amplifier offset as the AI Read VI retrieves the data. After the compensated voltage data from the AI Read VI is averaged, the voltage values are converted to temperature and linearized by using the Convert Thermocouple Reading VI, available on the **Functions»Data Acquisition»Signal Conditioning** palette. After completing the acquisition, remember to always clear the acquisition by using the AI Clear VI.

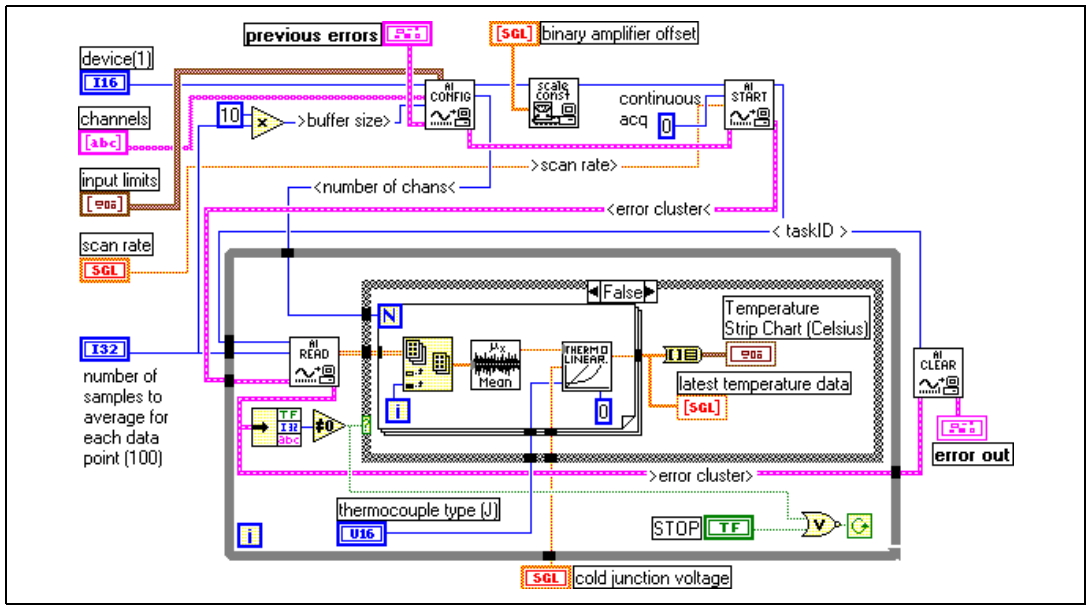


Figure 9-8. Continuously Acquiring Data Using Intermediate VIs

Refer to the SCXI Temperature Monitor VI in the `examples\daq\scxi\scxi1100.llb` for another temperature acquisition example using the SCXI-1100 module. This VI continually acquires thermocouple readings and sets an alarm if the temperature readings go above a user-defined limit.

You can use the SCXI-1100 examples with the SCXI-1122 module. Both modules have the capability to programmatically measure the amplifier offsets, and both modules need the CJC to linearize thermocouple measurements. The main differences between the two modules include the type of temperature sensors available on their terminal blocks and the way module channels are multiplexed.

The SCXI-1100 uses a CMOS multiplexer, which is capable of fast-channel multiplexing, whereas the SCXI-1122 uses an electromechanical relay to switch one of its 16 channels. Because the SCXI-1122 uses a relay, this module imposes a minimum interchannel delay of 10 ms, which affects the maximum possible scan rate. Scanning multiple SCXI-1122 channels many times can quickly wear out the relay. To avoid this, acquire data from the SCXI-1122 module one channel at a time. Refer to the *SCXI-1122 User Manual* or the SCXI-1122 Voltage VI in the `examples\daq\scxi\scxi1122.llb` for more information about reading SCXI-1122 channels.

Refer to the examples in the appropriate library for the module in `examples\daq\scxi` if you are measuring temperature with the SCXI-1120 and SCXI-1121 modules. This VI is similar to the VI used to measure temperature on the SCXI-1100. Both VIs average and linearize temperature data using the Intermediate analog input VIs. The main differences between the VIs are that the SCXI-1120/1121 VI does not measure the amplifier offset, and the input limits for the module and the temperature sensor are different from the input limits for the SCXI-1100.

The SCXI-1120 and SCXI-1121 modules do not have the internal switch used to programmatically ground the amplifiers as in the SCXI-1100 for the amplifier offset measurement. To determine the amplifier offset, you must manually wire the amplifier terminals to ground and use a separate VI to read the offset voltage. You also can manually calibrate the SCXI-1120 and SCXI-1121 to remove any amplifier offset on a channel-by-channel basis. Refer to the SCXI-1120 or SCXI-1121 user manuals for specific instructions.

Refer to the examples in the appropriate library for the module in `examples\daq\scxi` if you are measuring thermocouples with the SCXI-1125 or the SCXI-1112. These examples demonstrate how to scan the CJC channel (`cjtemp`) while scanning the thermocouple channels. By scanning the cold-junction sensor with the thermocouple channels, these examples are better suited to take temperature measurements over longer periods of time by accounting for temperature changes at the thermocouple junction inside the terminal block. The SCXI-1125 thermocouple example also demonstrates the ability to shunt the inputs and take an offset reading before collecting temperature data. This allows you to compensate for any offset drift due to operation at elevated temperatures or for offset produced by the system along the signal path.

Measuring Temperature with RTDs

Resistance-temperature detectors (RTDs) are temperature-sensing devices whose resistance increases with temperature. They are known for their accuracy over a wide temperature range. RTDs require current excitation to produce a measurable voltage. RTDs are available in 2-wire, 3-wire, or 4-wire configuration. The lead wires in the 4-wire configuration are resistance-matched. If you use a 2-wire or 3-wire RTD, they are unmatched. Resistance in the lead wires that connect your RTD to the measuring system will add error to your readings. If you are using lead lengths greater than 10 feet, you need to compensate for this lead resistance. RTDs also are classified by the type of metal they use. The most common metal is platinum.

Refer to Application Note 046, *Measuring Temperature with RTDs—A Tutorial* for more information about how the lead wires affect RTD measurements as well as general RTD information. You can find this note on the National Instruments Developer Zone, `zone.ni.com`

Signal conditioning is needed to interface an RTD to a DAQ device or an SCXI-1200 module. Signal conditioning required for RTDs include current excitation for the RTD, amplification of the measured signal, filtering of the signal to remove unwanted noise, and isolation of the RTD and monitored system from the host computer. Typically, you would use the SCXI-1121 module with RTDs because it easily performs all the signal conditioning listed previously. You must set up the excitation level, gain, and filter settings on the SCXI-1121 module with jumpers as well as in the configuration utility of your system.

The SC-2042 RTD is a signal conditioning device designed specifically for RTD measurement, and you can use it as an alternative to SCXI modules. Refer to the National Instruments catalog for more information.

You do not have to worry about CJC with RTDs as you do when measuring thermocouples. To build an application in LabVIEW, you can use the Easy I/O analog input VIs. If you are measuring multiple transducers on several different channels, you need to scan the necessary channels with little overhead. Because the Easy I/O VIs reconfigure your SCXI module every time your application performs an acquisition, it is recommended that you use the Intermediate analog input VIs instead.

Using the DAQ Channel Wizard to configure your channels can simplify the programming needed to measure your signal, as shown in Figure 9-9. LabVIEW configures the hardware with appropriate input limits and gain, measures the RTD, and scales the measurement for you. Enter the name of your configured channel in the **channels** input parameter. The acquired data is in the physical units you specify in the DAQ Channel Wizard.

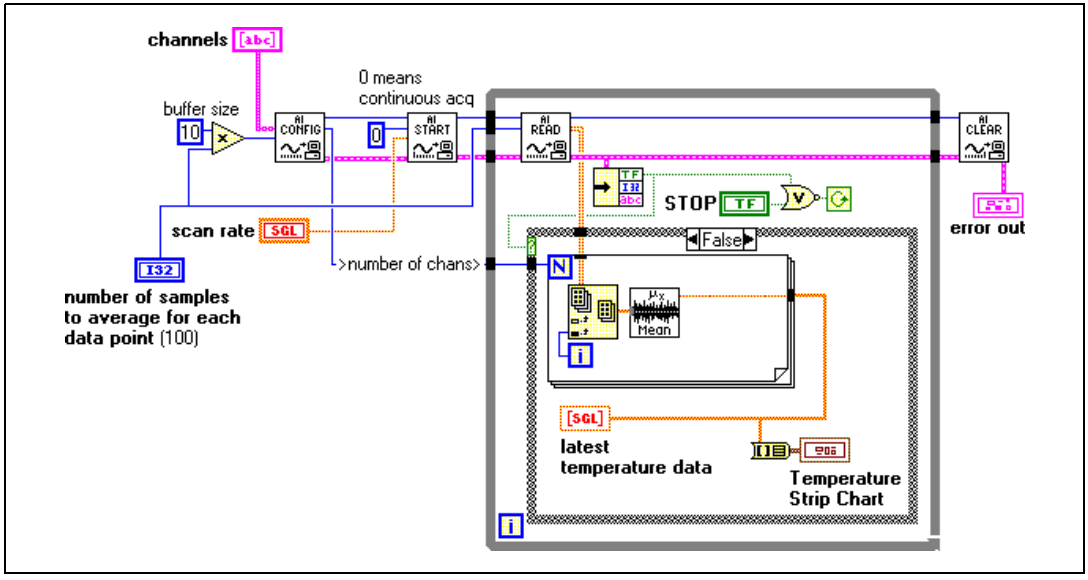


Figure 9-9. Measuring Temperature Using Information from the DAQ Channel Wizard

The VI in Figure 9-9 continually acquires data until an error occurs or you stop the VI from running. To perform a continuous hardware-timed acquisition, you must set up a buffer. In this example, the buffer is 10 times the number of points acquired for each channel. For each acquisition, your device averages the temperature data. After completing the acquisition, always clear the acquisition by using the AI Clear VI.

If you are not using the DAQ Channel Wizard, you must use the RTD Conversion VI in addition to specifying additional input parameters, as shown in Figure 9-10. The Convert RTD Reading VI, available on the **Functions»Data Acquisition»Signal Conditioning** palette, converts the voltage read from the RTD to a temperature representation.

Note Use the RTD conversion function in LabVIEW only for platinum RTDs. If you do not have a platinum RTD, the voltage-temperature relation is different, so you cannot use the LabVIEW conversion function.

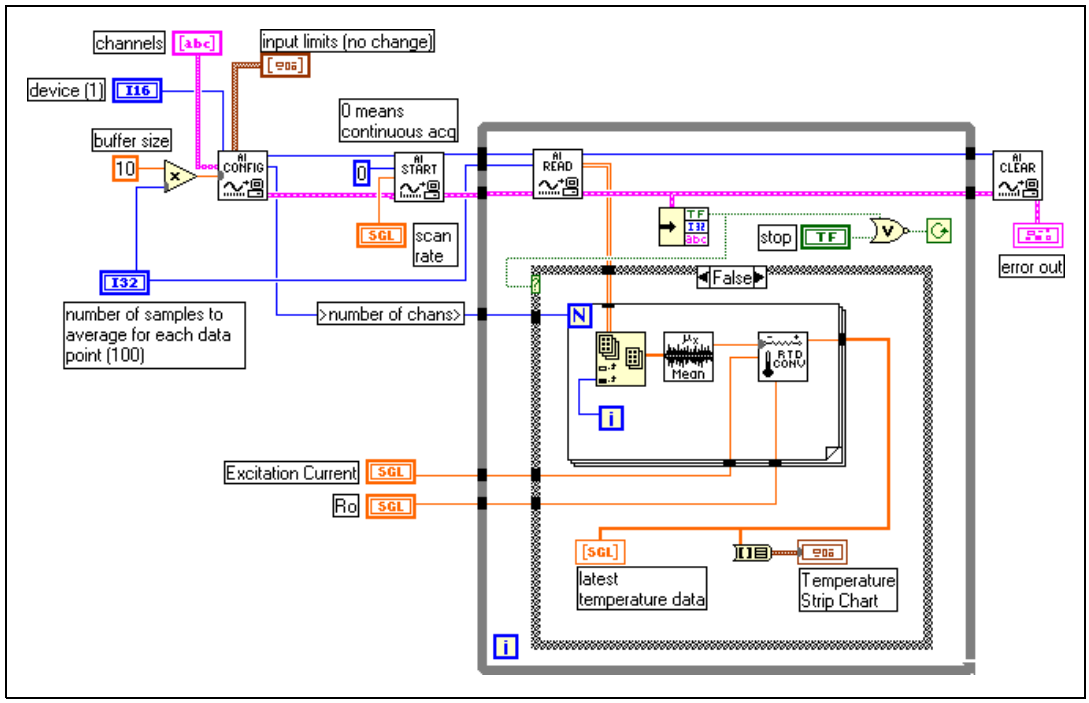


Figure 9-10. Measuring Temperature Using the Convert RTD Reading VI

The VI in Figure 9-10 continually acquires data until an error occurs or you stop the VI from executing. For continuous hardware-timed acquisition, you need to set up a buffer. In this example, the buffer is 10 times the number of points acquired for each channel. After your device averages the voltage data from the AI Read VI, it converts the voltage values to temperature. After completing the acquisition, remember to always clear the acquisition by using the AI Clear VI.

Measuring Pressure with Strain Gauges

Strain gauges give varying voltages in response to stress or vibrations in materials. Strain gauges are thin conductors attached to the material to be stressed. Resistance changes in parts of the strain gauge to indicate deformation of the material. Strain gauges require excitation (generally voltage excitation) and linearization of their voltage measurements. Depending on the strain-gauge configuration, another requirement for using strain gauges with SCXI is a configuration of resistors. As shown in Figure 9-11, the resistance from the strain gauges combined with the SCXI hardware form a diamond-shaped configuration of resistors, known as a *Wheatstone bridge*. When you apply a voltage to the bridge, the

differential voltage (V_m) varies as the resistor values in the bridge change. The strain gauge usually supplies the resistors that change value with strain.

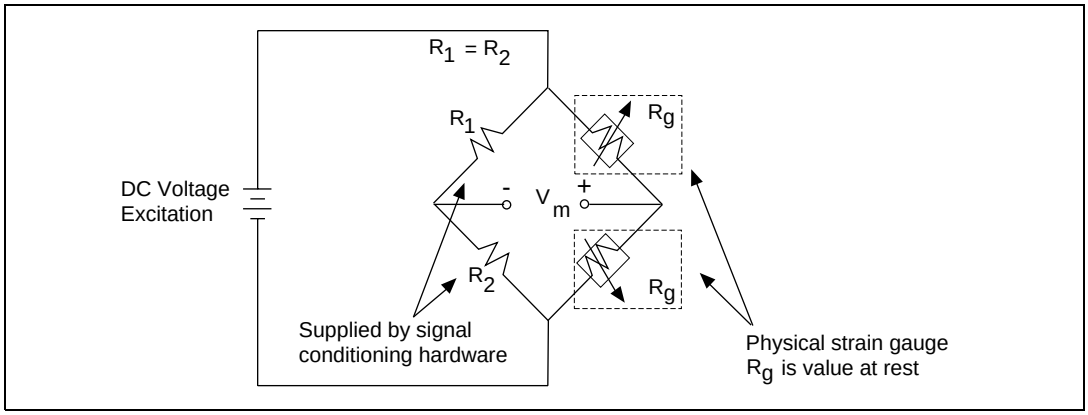


Figure 9-11. Half-Bridge Strain Gauge

Strain gauges come in full-bridge, half-bridge, and quarter-bridge configurations. For a full-bridge strain gauge, the four resistors of the Wheatstone bridge are physically located on the strain gauge itself. For a half-bridge strain gauge, the strain gauge supplies two resistors for the Wheatstone bridge while the SCXI module supplies the other two resistors, as shown above. For a quarter-bridge strain gauge, the strain gauge supplies only one of the four resistors for a Wheatstone bridge.

The SCXI-1520 is a dedicated strain measuring module, with software configurable bridge-completion, excitation, and resistance shunt switches, as well as filter and gain, on each of the 8 channels.

The SCXI-1121 and the SCXI-1122 modules are also commonly used with strain gauges because they include voltage or current excitation and internal Wheatstone bridge completion circuits. You also can use the signal conditioning device SC-2043SG as an alternative to SCXI modules. The device is designed specifically for strain-gauge measurements. For more information on this device, refer to your National Instruments catalog.

You can set up your SCXI module to amplify strain-gauge signals or filter noise from signals. Refer to your *Getting Started with SCXI* manual for the necessary hardware configuration and For information about setting up the excitation level, gain, and filter settings.

To build a strain-gauge application in LabVIEW, you can use the Easy I/O analog input VIs. If you are measuring multiple transducers on several

different channels, you need to scan the necessary channels as quickly as possible. Because the Easy I/O VIs reconfigure your SCXI module every time the VI is called, you should use the Intermediate analog input VIs as well as the Strain Gauge Conversion VI, as shown in Figure 9-12.
The Convert Strain Gauge Reading VI, available on the **Functions»Data Acquisition»Signal Conditioning** palette, converts the voltage read by the strain gauge to units of strain.

Using the DAQ Channel Wizard to configure your channels simplifies the programming required to measure your signal, as shown in Figure 9-12. LabVIEW configures the hardware with the appropriate input limits and gain, measures the strain, and scales the measurement for you. Enter the name of your configured channel in the **channels** input. You do not need to wire the **device** or **input limits** input. The acquired data is in the physical units you specified in the DAQ Channel Wizard.

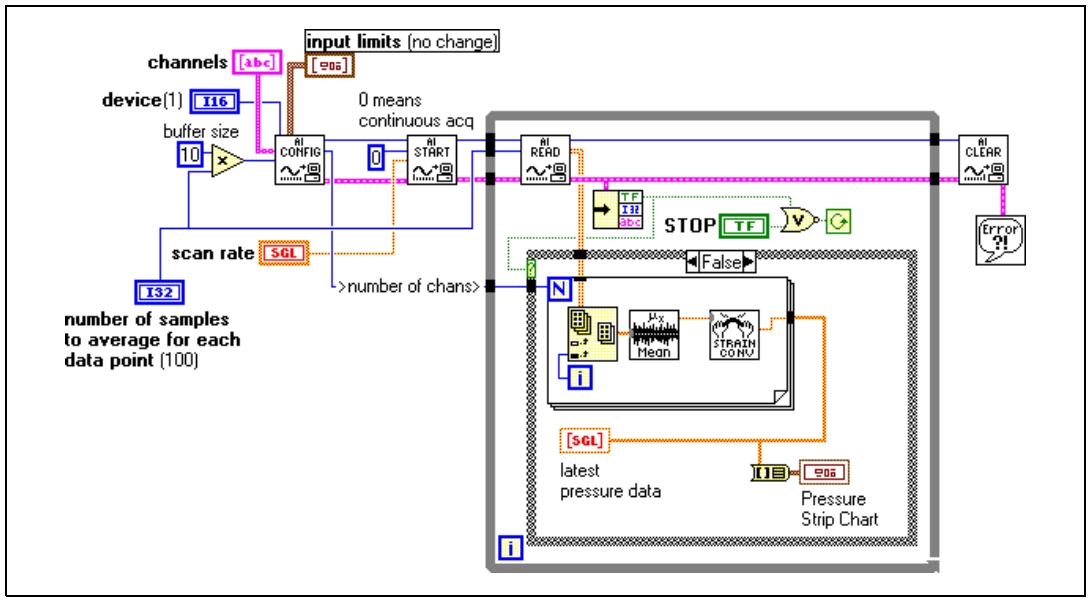


Figure 9-12. Measuring Pressure Using Information from the DAQ Channel Wizard

The VI in Figure 9-12 continually acquires data until an error occurs or you stop the VI from running. In order to perform continuous acquisition, you need to set up a buffer. In this example, the buffer is 10 times the number of points acquired for each channel. After your device averages the voltage data from the AI Read VI, it converts the voltage values to strain values. After completing the acquisition, always remember to clear the acquisition by using the AI Clear VI.

When measuring strain-gauge data, there are some parameters on the Convert Strain Gauge Reading VI you should know.

Vsg, the strain-gauge value, is the only parameter wired in Figure 9-12. The other parameters for this VI have default values, but those values may not be correct for your strain gauge. You should check the following parameters:

- **Rg**—The resistance of the strain gauge before strain is applied. You usually can ignore the lead resistance
- **Bridge Configuration**
- **Vex**—The excitation voltage
- **Vinit**—The voltage across the strain gauge before strain is applied (always measure at the beginning of the VI)
- **Rl**—The lead resistance
- **Rl**—For strain gauges unless the leads are several feet

Analog Output Application Example

You can output isolated analog signals using the SCXI-1124 analog output module. The remainder of this section describes how to generate signals with the SCXI-1124 when you do not use the DAQ Channel Wizard.

The SCXI-1124 can generate voltage and current signals. Refer to the , SCXI-1124 Update Channels VI in the `examples\daq\scxi\scxi1124.llb` for an example analog output VI. This VI uses the analog output Advanced VIs because the output mode (whether you have voltage or current data) must be accessible in order to change the value. The program calls the AO Group Config VI to specify the device and output channels. The AO Hardware Config VI specifies the output mode and the output range, or limit settings, for all the channels specified in the channels string. This advanced-level VI is the only place where you can specify a voltage or current output mode. If you are going to output voltages only, consider using the AO Config VI (an Intermediate VI), instead of the AO Group Config and AO Hardware Config VIs. You can program individual output channels of the SCXI-1124 for different output ranges by using the arrays for channels, output mode, and limit settings. The AO Single Update VI initiates the update of the SCXI-1124 output channels. To help debug your VIs, it is always helpful to display any errors, in this case using the Simple Error Handler VI.

Digital Input Application Example

To input digital signals through an SCXI chassis, you can use the SCXI-1162 and SCXI-1162HV modules and the Easy Digital VI, Read from Digital Port, as shown in Figure 9-13.

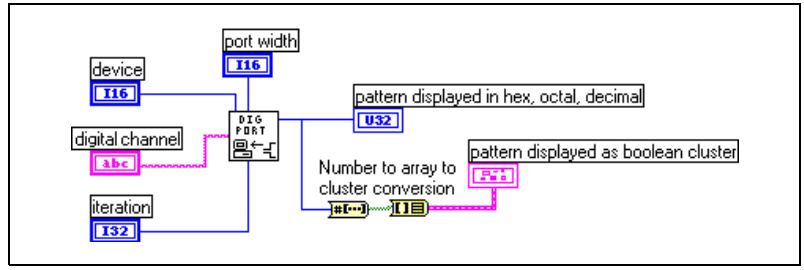


Figure 9-13. Inputting Digital Signals through an SCXI Chassis Using Easy Digital VIs

If you configure channels using the DAQ Channel Wizard, **digital channel** can consist of a digital channel name. The channel name can refer to either a port or a line in a port. You do not need to specify **device**, **line**, or **port width**, as these inputs are not used by LabVIEW if a channel name is specified in **digital channel**.

As an alternative, **digital channel** can be expressed in the `scx!mdy!0` format, where you are trying to input from the digital input module on slot `y` of chassis `x`. The last identifier is always port 0, because the whole module is considered one port. In this example, you also must specify **device** and **port width**. The **port width** should be the number of lines in a port on your SCXI module if you are operating in multiplexed mode. For the SCXI-1162 and SCXI-1162HV, the **port width** is 32 lines. If you are operating in parallel mode, the **port width** should be the number of lines on your DAQ device. The DIO-32F/DIO-32HS/6533 device can access all 32 lines of the SCXI modules at once by using the SCXI-1348 cable assembly. The DIO-24 and the DIO-96 devices can access only the first 24 lines of these modules when configured in parallel mode. For the fastest performance in parallel mode, you can use the appropriate onboard port numbers instead of the SCXI channel string syntax.

Use the iteration input to optimize your digital operation. When **iteration** is 0 (default), LabVIEW calls the DIO Port Config VI (an Advanced VI) to configure the port. If **iteration** is greater than zero, LabVIEW bypasses reconfiguration and remembers the last configuration, which improves performance. You can wire this input to an iteration terminal of a loop. With the DIO-24 and DIO-96 devices, every time you call the DIO Port Config VI, the digital line values are reset to default values. If you want

to maintain the integrity of the digital values from one loop iteration to another, do not set **iteration** to 0 except for the first iteration of the loop.

Refer to the SCXI-1162HV Digital Input VI in the `examples\daq\scxi\scxi1162.llb` for an example of SCXI digital input. Even though this VI uses Advanced VIs, it is functionally equivalent to the Easy I/O Digital VI, Read from Digital Port.

Note The DIO Port Config VI resets output lines on adjacent ports on the same 8255 chip for DIO-24, DIO-96, and Lab and 1200 Series devices.

Note If you also are using SCXI analog input modules, make sure your cabling DAQ device is cabled to one of them.

Digital Output Application Example

To output digital signals through an SCXI chassis, you can use the SCXI-1160, SCXI-1161, SCXI-1163, and SCXI-1163R modules and the digital Easy Digital VI, Write to Digital Port, as shown in Figure 9-14.

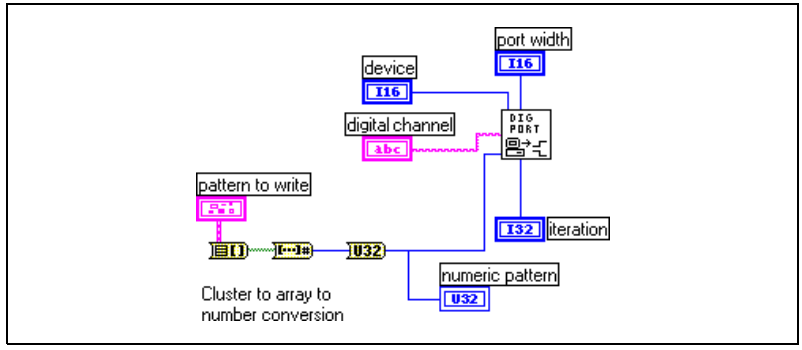


Figure 9-14. Outputting Digital Signals through an SCXI Chassis Using Easy Digital VIs

If you configure channels using the DAQ Channel Wizard, **digital channel** can consist of a digital channel name. The channel name can refer to either a port or a line in a port. You do not need to specify **device**, **line**, or **port width**, as these inputs are not used by LabVIEW if a channel name is specified in **digital channel**.

As an alternative, **digital channel** can be expressed in the `scx!mdy!0` format, where you are trying to output from the digital output module on slot `y` of chassis `x`. The last identifier is always port 0, because the whole module is considered one port. In this case, you also must specify **device**

and **port width**. The **port width** should be the number of lines on your SCXI module if you are operating in multiplexed mode. The SCXI-1160 has 16 relays, the SCXI-1161 has eight relays, and the SCXI-1163/1163R have 32 relays. You can not use the SCXI-1160 or SCXI-1161 in parallel mode. For the SCXI-1163/1163R the **port width** in parallel mode should be the number of lines on your DAQ device or SCXI-1200 module. The 6533 device can access all 32 lines of the SCXI-1163/1163R modules at once by using the SCXI-1348 cable assembly. The DIO-24 and the DIO-96 devices can access only the first 24 lines of the SCXI-1163/1163R when configured in parallel mode. For the fastest performance in parallel mode, you can use the appropriate onboard port numbers instead of the SCXI channel string syntax.

Use the **iteration** input to optimize your digital operation. When **iteration** is 0 (default), LabVIEW calls the DIO Port Config VI (an Advanced VI) to configure the port. If **iteration** is greater than zero, LabVIEW bypasses reconfiguration and remembers the last configuration, which improves performance. You can wire this input to an iteration terminal of a loop. Every time you call the DIO Port Config VI the digital line values are reset to default values. If you want to maintain the integrity of the digital values from one loop iteration to another, do not set iteration to 0 except for the first iteration of the loop.

Refer to the SCXI-116x Digital Output VI in the `examples\daq\scxi\scxi_dig.llb` for an example of SCXI digital output. Even though this VI uses Advanced VIs, it is functionally equivalent to the Easy Digital VI, Write to Digital Port.

Note If you also are using SCXI analog input modules, make sure your cabling DAQ device is cabled to one of them.

Multi-Chassis Applications

You can daisy-chain multiple SCXI-1000, SCXI-1000DC, or SCXI-1001 chassis using the SCXI-1350 or SCXI-1346 multichassis cable adapters and an MIO Series DAQ device other than the DAQPad-MIO-16XE-50. Every module in each of the chassis must be in multiplexed mode. Only one of the chassis will be connected directly to the DAQ device. Also, if you are using Remote SCXI with RS-485, you can daisy-chain up to 31 chassis on a single RS-485 port. Because you can configure only up to 16 devices on the NI-DAQ Configuration Utility, you can have only up to 16 SCXI-1200s in your system.

Note Lab Series devices, LPM devices, DAQCard-500, 516 devices, DAQCard-700, 1200 Series (other than SCXI-1200), and DIO-24 devices do *not* support multichassis applications.

If you use the DAQ Channel Wizard to configure your analog input channels, you simply address channels in multiple chassis by their channel names. You can combine channel names, separated by commas, to measure data from multiple modules in a daisy-chain configuration at the same time. For example, if you have a named channel called `temperature` on one module in the daisy chain and `pressure` on another module in the same daisy chain, your **channels** array could be `temperature, pressure`. You must enter the chassis in a sequential order in the NI-DAQ Configuration Utility, assigning the first chassis in the chain an ID number of 1, the second chassis an ID number of 2, and so forth.

If you are not using the DAQ Channel Wizard, there are special considerations for addressing the channels. When you daisy-chain multiple chassis to a single DAQ device (non-Remote SCXI), each chassis multiplexes all of its analog input channels into a separate onboard analog input channel. The first chassis in the chain uses onboard channel 0, the second chassis in the chain uses onboard channel 1, and so on. To access channels in the second chassis, you must select the correct onboard channel as well as the correct chassis ID. The string `ob1!sc2!md1!0` means channel 0 on the module in slot 1 of SCXI chassis 2, multiplexed into onboard channel 1. Remember to use the correct chassis ID number from the configuration utility and to put the jumpers from the power supply module in the correct position for each chassis.

When an MIO/AI Series device is cabled by a ribbon cable or shielded cable to multiple chassis, the number of reserved analog input channels depends on the number of chassis. On MIO Series devices, lines 0, 1, and 2 are unavailable. On MIO E Series devices, lines 0, 1, 2, and 4 are unavailable.

When you access digital SCXI modules, you do not use onboard channels. Therefore, if you have multiple chassis, you only have to choose the correct SCXI chassis ID and module slot.

When you use Remote SCXI to address analog input channels, specify the device number of the SCXI-1200 that is located in the same chassis containing the analog input module from which you take samples.

You can perform DAQ operations on channels in multiple SCXI chassis at the same time. For example, the first element of your **channels** array could be `ob0!sc1!md1!0:31`, and the second element of the **channels** array could be `ob1!sc2!md1!0:31`. Then, LabVIEW would scan 32 channels on module 1 of SCXI chassis 1, using onboard channel 0, then the 32 channels on module 1 in SCXI chassis 2, using onboard channel 1. Remember that the **scan rate** you specify is how many scans per second LabVIEW performs. For each scan, LabVIEW reads every channel in the **channels** array.

You can practice reading channels from different chassis by using the channel strings explained above in the Easy VIs.

SCXI Calibration—Increasing Signal Measurement Precision

Your SCXI module ships to you factory-calibrated for the specified accuracy. You need to recalibrate the module only if the precision of your signal measurement is not acceptable because of shifts in environmental conditions.

Note This chapter does not apply to the SCXI-1200. For calibration on the SCXI-1200, you should use the 1200 Calibrate VI, available on the **Functions»Data Acquisition»Calibration and Configuration** palette. If you are using an SCXI-1200 in a remote SCXI configuration, National Instruments recommends that you connect directly to your parallel port to perform calibration, because it works much faster.

EEPROM Calibration Constants

When you calibrate your SCXI module in LabVIEW, the calibration constants can be stored in *electronically erasable programmable read-only memory (EEPROM)*. The EEPROM stores calibration constant information in the memory of your module. There are three parts to the EEPROM: the factory area, the default load area, and the user area.

Note Only the SCXI-1102, SCXI-1102B, SCXI-1102C, SCXI-1104, SCXI-1104C, SCXI-1112, SCXI-1122, SCXI-1124, SCXI-1125, SCXI-1141, SCXI-1142, SCXI-1143, and SCXI-1520 have EEPROMs. All other SCXI modules do *not* store calibration constants.

Note The SCXI-1125 does *not* have a user area in its EEPROM.

- The *factory area* has a set of factory calibration constants already stored in it when you receive your SCXI module. You cannot write into the factory area, but you can read from it, so you can always access and use these factory constants if they are appropriate for your application.
- The *default load area* is where LabVIEW automatically looks to load calibration constants the first time you access the module. When the module is shipped, the default load area contains a copy of the factory calibration constants.

Note You can overwrite the constants stored in the default load area of EEPROM with a new set of constants using the SCXI Cal Constants VI.

- The *user area* is an area for you to store your own calibration constants that you calculate using the SCXI Cal Constants VI. You also can put a copy of your own constants in the default load area if you want LabVIEW to automatically load your constants for subsequent operations. You can read and write to the user area.

Note Use the user area in EEPROM to store any calibration constants that you may need to use later. This prevents you from accidentally overwriting your constants in the default load area, because you will have two copies of your new constants. You can revert to the factory constants by copying the factory area to the default load area without wiping out your new constants entirely.

Calibrating SCXI Modules

The SCXI Cal Constants VI in LabVIEW automatically calculates the calibration constants for your module with the precision you need for your particular application. You can find this VI on the **Functions»Data Acquisition»Calibration and Configuration** palette.

For the SCXI-1112, SCXI-1125, and SCXI-1520 modules, you can use the SCXI Calibrate VI for easy one-point calibration (SCXI-1125) or two-point calibration (SCXI-1112) without the separate function calls necessary with the SCXI Cal Constants VI. One- and two-point calibration are described in the *SCXI Calibration Methods for Signal Acquisition* section later in this chapter. You also can find the SCXI Calibrate VI on the **Functions»Data Acquisition»Calibration and Configuration** palette.

By default, calibration constants for the SCXI-1102, SCXI-1102B, SCXI-1102C, SCXI-1104, SCXI-1104C, SCXI-1112, SCXI-1122, SCXI-1125, SCXI-1141, SCXI-1142, SCXI-1143, and SCXI-1520 are loaded from the module EEPROM. The SCXI-1141, SCXI-1142, and SCXI-1143 have only gain adjust constants in the EEPROM. They do not have the binary zero offset. All other analog input modules do not have calibration constants by default and do not assume any binary offset and ideal gain settings. This means you must use one of the procedures described in the *SCXI Calibration Methods for Signal Acquisition* section to store calibration constants for your module if it is not an SCXI-1102, SCXI-1112, SCXI-1122, SCXI-1125, SCXI-1141, SCXI-1142, or SCXI-1143.

You can determine calibration constants based on your application setup, which includes your type of DAQ device, DAQ device settings, and cable assembly—combined with your SCXI module and its configuration settings.

Note If your SCXI module has independent gains on each channel, the calibration constants for each channel are stored at each gain setting.

SCXI Calibration Methods for Signal Acquisition

There are two ways you can calibrate your SCXI module—through *one-point calibration* or *two-point calibration*. Figure 9-15 illustrates why you may need to calibrate your SCXI module.

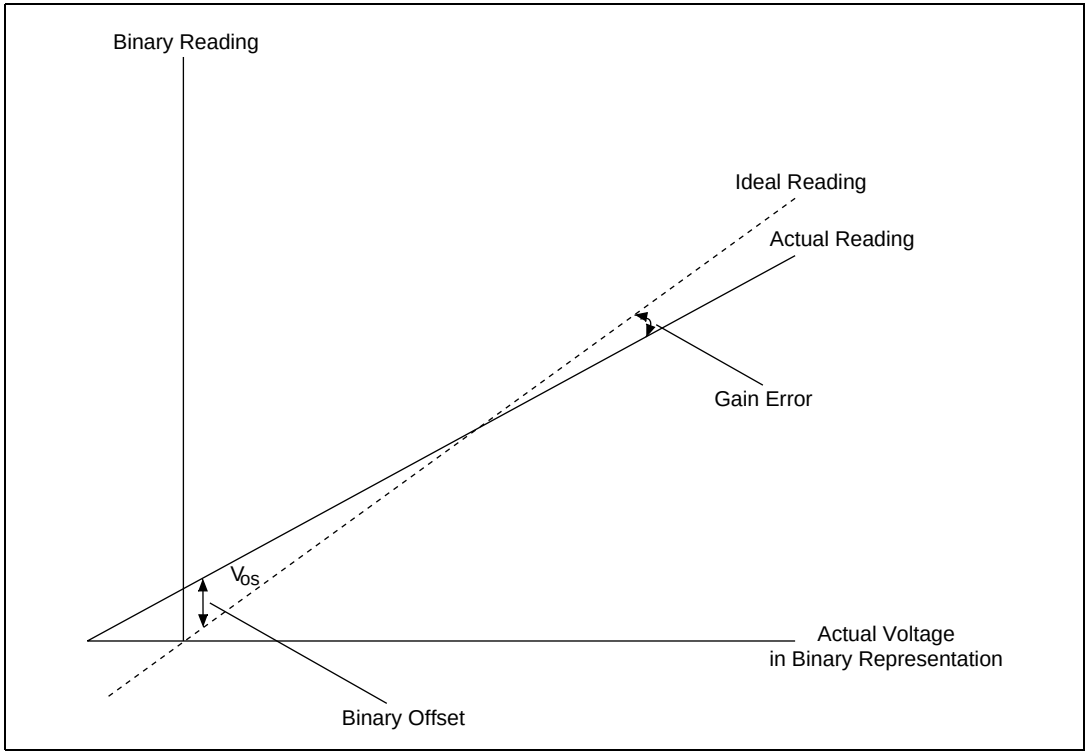


Figure 9-15. Ideal versus Actual Reading

Figure 9-15 shows the difference between the ideal reading and the actual reading. This difference is called V_{os}, or the **binary offset**, before the two readings intersect. The difference in slope between the actual and ideal readings is called the **gain error**.

One-point calibration removes the V_{os} (**binary offset**) by measuring a 0 V signal and comparing the actual reading to it. Two-point calibration removes the V_{os} (**binary offset**) *and* corrects gain error by first performing a one-point calibration. Then you measure a voltage at *x* volts and compare it to the actual reading. The *x* must be as close as possible to the full-scale range. The following sections explain how to perform a one-point and two-point calibration.

One-Point Calibration

Use one-point calibration when you need to adjust only the binary offset in your module. If you need to adjust both the binary offset and the gain error of your module, refer to the *Two-Point Calibration* section later in this chapter.

Note If you are using an E Series device, you should calibrate your DAQ device first using the E Series Calibrate VI.

Complete the following steps to perform a one-point calibration calculation in LabVIEW.

1. Make sure you set the SCXI gain to the gain you want to use in your application. If your modules have gain jumpers or DIP switches, set them appropriately. Refer to your SCXI module user manual for jumper or switch setting information. If your modules have software-programmable gain, use the **input limits** parameter in the AI Config VI to set gain.
2. Program the module for a single-channel operation by using the AI Config VI with the channel that you are calibrating as the **channels** parameter in the VI.
3. Ground your SCXI input channel to determine the binary zero offset. You should ground inputs because offset can vary at different voltage levels due to gain error. If you are using an SCXI-1100 or SCXI-1122, you can ground your input channels without external hookups by substituting the channel string with `calgnd` as the channel number. For other modules, you need to wire the positive and negative channel inputs together at the terminal block and wire them to the chassis ground.
4. Use the AI Single Scan VI to take several readings and average them for greater accuracy. Set the DAQ device gain settings to match the settings you plan to use in your application. By using the AI Start and AI Read VIs, instead of the AI Single Scan VI, you can average over an integral number of 60 or 50 Hz power line cycles (sine waves) to eliminate line noise. You now have your first volt/binary measurement: volt = 0.0 or the applied voltage at your input channel, and binary is your binary reading or binary average.
5. Use the SCXI Cal Constants VI with your volt/binary measurement from step 4 as the **Volt/Amp/Hz 1** and **Binary 1** inputs in your VI, respectively. (These input names may vary depending on your application setup.) For example, if your volt/binary measurement from step 4 was 0.00 V and 2, enter the values into your front panel controls.

Two-Point Calibration

The following steps show you how to perform a two-point calibration calculation in LabVIEW. Use two-point calibration when you need to correct both the binary offset and the gain error in your SCXI module.

Note If you are using an E Series device, you should calibrate your DAQ device first using the E Series Calibrate VI.

To perform a two-point calibration calculation in LabVIEW, follow steps 1 through 5 in the previous section, *One-Point Calibration*, then complete the following steps.

6. Now apply a known, stable, non-zero voltage to your input channel at the terminal block. This input voltage should be close to the upper limit of your input voltage range for the given gain setting. For example, if your input voltage range is –5 to 5 V, apply an input voltage that is as close to 5 V as possible, but does not exceed 5 V.
7. Take another binary reading or average of readings. If your binary reading is the maximum binary reading for your DAQ device, try a smaller input voltage. This is your second volt/binary measurement.
8. Use the SCXI Cal Constants VI with the first volt/binary measurement from step 4 as **Volt/Amp/Hz 1** and **Binary 1** inputs, and the second measurement from step 7 as **Volt/Amp/Hz 2** and **Binary 2** inputs of the VI. Your input names may vary depending on your application setup.
9. If you are using SCXI-1102 or SCXI-1122 inputs, you can save the constants in the module user area in EEPROM. Store constants in the user area as you are calibrating, and use the SCXI Cal Constants VI again at the end of your calibration sequence to copy the calibration table in the user area to the default load area in EEPROM. Remember, constants stored in the default load area can be overwritten. If you want to use a set of constants later, keep a copy of the constants stored in the user area in EEPROM.

Note If you are storing calibration constants in the SCXI-1102 or SCXI-1122 EEPROM, your binary offset and gain adjust factors must not exceed the ranges given in the respective module user manuals.

For other analog input modules, you must store the constants in the memory. Unfortunately, calibration constants stored in the memory are lost at the end of a program session. You can solve this problem by creating a file and saving the calibration constants to this file. You can load them

again in subsequent application runs by passing them into the SCXI Cal Constants or the Scale Constant Tuner VIs.

Calibrating SCXI Modules for Signal Generation

When you output a voltage or current value to your SCXI analog output module, LabVIEW 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. By default, calibration constants for the SCXI-1124 are loaded into the memory from the EEPROM default load area.

Recalibrate your SCXI analog output module by following these steps.

1. Use the AO Single Update VI to output a binary value. If you are calibrating a voltage output range, enter `0` in the **binary data** input of the VI. If you are calibrating current range, enter `255` into the **binary data** input of the VI.
2. Measure the output voltage or current at the output channel with a voltmeter or ammeter. This is your first volt/binary measurement: **Binary 1** = 0, and **Volt/Amp/Hz 1** is the voltage or current you measured at the output.
3. Use the AO Single Update VI to output a binary value of 4,095.
4. Measure the output voltage or current at the output channel. This is your second volt/binary measurement: **Binary 2** should be `4095` and **Volt/Amp 2** is the voltage or current you measured at the output.
5. Use SCXI Cal Constants VI with the first voltage/binary measurement from step 2 as the **Volt/Amp/Hz 1** and **Binary 1** inputs and the second measurement from step 4 as the **Volt/Amp/Hz 2** and **Binary 2** inputs of the VI.

You can save the constants on the module in the user area in EEPROM. Use the user area as you are calibrating, and use the SCXI Cal Constants VI again at the end of your calibration sequence to copy the calibration table in the user area to the default load area in EEPROM. Remember, you can overwrite constants stored in the default load area. If you want to use a set of constants later, keep a copy of the constants stored in the user area in EEPROM.

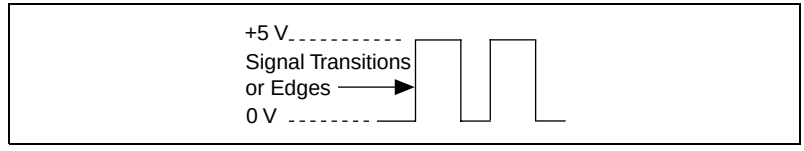
Repeat the procedure above for each channel and range you want to calibrate. Subsequent analog outputs will use your new constants to scale voltage or current to the correct binary value.

10

High-Precision Timing (Counters/Timers)

Things You Should Know about Counters

Counters add counting or high-precision timing to your DAQ system. Counters respond to and output TTL signals—square-pulse signals that are 0 V (low) or 5 V (high) in value. The following diagram shows a TTL signal.



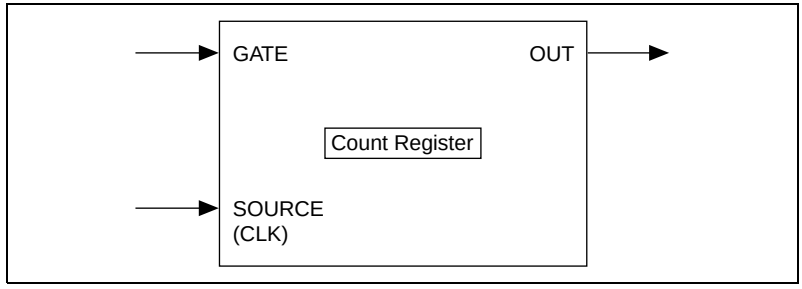
Although counters count only the signal transitions (edges) of a TTL source signal, you can use this counting capability in many ways:

- You can generate square TTL pulses for clock signals and triggers for other DAQ applications.
- You can measure the pulse width of TTL signals.
- You can measure the frequency and period of TTL signals.
- You can count TTL signal transitions or elapsed time.
- You can divide the frequency of TTL signals.
- You can measure position using quadrature encoders.

Some of the advanced counters also allow you to make any of the above measurements successively and return the measured values in a data buffer.

Knowing the Parts of Your Counter

The following illustration shows a basic model of a counter.



A counter consists of a SOURCE (or CLK) input pin, a GATE input pin, an OUT output pin, and a count register. In plug-in device diagrams, these counter parts are called SOURCE*n* (or CLK*n*), GATE*n*, and OUT*n*, where *n* is the number of the counter.

Edges are counted at the SOURCE input. The count register can be preloaded with a count value, and the counter increments or decrements the count register for each counted edge. The count register value always reflects the current count of signal edges. Reading the count register does not change its value. You can use the GATE input to control when counting occurs in your application. You also can use a counter with no gating, allowing the software to initiate the counting operation.

The OUT pin can be toggled according to available counter programming modes to generate various TTL pulses and pulse trains.

Use the OUT signal of a counter to generate various TTL pulse waveforms. If you are incrementing the count register value, you can configure the OUT signal to either toggle signal states or pulse when the count register reaches a certain value. The highest value of a counter is called the *terminal count* (TC). If you are decrementing, the count register TC value is 0. If you chose to have pulsed output, the counter outputs a high pulse that is equal in time to one cycle of the counter SOURCE signal, which can be either an internal or external signal. If you chose a toggled output, the state of the output signal changes from high to low or from low to high. For more control over the length of high and low outputs, use a toggled output.

Knowing Your Counter Chip

Most National Instruments DAQ devices contain one of four different counter chips: the TIO-ASIC, the DAQ-STC, the Am9513, or the 8253/54 chip. Typically, 660*x* devices use the TIO-ASIC chip. E Series devices (for example, the PCI-MIO-16E-1) use the DAQ-STC chip. Legacy-type MIO devices (for example the AT-MIO-16) use the Am9513 chip. Low-cost Lab/1200-type devices (for example the PCI-1200) use the 8253/54 chip. If you are not sure which chip your device uses, refer to your hardware documentation.

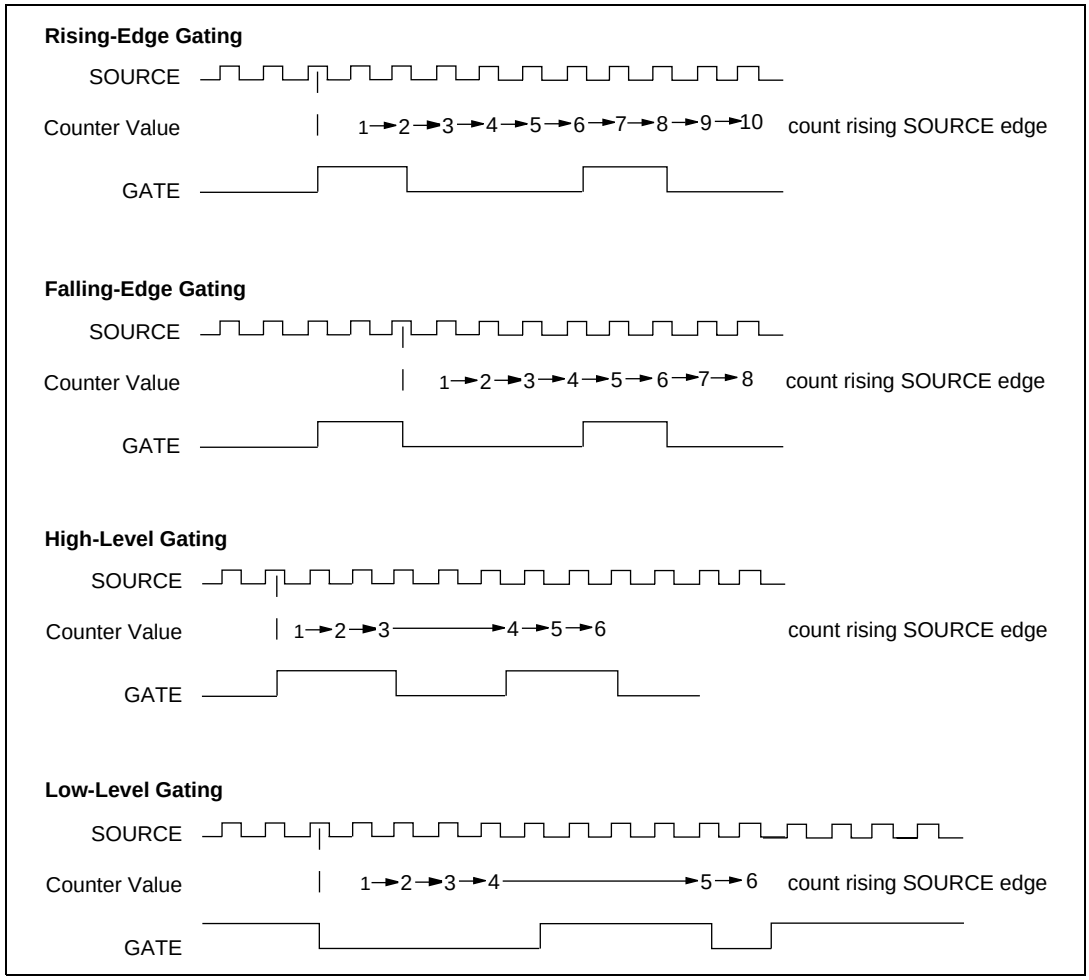


Figure 10-1. Counter Gating Modes

TIO-ASIC

You can configure the TIO-ASIC to count either low-to-high or high-to-low transitions of the SOURCE input. The counter has a 32-bit count register with a counting range of 0 to 2^{32}–1. It can be configured to increment or decrement for each counted edge. Furthermore, you can use an external digital line to control whether the count register increments or decrements, which is useful for encoder applications. Of the gating modes shown in Figure 10-1, the gating modes the TIO-ASIC supports depends upon the application. This counter chip supports buffered counter measurements. You can set the configuration parameters described above using the Advanced VI, Counter Set Attribute.

DAQ-STC

You can configure the DAQ-STC to count either low-to-high or high-to-low transitions of the SOURCE input. The counter has a 24-bit count register with a counting range of 0 to 2^{24}–1. It can be configured to increment or decrement for each counted edge. Furthermore, you can use an external line to control whether the count register increments or decrements, which is useful for encoder applications. Of the gating modes shown in Figure 10-1, the gating modes the DAQ-STC supports depends upon the application. You can set the configuration parameters discussed above using the Advanced VI, Counter Set Attribute.

Am9513

You can configure the Am9513 to count either low-to-high or high-to-low transitions of the SOURCE input. The counter has a 16-bit count register with a counting range of 0 to 65,535, and can be configured to increment or decrement for each counted edge. The Am9513 supports all of the gating modes shown in Figure 10-1. You can set the configuration parameters discussed above using the Advanced VI, CTR Mode Config.

8253/54

The 8253/54 chip counts low-to-high transitions of the CLK input. The counter has a 16-bit count register with a counting range of 65,535 to 0 that decrements for each counted edge. Of the gating modes shown in Figure 10-1, the 8253/54 supports only high-level gating. For single-pulse output, the 8253/54 can create only negative-polarity pulses. For this reason, some applications require the use of a 7404 inverter chip to produce a positive pulse. The 14-pin 7404 is a common chip available from many electronics store, and can be powered with the 5 V available on most DAQ devices. Figure 10-2 shows how to wire a 7404 chip to invert a signal.

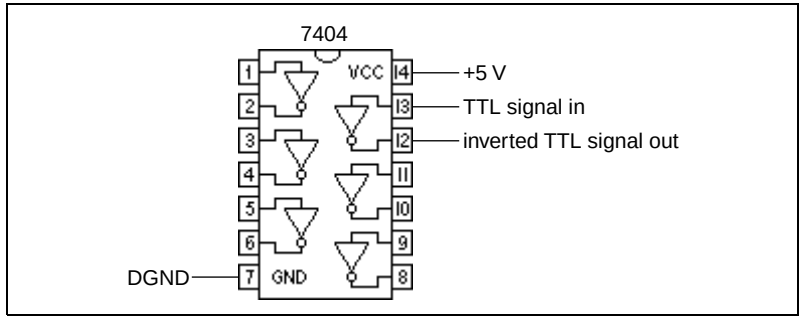


Figure 10-2. Wiring a 7404 Chip to Invert a TTL Signal

Note Refer to the VIs in the `examples\daq\counter` library for more information about the features of your counter chip.

Generating a Square Pulse or Pulse Trains

This section describes the ways you can generate a square pulse or multiple pulses (called *pulse trains*) using the counters available on your DAQ device with the example VIs in LabVIEW. All LabVIEW counter examples are in the `examples\daq\counter` library.

Generating a Square Pulse

There are many applications where you may need to generate TTL pulses. TTL pulses can be used as clock signals, gates, and triggers. You can use a pulse train of known frequency to determine an unknown TTL pulse width. You also can use a single pulse of known duration to determine an unknown TTL signal frequency, or use a single pulse to trigger an analog acquisition.

There are two basic types of counter signal generation—*toggled* and *pulsed*. When a counter reaches a certain value, a counter configured for toggled output changes the state of the output signal, while a counter configured for pulsed output outputs a single pulse. The width of the pulse is equal to one cycle of the counter SOURCE signal.

The following is a list of terms you should know before outputting a pulse or pulse train using LabVIEW:

- *Phase 1* refers to the first phase or delay to the pulse.
- *Phase 2* refers to the second phase or the pulse itself.

- *Period* is the sum of phase 1 and phase 2.
- *Frequency* is the reciprocal of the period (1/period).

In LabVIEW, you can adjust and control the times of phase 1 and phase 2 in your counting operation. You do this by specifying a *duty cycle*. The duty cycle equals

$$\frac{phase\ 2}{period},$$

where *period = phase 1 + phase 2*.

Examples of various duty cycles are shown in Figure 10-3. The first line shows a duty cycle of 0.5, where phase 1 and phase 2 are the same duration. A signal with a 0.5 duty cycle acts as a SOURCE for counter operations. The second line shows a duty cycle of 0.1, where phase 1 has increased and phase 2 has decreased. The final line shows a large duty cycle of 0.9, where phase 1 is very short and the phase 2 duration is longer.

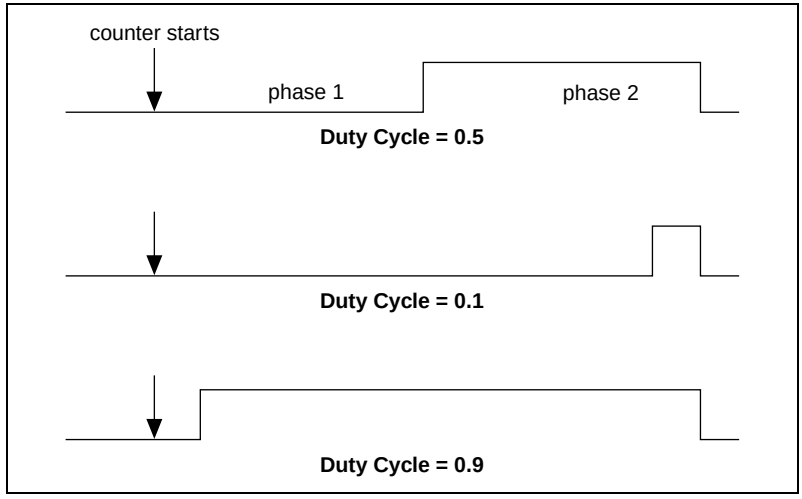


Figure 10-3. Pulse Duty Cycles

Note A high duty cycle denotes a long pulse phase relative to the delay phase.

How you generate a square pulse varies depending upon which counter chip your DAQ hardware has. If you are unsure which chip your device uses, refer to your hardware documentation.

TIO-ASIC, DAQ-STC, and Am9513

When generating a pulse or pulse train with the TIO-ASIC, DAQ-STC, or Am9513 chip, you can define the polarity of the signal as positive or negative. Figure 10-4 shows these pulse polarities. Notice that for a signal with a positive polarity, the initial state is low, while a signal with a negative polarity has a high initial state.

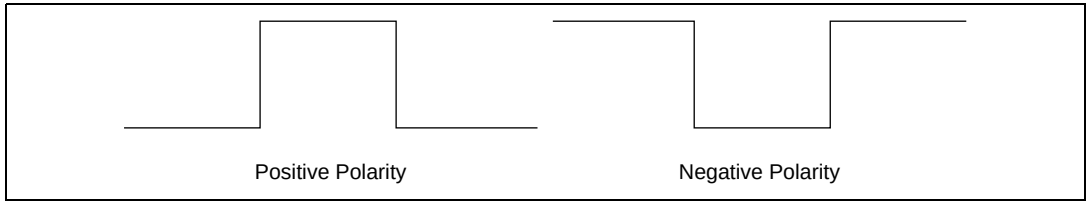


Figure 10-4. Positive and Negative Pulse Polarity

Each counter-generated pulse consists of two phases. If the counter is configured to output a signal with positive polarity and toggled output, as shown in the following diagram, the period of time from when the counter starts counting to the first rising edge is called phase 1. The time between the rising and the following falling edge is called phase 2. If you configure the counter to generate a continuous pulse train, the counter repeats this process many times as shown on the bottom line of Figure 10-5.

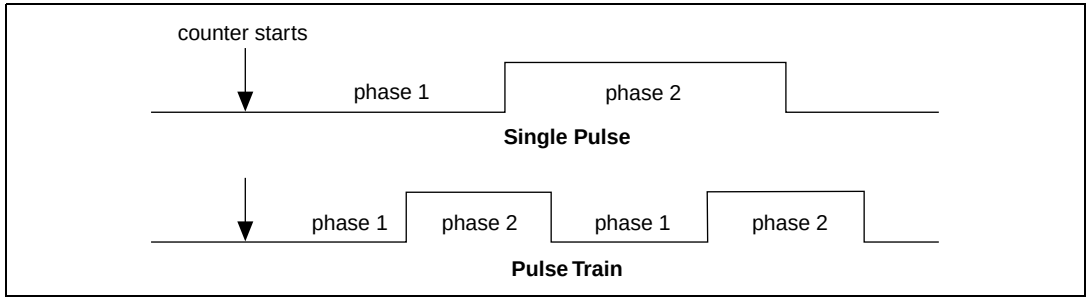


Figure 10-5. Pulses Created with Positive Polarity and Toggled Output

8253/54

When generating a pulse with the 8253/54 chip, the hardware limits you to a negative polarity pulse, as shown in Figure 10-4. The period of time from when the counter starts counting to the falling edge is called phase 1. The time between the falling and following rising edge is called phase 2. Figure 10-6 shows these phases for a single negative-polarity pulse.

To create a positive-polarity pulse, you can connect your negative-polarity pulse to an external 7404 inverter chip.

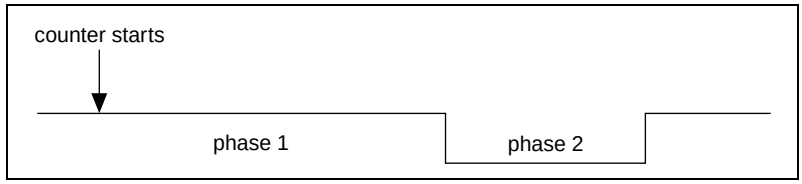


Figure 10-6. Phases of a Single Negative Polarity Pulse

When generating a pulse train with the 8253/54 chip, the hardware limits you to positive polarity pulses. Furthermore, the value loaded in the count register is divided equally to create phase 1 and phase 2. This means you will always get a 0.5 duty cycle if the count register is loaded with an even number. If you load the count register with an odd number, phase 1 will be longer than phase 2 by one cycle of the counter CLK signal.

Now that you know the terms involving generating a single square pulse or a pulse train, you can learn about the LabVIEW VIs and the physical connections needed to implement your application.

Generating a Single Square Pulse

You can use a single pulse to trigger analog acquisition or to gate another counter operation. You also can use a single pulse to stimulate a device or circuit for which you need to acquire and test the response.

TIO-ASIC, DAQ-STC, Am9513

Figure 10-7 shows two ways to connect your counter to generate a square pulse. In the Basic Connection, the edges of the internal SOURCE signal are counted to generate the output signal, the GATE is not used (software start), and the pulse signal on the OUT pin gets connected to your device. For optional connections, you will acquire an external SOURCE from your device which is also gated by your device. You can use either or both of these options.

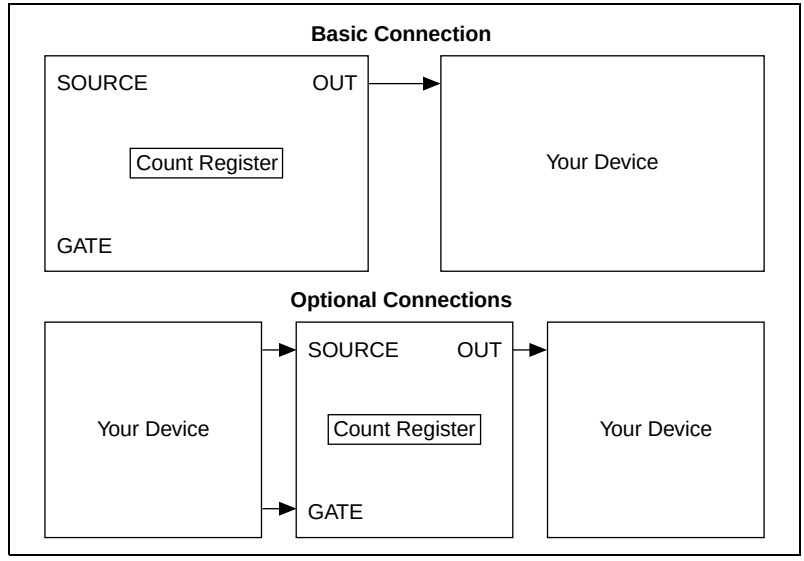


Figure 10-7. Physical Connections for Generating a Square Pulse

Open the Generate Single Pulse (DAQ-STC), Generate Single Pulse (NI-TIO), or Generate Delayed Pulse-Easy (9513) examples and study their block diagrams.

The Generate Delayed Pulse VI, available on the **Functions»Data Acquisition»Counter** palette, tells your device to generate a single delayed pulse. This VI is self-contained and checks for errors automatically. With the Generate Delayed Pulse VI, you must connect the **pulse delay** (phase 1) and **pulse width** (phase 2) controls to define the output pulse. Sometimes the **actual delay** and **actual width** are not the same as you specified.

To gain more control over when the counter begins generating a single square pulse, use Intermediate VIs instead of the Easy VIs. You also can use the example Delayed Pulse-Int (9513) VI, available in the `examples\daq\counter\Am9513.llb`. This example shows how to generate a single pulse using Intermediate level VIs. The Delayed Pulse Generator Config VI configures the counter, and the Counter Start VI generates the TTL signal. An example of this is generating a pulse after meeting certain conditions. If you use the Easy Counter VI, the VI configures and then immediately starts the pulse generation. With the Intermediate VIs, you can configure the counter long before the actual pulse generation begins. As soon as you want a pulse to be generated, the counter can immediately begin without having to configure the counter. In this situation, using

Intermediate VIs improves performance. You must stop the counter if you want to use it for other purposes.

8253/54

Refer to the Delayed Pulse (8253) VI in the `examples\daq\counter\8253.llb` for an example of how to generate a negative polarity pulse. Due to the nature of the 8253/54 chip, three counters are used to generate this pulse. Because only **counter 0** is internally connected to a clock source, it is used to generate the timebase. **counter 1** is used to create the pulse delay that gates **counter 2**. **counter 2** is used to generate the pulse, which occurs on the OUT pin. Using multiple counters requires external wiring, which is shown in Figure 10-8 and described on the front panel of the VI.

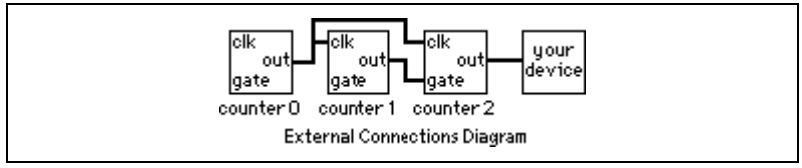


Figure 10-8. External Connections Diagram from the Front Panel of Delayed Pulse (8253) VI

The block diagram uses a sequence structure to divide the basic tasks involved. In frame 0 of the sequence all of the counters are reset. Notice that counters 1 and 2 are reset so their output states start out high.

In frame 1 of the sequence, the counters are set up for different counting modes. **counter 0** is set up to generate a timebase using the ICTR Timebase Generator subVI. **counter 1** is set up to toggle its output (low-to-high) when it reaches terminal count (TC). This toggled output is used to gate **counter 2**. **counter 2** is set up to output a low pulse when its gate goes high.

In frame 2 of the sequence, a delay occurs so the delayed pulse has time to complete before the example can run again. This is useful if the example is used as a subVI that is called repeatedly.

While this example works well for most pulses, it has limitations when your pulse delay gets very short (in the microsecond range), or when the ratio of pulse delay to pulse width gets very large.

Generating a Pulse Train

There are two types of pulse trains: *continuous* and *finite*. You can use a continuous pulse train as the SOURCE (CLK) of another counter or as the clock for analog acquisition (or generation). You can use a finite pulse train as the clock of an analog acquisition that acquires a predetermined number of points, or to provide a finite clock to an external circuit.

Generating a Continuous Pulse Train

How you generate a continuous pulse varies depending upon which counter chip your DAQ hardware has. If you are not sure which chip your device uses, refer to your hardware manual.

TIO-ASIC, DAQ-STC, Am9513

Figure 10-9 shows how to connect your counter and device to generate a continuous pulse train. The edges of the internal source signal are counted to generate the output signal. You obtain the continuous pulse train for your external device from the counter OUT pin. You also can gate the operation with a signal connected to the GATE input pin. Instead of having an internal timebase as your SOURCE, you can connect an external signal.

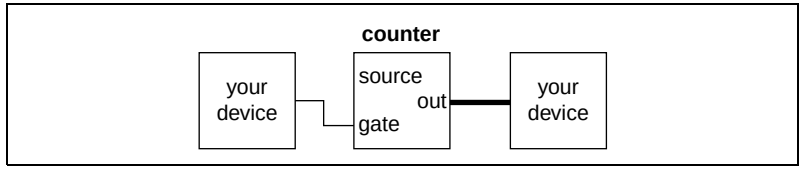


Figure 10-9. Physical Connections for Generating a Continuous Pulse Train

Open the Generate Pulse Train (DAQ-STC), Generate Pulse Train (NI-TIO), or Cont Pulse Train-Easy (9513) VIs, available in the `examples\daq\counter` library and study their block diagrams.

8253/54

Figure 10-10 shows how to connect your counter and device to generate a continuous pulse train. If you use counter 0, an internal source is counted to generate the output signal. If you use counter 1 or 2, you will need to connect your own source to the CLK pin. You obtain the continuous pulse train for your external device from the counter OUT pin.

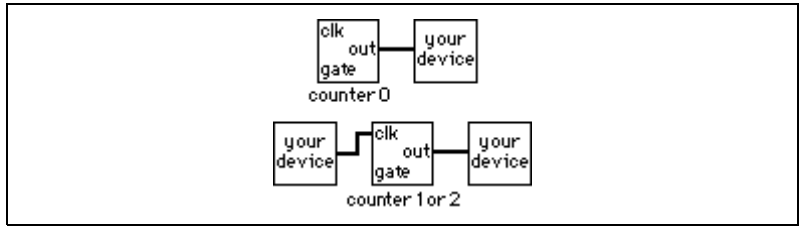


Figure 10-10. External Connections Diagram from the Front Panel of Cont Pulse Train (8253) VI

Refer to the Cont Pulse Train (8253) VI in the `examples\daq\counter\8253.llb` for an example of how to use the Generate Pulse Train (8253) VI to generate a continuous pulse train. When using **counter 0** with this VI, you can specify the desired frequency. The actual frequency shows the closest frequency to your desired frequency that the counter was able to achieve. The actual duty cycle will be as close to 0.5 as possible for your actual frequency. When using **counter 1** or **counter 2**, you specify the divisor factor N to be used to divide your supplied source. You also can enter the user-supplied timebase if you want the VI to calculate your actual frequency and actual duty cycle. When you click the **STOP** button, the While Loop stops, and a call to ICTR Control resets the counter, stopping the generation.

Generating a Finite Pulse Train

How you generate a finite pulse varies depending upon which counter chip your DAQ hardware has. If you are not sure which chip your device uses, refer to your hardware manual.

You can use the Easy I/O VI, Generate Pulse Train, or a stream of Intermediate VIs to generate a finite pulse train. With either technique, you must use two counters as shown in Figure 10-11. The maximum number of pulses in the pulse train is 2^{16}–1 for Am9513 devices and 2^{24}–1 for DAQ-STC devices.

Figure 10-11 shows the physical connections to produce a finite pulse train on the OUT pin of a counter. **counter** generates the finite pulse train with high-level gating. **counter-1** provides **counter** with a long enough gate pulse to output the number of desired pulses. You must externally connect the OUT pin of the **counter-1** to the GATE pin of **counter**. You also can gate **counter-1**.

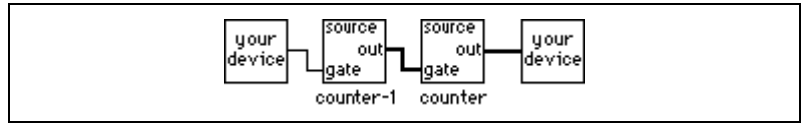


Figure 10-11. Physical Connections for Generating a Finite Pulse Train

Open the Finite Pulse Train (DAQ-STC), Finite Pulse Train (NI-TIO), or Finite Pulse Train-Easy (9513) VIs, available in the `examples\daq\counter` library, and study the block diagrams.

8253/54

Generating a finite pulse train with the 8253/54 chip uses all three counters. Figure 10-12 shows how to externally connect your counters. Because **counter 0** is internally connected to a clock source, **counter 0** is used to generate the timebase used by **counter 1** and **counter 2**. **counter 1** generates a single low pulse used to gate **counter 2**. Because **counter 2** must be gated with a high pulse, the output of **counter 1** is passed through a 7404 inverter chip prior to being connected to the GATE of **counter 2**. **counter 2** is set up to generate a pulse train at its OUT pin.

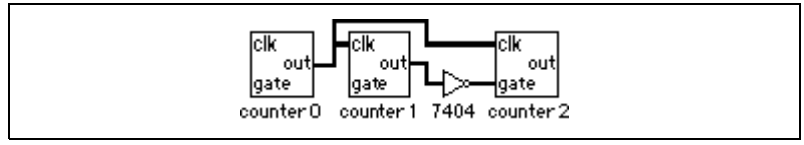


Figure 10-12. External Connections Diagram from the Front Panel of Finite Pulse Train (8253) VI

Refer to the Finite Pulse Train (8253) VI in the `examples\daq\counter\8253.llb` for an example of how to generate a finite pulse train. This example uses a sequence structure to divide the basic tasks involved. In frame 0 of the sequence, all of the counters are reset. Notice **counter 1** is reset so its output state starts high.

In frame 1 of the sequence, the counters are set up for different counting modes. **counter 0** is set up to generate a timebase using the ICTR Timebase Generator VI. **counter 1** is set up to output a single low pulse using the ICTR Control VI. **counter 2** is set up to output a pulse train using the ICTR Timebase Generator VI.

In frame 2 of the sequence, a delay occurs so that the finite pulse train has time to complete before the example can be run again. This is useful if the example is used as a subVI where it may get called over and over.

Counting Operations When All Your Counters Are Used

The DAQ-STC and Am9513 have counting operations available even when all the counters have been used.

DAQ-STC devices feature a FREQ_OUT pin, and Am9513 devices feature an FOUT pin. You can generate a 0.5 duty cycle square wave on these pins without using any of the available counters.

The CTR Control VI, available on the **Functions»Data Acquisition» Counter»Advanced Counter»AM9513 & Compatibility** palette, enables and disables the FOUT signal and sets the square wave frequency. The square wave frequency is defined by the FOUT timebase signal divided by the FOUT divisor.

Note If you are using NI-DAQ 6.5 or higher, National Instruments recommends you use the new Advanced Counter VIs, such as Counter Group Config, Counter Get Attribute, Counter Set Attribute, Counter Buffer Read, and Counter Control.

You also can refer to the Generate Pulse Train on FREQ_OUT VI in the `examples\daq\counter\DAQ-STC.llb` or the Generate Pulse Train on FOUT VI in the `examples\daq\counter\Am9513.llb` for examples of how to generate a pulse train on these outputs.

Knowing the Accuracy of Your Counters

When you generate a waveform, there can be an uncertainty of up to one timebase period between the start signal and the first counted edge of the timebase. This is due to the uncertainty in the exact relation of the start signal, which the software calls or the gate signal supplies to the first edge of the timebase, as shown in Figure 10-13.

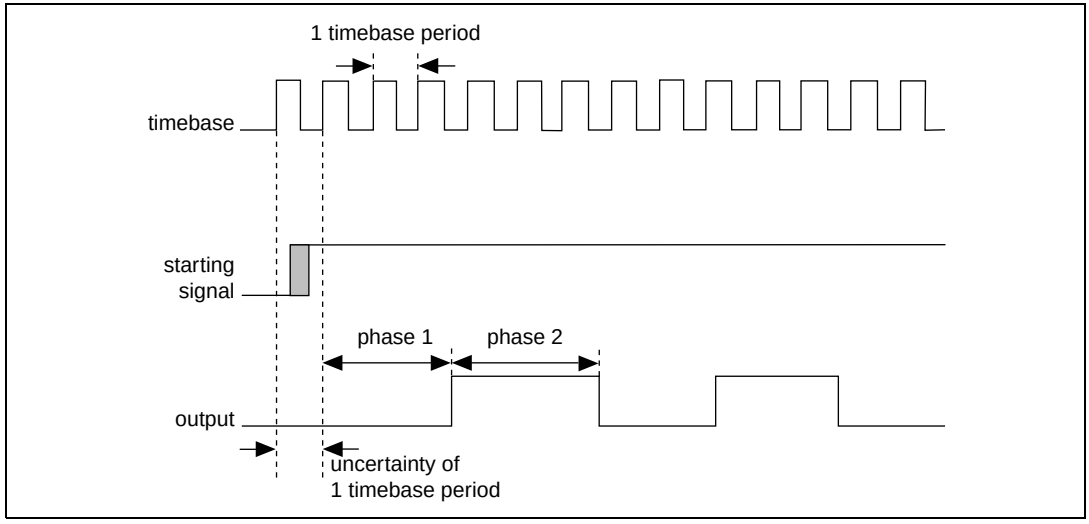


Figure 10-13. Uncertainty of One Timebase Period

8253/54

In addition to the previously described uncertainty, the 8253/54 chip has an additional uncertainty when used in mode 0. Mode 0 generates a low pulse for a chosen number of clock cycles, but a software delay is involved. This delay occurs because with mode 0 the counter output is set low by a software write to the mode setting. Afterward the count can be loaded and the counter starts counting down. The time between setting the output to low and loading the count is included in the output pulse. This time was found to be 20 µs when tested on a 200 MHz Pentium computer.

Stopping Counter Generations

You can stop a counting operation in several ways. You can restart a counter for the same operation it just completed, you can reconfigure it to do something else, or you can call a specific VI to stop it. All of these methods allow you to use counters for different operations without resetting the entire device.

DAQ-STC, Am9513

Figure 10-14 shows how to stop a counter using the Intermediate VI, Counter Stop. Notice that the Wait+ (ms) VI is called before Counter Stop. The Wait+ (ms) VI allows you to wire a time delay so that the previous counter operation has time to complete before the Counter Stop VI is called. The Wait+ (ms) and Counter Stop VIs are available on the **Functions»Data Acquisition»Counter»Intermediate Counter** palette.

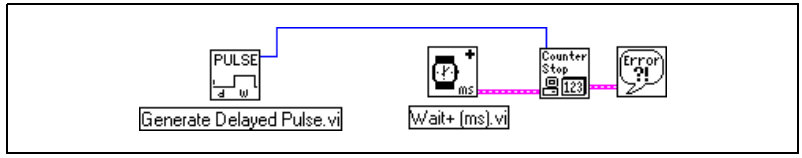


Figure 10-14. Using the Generate Delayed Pulse and Stopping the Counting Operation

To stop a generated pulse train, you can use another Generate Pulse Train VI with the number of pulses input set to –1, shown in Figure 10-15. This example expects that a pulse train is already being generated. The call to Generate Pulse Train VI stops the counter, and the call to Generate Delayed Pulse VI sets the counter up for a different operation.

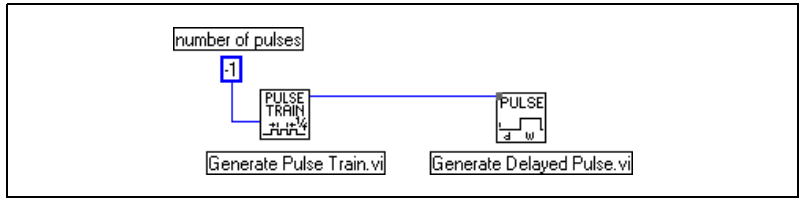


Figure 10-15. Stopping a Generated Pulse Train

8253/54

Calling ICTR Control VI with a control code of 7 (reset) can stop a counter on the 8253/54 chip.

Measuring Pulse Width

This section describes how you can use a counter to measure pulse width. There are several reasons you may need to determine pulse width. For example, to determine the duration of an event, set your application to measure the width of a pulse that occurs during that event. Another example is determining the interval between two events. In this case, you measure the pulse width between the two events. An example of when you might use this type of application is determining the time interval between two boxes on a conveyor belt or the time it takes one box to be processed through an operation. The event is an edge every time a box goes by a point, which prompts a digital signal to change in value. All LabVIEW counter examples are located in the `labview\examples\daq\counter` library.

Measuring a Pulse Width

You can measure an unknown pulse width by counting the number of pulses of a faster known frequency that occur during the pulse to be measured. Connect the pulse you want to measure to the GATE input pin and a signal of known frequency to the SOURCE (CLK) input pin, as shown in Figure 10-16. The pulse of unknown width (T_{pw}) gates the counter configured to count a timebase clock of known period (T_s). The pulse width equals the timebase period times the count, or: $T_{pw} = T_s \times count$. The SOURCE (CLK) input can be an external or internal signal.

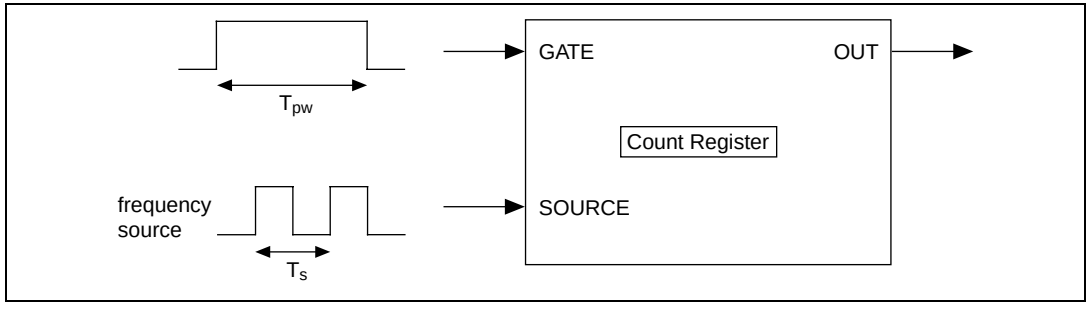


Figure 10-16. Counting Input Signals to Determine Pulse Width

An internal signal is based upon the type of counter chip on your DAQ device. With TIO-ASIC devices, you can choose internal timebases of 20 MHz, 100 kHz, and a device-specific maximum timebase. With DAQ-STC devices, you have a choice between internal timebases of 20 MHz and 100 kHz. With Am9513 devices, you can choose internal timebases of 1 MHz, 100 kHz, 10 kHz, 1 kHz, and 100 Hz. With 8253/54

devices, the internal timebase is either 2 MHz or 1 MHz, depending on which device you have.

Figure 10-17 shows how to physically connect the counter on your device to measure pulse width.

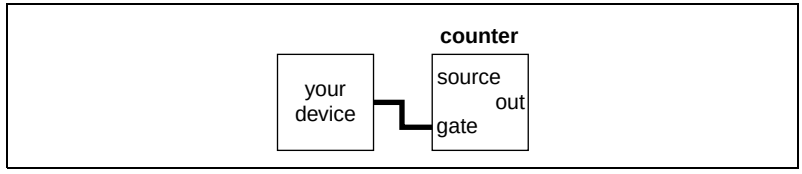


Figure 10-17. Physical Connections for Determining Pulse Width

Determining Pulse Width

How you determine a pulse width depends upon which counter chip is on your DAQ device. If you are not sure which counter chip your DAQ device has, refer to your hardware user manual.

Open the Measure Pulse (DAQ-STC) and Measure Pulse (NI-TIO) examples and study their block diagrams.

Am9513

Open the block diagram of the Measure Pulse-Easy (9513) VI, available on the **Functions»Data Acquisition»Counter** palette. This VI uses the Easy VI, Measure Pulse Width or Period.

The Measure Pulse Width or Period VI counts the number of cycles of the specified timebase, depending on your choice from the type of measurement menu located on the front panel of the VI. The measurement menu choices for this VI are the following:

- Measure high pulse width
- Measure low pulse width
- Measure period (rising edge to rising edge)
- Measure period (falling edge to falling edge)
- High pulse width (multiple pulses, DAQ-STC)
- Low pulse width (multiple pulses, DAQ-STC)

Either menu choice can be used to measure the width of a single pulse or to measure a pulse within a train of multiple pulses. However, the pulse must occur after the counter starts. Because the counter uses high-level gating, it might be difficult to measure a pulse within a fast pulse train. If the counter is started in the middle of a pulse, it measures the remaining width of that pulse.

The timebase you choose determines how long a pulse you can measure with the 16-bit counter. For example, the 100 Hz timebase allows you to measure a pulse up to $2^{16} \times 10$ ms = 655 seconds long. The 1 MHz timebase allows you to measure a pulse up to 65 ms long. Because a faster timebase yields a more accurate pulse-width measurement, it is best to use the fastest timebase possible without the counter reaching terminal count (TC).

The **valid?** output of the example VI indicates whether the counter measured the pulse without overflowing (reaching TC). However, **valid?** does not tell you whether a whole pulse was measured when measuring a pulse within a pulse train.

8253/54

Open the block diagram of the Measure Short Pulse Width (8253) VI located in the `examples\daq\counter\8253.llb`.

This VI counts the number of cycles of the internal timebase of **Counter 0** to measure a high pulse width. You can measure a single pulse or a pulse within a train of multiple pulses. However, the pulse must occur after the counter starts. This means it may be difficult to measure a pulse within a fast pulse train because the counter uses high-level gating. To measure a low pulse width, insert a 7404 inverter chip between your pulse source and the GATE input of counter 0.

On the Measure Short Pulse Width (8253) VI block diagram, the first call to ICTR Control VI sets up counting mode 4, which tells the counter to count down while the gate input is high. The Get Timebase (8253) VI is used to get the timebase of your DAQ device. A DAQ device with an 8253/54 counter has an internal timebase of either 1 MHz or 2 MHz, depending on the device. Inside the While Loop, ICTR Control VI is called to continually read the count register until one of four conditions are met:

- The count register value has decreased but is no longer changing. It is finished measuring the pulse.
- The count register value is greater than the previously read value. An overflow has occurred.

- An error has occurred.
- Your chosen time limit has been reached.

After the While Loop, the final count is subtracted from the originally loaded count of 65,535 and multiplied by the timebase period to yield the pulse width. Finally, the last ICTR Control VI resets the counter. Notice that this VI uses only **Counter 0**. If **Counter 0** has an internal timebase of 2 MHz, the maximum pulse width you can measure is $2^{16} \times 0.5\ \mu s = 32$ ms. Refer to the information found in **Context Help** for a complete description of this example.

Controlling Your Pulse Width Measurement

How you control your pulse-width measurement depends upon which counter chip is on your DAQ device. If you are not sure which counter chip your DAQ device has, refer to your hardware user manual.

TIO-ASIC, DAQ-STC, or Am9513

Figure 10-18 shows one approach to measuring pulse width using the Intermediate VIs Pulse Width or Period Meas Config, Counter Start, Counter Read, and Counter Stop. You can use these VIs to control when the measurement of the pulse widths begins and ends. The Pulse Width or Period Config VI configures a counter to count the number of cycles of a known internal timebase. The Counter Start VI begins the measurement. The Counter Read VI determines if the measurement is complete and displays the count value. After the While Loop is stopped, the Counter Stop VI stops the counter operation. Finally, the General Error Handler VI notifies you of any errors.

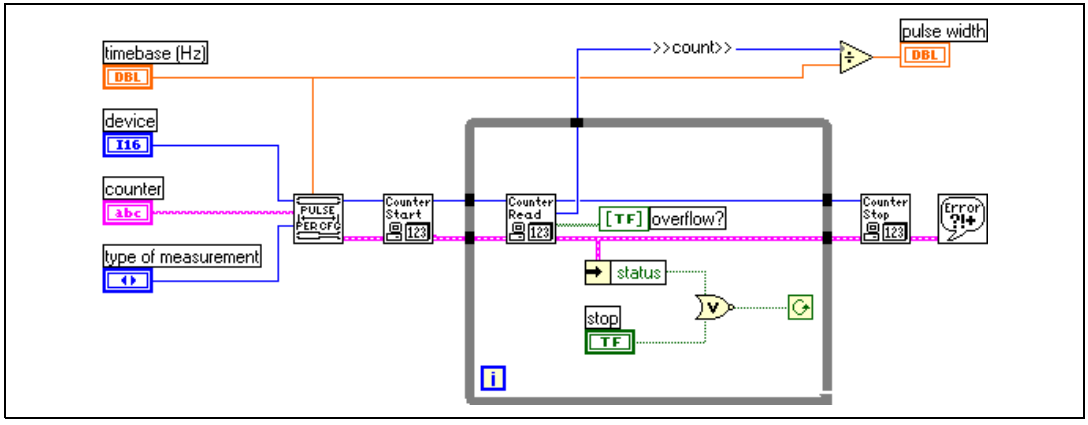


Figure 10-18. Measuring Pulse Width with Intermediate VIs

Buffered Pulse and Period Measurement

With the TIO-ASIC and DAQ-STC chips, LabVIEW provides a buffer for counter operations. You typically use buffered counter operations when you have a gate signal to trigger a counter several times.

Open the Measure Buffered Pulse (DAQ-STC) and Measure Buffered Pulse (NI-TIO) examples in the `examples\daq\counter` library and study the block diagrams.

Note If you are using NI-DAQ 6.5 or higher, National Instruments recommends you use the new Advanced Counter VIs, such as Counter Group Config, Counter Get Attribute, Counter Set Attribute, Counter Buffer Read, and Counter Control.

Increasing Your Measurable Width Range

The maximum counting range of a counter and the chosen internal timebase determine how long of a pulse width can be measured. The internal timebase acts as the SOURCE. When measuring the pulse width of a signal, you count the number of source edges that occur during the pulse being measured. The counted number of SOURCE edges cannot exceed the counting range of the counter. Slower internal timebases allow you to measure longer pulse widths, but faster timebases give you a more accurate pulse-width measurement. If you need a slower timebase than is available on your counter as shown in Table 10-1, set up an additional counter for pulse-train generation and use the OUT of that counter as the SOURCE of the counter measuring pulse width.

Table 10-1. Internal Counter Timebases and Their Corresponding Maximum Pulse Width, Period, or Time Measurements

Counter Type	Internal Timebases	Maximum Measurement
TIO-ASIC	80 MHz*	53.69 s
	20 MHz	214.748 s
	100 kHz	11 h 55 m 49.67 s
DAQ-STC	20 MHz	838 ms
	100 kHz	167 s

Table 10-1. Internal Counter Timebases and Their Corresponding Maximum Pulse Width, Period, or Time Measurements (Continued)

Counter Type	Internal Timebases	Maximum Measurement
Am9513	1 MHz	65 ms
	100 kHz	655 ms
	10 kHz	6.5 s
	1 kHz	65 s
	100 Hz	655 s
8253/54	2 MHz**	32 ms
	1 MHz**	65 ms

* Some devices have a maximum timebase of 20 MHz.

** A DAQ device with an 8253/54 counter has one of these internal timebases available on counter 0, but not both.

Measuring Frequency and Period

This section describes the various ways you can measure frequencies and periods of TTL signals using the counters on your DAQ device. One cycle of a signal, known as the period, is measured in units of time, usually seconds. The inverse of period is frequency, which is measured in cycles per second or hertz (Hz). The rate of your signal and the type of counter on your DAQ device determine whether you use frequency or period measurement. An example of when you would want to know the frequency of a signal is if you need to monitor the shaft speed of a motor.

Knowing How and When to Measure Frequency and Period

A common way to measure the frequency of a signal is to measure the number of pulses that occur during a known time period. Figure 10-19 illustrates the measurement of a pulse train of an unknown frequency (f_s) by using a pulse of a known width (T_G). The frequency of the waveform equals the count divided by the known pulse width (frequency = count/T_G). The period is the reciprocal of the measured frequency (period = 1/f_s). You typically use frequency measurement for high-frequency signals where the signal to be measured is approaching or faster than the chosen internal timebase.

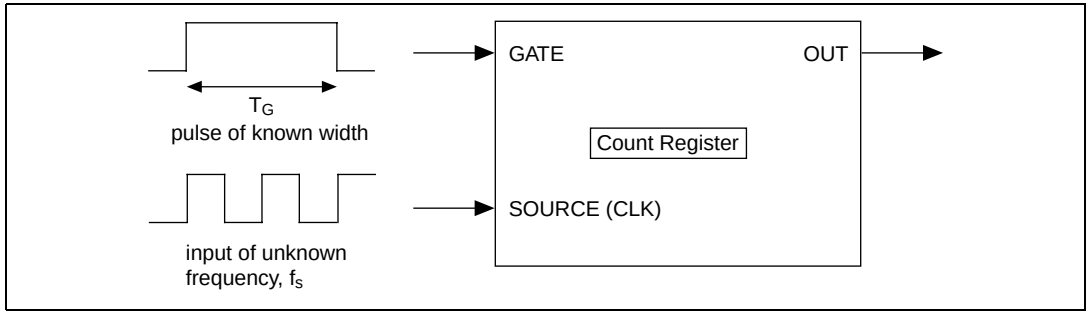


Figure 10-19. Measuring Square Wave Frequency

TIO-ASIC, DAQ-STC, Am9513

For period measurement, you count the number of pulses of a known frequency (f_s) during one period of the signal to be measured. As shown in Figure 10-20, the signal of a known frequency is connected to the SOURCE, and the signal to be measured is connected to the GATE. The period is the count divided by the known frequency (T_G = count/f_s).

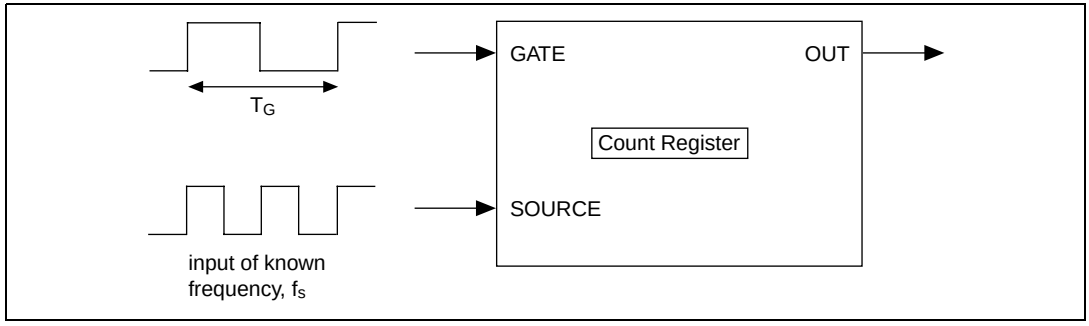


Figure 10-20. Measuring a Square Wave Period

You typically use period measurement for low-frequency signals where the signal to be measured is significantly slower than the chosen internal timebase. The internal timebases for the TIO-ASIC are 20 MHz, 100 kHz, and a device-specific maximum timebase. The internal timebases for the DAQ-STC are 20 MHz and 100 kHz. The internal timebases for the Am9513 are 1 MHz, 100 kHz, 10 kHz, 1 kHz, and 100 Hz. Whether you use period measurement or frequency measurement, you always can obtain

the other measurement by taking the inverse of the current one as shown in the following equations.

$$period\ measurement = \frac{1}{frequency\ measurement}$$

$$frequency\ measurement = \frac{1}{period\ measurement}$$

8253/54

The 8253/54 chip does not support period measurement, but you can use frequency measurement for a pulse train and take the inverse to get the period. The frequency examples discussed in this chapter calculate the period for you.

Connecting Counters to Measure Frequency and Period

Figure 10-21 shows typical external connections for measuring frequency. In the figure, your device provides the signal with the frequency to be measured to the SOURCE (CLK) of **counter**. It optionally can control the GATE of **counter-1**. The OUT of **counter-1** supplies a known pulse to the GATE of **counter**. Finally, **counter** counts the number of cycles of the unknown pulse during the known GATE pulse.

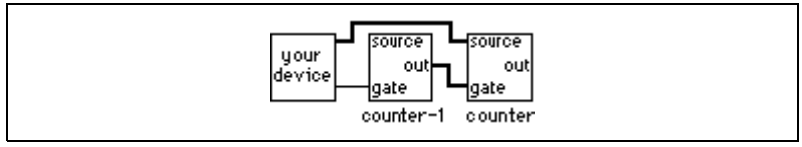


Figure 10-21. External Connections for Frequency Measurement

TIO-ASIC, DAQ-STC, Am9513

Figure 10-22 shows typical external connections for measuring period. In the figure, your device provides the signal with the period to be measured to the GATE of **counter**. A timebase of known frequency is supplied to the SOURCE. This usually is an internal timebase, but it can be externally supplied. The counting range of your counter must not be exceeded during the period measurement. The range of the Am9513 is 65,335; the range of the DAQ-STC is 16,777,216; and the range of the TIO-ASIC is 2^{32}–1. If the counting range is exceeded, select a slower internal timebase.

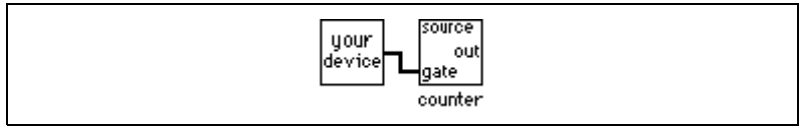


Figure 10-22. External Connections for Period Measurement

Measuring the Frequency and Period of High-Frequency Signals

How you measure the frequency and period of high-frequency signals (higher than 1000 Hz) depends on the counter chip on your DAQ device. If you are not sure which chip your DAQ device has, refer to your hardware user manual.

TIO-ASIC, DAQ-STC

Open the Measure Frequency (DAQ-STC) and Measure Frequency (NI-TIO) VIs in the `examples\daq\counter` library and study the block diagrams.

The counter counts the number of rising edges of a TTL signal at the SOURCE of counter during a known pulse at the GATE of **counter**. The width of that known pulse is determined by **gate width**. **Frequency** is the output for this example, and **period** is calculated by taking the inverse of the frequency.

Am9513

Refer to the Measure Frequency-Easy (9513) VI in the `examples\daq\counter\Am9513.llb` for an example of how to use the Easy VI, Measure Frequency, available on the **Functions»Data Acquisition» Counter** palette.

This VI initiates the counter to count the number of rising edges of a TTL signal at the SOURCE of counter during a known pulse at the GATE of counter. The width of that known pulse is determined by gate width. Frequency is the output for this example, and period is calculated by taking the inverse of the frequency. The **valid?** output lets you know if the measurement completed without an overflow. The number of counters to use input lets you choose one counter for 16-bit measurement or two counters for 32-bit measurement. Remember, you must externally wire your signal to be measured to the SOURCE of counter, and the OUT of **counter-1** must be wired to the GATE of counter.

TIO-ASIC, DAQ-STC, Am9513

If you need more control over when your frequency measurement begins and ends, use the Intermediate VIs instead of the Easy VIs. Figure 10-23 shows one approach for this that uses the Event or Time Counter Config, Adjacent Counters, Delayed Pulse Generator Config, Counter Start, CTR Control, Counter Read, and Counter Stop VIs. The Delayed Pulse Generator Config VI configures **counter** to count the number of pulses while its GATE is high. The Adjacent Counters VI is used to determine the correct **counter-1**. The Delayed Pulse Generator Config VI then configures **counter-1** to generate a single pulse for the GATE signal. The Counter Start VI begins the counting operation for **counter** first, then **counter-1**. The CTR Control VI is an Advanced VI that is used to check if the GATE pulse has completed. The Counter Read VI returns the count value from **counter**, which is used to determine the frequency and pulse width. Finally, the Counter Stop VI stops the counter operation.

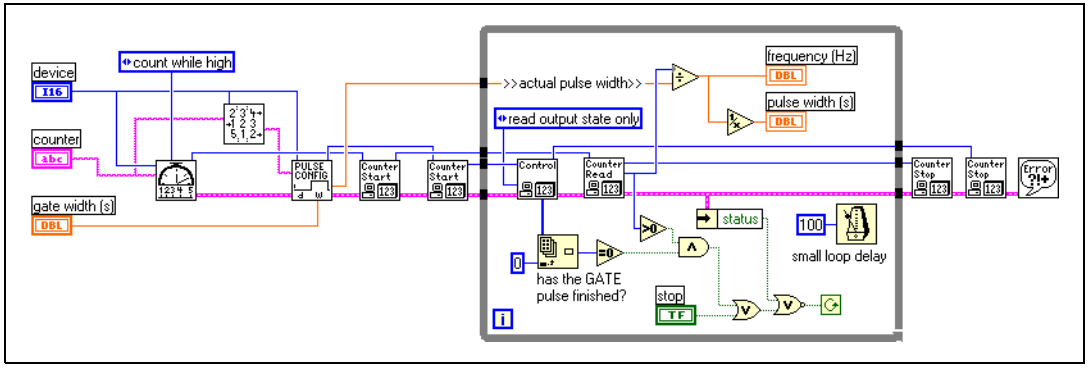


Figure 10-23. Frequency Measurement Example Using Intermediate VIs

8253/54

Refer to the Measure Hi Frequency (8253) VI in the `examples\daq\counter\8253.llb` for an example of how to initiate the counter to count the number of rising edges of a TTL signal at the CLK of **counter** during a known pulse at the GATE of **counter**. The known pulse is created by **counter 0**, and its width is determined by **gate width**. The maximum width of the pulse is 32 ms if your DAQ device has a 2 MHz internal timebase, and 65 ms if your DAQ device has a 1 MHz internal timebase. This maximum pulse is why this example only reads frequencies higher than 1 kHz. A frequency of 1 kHz generates 32 cycles during the 32 ms pulse. As this cycle count decreases (as with lower frequencies), the frequency measurement becomes less accurate. Frequency is the output for this example, and period is determined by taking the inverse of the frequency. You must externally wire the signal to be measured to the CLK of **counter**, and the OUT of **counter 0** must be wired through a 7404 inverter chip to the GATE of **counter**.

Notice the ICTR Control, Get Timebase (8253), and Wait + (ms) VIs on the block diagram. The first two ICTR Control VIs reset **counter** and **counter 0**. The next ICTR Control sets up **counter** to count down while its GATE input is high. The Get Timebase (8253) VI returns the internal timebase period for **counter 0** of the device. This value is multiplied by the gate width to yield the count to be loaded into the count register of **counter 0**. The next ICTR Control VI loads this count and sets up **counter 0** to output a low pulse, during which cycles of the signal to be measured are counted.

One advantage of this example is that it uses only two counters. However, this example has two notable limitations. One limitation is that it cannot accurately measure low frequencies. Refer to the Measure Lo Frequency (8253) VI in the `examples\daq\counter\8253.llb` if you need to measure lower frequencies. This VI uses three counters. The other limitation is that there is a software dependency, which causes **counter 0** to output a pulse slightly longer than the count it is given. This is the nature of the 8253 chip, and it can increase the readings of high frequencies. Use the Measure Hi Frequency–DigStart (8253) VI in the `examples\daq\counter\8253.llb` to avoid this software delay.

Measuring the Period and Frequency of Low-Frequency Signals

How you measure the period and frequency of low-frequency signals (lower than 1000 Hz) depends on which counter chip is on your DAQ device. If you are not sure which chip your DAQ device has, refer to your hardware documentation.

TIO-ASIC, DAQ-STC

Open the Measure Period (DAQ-STC) and Measure Period (NI-TIO) VIs, available in the `examples\daq\counter` library, and study the block diagrams.

You connect your signal of unknown period to the GATE of **counter**. The counter measures the period between successive rising edges of your TTL signal by counting the number of internal **timebase** cycles that occur during the period. The **period** is the count divided by the timebase. The **frequency** is determined by taking the inverse of the **period**. You must choose **timebase** such that the counter does not reach its highest value, or terminal count (TC).

Am9513

Refer to the Measure Period-Easy (9513) VI, in the `examples\daq\counter\Am9513.llb` for an example of how to use the Easy VI, Measure Pulse Width or Period, available on the **Functions»Data Acquisition»Counter** palette.

You connect your signal of unknown period to the GATE of **counter**. The counter measures the period between successive rising edges of your TTL signal by counting the number of internal **timebase** cycles that occur during the period. The **period** is the count divided by the timebase. The **frequency** is determined by taking the inverse of the **period**. The **valid?** output indicates if the period was measured without overflow. Overflow occurs when the counter reaches its highest value, or terminal count (TC). You must choose **timebase** such that it does not reach TC. With a timebase of 1 MHz, the Am9513 can measure a period up to 65 ms. With a timebase of 100 Hz, you can measure a period up to 655 seconds.

TIO-ASIC, DAQ-STC, Am9513

If you need more control over when period measurement begins and ends, use the Intermediate VIs instead of the Easy VIs. Figure 10-24 shows how to measure period and frequency.

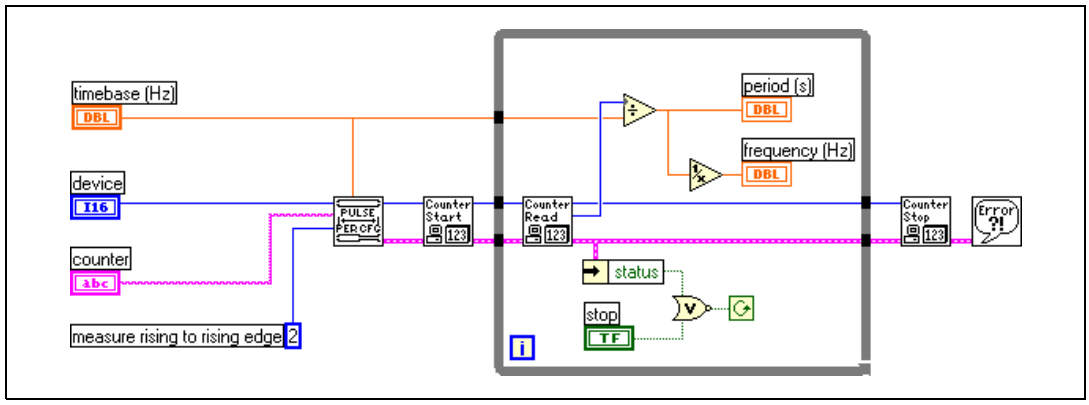


Figure 10-24. Measuring Period Using Intermediate Counter VIs

The Intermediate VIs used in Figure 10-24 include Pulse Width or Period Meas Config, Counter Start, Counter Read, and Counter Stop. The Pulse Width or Period Meas Config VI configures the counter for period measurement. The Counter Start VI begins the counting operation. The Counter Read VI returns the count value from the counter, which is used to determine the period and frequency. The Counter Stop VI stops the counter operation.

8253/54

The 8253/54 chip does not support period measurement, but you can use frequency measurement for a pulse train and take the inverse to get the period. The Measure Lo Frequency (8253) VI, available in the `examples\daq\counter\8253.llb`, measures frequency and calculates the period for you.

Counting Signal Highs and Lows

This section describes the various ways you can count TTL signals using the counters on your DAQ device. Counters can count external events such as rising and falling edges on the SOURCE (CLK) input pin. They also can count elapsed time using the rising and falling edges of an internal timebase. An example of counting events is calculating the output of a production line. An example of counting time is calculating how long it takes to produce one item on a production line.

Connecting Counters to Count Events and Time

Figure 10-25 shows typical external connections for counting events. In the figure, **your device** provides the TTL signal to be counted, and it is connected to the SOURCE (CLK) of **counter**. The number of events counted is determined by reading the count register of **counter**.

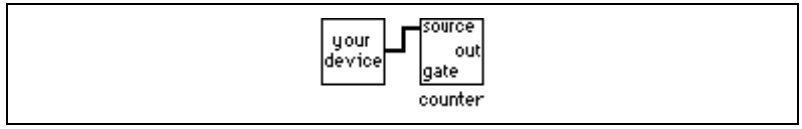


Figure 10-25. External Connections for Counting Events

Figure 10-26 shows typical external connections for counting elapsed time. In the figure, **your device** provides a pulse to the GATE of **counter**. While the gate pulse is high, **counter** counts a known internal timebase. Dividing the count by the internal timebase determines the elapsed time.

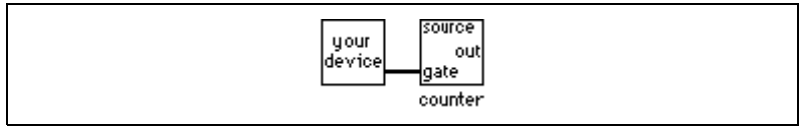


Figure 10-26. External Connections for Counting Elapsed Time

Am9513

With the Am9513, you can extend the counting range of a counter by connecting the OUT of one counter to the SOURCE of the next higher order counter (counter+1). This is called *cascading* counters. By cascading counters you can increase your counting range from a 16-bit counting range of 65,535 to a 32-bit counting range of 4,294,967,295. The Am9513 chip has a set of five counters where higher-order counters can be cascaded. The TIO-10 device has two Am9513 chips for a total of 10 counters. Table 10-2

identifies adjacent counters on the Am9513 (one and two chips). This information is useful when cascading counters.

Table 10-2. Adjacent Counters for Counter Chips

Next Lower Counter	Counter	Next Higher Counter
5	1	2
1	2	3
2	3	4
3	4	5
4	5	1
10	6	7
6	7	8
7	8	9
8	9	10
9	10	6

Figure 10-27 shows typical external connections for cascading counters when counting events. Notice that the OUT of **counter** is connected to the SOURCE of **counter+1**.

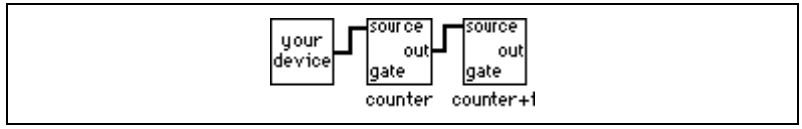


Figure 10-27. External Connections to Cascade Counters for Counting Events

Figure 10-28 shows typical external connections for cascading counters when counting elapsed time. Notice that the OUT of **counter** is connected to the SOURCE of **counter+1**.

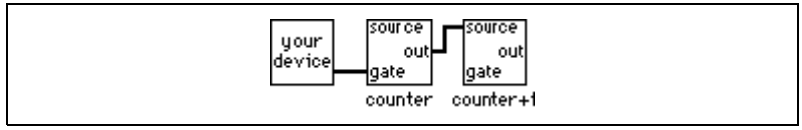


Figure 10-28. External Connections to Cascade Counters for Counting Elapsed Time

Counting Events

How you count events depends upon which counter chip is on your DAQ device. If you are not sure which counter your DAQ device has, refer to your hardware user manual.

TIO-ASIC, DAQ-STC

Open the Count Edges (DAQ-STC) and Count Edges (NI-TIO) VIs in the `daq\examples\counter` library and study the block diagrams.

Am9513

The Count Events-Easy (9513) VI, available in the `examples\daq\counter\Am9513.llb`, uses the Easy VI, Count Events or Time, available on the **Functions»Data Acquisition»Counter** palette.

This VI initiates the counter to count the number of rising edges of a TTL signal at the SOURCE of counter. The counter continues counting until you click the **STOP** button. You must externally wire your signal to be counted to the SOURCE of **counter**. Additionally, you can cascade two counters by choosing **two counters (32-bits)** in the **number of counters to use** menu. This extends your counting range to over 4 billion. You must also wire the OUT of **counter** to the SOURCE of **counter+1** for this increased counting range.

If you need more control over when your event counting begins and ends, use the Intermediate VIs instead of the Easy VIs. Refer to the Count Events-Int (9513) VI in the `examples\daq\counter\Am9513.llb` for an example of using Intermediate VIs for more control over when your event counting begins and ends.

This example uses the Intermediate VIs Event or Time Counter Config, Counter Start, Counter Read, and Counter Stop. The Event or Time Counter Config VI configures **counter** to count the number of rising edges of a TTL signal at its SOURCE input pin. The Counter Start VI begins the counting operation for **counter**. The Counter Read VI returns the count until you click the **STOP** button or an error occurs. Finally, the Counter Stop VI stops the counter operation. You must externally wire your signal to be counted to the SOURCE of **counter**. You also can gate counter with a pulse to control when it starts and stops counting. To do this, wire your pulse to the GATE of **counter**, and choose the appropriate **gate mode** from the front panel menu. Additionally, you can cascade two counters by choosing **two counters (32-bits)** in the **number of counters to use** menu. This extends

your counting range to over 4 billion. You must also wire the OUT of **counter** to the SOURCE of **counter+1** for this increased counting range.

8253/54

The Count Events (8253) VI, available in the `examples\daq\counter\8253.llb`, uses the Intermediate VI, ICTR Control, available on the **Functions»Data Acquisition»Counter»Intermediate Counter** palette.

This VI initiates the counter to count the number of rising edges of a TTL signal at the CLK of **counter**. Looking at the block diagram, the first call to ICTR Control loads the count register and sets up **counter** to count down. The second call to ICTR Control reads the count register. Inside the first While Loop, the count is read until it changes. While the count register has previously been loaded, the new value is not active until the first edge is counted on the CLK pin. Once the first edge comes in, the second While Loop takes over and continually reads the count until you click the **STOP** button or an error occurs. You must externally wire your signal to be counted to the CLK of counter.

Counting Elapsed Time

How you count elapsed time depends upon which counter chip is on your DAQ device. If you are not sure which chip your DAQ device has, refer to your hardware user manual.

TIO-ASIC, DAQ-STC

Open the Count Time-Easy (DAQ-STC) and Count Edges (NI-TIO) VIs, available in the `examples\daq\counter` library, and study the block diagrams.

The counter is incremented by every rising edge of its source, typically a known internal timebase, and the elapsed time is calculated by the value of counter times the period of the internal timebase.

Am9513

The Count Time-Easy (9513) VI, available in the `examples\daq\counter\Am9513.llb`, uses the Easy VI, Count Events or Time, available on the **Functions»Data Acquisition»Counter** palette.

This VI initiates the counter to count the number of rising edges of a known internal timebase at the SOURCE of **counter**. The Count Events or

Time VI takes care of dividing the count by the timebase frequency to determine the **elapsed time**. The counter continues timing until you click the **STOP** button. You do not need to make any external connections if the **number of counters to use** menu is set to **one counter (16-bits)**. If you **set the number of counters to use** menu to **two counters (32-bits)**, you must externally wire the OUT of **counter** to the SOURCE of **counter+1**. The length of time that can be counted depends on the maximum count of the counter(s) and the chosen **timebase**. For example, the 65,535 (16-bit) count of the Am9513 and a timebase of 1 MHz can count time for 65 ms. Using the 100 Hz timebase and two counters (32-bits), you can count time for over a year.

If you need more control over when your elapsed timing begins and ends, use the Intermediate VIs instead of the Easy VIs. Refer to the Count Time-Int (9513) VI in the `examples\daq\counter\Am9513.llb` for an example of how to use Intermediate VIs when you need more control over when your elapsed timing begins and ends.

This example uses the Intermediate VIs Event or Time Counter Config, Counter Start, Counter Read, and Counter Stop. The Event or Time Counter Config VI configures **counter** to count the number of rising edges of a known internal timebase. The Counter Start VI begins the counting operation for **counter**. The Counter Read VI returns the count until you click the **STOP** button or an error occurs. The count value is divided by the **timebase** to determine the **elapsed time**. Finally, the Counter Stop VI stops the counter operation. You also can gate **counter** with a pulse to control when it starts and stops timing. To do this, wire your pulse to the GATE of **counter**, and choose the appropriate **gate mode** from the front panel menu. Additionally, you can cascade two counters by choosing **two counters (32-bits)** in the **number of counters to use** menu. This extends your elapsed time range. You must also wire the OUT of **counter** to the SOURCE of **counter+1** for this increased range.

8253/54

The Count Time (8253) VI, available in the `examples\daq\counter\8253.llb`, uses the ICTR Control VI, available on the **Functions»Data Acquisition»Counter»Intermediate Counter** palette.

This VI initiates the counter to count the number of rising edges of a TTL timebase at the CLK of **counter**. **Counter 0** creates the timebase. Looking at the block diagram, the Timebase Generator (8253) VI sets up **Counter 0** to generate a timebase by dividing down its internal timebase. The first call to ICTR Control loads the count register and sets up **counter**

to count down. Inside the While Loop, ICTR Control reads the count, which is divided by the actual timebase frequency to determine the **elapsed time**. The elapsed time increments until you click the **STOP** button or an error occurs. The last two calls to ICTR Control reset **Counter 0** and **counter**. Remember, you must externally wire the OUT of **Counter 0** to the CLK of **counter**. You also can gate **counter** with a pulse to control when it starts and stops timing. To do this, wire your pulse to the GATE of **counter**. Refer to the information found in the **Context Help** for a complete description of this example.

Dividing Frequencies

Dividing TTL frequencies is useful if you want to use an internal timebase and the frequency you need does not exist. You can divide an existing internal frequency to get what you need. You also can divide the frequency of an external TTL signal. Frequency division results in a pulse or pulse train from a counter for every *N* cycles of an internal or external source. Counters can only decrease (divide down) the frequency of the source signal. The resulting frequency is equal to the input frequency divided by *N* (timebase divisor). *N* must be an integer number greater than 1. Performing frequency division on an internal signal is called a *down counter*. Frequency division on an external signal is called a *signal divider*. Figure 10-29 shows typical wiring for frequency division.

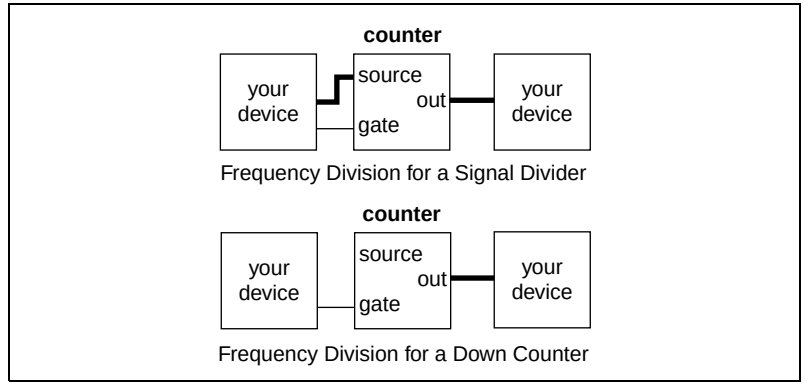


Figure 10-29. Wiring Your Counters for Frequency Division

TIO-ASIC or DAQ-STC

Open the Generate Pulse Train (DAQ-STC) and Generate Pulse Train (NI-TIO) VIs, available in the `examples\daq\counter` library and study the block diagrams.

Am9513

Figure 10-30 shows an example of a signal divider. It uses the Intermediate counter VIs Down Counter or Divide Config, Counter Start, and Counter Stop.

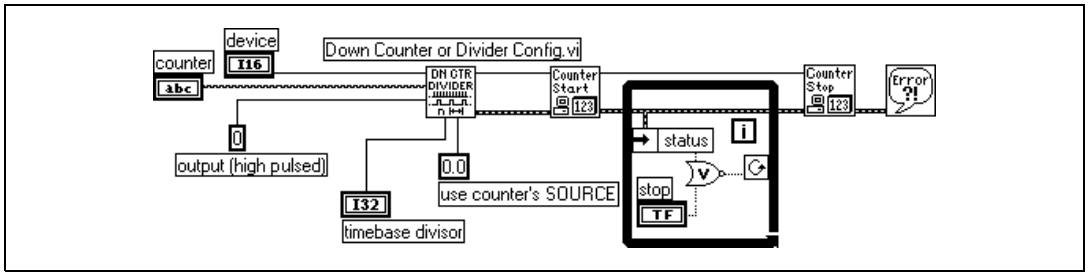


Figure 10-30. Programming a Single Divider for Frequency Division

The Down Counter or Divide Config VI configures the specified counter to divide the SOURCE signal by the **timebase divisor** value and output a signal when the counter reaches its terminal count (TC). Using Down Counter or Divide Config VI, you can configure the type of output to be pulsed or toggled. The block diagram in Figure 10-30 outputs a high pulse lasting one cycle of the source signal once the counter reaches its TC. The previous block diagram counts the rising edges of the SOURCE signal, the default value of the **source edge** input.

The Counter Start VI tells the counter to start counting the SOURCE signal edges. The counter stops the frequency division only when you click the **STOP** button. The Counter Stop VI stops the counter immediately and clears the count register. It is a good idea to always check your errors at the end of an operation to see if the operation was successful.

You can alter the Down Counter or Divide Config VI to create a down counter. To do this, change the timebase value from 0.0 (external SOURCE) to a frequency available on your counter. With the Am9513 chip, you can choose timebases of 1 MHz, 100 kHz, 10 kHz, 1 kHz, and 100 Hz. With the DAQ-STC chip, you can choose timebases of 20 MHz and 100 kHz.

Instead of triggering frequency division for signal dividers and down counters by software, as previously described, you can trigger using the GATE signal. You can trigger while the GATE signal is high, low, or on the rising or falling edge.

8253/54

To divide a frequency with the 8253/54 counter chip, use the example Cont Pulse Train (8253) VI, available in the `examples\daq\8253.llb`.

Part III

Measurement Analysis in LabVIEW

This part explains how to analyze your measurements in LabVIEW.

Part III, *Measurement Analysis in LabVIEW*, contains the following chapters:

- Chapter 11, *Introduction to Measurement Analysis in LabVIEW*, introduces digital signal processing and the LabVIEW Analysis VIs.
- Chapter 12, *DC/RMS Measurements*, describes how to use the two most common measurements of a signal.
- Chapter 13, *Frequency Analysis*, explains how to analyze dynamic signals using frequency analysis.
- Chapter 14, *Distortion Measurements*, explains harmonic distortion, THD, and SINAD.
- Chapter 15, *Limit Testing*, explains how to use limit testing, or mask testing, to monitor a waveform.
- Chapter 16, *Digital Filtering*, explains various types of filters you can use and how to decide which one to use.
- Chapter 17, *Signal Generation*, describes common test signals and how you can generate them.

11

Introduction to Measurement Analysis in LabVIEW

Digital signals are everywhere in the world around us. Telephone companies use digital signals to represent the human voice. Radio, TV, and hi-fi sound systems are all gradually converting to the digital domain because of its superior fidelity, noise reduction, and signal processing flexibility. Data is transmitted from satellites to earth ground stations in digital form. NASA's pictures of distant planets and outer space are often processed digitally to remove noise and extract useful information. Economic data, census results, and stock market prices are all available in digital form. Because of the many advantages of digital signal processing, analog signals are also converted to digital form before they are processed with a computer.

This chapter provides a background in basic digital signal processing and an introduction to the LabVIEW Measurement Analysis VIs.

The Importance of Data Analysis

The importance of integrating analysis libraries into engineering stations is that the raw data, as shown Figure 11-1, does not always immediately convey useful information. Often you must transform the signal, remove noise disturbances, correct for data corrupted by faulty equipment, or compensate for environmental effects, such as temperature and humidity.

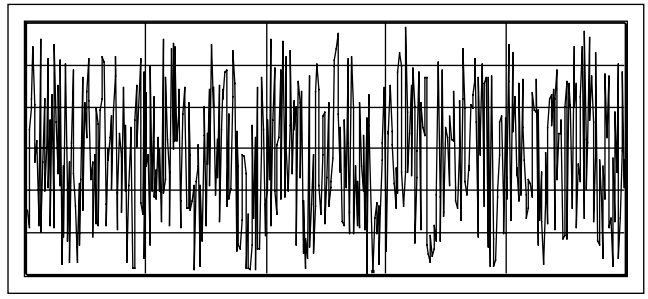


Figure 11-1. Raw Data

By analyzing and processing the digital data, you can extract the useful information from the noise and present it in a form more comprehensible than the raw data, as shown in Figure 11-2.

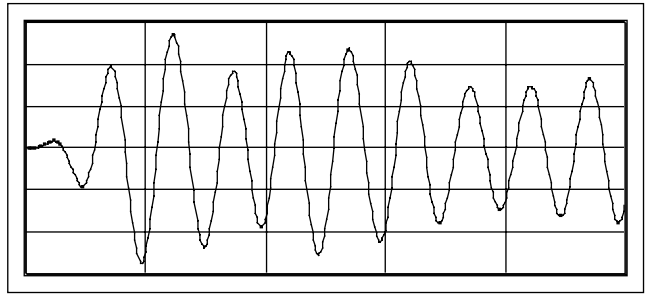


Figure 11-2. Processed Data

The LabVIEW block diagram programming approach and the extensive set of LabVIEW Measurement Analysis VIs simplify the development of analysis applications.

The LabVIEW Measurement Analysis VIs give you the most recent data analysis techniques using VIs that you can wire together. Instead of worrying about implementation details for analysis routines, as you do in conventional programming languages, you can concentrate on solving your data analysis problems.

Data Sampling

Sampling Signals

To use digital signal processing techniques, you must first convert an analog signal into its digital representation. In practice, this is implemented by using an analog-to-digital (A/D) converter. Consider an analog signal $x(t)$ that is sampled every Δt seconds. The time interval Δt is known as the sampling interval or sampling period. Its reciprocal, $1/\Delta t$, is known as the sampling frequency, with units of samples/second. Each of the discrete values of $x(t)$ at $t = 0$, Δt, $2\Delta t$, $3\Delta t$, etc., is known as a sample. Thus, $x(0)$, $x(\Delta t)$, $x(2\Delta t)$, ..., are all samples. The signal $x(t)$ can thus be represented by the discrete set of samples

$$\{x(0), x(\Delta t), x(2\Delta t), x(3\Delta t), \ldots, x(k\Delta t), \ldots\}$$

Figure 11-3 below shows an analog signal and its corresponding sampled version. The sampling interval is Δt. Observe that the samples are defined at discrete points in time.

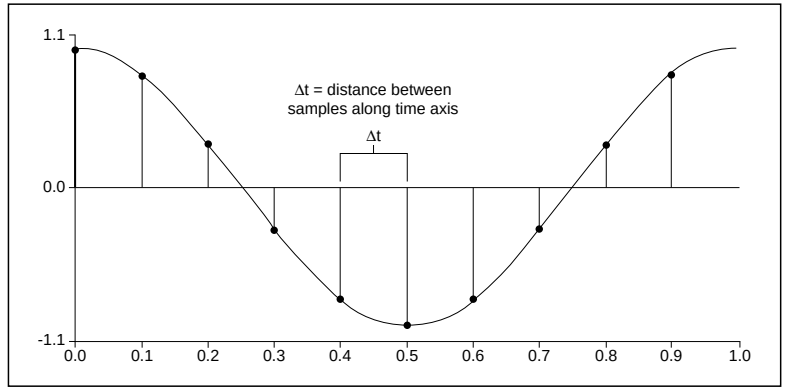


Figure 11-3. Analog Signal and Corresponding Sampled Version

The following notation represents the individual samples:

$$x[i] = x(i\Delta t)$$

for

$$i = 0, 1, 2, \ldots$$

If N samples are obtained from the signal $x(t)$, then $x(t)$ can be represented by the sequence

$$X = \{x[0], x[1], x[2], x[3], \ldots, x[N-1]\}$$

This is known as the digital representation or the sampled version of $x(t)$. Note that the sequence $X = \{x[i]\}$ is indexed on the integer variable i, and does not contain any information about the sampling rate. So by knowing just the values of the samples contained in X, you will have no idea of what the sample rate is.

Sampling Considerations

A/D converters (ADCs) are an integral part of National Instruments DAQ boards. One of the most important parameters of an analog input system is the rate at which the DAQ device samples an incoming signal. The sampling rate determines how often an analog-to-digital (A/D) conversion takes place. A fast sampling rate acquires more points in a given

time and can therefore often form a better representation of the original signal than a slow sampling rate. Sampling too slowly may result in a poor representation of your analog signal. Figure 11-4 shows an adequately sampled signal, as well as the effects of undersampling. The effect of undersampling is that the signal appears as if it has a different frequency than it truly does. This misrepresentation of a signal is called an alias.

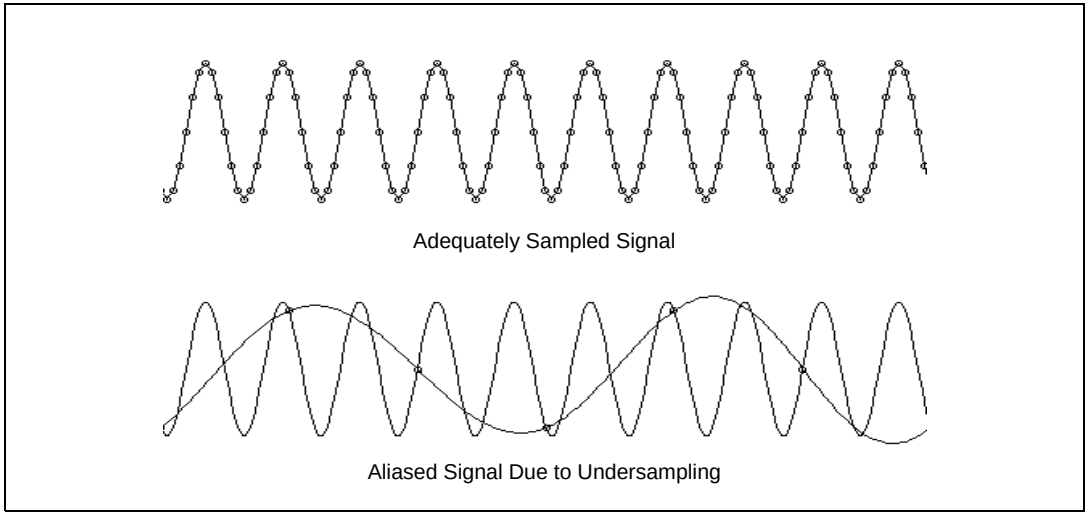


Figure 11-4. Aliasing Effects of an Improper Sampling Rate

According to Shannon's theorem, to avoid aliasing you must sample at a rate greater than twice the maximum frequency component in the signal you are acquiring. For a given sampling rate, the maximum frequency that can be represented accurately, without aliasing, is known as the **Nyquist frequency**. The Nyquist frequency is one half the sampling frequency. Signals with frequency components above the Nyquist frequency will appear aliased between DC and the Nyquist frequency. The alias frequency is the absolute value of the difference between the frequency of the input signal and the closest integer multiple of the sampling rate. Figures 11-5 and 11-6 illustrate this phenomenon. For example, assume f_s, the sampling frequency, is 100 Hz. Also, assume the input signal contains the following frequencies—25 Hz, 70 Hz, 160 Hz, and 510 Hz. These frequencies are shown in Figure 11-5.

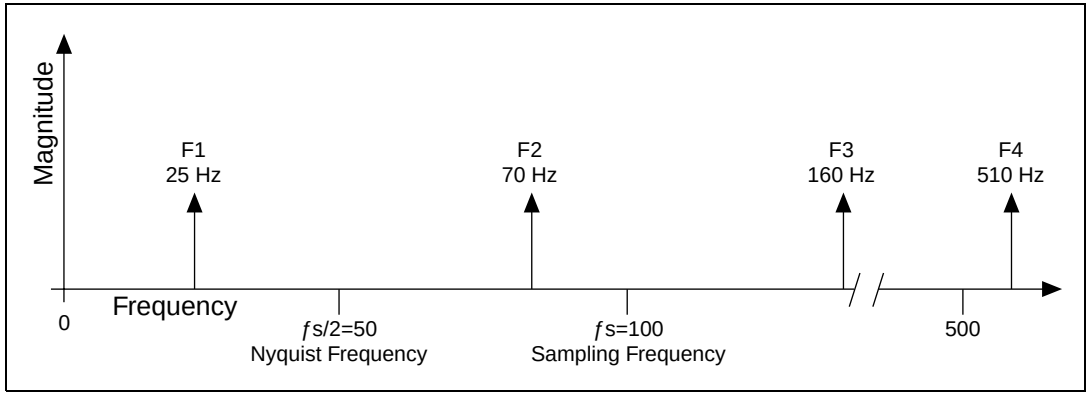


Figure 11-5. Actual Signal Frequency Components

In Figure 11-6, we see that frequencies below the Nyquist frequency (*f*s/2=50 Hz) are sampled correctly. Frequencies above the Nyquist frequency appear as aliases. For example, F1 (25 Hz) appears at the correct frequency, but F2 (70 Hz), F3 (160 Hz), and F4 (510 Hz) have aliases at 30 Hz, 40 Hz, and 10 Hz, respectively. To calculate the alias frequency, use the following equation:

Alias Freq. = ABS (Closest Integer Multiple of Sampling Freq. – Input Freq.)

where ABS means “the absolute value.” For example,

Alias F2 = |100 – 70| = 30 Hz
Alias F3 = |(2)100 – 160| = 40 Hz
Alias F4 = |(5)100 – 510| = 10 Hz

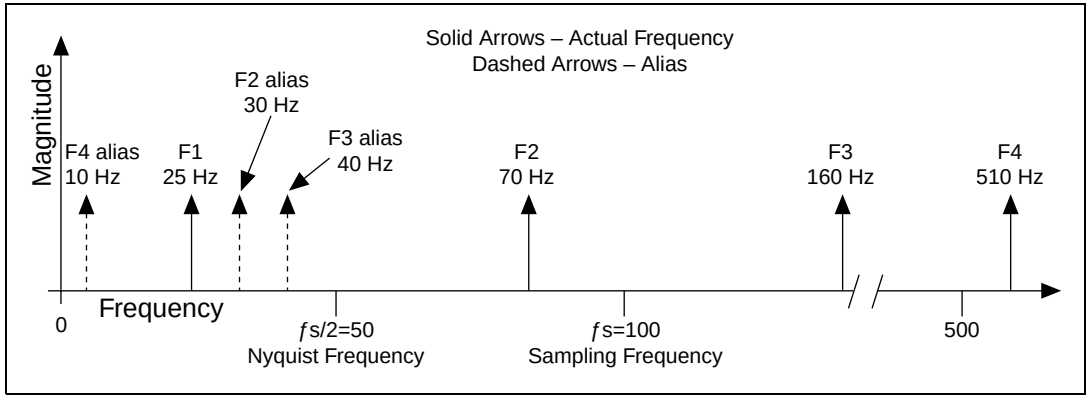


Figure 11-6. Signal Frequency Components and Aliases

A question often asked is, “How fast should I sample?” Your first thought may be to sample at the maximum rate available on your DAQ device. However, if you sample very fast over long periods of time, you may not have enough memory or hard disk space to hold the data. Figure 11-7 shows the effects of various sampling rates. In case a, the sine wave of frequency *f* is sampled at the same frequency f_s (samples/sec) = *f* (cycles/sec), or at 1 sample per cycle. The reconstructed waveform appears as an alias at DC. As you increase the sampling to 7 samples/4 cycles, as in case b, the waveform increases in frequency, but aliases to a frequency less than the original signal (3 cycles instead of 4). The sampling rate in case b is $f_s = 7/4\ f$. If you increase the sampling rate to $f_s = 2f$, the digitized waveform has the correct frequency (same number of cycles), and can be reconstructed as the original sinusoidal wave, as shown in case c. For time-domain processing, it may be important to increase your sampling rate so that the samples more closely represent the original signal. By increasing the sampling rate to well above *f*, say to $f_s=10f$, or 10 samples/cycle, you can accurately reproduce the waveform, as shown in case d.

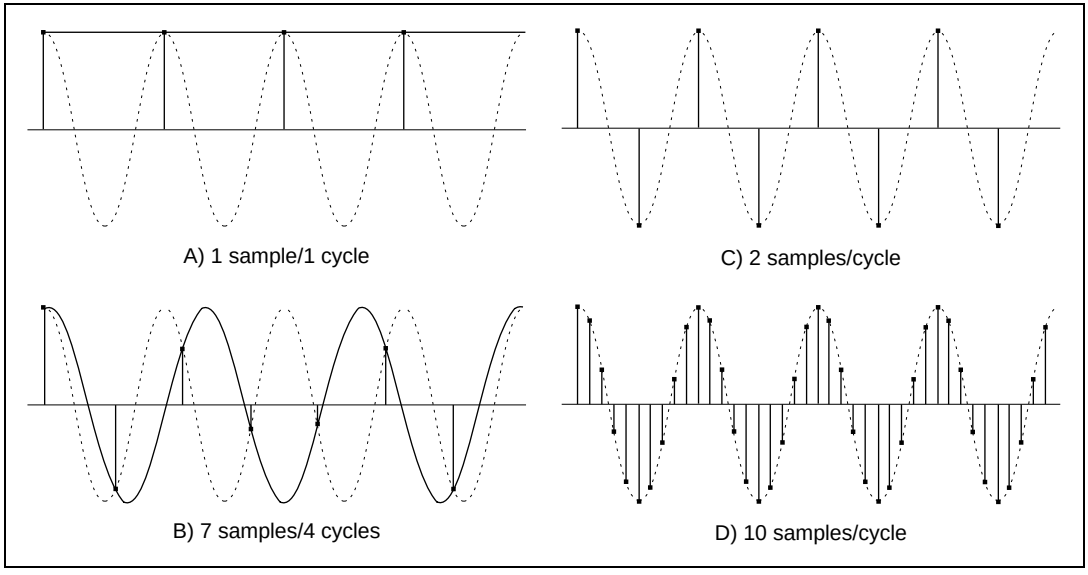


Figure 11-7. Effects of Sampling at Different Rates

Why Do You Need Anti-Aliasing Filters?

We have seen that the sampling rate should be at least twice the maximum frequency of the signal that we are sampling. In other words, the maximum frequency of the input signal should be less than or equal to half of the sampling rate. But how do you ensure that this is definitely the case in

practice? Even if you are sure that the signal being measured has an upper limit on its frequency, pickup from stray signals (such as the powerline frequency or from local radio stations) could contain frequencies higher than the Nyquist frequency. These frequencies may then alias into the desired frequency range and thus give us erroneous results.

To be completely sure that the frequency content of the input signal is limited, a low pass filter (a filter that passes low frequencies but attenuates the high frequencies) is added before the sampler and the ADC. This filter is called an anti-alias filter because by attenuating the higher frequencies (greater than Nyquist), it prevents the aliasing components from being sampled. Because at this stage (before the sampler and the ADC) we are still in the analog world, the anti-aliasing filter is an analog filter.

An ideal anti-alias filter is as shown in Figure 11-8 (a).

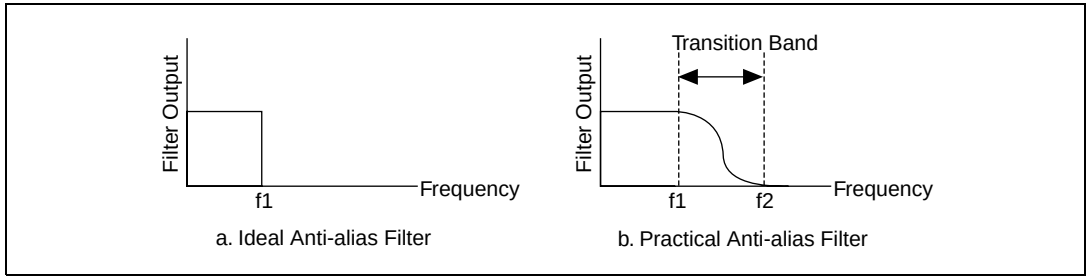


Figure 11-8. Ideal versus Practical Anti-Alias Filter

It passes all the desired input frequencies (below f_1) and cuts off all the undesired frequencies (above f_1). However, such a filter is not physically realizable. In practice, filters look as shown in figure (b) above. They pass all frequencies < f_1, and cut-off all frequencies > f_2. The region between f_1 and f_2 is known as the transition band, which contains a gradual attenuation of the input frequencies. Although you want to pass only signals with frequencies < f_1, those signals in the transition band could still cause aliasing. Therefore, in practice, the sampling frequency should be greater than two times the highest frequency in the transition band. So, this turns out to be more than two times the maximum input frequency (f_1). That is one reason why you may see that the sampling rate is more than twice the maximum input frequency.

Why Use Decibels?

On some instruments, you will see the option of displaying the amplitude in a linear or decibel (dB) scale. The linear scale shows the amplitudes as they are, whereas the decibel scale is a transformation of the linear scale

into a logarithmic scale. We will now see why this transformation is necessary.

Suppose that you want to display a signal with very large as well as very small amplitudes. Let us assume you have a display of height 10 cm, and will utilize the entire height of the display for the largest amplitude. So, if the largest amplitude in the signal is 100 V, a height of 1 cm of the display corresponds to 10 V. If the smallest amplitude of the signal is 0.1 V, this corresponds to a height of only 0.1 mm. This will barely be visible on the display.

To see all the amplitudes, from the largest to the smallest, you need to change the amplitude scale. Alexander Graham Bell invented a unit, the Bell, which is logarithmic, compressing large amplitudes and expanding the small amplitudes. However, the Bell was too big of a unit, so commonly the decibel (1/10th of a Bell) is used. The decibel (dB) is defined as

$$\text{one dB} = 10 \log_{10} (\text{Power Ratio}) = 20 \log_{10} (\text{Voltage Ratio})$$

The Table 11-1 shows the relationship between the decibel and the Power and Voltage Ratios.

Table 11-1. Decibels and Power and Voltage Ratio Relationship

dB	Power Ratio	Voltage Ratio
+40	10000	100
+20	100	10
+6	4	2
+3	2	1.4
0	1	1
–3	1/2	1/1.4
–6	1/4	1/2
–20	1/100	1/10
–40	1/10000	1/100

Thus, you see that the dB scale is useful in compressing a wide range of amplitudes into a small set of numbers.

12

DC/RMS Measurements

Two of the most common measurements of a signal are its direct current (DC) and root mean square (RMS) levels. This chapter introduces measurement analysis techniques for making DC and RMS measurements of a signal.

What Is the DC Level of a Signal?

You can use DC measurements to define the value of a static or slowly varying signal. DC measurements can be both positive and negative. The DC value usually is constant within a specific time window. You can track and plot slowly moving values, such as temperature, as a function of time using a DC meter. In that case, the observation time that results in the measured value has to be short compared to the speed of change for the signal. Figure 12-1 illustrates an example DC level of a signal.

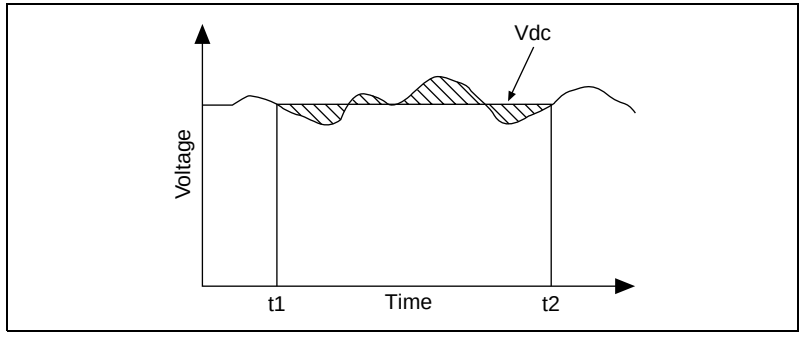


Figure 12-1. DC Level of a Signal

The DC level of a continuous signal *V(t)* from time *t1* to time *t2* is given by the equation:

$$V_{dc} = \frac{1}{(t2 - t1)} \cdot \int_{t1}^{t2} V(t)dt$$

where *t2 – t1* represents the integration time or measurement time.

For digitized signals, the discrete-time version of the previous equation is given by:

$$V_{dc} = \frac{1}{N} \cdot \sum_{i=1}^{N} V_i$$

For a sampled system, the DC value is defined as the mean value of the samples acquired in the specified measurement time window.

Between pure DC signals and fast-moving dynamic signals is a “gray zone” where signals become more complex, and measuring the DC level of these signals becomes quite challenging.

Real world signals often contain a significant amount of dynamic influence. Often, you do not want the dynamic part of the signal. The DC measurement identifies the static DC signal hidden in the dynamic signal, for example, the voltage generated by a thermocouple in an industrial environment, where external noise or hum from the main power can disturb the DC signal significantly.

What Is the RMS Level of a Signal?

The RMS of a signal is the square root of the mean value of the squared signal. RMS measurements are always positive. Use RMS measurements when a representation of energy is needed. You usually acquire RMS measurements on dynamic signals (signals with relatively fast changes) like noise or periodic signals. Refer to the *How to Measure AC Voltage* section in Chapter 4, *Example Measurements*, for more information about when to use RMS measurements.

The RMS level of a continuous signal *V(t)* from time *t1* to time *t2* is given by the equation:

$$V_{rms} = \sqrt{\frac{1}{(t2 - t1)} \cdot \int_{t1}^{t2} V^2(t)dt}$$

where *t2 – t1* represents the integration time or measurement time.

The RMS level of a discrete signal V_i is given by the equation:

$$V_{rms} = \sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} V_i^2}$$

One difficulty is encountered when measuring the dynamic part of a signal using an instrument that does not offer an AC-coupling option. A true RMS measurement includes the DC part in the measurement, a measurement you might not want.

Averaging to Improve the Measurement

Instantaneous DC measurements of a noisy signal can vary randomly and significantly, as shown in Figure 12-2. You can measure a more accurate value by averaging out the noise that is superimposed on the desired DC level. In a continuous signal, the averaged value between two times, *t1* and *t2*, is defined as the signal integration between *t1* and *t2*, divided by the measurement time, *t2 – t1*, as shown in Figure 12-1. The area between the averaged value V_{dc} and the signal that is above V_{dc} is equal to the area between V_{dc} and the signal that is under V_{dc}. For a sampled signal, the average value is the sum of the voltage samples divided by the measurement time in samples, or the mean value of the measurement samples. Refer to the *Averaging a Scan Example* section in Chapter 4, *Example Measurements*, for more information about averaging in LabVIEW.

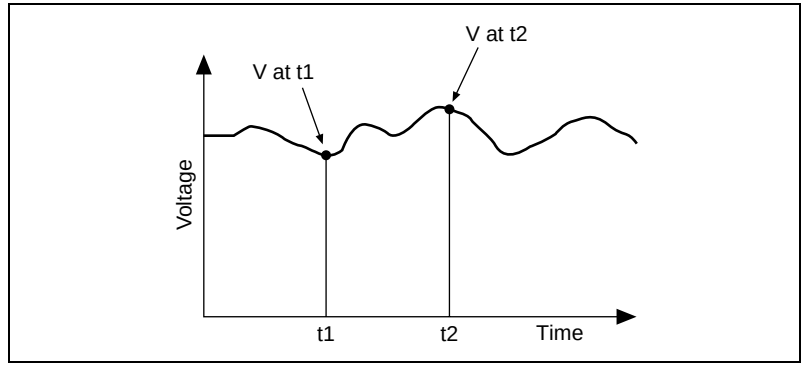


Figure 12-2. Instantaneous DC Measurements

An RMS measurement is an averaged quantity, because it is the average energy in the signal over a measurement period. You can improve the RMS measurement accuracy by using a longer averaging time, equivalent to the integration time or measurement time.

There are several different strategies to use for making DC and RMS measurements, each dependent on the type of error or noise sources. When choosing a strategy, you must decide if accuracy or speed of the measurement is more important.

Common Error Sources Affecting DC and RMS Measurements

Some common error sources for DC measurements are single frequency components (or tones), multiple tones, or random noise. These same error signals can interfere with RMS measurements, so in many cases the approach taken to improve RMS measurements is the same as for DC measurements.

DC Overlapped with Single Tone

Consider the case where the signal you measure is composed of a DC signal and a single sine tone. The average of a single period of the sine tone is ideally zero, because the positive half-period of the tone cancels the negative half-period.

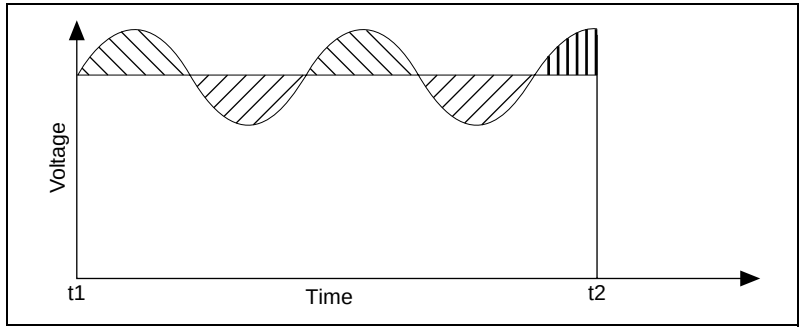


Figure 12-3. DC Signal Overlapped with Single Tone

Any remaining partial period, shown in Figure 12-3 with vertical hatching, introduces an error in the average value and, therefore, in the DC measurement. Increasing the averaging time reduces this error, because the integration is always divided by the measurement time *t2 – t1*. If you know

the period of the sine tone, you can take a more accurate measurement of the DC value by using a measurement period equal to an integer number of periods of the sine tone. The most severe error occurs when the measurement time is a half-period different from an integer number of periods of the sine tone, because this is the maximum area under or over the signal curve.

Defining the Equivalent Number of Digits

Defining the Equivalent Number of Digits (ENOD) makes it easier to relate a measurement error to a number of digits, similar to digits of precision. ENOD translates measurement accuracy into a number of digits.

$$\text{ENOD} = \log_{10}(\text{Relative Error})$$

A 1 percent error corresponds to 2 digits of accuracy, and a 1 part per million error corresponds to 6 digits of accuracy ($\log_{10}(0.000001) = 6$).

ENOD is only an approximation that tells you what order of magnitude of accuracy you can achieve under specific measurement conditions. This accuracy does not take into account any error introduced by the measurement instrument or data acquisition hardware itself. ENOD only gives you a tool for computing the reliability of a specific measurement technique.

Thus, the ENOD should at least match the accuracy of the measurement instrument or measurement requirements. For example, it is not necessary to use a measurement technique with an ENOD of 6 digits if your instrument has an accuracy of only 0.1 percent (3 digits). Similarly, you do not get the six digits of accuracy from your 6-digit accurate measurement instrument if your measurement technique is limited to an ENOD of only 3 digits.

DC Plus Sine Tone

Figure 12-4 shows that for a 1.0 VDC signal overlapped with a 0.5 V single sine tone, the worst ENOD increases with measurement time, x-axis shown in periods of the additive sine tone, at a rate of approximately 1 additional digit for 10 times more measurement time. To achieve 10 times more accuracy, you need to increase your measurement time by a factor of 10. In other words, accuracy and measurement time are related through a first-order function.

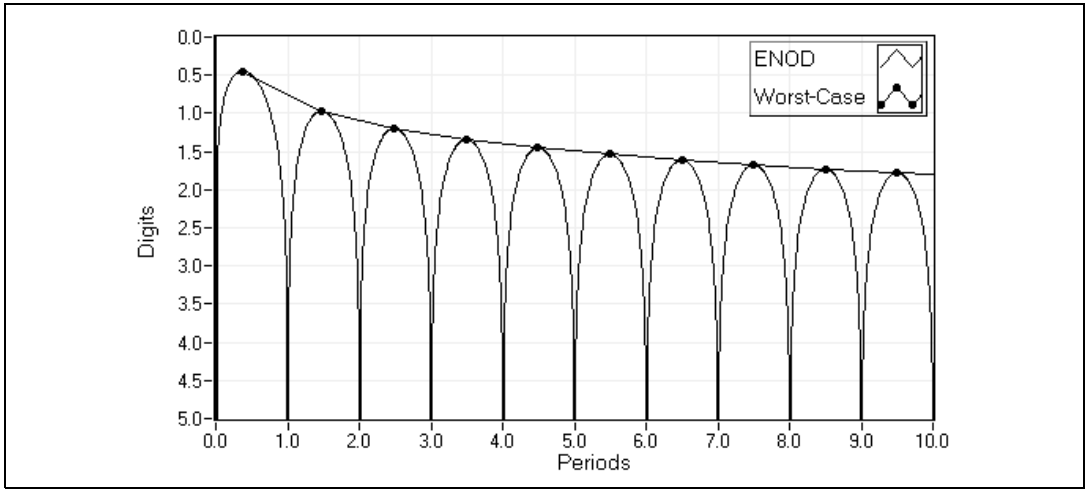


Figure 12-4. Digits vs Measurement Time for 1 VDC Signal with 0.5 Single Tone

Windowing to Improve DC Measurements

The worst ENOD for a DC signal plus a sine tone occurs when the measurement time is at half-periods of the sine tone. You can greatly reduce these errors due to non-integer number of cycles by using a weighting function before integrating to measure the desired DC value. The most common weighting or window function is the Hann window, commonly known as the Hanning window.

Figure 12-5 shows a dramatic increase in accuracy from the use of the Hann window. The accuracy as a function of the number of sine tone periods is improved from a first-order function to a third-order function. In other words, you can achieve 1 additional digit of accuracy for every $10^{1/3} = 2.15$ times more measurement time using the Hann window instead of 1 digit for every 10 times more measurement time without using a window. As in the non-windowing case, the DC level is 1.0 V and the single tone peak amplitude is 0.5 V.

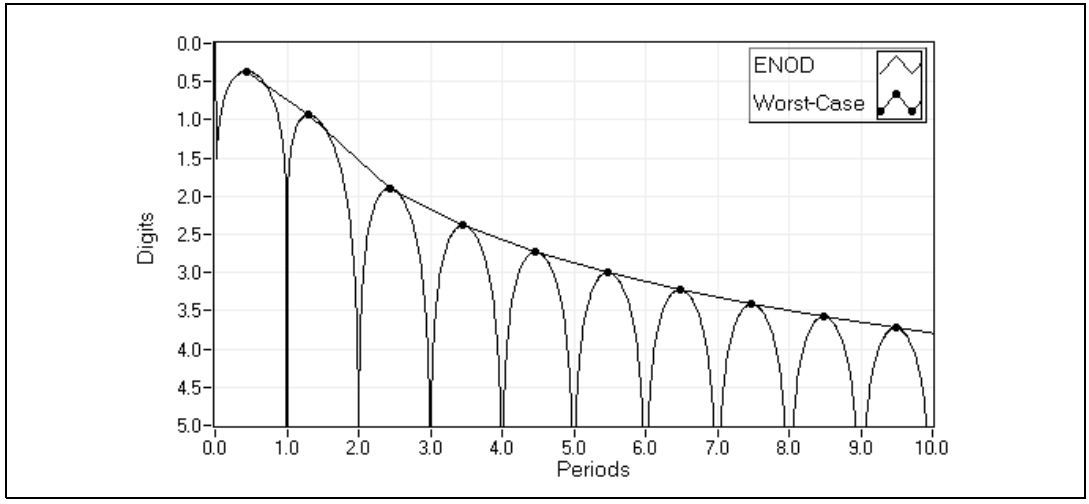


Figure 12-5. Digits vs Measurement Time for DC+Tone Using Hann Window

You can use other types of window functions to further reduce the necessary measurement time or greatly increase the resulting accuracy. Figure 12-6 shows that the Low Sidelobe (LSL) window can achieve more than six ENOD of worst accuracy when averaging your DC signal over only five periods of the sine tone (same test signal).

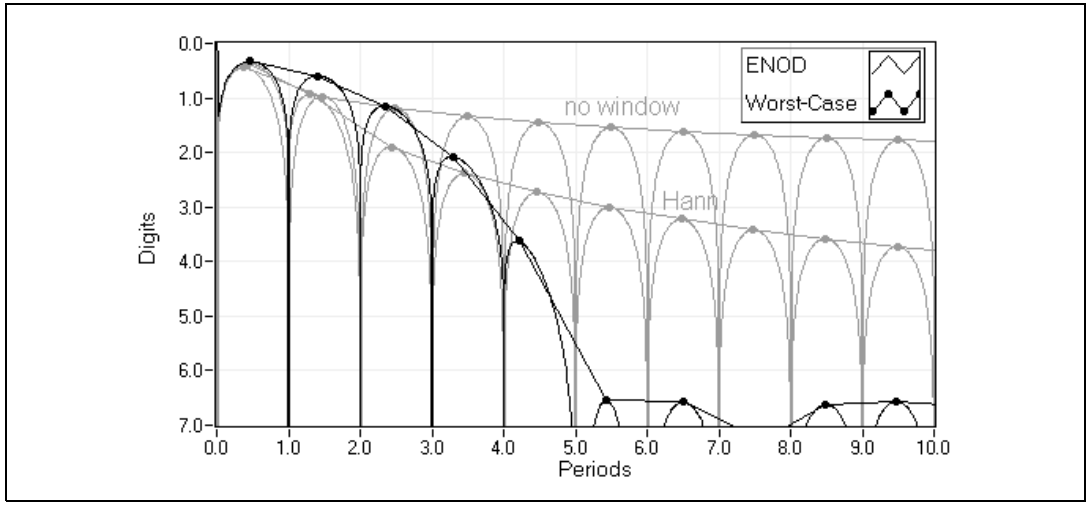


Figure 12-6. Digits vs Measurement Time for DC+Tone Using LSL Window

RMS Measurements Using Windows

Like DC measurements, the worst ENOD for measuring the RMS level of signals sometimes can be significantly improved by applying a window to the signal before RMS integration. For example, if you measure the RMS level of the DC signal plus a single sine tone, the most accurate measurements are made when the measurement time is an integer number of periods of the sine tone. Figure 12-7 shows that the worst ENOD varies with measurement time (in periods of the sine tone) for various window functions. Here, the test signal contains 0.707 VDC with 1.0 V peak sine tone.

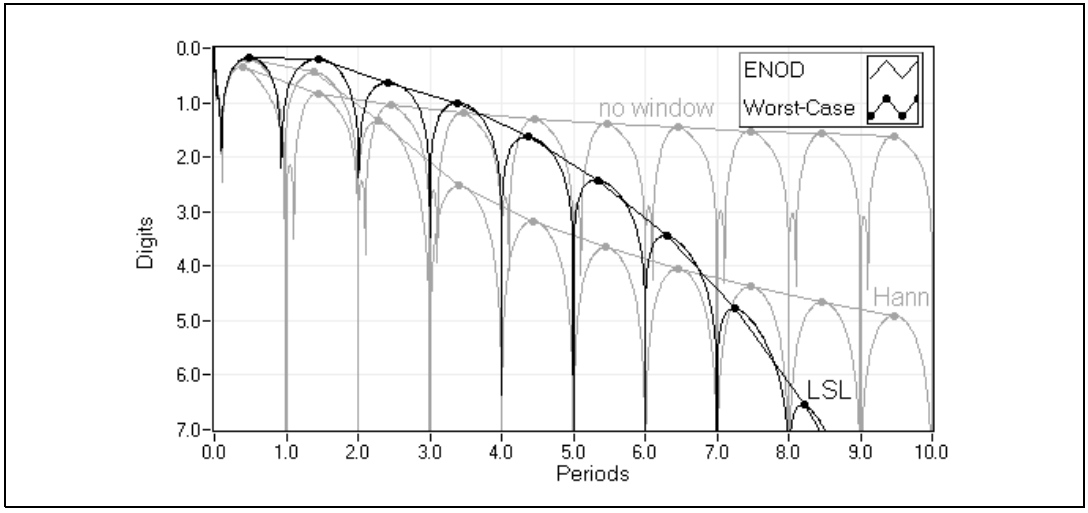


Figure 12-7. Digits vs Measurement Time for RMS Measurements

Applying the window to the signal increases RMS measurement accuracy significantly, but the improvement is not as large as in DC measurements. For this example, the LSL window achieves six digits of accuracy when the measurement time reaches eight periods of the sine tone.

Using Windows with Care

Window functions can be very useful to improve the speed of your measurement, but you must be careful. The Hann window is a general window recommended in most cases. Use more advanced windows like the LSL window only if you know enough about your signal that you are sure the window is not doing more damage than good. For example, you can significantly reduce RMS measurement accuracy if the signal you want to

measure is composed of many frequency components close to each other in the frequency domain.

You also must make sure that the window is scaled correctly or that you update scaling after applying the window. The most useful window functions are pre-scaled by their *coherent gain*—the mean value of the window function—so that the resulting mean value of the scaled window function is always 1.00. DC measurements do not need to be scaled when using a properly scaled window function. For RMS measurements, each window has a specific *equivalent noise bandwidth* that you must use to scale integrated RMS measurements. You must scale RMS measurements using windows by the reciprocal of the square root of the equivalent noise bandwidth.

Rules for Improving DC and RMS Measurements

Use the following guidelines when determining a strategy for improving your DC and RMS measurements:

- If your signal is overlapped with a single tone, longer integration times increase the accuracy of your measurement. If you know the exact frequency of the sine tone, use a measurement time that corresponds to an exact number of sine periods. If you do not know the frequency of the sine tone, apply a window, such as a Hann window, to significantly reduce the measurement time needed to achieve a specific accuracy.
- If your signal is overlapped with many independent tones, increasing measurement time increases the accuracy of the measurement. As in the single tone case, using a window significantly reduces the measurement time needed to achieve a specific accuracy.
- If your signal is overlapped with noise, do not use a window. In this case, you can increase the accuracy of your measurement by increasing the integration time or by pre-processing or conditioning your noisy signal with a lowpass (or bandstop) filter.

RMS Levels of Specific Tones

You can always improve the accuracy of an RMS measurement by choosing a specific measurement time (to contain an integer number of cycles of your sine tones) or by using a window function. The measurement of the RMS value is based only on the time domain knowledge of your signal. You can use advanced techniques when you are interested in a specific frequency or narrow frequency range.

You can use bandpass or bandstop filtering before RMS computations to measure the RMS power in a specific band of frequencies. You also can use the Fast Fourier Transform (FFT) to pick out specific frequencies for RMS processing. Refer to Chapter 13, *Frequency Analysis*, for more information about the FFT.

The RMS level of a specific sine tone that is part of a complex or noisy signal can be extracted very accurately using frequency domain processing, leveraging the power of the FFT, and utilizing the benefits of windowing.

13

Frequency Analysis

Frequency analysis is a general-purpose tool used for a wide variety of applications dealing with dynamic signals, including electrical and mechanical engineering, sound and vibration measurements, production testing, and biomedical applications.

Frequency vs. Time Domain

Fourier's theorem states that any waveform in the time domain can be represented by the weighted sum of sines and cosines. The same waveform can then be represented in the frequency domain as a pair of amplitude and phase values at each component frequency.

You can generate any waveform by adding up sine waves, each with a particular amplitude and phase. Figure 13-1 shows the original waveform, labeled *sum*, and its component frequencies. The fundamental frequency is shown at the frequency *f0*, the second harmonic at frequency *2f0*, and the third harmonic at frequency *3f0*.

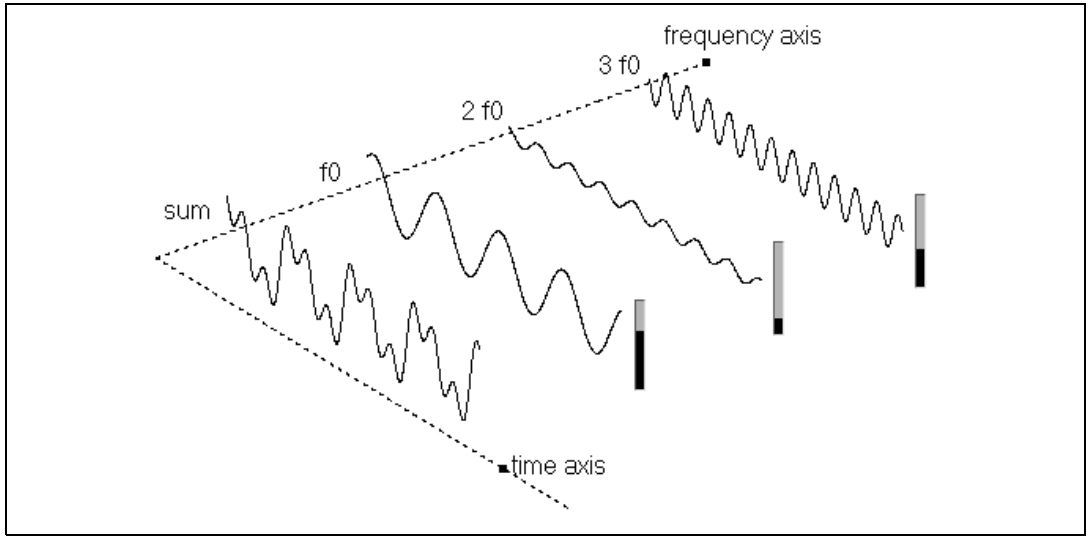


Figure 13-1. Signal Formed by Adding Three Frequency Components

In the frequency domain, you can conceptually separate the sine waves that add to form the complex time-domain signal. Figure 13-1 shows single frequency components, which spread out in the time domain, as distinct impulses in the frequency domain. The amplitude of each frequency line is the amplitude of that frequency component's time waveform.

Some measurements, such as harmonic distortion, are very difficult to quantify by inspecting the time waveform on an oscilloscope. When the same signal is displayed in the frequency domain by an FFT Analyzer, also known as a Dynamic Signal Analyzer, you easily can measure the harmonic frequencies and amplitudes.

Aliasing

According to Shannon's sampling theorem, the highest frequency (Nyquist frequency: f_N) that can be analyzed is $f_N = f_s/2$, where f_s is the sampling frequency. Any analog frequency greater than f_N after sampling appears as a frequency between 0 and f_N. Such a frequency is known as an alias frequency. In the digital (sampled) domain, there is no way to distinguish these alias frequencies from the frequencies that actually lie between 0 and f_N. Therefore these alias frequencies need to be removed from the analog signal before sampling by the A/D converter.

In order to remove these components present at frequencies higher than the Nyquist frequency, you must use an analog lowpass filter. The anti-aliasing analog lowpass filter should exhibit a flat passband frequency response with a good high-frequency alias rejection and a fast roll-off in the transition band.

FFT Fundamentals

The Fast Fourier Transform (FFT) is a fast version of the Discrete Fourier Transform (DFT). The DFT transforms digital time domain signals into the digital frequency domain.

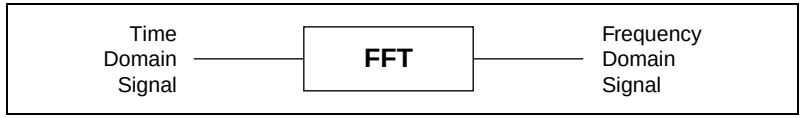


Figure 13-2. FFT Transforms Time-Domain Signals into the Frequency Domain

Each frequency component is the result of a dot product of the time domain signal with the complex exponential at that frequency and is given by the equation:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\left(\frac{2\pi nk}{N}\right)}$$

$$= \sum_{n=0}^{N-1} x(n)\left[\cos\left(\frac{2\pi nk}{N}\right) - j\sin\left(\frac{2\pi nk}{N}\right)\right]$$

The DC component is the dot product of *x(n)* with [cos(0) – *j*sin(0)], or with 1.0.

The first bin, or frequency component, is the dot product of *x(n)* with cos(2π*n*/*N*) – *j*sin(2π*n*/*N*). Here, cos(2π*n*/*N*) is a single cycle of the cosine wave, and sin(2π*n*/*N*) is a single cycle of a sine wave.

In general, bin *k* is the dot product of *x(n)* with *k* cycles of the cosine wave for the real part of *X(k)* and the sine wave for the imaginary part of *X(k)*.

The use of the FFT for frequency analysis implies two important relationships.

The first relationship links the highest frequency that can be analyzed to the sampling frequency and is given by the equation:

$$F_{max} = \frac{f_s}{2}$$

where F_{max} is the highest frequency that can be analyzed, and f_s is the sampling frequency.

Refer to the *Aliasing* section in this chapter for more information about F_{max}.

The second relationship links the frequency resolution to the total acquisition time, which is related to the sampling frequency and the block size of the FFT and is given by the equation:

$$\Delta f = \frac{1}{T} = \frac{f_s}{N}$$

where Δf is the frequency resolution, T is the acquisition time, f_s is the sampling frequency, and N is the block size of the FFT.

Fast FFT Sizes

Direct implementation of the DFT on N data samples requires approximately N^2 complex operations and, therefore, is a time-consuming process. However, when the size of the sequence is a power of 2,

$$N = 2^m \text{ for } m = 1, 2, 3,\ldots$$

a fast algorithm can compute the DFT with approximately $N \log_2(N)$ operations. This makes the calculation of the DFT much faster. This algorithm can compute the FFT in place, so it is highly memory efficient. Examples of sequence sizes where you can use this algorithm are 512, 1024, and 2048.

In addition, another optimized algorithm is used for short DFTs of lengths 2, 3, 4, 5, 8, and 10. As a result, when the size of the sequence is not a power of 2, but can be factored as

$$N = 2^m 3^k 5^j \text{ for } m, k, j = 0, 1, 2, 3,\ldots,$$

the DFT can be computed with speeds comparable to the radix-2 FFT, but requires more memory. It can be used for sequence sizes such as 640, 480, 1000, and 2000.

When the sequence size cannot be factored into sizes that are in the set of short DFTs, a Chirp-Z implementation of the DFT is used. This is much faster than the direct evaluation of the DFT expression. This algorithm uses more memory than the prime-factor algorithms, because it must allocate additional buffers for storing intermediate results during processing.

Magnitude and Phase

The FFT spectrum output produces complex numbers. In other words, every frequency component has a magnitude and phase. The phase is relative to the start of the time record or relative to a single-cycle cosine wave starting at the beginning of the time record. Single-channel phase measurements are stable only if the input signal is triggered. Dual-channel phase measurements compute phase differences between channels so that if the channels are simultaneously sampled, triggering usually is not necessary.

Normally the magnitude of the spectrum is displayed. The magnitude is the square root of the sum of the squares of the real and imaginary parts.

The phase is the arctangent of the ratio of the imaginary and real parts, and is usually between π and $-\pi$ radians (between 180 and –180 degrees).

Windowing

In practical applications, you obtain only a finite number of samples of the signal. The FFT assumes that this time record repeats. If you have an integral number of cycles in your time record, the repetition is smooth at the boundaries. However, in practical applications, you usually have a nonintegral number of cycles. In such cases the repetition results in discontinuities at the boundaries, as shown in Figure 13-3. These artificial discontinuities were not originally present in your signal and result in a smearing or leakage of energy from your actual frequency to all other frequencies. This phenomenon is known as spectral leakage. The amount of leakage depends on the amplitude of the discontinuity, a larger one causing more leakage.

A signal that is exactly periodic in the time record is composed of sine waves with exact integral cycles within the time record. Such a perfectly periodic signal has a spectrum with energy contained in exact frequency bins.

A signal that is not periodic in the time record has a spectrum with energy split or spread across multiple frequency bins. The FFT spectrum models the time domain as if the time record repeated itself forever. It assumes that the analyzed record is just one period of an infinitely-repeating periodic signal.

Because the amount of leakage is dependent on the amplitude of the discontinuity at the boundaries, you can use windowing to reduce the size of the discontinuity and hence reduce spectral leakage. Windowing consists of multiplying the time-domain signal by another time-domain waveform, known as a window, whose amplitude tapers gradually and smoothly towards zero at edges. The result is a windowed signal with very small or no discontinuities, and therefore reduced spectral leakage. There are many different types of windows. The one you choose depends on your application.

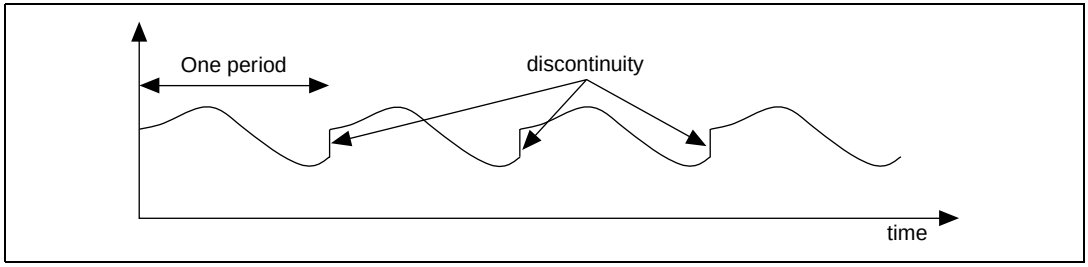


Figure 13-3. Periodic Waveform Created from Sampled Period

Some common window functions are uniform (no window or rectangular), Hanning, Hamming, Flat Top, Blackman-Harris, Force, and Exponential. Choosing the correct window requires some prior knowledge of the signal that you are analyzing. Table 13-1 shows different types of signals and the window choices that you can use with them.

Table 13-1. Signals and Window Choices

Type of Signal	Window
Transients whose duration is shorter than the length of the window	Rectangular
Transients whose duration is longer than the length of the window	Exponential, Hanning
General-purpose applications	Hanning
System analysis (frequency response measurements)	Hanning (for random excitation), Rectangular (for pseudo-random excitation)
Separation of two tones with frequencies very close to each other, but with widely differing amplitudes	Kaiser-Bessel
Separation of two tones with frequencies very close to each other, but with almost equal amplitudes	Rectangular
Accurate single tone amplitude measurements	Flat Top

Averaging to Improve the Measurement

Averaging successive measurements usually improves measurement accuracy. Averaging is usually performed on measurement results or on individual spectra, but not directly on the time record.

Common averaging modes include the following:

- RMS averaging
- Vector averaging
- Peak hold

RMS averaging reduces signal fluctuations but not the noise floor. The noise floor is not reduced because RMS averaging averages the energy, or power, of the signal. RMS averaging also causes averaged RMS quantities of single-channel measurements to have zero phase. RMS averaging for dual-channel measurements preserves important phase information.

Vector averaging eliminates noise from synchronous signals. Vector averaging computes the average of complex quantities directly. The real part is averaged separately from the imaginary part. This can reduce the noise floor for random signals, because they are not phase-coherent from one time record to the next. The real and imaginary parts are averaged separately, reducing noise but usually requiring a trigger.

Peak hold averaging retains the peak levels of the averaged quantities. Peak hold is performed at each frequency line separately, retaining peak levels from one FFT record to the next.

Equations for Averaging

Averaged measurements are computed according to the following equations.

RMS Averaging

FFT Spectrum	$\sqrt{\langle X^* \bullet X \rangle}$
power spectrum	$\langle X^* \bullet X \rangle$
cross spectrum	$\langle X^* \bullet Y \rangle$
frequency response	$\frac{\langle X^* \bullet Y \rangle}{\langle X^* \bullet X \rangle}$ (H1)

$$\left\langle \frac{Y^* \bullet Y}{Y^* \bullet X} \right\rangle \text{ (H2)}$$

$$H3 = \frac{(H1 + H2)}{2}$$

where X is the complex FFT of signal x (stimulus),

Y is the complex FFT of signal y (response),

X^* is the complex conjugate of X,

Y^* is the complex conjugate of Y, and

$\langle X \rangle$ is the average of X, real and imaginary parts being averaged separately.

Vector Averaging

FFT Spectrum	$\langle X \rangle$
power spectrum	$\langle X^* \rangle \bullet \langle X \rangle$
cross spectrum	$\langle X^* \rangle \bullet \langle Y \rangle$
frequency response	$\frac{\langle Y \rangle}{\langle X \rangle}$ (H1 = H2 = H3)

where X is the complex FFT of signal x (stimulus),

Y is the complex FFT of signal y (response),

X^* is the complex conjugate of X, and

$\langle X \rangle$ is the average of X, real and imaginary parts being averaged separately.

Peak Hold

FFT spectrum	$MAX\sqrt{(X^* \bullet X)}$
power spectrum	$MAX(X^* \bullet X)$

where X is the complex FFT of signal x (stimulus), and

X^* is the complex conjugate of X.

When performing RMS or vector averaging, each new spectral record can be weighted using either linear or exponential weighting.

Linear weighting combines N spectral records with equal weighting. When the number of averages has been completed, the analyzer stops averaging and presents the averaged results.

Exponential weighting emphasizes new spectral data more than old and is a continuous process.

Weighting is applied according to the following equation:

$$Y_i = \frac{N-1}{N} Y_{i-1} + \frac{1}{N} X_i$$

where X_i is the result of the analysis performed on the i^{th} block, and

Y_i is the result of the averaging process from X_1 to X_i,

$N = i$ for linear weighting, and

N is a constant for exponential weighting ($N = 1$ for $i = 1$).

Single-Channel Measurements—FFT, Power Spectrum

The FFT of a real signal returns a complex output, having a real and an imaginary part. The power in each frequency component represented by the FFT is obtained by squaring the magnitude of that frequency component. Because of this, the power spectrum is always real and all the phase information is lost. If you want phase information, you must use the FFT, which gives you a complex output.

You can use the power spectrum in applications where phase information is not necessary, for example, to calculate the harmonic power in a signal. You can apply a sinusoidal input to a nonlinear system and see the power in the harmonics at the system output.

Dual-Channel Measurements—Frequency Response

When analyzing two simultaneously sampled channels, you usually want to know the differences between the two channels rather than the properties of each.

In a typical dual-channel analyzer, as shown in Figure 13-4, the instantaneous spectrum is computed using a window function and the FFT for each channel. The averaged FFT spectrum, auto power spectrum, and cross power spectrum are computed and used in estimating the frequency response function. You also can use the coherence function to check the validity of the frequency response function.

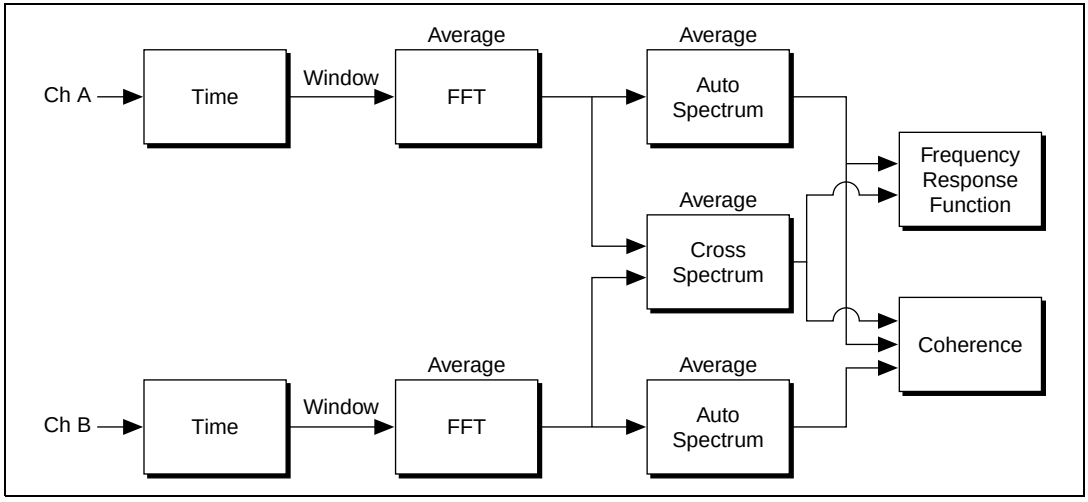


Figure 13-4. Dual-Channel Frequency Analysis

The frequency response of a system is described by the magnitude, |H|, and phase, ∠H, at each frequency. The gain of the system is the same as its magnitude and is the ratio of the output magnitude to the input magnitude at each frequency. The phase of the system is the difference of the output phase and input phase at each frequency.

14

Distortion Measurements

This chapter defines harmonic distortion, THD, and SINAD, and explains when to use distortion measurements.

What Is Distortion?

When a pure single-frequency sine wave is applied to a perfectly linear system, it produces an output that has the same frequency as that of the input sine wave, but with possible changes in the amplitude and/or phase. This also is true when a composite signal consisting of several sine waves is applied at the input. The output signal consists of the same frequencies but with different amplitudes and/or phases.

Many real-world systems act as nonlinear systems when their input limits are exceeded, resulting in distorted output signals. If the input limits of a system are exceeded, the output consists of one or more frequencies that did not originally exist at the input. For example, if the input to a nonlinear system consists of two frequencies f_1 and f_2, the frequencies at the output could be f_1 and harmonics (integer multiples) of f_1, f_2 and harmonics of f_2, and sums and differences of f_1, f_2, and the harmonics of f_1 and f_2. The number of new frequencies at the output, their corresponding amplitudes, and their relationships with respect to the original frequencies vary depending on the transfer function. Use distortion measurements to quantify the degree of nonlinearity of a system. Some common distortion measurements include Total Harmonic Distortion (THD), Total Harmonic Distortion + Noise (THD + N), Signal Noise and Distortion (SINAD), and Intermodulation Distortion.

Application Areas

You can make distortion measurements for many devices, such as A/D and D/A converters, audio processing devices, such as preamplifiers, equalizers, and power amplifiers, analog tape recorders, cellular phones, radios, TVs, stereos, and loudspeakers.

Measurements of harmonics often provide a good indication of the cause of the nonlinearity. For example, nonlinearities that are not symmetrical around zero produce mainly even harmonics, whereas symmetrical

nonlinearities result in the production of mainly odd harmonics. You can use distortion measurements to diagnose faults such as bad solder joints, torn speaker cones, and components that have been incorrectly installed. Nonlinearities are not always undesirable, however. For example, many musical sounds are produced specifically by driving a device into its nonlinear region.

Harmonic Distortion

When a signal, *x(t)*, of a particular frequency (for example, f_1) is passed through a nonlinear system, the output of the system consists of not only the input frequency (f_1), but also its harmonics ($f_2 = 2f_1$, $f_3 = 3f_1$, $f_4 = 4f_1$, and so on). The number of harmonics, and their corresponding amplitudes, that are generated depends on the degree of nonlinearity of the system. In general, the more the nonlinearity, the higher the harmonics, and vice versa.

An example of a nonlinear system is a system where the output *y(t)* is the cube of the input signal *x(t)*, as shown in Figure 14-1.

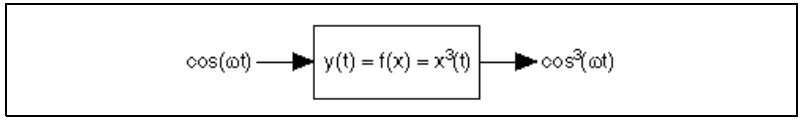


Figure 14-1. Example Nonlinear System

So, if the input is

$$x(t) = \cos(\omega t)$$

the output is

$$x^3(t) = 0.5\cos(\omega t) + 0.25[\cos(\omega t) + \cos(3\omega t)]$$

Therefore, the output contains not only the input fundamental frequency of ω, but also the third harmonic of 3ω.

Total Harmonic Distortion

To determine the amount of nonlinear distortion that a system introduces, you need to measure the amplitudes of the harmonics that were introduced by the system relative to the amplitude of the fundamental. Harmonic distortion is a relative measure of the amplitudes of the harmonics as compared to the amplitude of the fundamental. If the amplitude of the

fundamental is A_1 and the amplitudes of the harmonics are A_2 (second harmonic), A_3 (third harmonic), A_4 (fourth harmonic), and so on, then the total harmonic distortion (THD) is given by

$$\text{THD} = \frac{\sqrt{A_2^2 + A_3^2 + A_4^2 + \ldots}}{A_1}$$

The percentage total harmonic distortion (%THD) is given by the equation:

$$\text{percentageTHD} = 100 \cdot \frac{\sqrt{A_2^2 + A_3^2 + A_4^2 + \ldots}}{A_1}$$

Thus, measurement of the total harmonic distortion requires measuring the amplitudes of the fundamental frequency and the amplitudes of the individual harmonics. A common cause of harmonic distortion is clipping that occurs when a system is driven beyond its capabilities. Symmetrical clipping results in odd harmonics, but asymmetrical clipping creates both even and odd harmonics.

Real-world signals are usually noisy. The system also can introduce additional noise into the signal. A useful measure of distortion, which also takes into account the amount of noise power, is total harmonic distortion + noise (THD + N) and is given by the equation:

$$\text{THD} + N = \frac{\sqrt{A_2^2 + A_3^2 + \ldots + N^2}}{\sqrt{A_1^2 + A_2^2 + A_3^2 + \ldots N^2}}$$

where N is the noise power.

The percentage total harmonic distortion + noise (%THD + N) is given by the equation:

$$\text{percentageTHD} + N = 100 \cdot \frac{\sqrt{A_2^2 + A_3^2 + \ldots + N^2}}{\sqrt{A_1^2 + A_2^2 + A_3^2 + \ldots N^2}}$$

Thus, measurement of THD + N requires measuring the amplitude of the fundamental frequency and the power present in the remaining signal after the fundamental frequency has been removed.

THD + N also includes the noise, a low measurement not only means that the system has a low amount of harmonic distortion, it also means that the contribution from the AC mains hum, wideband white noise, and other interfering signals is low. Measurements of THD or THD + N are usually specified in terms of the highest order harmonic that has been present in the measurement, for example, THD through the seventh harmonic or THD + N through the third harmonic.

SINAD

Another measurement that takes into account both harmonics and noise is SINAD. SINAD is given by the equation:

$$\text{SINAD} = \frac{\text{Fundamental} + \text{Noise} + \text{Distortion}}{\text{Noise} + \text{Distortion}}$$

SINAD is the reciprocal of THD + N. You can use SINAD to characterize the performance of FM receivers in terms of sensitivity, adjacent channel selectivity, and alternate channel selectivity.

15

Limit Testing

You can use limit testing to monitor a waveform and determine if it always satisfies a set of conditions, usually upper and lower limits. The region bounded by the specified limits is a mask. The result of a limit or mask test is generally a pass or fail.

Setting Up an Automated Test System

You can use the same method to create and control many different automated test systems. Complete the following basic steps to set up an automated test system for limit mask testing.

1. Configure the measurement by specifying arbitrary upper and lower limits. This defines your mask or region of interest.
2. Acquire data using a DAQ device.
3. Monitor the data to make sure it always falls within the specified mask.
4. Log the Pass/Fail results from step 3 to a file or visually inspect the input data and the points that fall outside the mask.

Repeat steps 2 through 4 to continue limit mask testing.

The following sections examine steps 1 and 3 in further detail. Assume that the signal to be monitored starts at $x = x_0$ and all the data points are evenly spaced. The spacing between each point is denoted by dx.

Specifying a Limit

Limits are classified into two types: continuous limits and segmented limits, as shown in Figure 15-1. The top graph in Figure 15-1 shows a continuous limit. A continuous limit is specified using a set of x and y points $\{\{x_1, x_2, x_3, \ldots\}, \{y_1, y_2, y_3, \ldots\}\}$. Completing step 1 creates a limit with the first point at x_0 and all other points at an uniform spacing of dx ($x_0 + dx$, $x_0 + 2dx$, …). This is done through a linear interpolation of the x and y values that define the limit. In Figure 15-1, black dots represent the points at which the limit is defined and the solid line represents the limit you create. Creating the limit in step 1 reduces test times in step 3. If the

spacing between the samples changes, you can repeat step 1. Notice that the limit is undefined in the region $x_0 < x < x_1$ and for $x > x_4$.

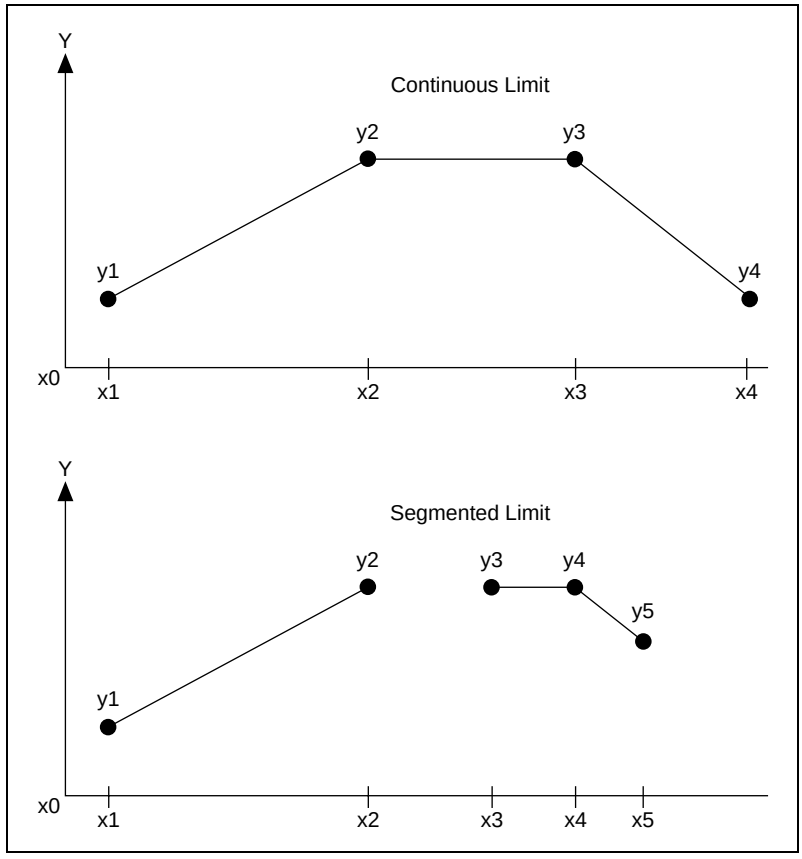


Figure 15-1. Continuous vs. Segmented Limit Specification

The bottom graph of Figure 15-1 shows a segmented limit. The first segment is defined using a set of x and y points $\{\{x_1, x_2\}, \{y_1, y_2\}\}$. The second segment is defined using a set of points $\{x_3, x_4, x_5\}$ and $\{y_3, y_4, y_5\}$. You can define any number of such segments. As with continuous limits, step 1 uses linear interpolation to create a limit with the first point at x_0 and all other points with an uniform spacing of dx. Notice that the limit is undefined in the region $x_0 < x < x_1$ and in the region $x > x_5$. Also notice the limit is undefined in the region $x_2 < x < x_3$.

Specifying a Limit Using a Formula

You can specify limits using formulas. Such limits are best classified as segmented limits. Each segment is defined by start and end frequencies and a formula. For example, the ANSI T1-413 recommendation specifies the limits for the transmit and receive spectrum of an ADSL signal in terms of formula. Table 15-1, which only includes a part of the specification, shows the start and end frequencies and the upper limits of the spectrum for each segment.

Table 15-1. ADSL Signal Recommendations

Start KHz	End KHz	Maximum (Upper Limit) Value (dBm/Hz)
0.3	4.0	–97.5
4.0	25.9	$-92.5 + 21.5\log_2(f/4000)$
25.9	138.0	–34.5
138.0	307.0	$-34.5 - 48.0\log_2(f/138000)$
307.0	1221.0	–90

The limit is specified as an array of a set of *x* and *y* points, [{0.3, 4.0}{–97.5, –97.5}, {4.0, 25.9}{$-92.5 + 21.5\log_2(f/4000)$, {$-92.5 + 21.5\log_2(f/4000)$}, …, {307.0, 1221.0}{–90, –90}]. Each element of the array corresponds to a segment.

Figure 15-2 shows the segmented limit specified using formula as shown in Table 15-1. The *x* axis is on a logarithmic scale.

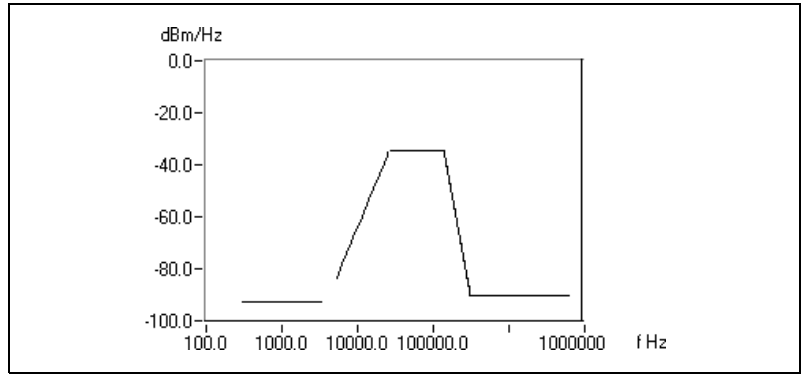


Figure 15-2. Segmented Limit Specified Using Formula

Limit Testing

After you define your mask, you acquire a signal using a DAQ device. The sample rate is set at 1/*dx* S/s. Compare the signal with the limit. In step 1, you create a limit value at each point where the signal is defined. In step 3, you compare the signal with the limit. For the upper limit, if the data point is less than or equal to the limit point, the test passes. If the data point is greater than the limit point, the test fails. For the lower limit, if the data point is greater than or equal to the limit point, the test passes. If the data point is smaller than the limit point, the test fails.

Figure 15-3 shows the result of limit testing in a continuous mask case. Here, the test signal falls within the mask at all the points it is sampled, other than points b and c. Thus the limit test fails. We do not test point d because it falls outside the mask.

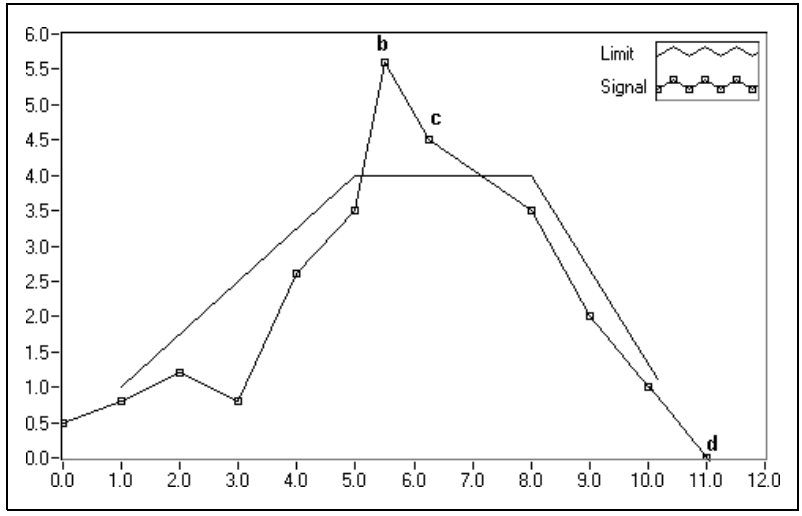


Figure 15-3. Result of Limit Testing with a Continuous Mask

Figure 15-4 shows the result of limit testing in a segmented mask case. Here, all the points fall within the mask. Points b and c are not tested because the mask is undefined at those points. Thus the limit test passes. Point d is not tested because it falls outside the mask.

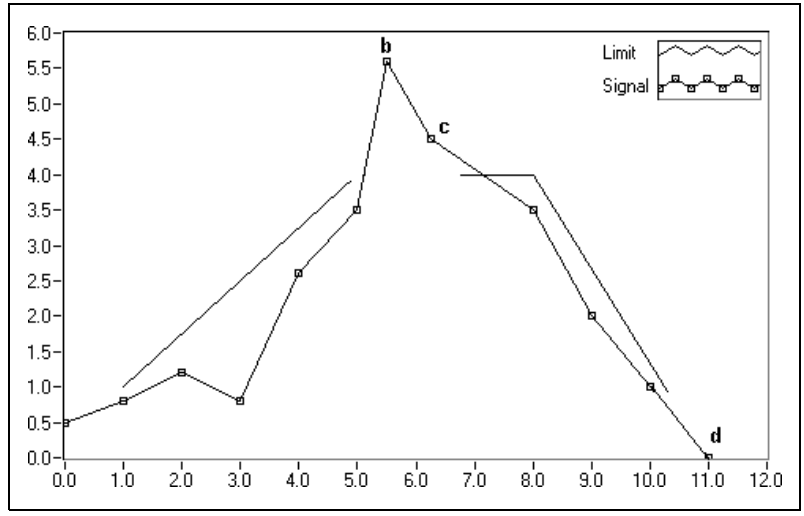


Figure 15-4. Result of Limit Testing with a Segmented Mask

Applications

You can use limit mask testing in a wide range of test and measurement applications. For example, you can use limit mask testing to determine that the power spectral density of ADSL signals meets the recommendations laid out in the ANSI T1-413 specification. Refer to the *Specifying a Limit Using a Formula* section in this chapter for more information about ADSL signal limits.

The following sections provide examples of when you can use limit mask testing. In all these examples, the specifications are recommended by standards-generating bodies, such as the CCITT, ITU-T, ANSI and IEC, to ensure that all the test and measurement systems conform to a universally accepted standard. In some other cases, the limit testing specifications are proprietary and are strictly enforced by companies for quality control.

Modem Manufacturing Example

Limit testing is used in modem manufacturing to make sure the transmit spectrum of the line signal meets the V.34 modem specification as shown in Figure 15-5.

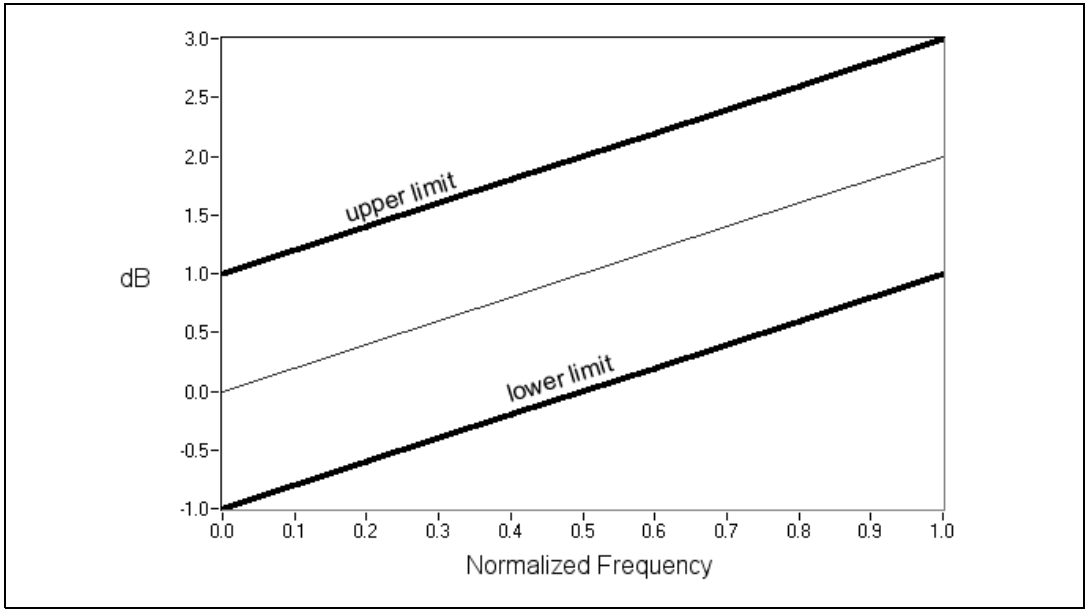


Figure 15-5. Upper and Lower Limit for V.34 Modem Transmitted Spectrum

The ITU-T V.34 Recommendation contains specifications for a modem operating at data signaling rates up to 33600 bits/s. It specifies that the spectrum for the line signal that transmits data conforms to the template shown in Figure 15-5. For example, for a normalized frequency of 1.0, the spectrum must always lie between 3 and 1 dB. All the modems must meet this specification. A modem manufacturer can set up an automated test system to monitor the transmit spectrum for the signals that the modem outputs. If the spectrum conforms to the specification, the modem passes the test and is ready for customer use. Recommendations such as the ITU-T V.34 are essential to ensure interoperability between modems from different manufacturers and to provide high-quality service to customers.

Digital Filter Design Example

You also can use limit mask testing in the area of digital filter design. You may want to design lowpass filters with a passband ripple of 10 dB and stopband attenuation of 60 dB. You can use limit testing to make sure the frequency response of the filter always meets these specifications. The first step in this process is to specify the limits. You can specify a lower limit of –10 dB in the passband region and an upper limit of –60 dB in the stopband region, as shown in Figure 15-6. After you specify these limits, you can run the actual test repeatedly to make sure that all the frequency responses of all the filters are designed meet these specifications.

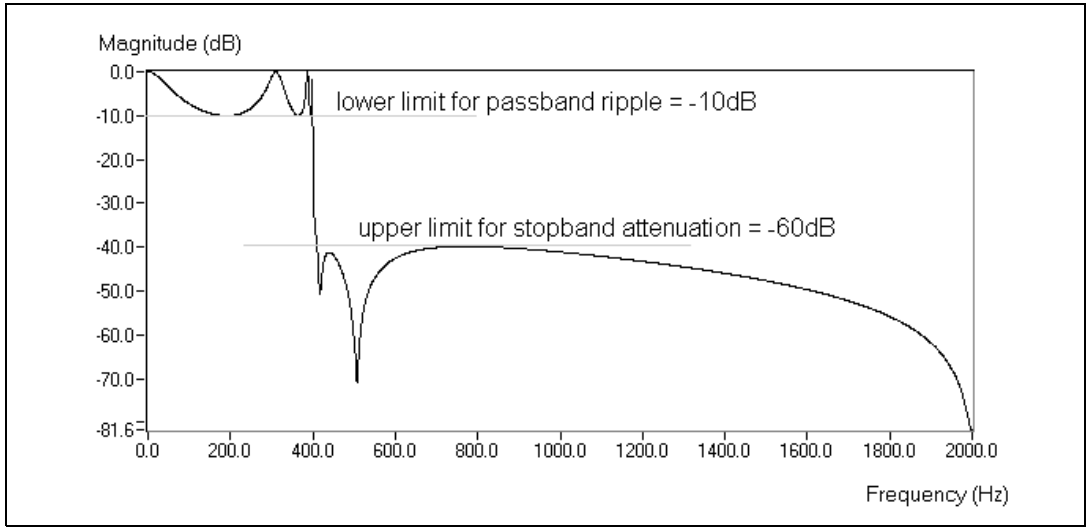


Figure 15-6. Limit Test of a Lowpass Filter Frequency Response

Pulse Mask Testing Example

The ITU-T G.703 recommendation specifies the pulse mask for signals with bit rates, $n \times 64$, where n is between 2 and 31. Figure 15-7 shows the pulse mask for interface at 1544 kbits/s. Signals with this bit rate are also referred to as T1 signals. T1 signals must lie in the mask specified by the upper and lower limit. These limits are set to properly enable the interconnection of digital network components to form a digital path or connection.

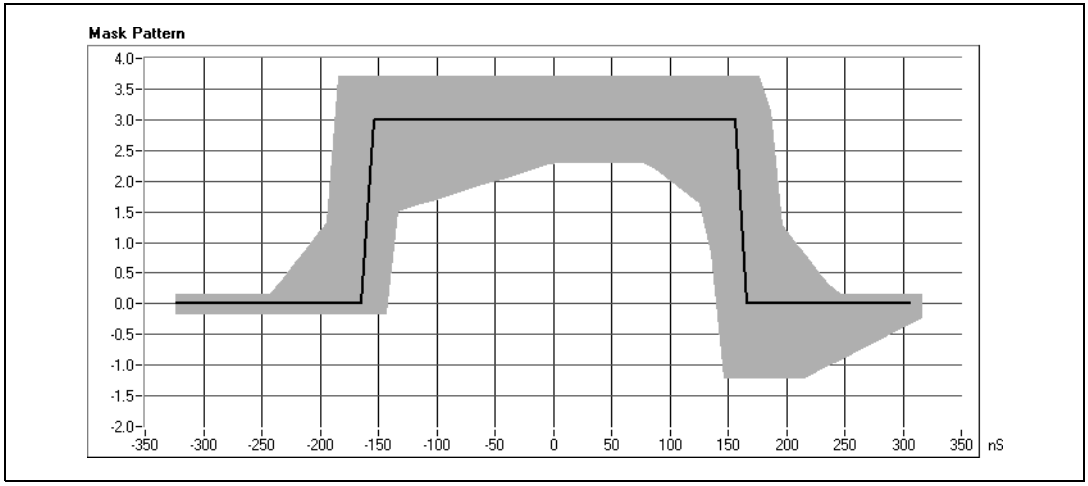


Figure 15-7. Pulse Mask Testing on T1/E1 Signals

16

Digital Filtering

This chapter introduces the concept of filtering, compares analog and digital filters, discusses Finite Infinite Response (FIR) and Infinite Impulse Response (IIR) filters, and helps you determine how to choose the most appropriate digital filter.

What Is Filtering?

Filtering is the process by which the frequency content of a signal is altered. The implicit assumption is that the signal content of interest is separable from the raw signal. Classical linear filtering assumes that the signal content of interest is distinct from the remainder of the signal in the frequency domain (Fourier Transform). Filtering is one of the most commonly used signal processing techniques. For example, consider the bass and treble controls on your stereo system. The bass control alters the low-frequency content of a signal, and the treble control alters the high-frequency content. By varying these controls, you are filtering the audio signal. Removing noise and performing decimation (lowpass filtering the signal and reducing the sample rate) are other filtering applications.

Advantages of Digital Filtering over Analog Filtering

An analog filter has an analog signal at both its input and its output. Both the input, *x(t)*, and output, *y(t)*, are functions of a continuous variable *t* and can have an infinite number of values. Analog filter design is about 50 years older than digital filter design. This type of filter design is often reserved for specialists because it requires advanced mathematical knowledge and understanding of the processes involved in the system affecting the filter. Modern sampling and digital signal processing tools have made it possible to replace analog filters with digital filters in applications that require flexibility and programmability, such as audio, telecommunications, geophysics, and medical monitoring.

Some advantages of digital filters over analog filters include the following:

- Digital filters are software-programmable and therefore are easy to build and test.
- Digital filters require only the arithmetic operations of multiplication and addition/subtraction and therefore are easier to implement.
- Digital filters do not drift with temperature or humidity or require precision components.
- Digital filters have a superior performance-to-cost ratio.
- Digital filters do not suffer from manufacturing variations or aging.

Common Digital Filters

Digital filters can be classified in many ways. The traditional approach is to first classify a filter based on the values upon which it operates. The simplest filters are those that operate on input values only. These filters are called Moving Average (MA) filters or Finite Impulse Response (FIR) filters. These filters perform a convolution of the filter coefficients with a sequence of input values, producing an equally numbered sequence of output values. The term FIR is used because if a single impulse is present at the input of the filter and all subsequent inputs are zero, then the output of the filter becomes zero after some finite time, equal to the number of filter coefficients.

If a filter operates on current and previous input values and current and previous output values, then the filter is termed Infinite Impulse Response (IIR) or Auto Regressive Moving Average (ARMA). The impulse response of such a filter is infinite in the sense that the response of the filter to an impulse never goes to zero.

Each type of filter has advantages and disadvantages. Filter design, as with all other engineering practices, involves tradeoffs. FIR filters are simple, and can be designed to provide a linear phase response or constant group delay. IIR filters can achieve the same level of attenuation as FIR filters with far fewer coefficients. This means that the IIR filter can be significantly faster and more efficient.

Ideal Filters

Filters alter or remove unwanted frequencies. Depending on the frequency range that they either pass or attenuate (reject), they can be classified into the following types:

- A *lowpass filter* passes low frequencies but attenuates high frequencies.
- A *highpass filter* passes high frequencies but attenuates low frequencies.
- A *bandpass filter* passes a certain band of frequencies.
- A *bandstop filter* attenuates a certain band of frequencies.

The ideal frequency response of these filters is shown in Figure 16-1.

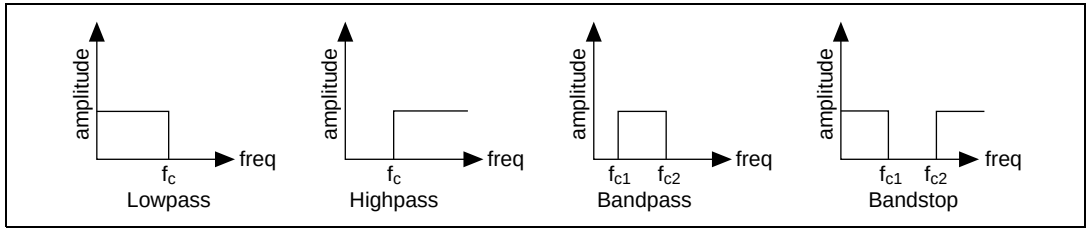


Figure 16-1. Ideal Frequency Response

The lowpass filter passes all frequencies below f_c, whereas the highpass filter passes all frequencies above f_c. The bandpass filter passes all frequencies between f_{c1} and f_{c2}, whereas the bandstop filter attenuates all frequencies between f_{c1} and f_{c2}. The frequency points f_c, f_{c1} and f_{c2} are known as the cut-off frequencies of the filter. When designing filters, you need to specify these cut-off frequencies.

The frequency range that is passed through the filter is known as the *passband* (PB) of the filter. An ideal filter has a gain of one (0 dB) in the passband so that the amplitude of the signal neither increases nor decreases. The *stopband* (SB) corresponds to that range of frequencies that do not pass through the filter at all and are attenuated. The passband and the stopband for the different types of filters are shown in Figure 16-2.

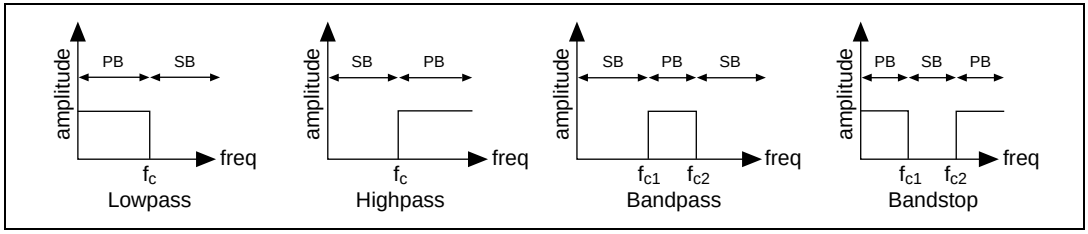


Figure 16-2. Passband and Stopband

Note that while the lowpass and highpass filters have one passband and one stopband, the bandpass filter has one passband and two stopbands, and the bandstop filter has two passbands and one stopband.

Practical (Nonideal) Filters

Ideally, a filter should have a unit gain (0 dB) in the passband, and a gain of zero (–infinity dB) in the stopband. However, in a real implementation, not all of these criteria can be fulfilled. In practice, there is always a finite transition region between the passband and the stopband. In this region, the gain of the filter changes gradually from one (0 dB) in the passband to zero (–infinity dB) in the stopband.

The Transition Band

The following diagrams show the passband, the stopband, and the transition region (TR) for the different types of nonideal filters. Note that the passband is now the region where the frequency range within which the gain of the filter varies from 0 dB to –3 dB.

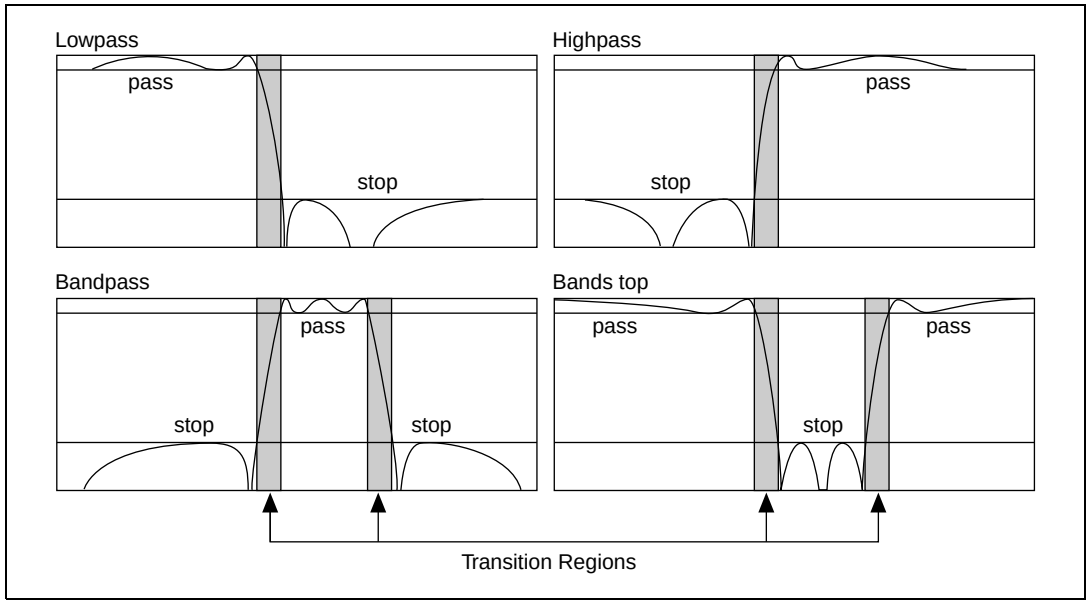


Figure 16-3. Nonideal Filters

Passband Ripple and Stopband Attenuation

In many applications, you can allow the gain in the passband to vary slightly from unity. This variation in the passband is called the *passband ripple* and is the difference between the actual gain and the desired gain of unity. The *stopband attenuation*, in practice, cannot be infinite, and you must specify a value with which you are satisfied. Both the passband ripple and the stopband attenuation are measured in decibels (dB), defined by the equation:

$$\mathrm{dB} = 20\log\left(\frac{A_o(f)}{A_i(f)}\right)$$

where *log* denotes the base 10 logarithm, and $A_i(f)$ and $A_o(f)$ are the amplitudes at a particular frequency f before and after the filtering, respectively.

For example, for –0.02 dB passband ripple, the formula gives:

$$-0.02 = 20\log\left(\frac{A_o(f)}{A_i(f)}\right)$$

$$\frac{A_o(f)}{A_i(f)} = 10^{-0.001} = 0.9977$$

which shows that the ratio of input and output amplitudes is close to unity.

You can view practical filter design as approximating the ideal desired magnitude response subject to certain constraints. The ideal passband and stopband are flat and constant. Practical filter passbands and stopbands may have ripples. Ideal filters have no transition region. Practical filters have transition regions. Practical filter design allows tradeoffs between these different components (passband ripple, stopband ripple, stopband attenuation, transition region width) subject to the filter structure (FIR or IIR) and the design algorithm.

FIR Filters

FIR filters have several different design methods. FIR filters have ripple in the magnitude response, so the design problem can be restated as how you can design a filter that has a magnitude response as close to the ideal as possible and distributes the ripple in a desired fashion. For example, a lowpass filter has an ideal characteristic magnitude response. A particular application may allow some ripple in the passband and more ripple in the stopband. The filter design algorithm should balance the relative ripple requirements while producing the sharpest transition region.

The simplest approach is the Windowed FIR design. The Windowed FIR design takes the inverse FFT of the desired magnitude response and applies a time domain window to the result. The advantages of this method are conceptual simplicity and ease of implementation. The disadvantages are the inefficiency and difficulty in specification. For a given number of taps, the Windowed FIR design does not distribute ripple equally and has a wider transition band than other designs. It also is difficult to specify a cut-off frequency that has a particular attenuation. To design a Windowed FIR filter, you must specify the ideal cut-off frequency, the sampling frequency, the number of taps, and the window type.

The other main FIR design approach uses the Parks-McClellan algorithm, also known as Remez Exchange. This is an iterative algorithm that

produces filters with a magnitude response for which the weighted ripple is evenly distributed over the passband and stopband and that have a sharp transition region. The advantage of this approach is the optimal response of the designed filter. The disadvantages are the complexity and length of time required to design. Park-McClellan design time is much longer than the Windowed approach. A specialization of the Parks-McClellan approach is equiripple FIR design. The only difference between them is the equiripple design weights the passband and stopband ripple equally. To design an FIR filter using the equiripple approach, you must specify the cut-off frequency, the number of taps, the filter type, and pass and stop frequencies. The cut-off frequency for equiripple designs specifies the edge of the passband and/or the stopband. Equiripple filters have a ripple in the passband that causes the magnitude response in the passband to be greater than or equal to 1. Similarly, the magnitude response in the stopband is always less than or equal to the stopband attenuation. For example, if you specify a lowpass filter, the passband cut-off frequency is the highest or largest frequency for which the passband conditions hold true. Similarly, the stopband cut-off is the lowest frequency for which the stopband conditions are met. Both design approaches deliver FIR filters with a linear phase characteristic.

When you use conventional techniques to design FIR filters with especially narrow bandwidths, the resulting filter lengths can be very long. FIR filters with long filter lengths often require lengthy design and implementation times and are more susceptible to numerical inaccuracy. In some cases, conventional filter design techniques, such as the Parks-McClellan algorithm, might fail the design altogether.

IIR Filters

IIR filters are filters that may or may not have ripple in the passband and/or the stopband. Digital IIR filter design derives from the classical analog designs. These designs are Butterworth, Chebyshev, inverse Chebyshev, Elliptic, and Bessel.

Butterworth Filters

A smooth response at all frequencies and a monotonic decrease from the specified cut-off frequencies characterize the frequency response of Butterworth filters. Butterworth filters are maximally flat, the ideal response of unity in the passband and zero in the stopband. The half power frequency or the 3 dB down frequency corresponds to the specified cut-off frequencies.

Figure 16-4 shows the response of a lowpass Butterworth filter. The advantage of Butterworth filters is a smooth, monotonically decreasing frequency response. After you set the cut-off frequency, LabVIEW sets the steepness of the transition proportional to the filter order. Higher order Butterworth filters approach the ideal lowpass filter response.

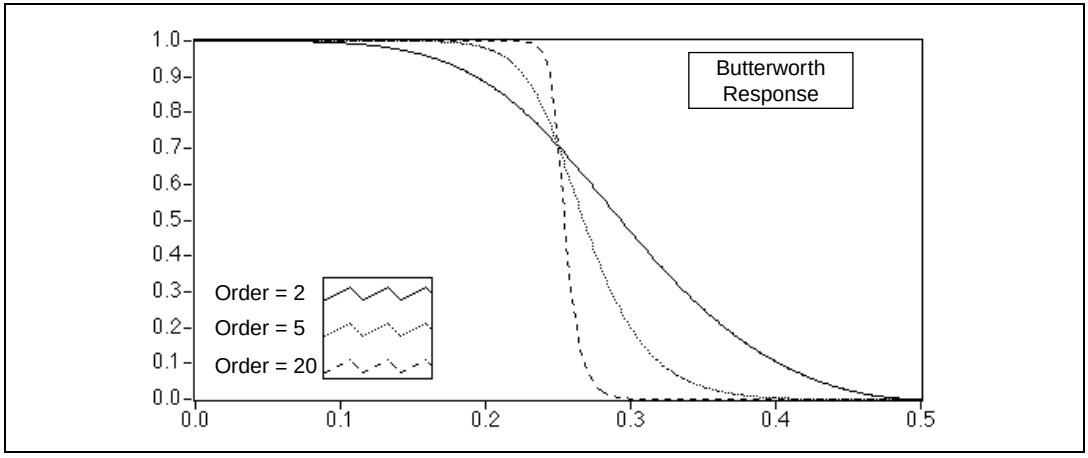


Figure 16-4. Butterworth Filter Response

Chebyshev Filters

Butterworth filters do not always provide a good approximation of the ideal filter response because of the slow rolloff between the passband (the portion of interest in the spectrum) and the stopband (the unwanted portion of the spectrum).

Chebyshev filters minimize peak error in the passband by accounting for the maximum absolute value of the difference between the ideal filter and the filter response you want (the maximum tolerable error in the passband). The frequency response characteristics of Chebyshev filters have an equiripple magnitude response in the passband, monotonically decreasing magnitude response in the stopband, and a sharper rolloff than Butterworth filters.

Figure 16-5 shows the response of a lowpass Chebyshev filter. Notice that the equiripple response in the passband is constrained by the maximum tolerable ripple error and that the sharp rolloff appears in the stopband. The advantage of Chebyshev filters over Butterworth filters is that Chebyshev filters have a sharper transition between the passband and the stopband with a lower order filter. This produces smaller absolute errors and higher execution speeds.

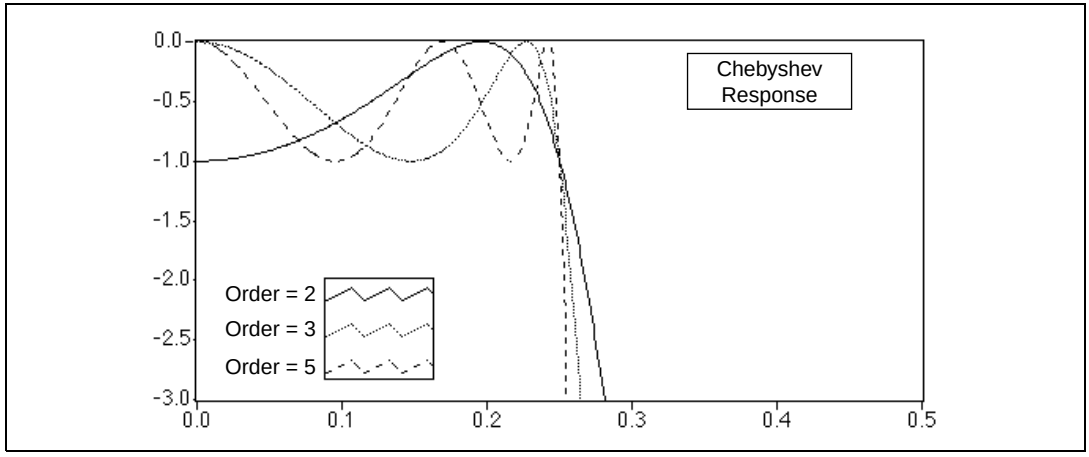


Figure 16-5. Chebyshev Filter Response

Chebyshev II or Inverse Chebyshev Filters

Chebyshev II, also known as inverse Chebyshev or Type II Chebyshev filters, are similar to Chebyshev filters, except that Chebyshev II filters distribute the error over the stopband (as opposed to the passband), and Chebyshev II filters are maximally flat in the passband (as opposed to the stopband).

Chebyshev II filters minimize peak error in the stopband by accounting for the maximum absolute value of the difference between the ideal filter and the filter response you want. The frequency response characteristics of Chebyshev II filters are equiripple magnitude response in the stopband, monotonically decreasing magnitude response in the passband, and a rolloff sharper than Butterworth filters.

Figure 16-6 plots the response of a lowpass Chebyshev II filter. Notice that the equiripple response in the stopband is constrained by the maximum tolerable error and that the smooth monotonic rolloff appears in the stopband. The advantage of Chebyshev II filters over Butterworth filters is that Chebyshev II filters give a sharper transition between the passband and the stopband with a lower order filter. This difference corresponds to a smaller absolute error and higher execution speed. One advantage of Chebyshev II filters over regular Chebyshev filters is that Chebyshev II filters distribute the error in the stopband instead of the passband.

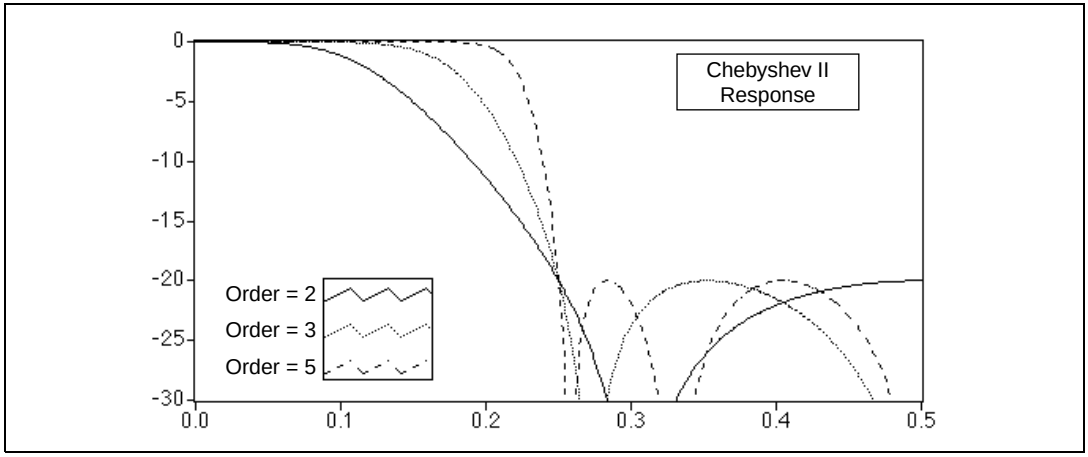


Figure 16-6. Chebyshev II Filter Response

Elliptic (or Cauer) Filters

Elliptic filters minimize the peak error by distributing it over the passband and the stopband. Equiripples in the passband and the stopband characterize the magnitude response of elliptic filters. Compared with the same order Butterworth or Chebyshev filters, the elliptic design provides the sharpest transition between the passband and the stopband. For this reason, elliptic filters are widely used.

Figure 16-7 plots the response of a lowpass elliptic filter. Notice that the ripple in both the passband and stopband is constrained by the same maximum tolerable error (as specified by ripple amount in decibels). Also, notice the sharp transition edge for even low-order elliptic filters.

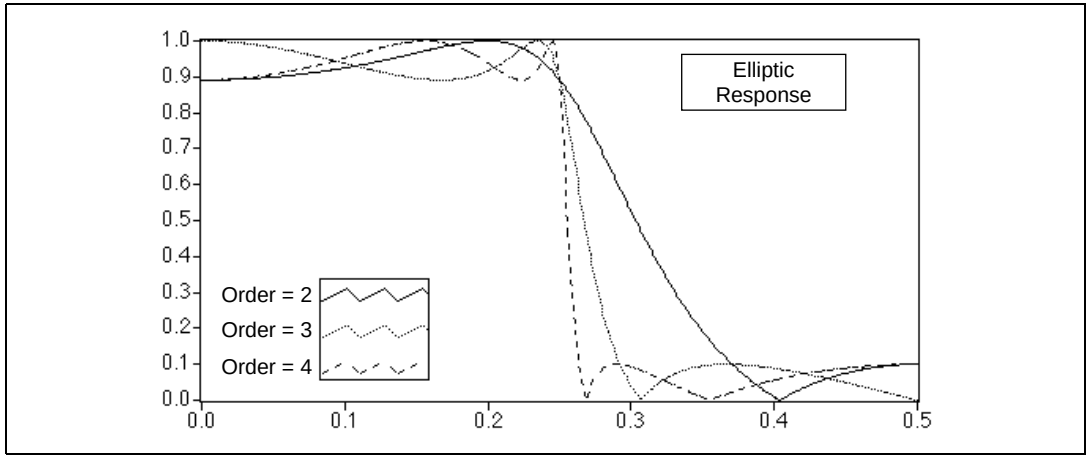


Figure 16-7. Elliptic Filter Response

Bessel Filters

You can use Bessel filters to reduce nonlinear phase distortion inherent in all IIR filters. In higher order filters and those with a steeper rolloff, this condition is more pronounced, especially in the transition regions of the filters. Bessel filters have maximally flat response in both magnitude and phase. Furthermore, the phase response in the passband of Bessel filters, which is the region of interest, is nearly linear. Like Butterworth filters, Bessel filters require high-order filters to minimize the error and, for this reason, are not widely used. You can also obtain linear phase response using FIR filter designs.

Figure 16-8 and Figure 16-9 plot the response of a lowpass Bessel filter. Notice that the response is smooth at all frequencies, as well as monotonically decreasing in both magnitude and phase. Also, notice that the phase in the passband is nearly linear.

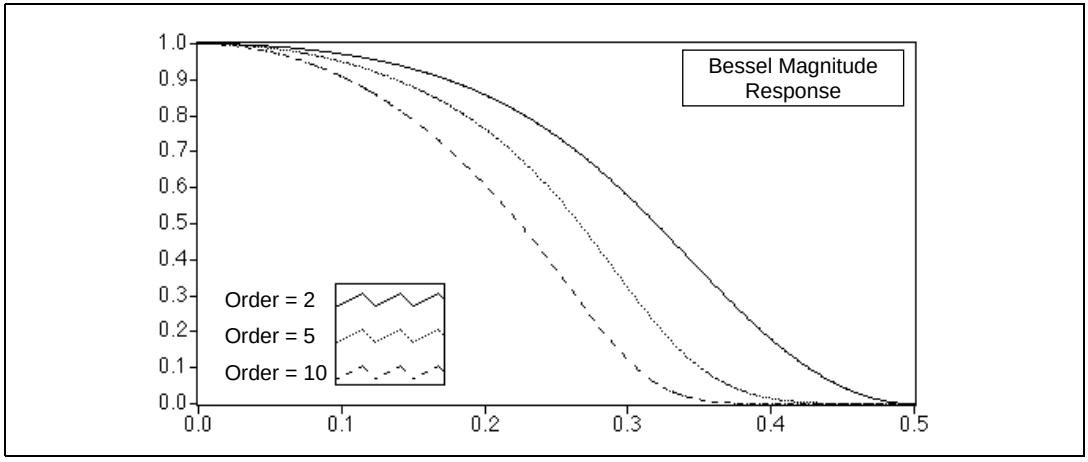


Figure 16-8. Bessel Magnitude Filter Response

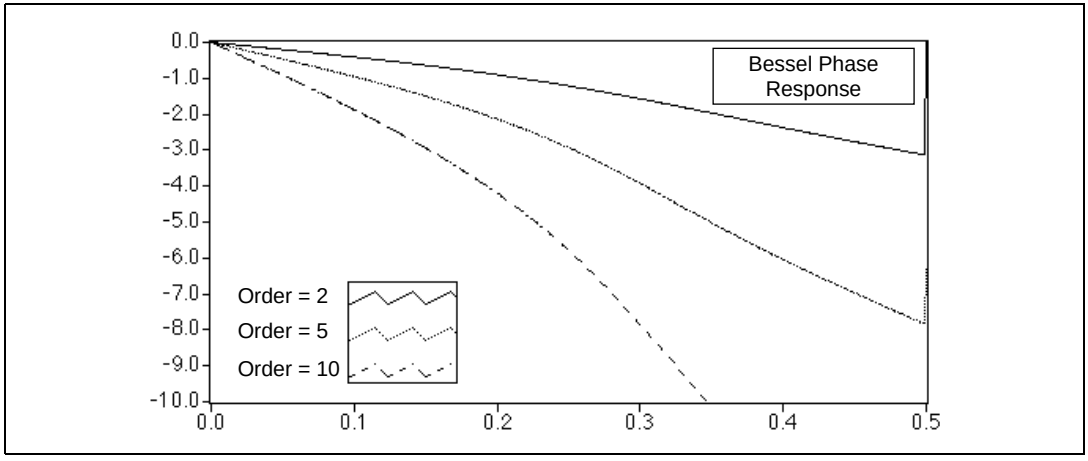


Figure 16-9. Bessel Phase Filter Response

Choosing and Designing a Digital Filter

Some of the factors affecting the choice of a suitable filter are whether you require linear phase, whether you can tolerate ripples, and whether a narrow transition band is required. Use Figure 16-10 as a guideline for selecting the correct filter. Keep in mind that in practice, you may need to experiment with several different options before finding the best one.

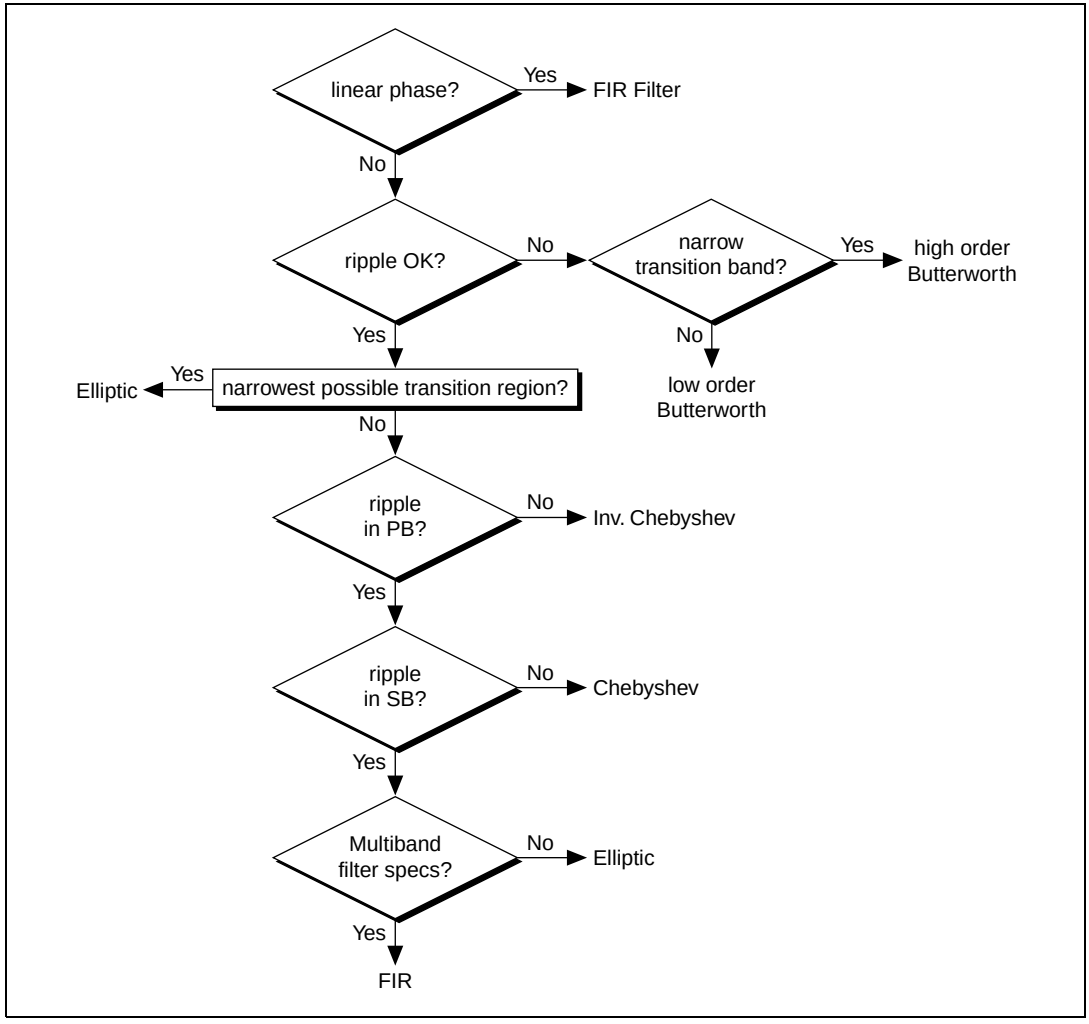


Figure 16-10. Filter Flowchart

After you choose the type of filter, you must specify the design parameters. The first filter design parameter to consider is sampling rate. The maximum frequency component of the signal of interest usually determines the sampling rate. A common rule of thumb is to choose a sampling rate that is 10 times the highest frequency component of the signal of interest. The possible tradeoff occurs when the cutoff frequency of the filter must be very close to either DC or the Nyquist frequency. At these points, a filter may converge more slowly. The solution is to increase the sampling rate if the

cutoff is too close to Nyquist, or reduce the sampling rate if the cutoff is too close to DC. In practice, a particular sampling rate is chosen and adjusted only if there are problems.

17

Signal Generation

The generation of signals is an important part of any test or measurement system. Common test signals include the sine wave, the square wave, the triangle wave, the sawtooth wave, several types of noise waveforms, and multitone signals consisting of a superposition of sine waves.

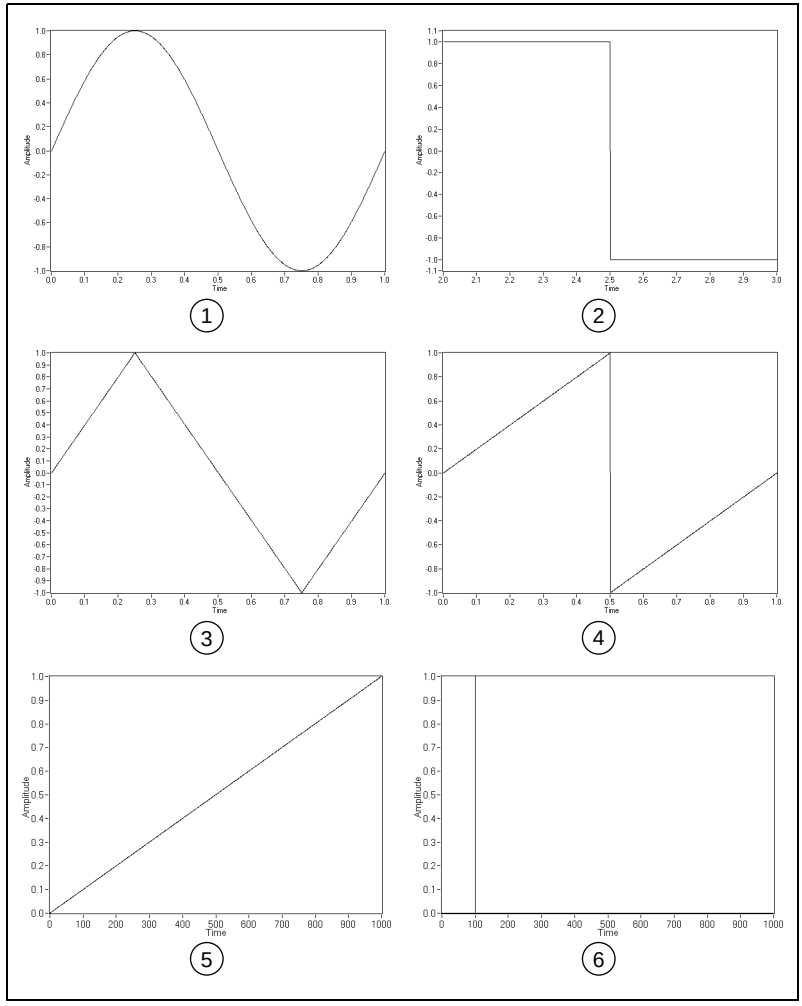
The most common signal for audio testing is the sine wave. A single sine wave is often used to determine the amount of harmonic distortion introduced by a system. Multiple sine waves are widely used to measure the intermodulation distortion or to determine the frequency response. The following table lists the signals used for some typical measurements.

Table 17-1. Typical Measurements and Signals

Measurement	Signal
Total Harmonic Distortion	Sine wave
Intermodulation Distortion	Multitone (two sine waves)
Frequency Response	Multitone (many sine waves, Impulse, Chirp)
Interpolation	Sinc
Rise Time, Fall Time, Overshoot, Undershoot	Pulse
Jitter	Square wave

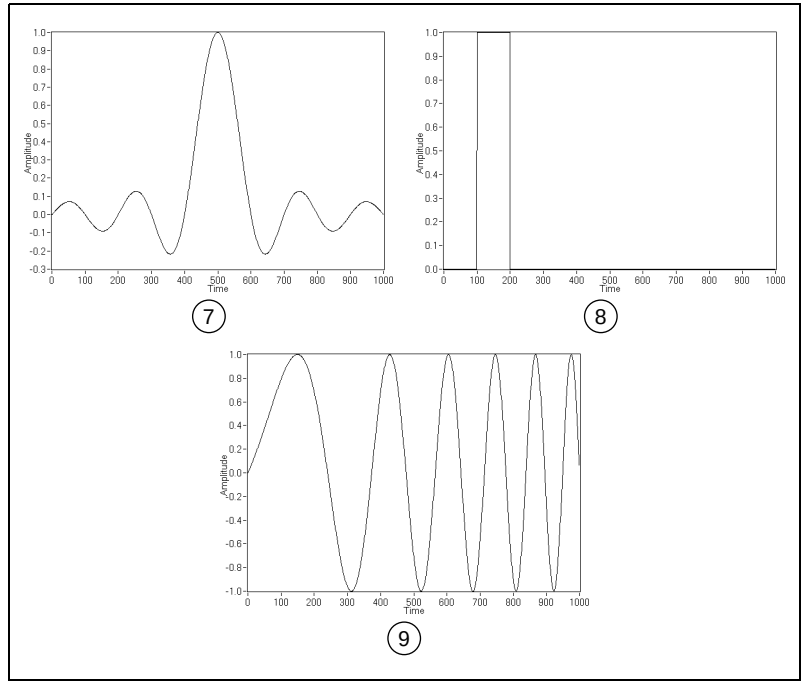
Common Test Signals

These signals form the basis for many tests and are used to measure the response of a system to a particular stimulus. Some of the common test signals available in most signal generators are as shown in Figure 17-1 and Figure 17-2.



1	Sine Wave	2	Square Wave
3	Triangle Wave	4	Sawtooth
5	Ramp	6	Impulse

Figure 17-1. Common Test Signals



7 Sinc 8 Pulse 9 Chirp

Figure 17-2. Common Test Signals (continued)

It is useful to view these signals in terms of their frequency content. For example, a sine wave has a single frequency component. A square wave consists of the superposition of many sine waves at odd harmonics of the fundamental frequency. The amplitude of each harmonic is inversely proportional to its frequency. Similarly, the triangle and sawtooth waves also have harmonic components that are multiples of the fundamental frequency. An impulse contains all frequencies that can be represented for a given sampling rate and number of samples. Chirp patterns have discrete frequencies that lie within a certain range. These frequencies depend on the sampling rate, the start and end frequencies, and the number of samples.

Multitone Generation

The common test signals, except for the sine wave, do not allow full control over their spectral content. For example, the harmonic components of a square wave are fixed in frequency, phase, and amplitude relative to the fundamental. On the other hand, multitone signals can be generated with a specific amplitude and phase for each individual frequency component.

A multitone signal is the superposition of several sine waves or tones, each with a distinct amplitude, phase, and frequency. A multitone signal is typically created so that an integer number of cycles of each individual tone are contained in the signal. If an FFT of the entire multitone signal is computed, then each of the tones falls exactly onto a single frequency bin. This means there is no spectral spread or leakage.

Multitone signals are a part of many test specifications and allow the fast and efficient stimulus of a system across an arbitrary band of frequencies. Multitone test signals are used to determine the frequency response of a device, and with appropriate selection of frequencies, can also be used to measure such quantities as intermodulation distortion.

Crest Factor

The relative phases of the constituent tones with respect to each other determines the crest factor of a multitone signal with specified amplitude. The crest factor is defined as the ratio of the peak magnitude to the RMS value of the signal. For example, a sine wave has a crest factor of 1.414:1.

For the same maximum amplitude, a multitone signal with a large crest factor contains less energy than one with a smaller crest factor. Another way to express this is to say that a large crest factor means that the amplitude of a given component sine tone is lower than the same sine tone in a multitone signal with a smaller crest factor. A higher crest factor results in individual sine tones with lower signal-to-noise ratios. Proper selection of phases is therefore critical to generating a useful multitone signal.

To avoid clipping, the maximum value of the multitone signal should not exceed the maximum capability of the hardware that generates the signal. This places a limit on the maximum amplitude of the signal. You can generate a multitone signal with a specific amplitude by different combinations of the phase relationships and amplitudes of the constituent sine tones. It is usually better to generate a signal choosing amplitudes and phases that result in a lower crest factor.

Phase Generation

There are two general schemes for generating tone phases of multitone signals. The first is to make the phase difference between adjacent-frequency tones vary linearly from 0 to 360 degrees. This allows the creation of multitone signals with very low crest factors, but the multitone signals then have some potentially undesirable characteristics. This sort of multitone signal is very sensitive to phase distortion. If, in the course of generating the multitone signal, the hardware or signal path

induces non-linear phase distortion, then the crest factor can vary considerably. In addition, a multitone signal with this sort of phase relationship may display some repetitive time domain characteristics that are possibly undesirable. This is shown in the multitone signal in Figure 17-3.

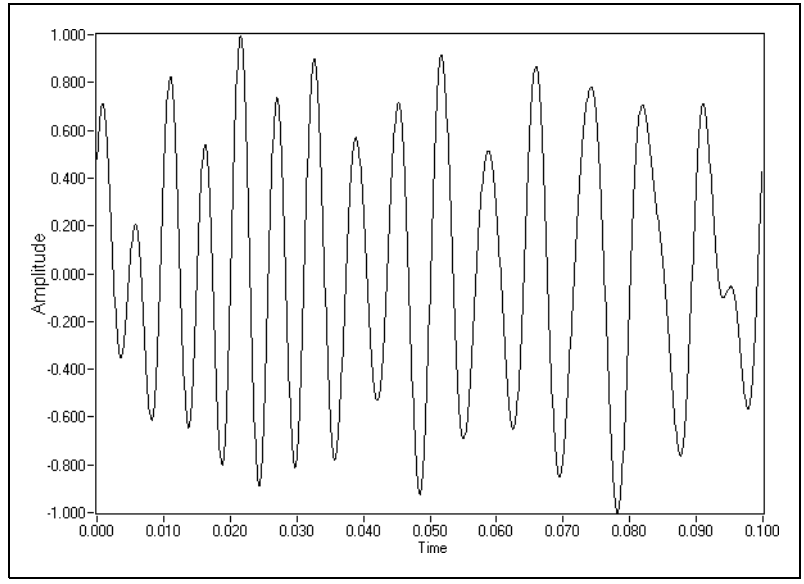


Figure 17-3. Multitone Signal with Linearly Varying Phase Difference between Adjacent Tones

Observe that it resembles a chirp signal in that its frequency appears to decrease from left to right. This is characteristic of multitone signals generated by linearly varying the phase difference between adjacent frequency tones. It is often desirable to have a signal that is more noise-like than this.

Another way to generate the tone phases is to vary them randomly. As the number of tones increases the multitone signal will have amplitudes that are nearly Gaussian in distribution. Figure 17-4 illustrates a signal created using this method.

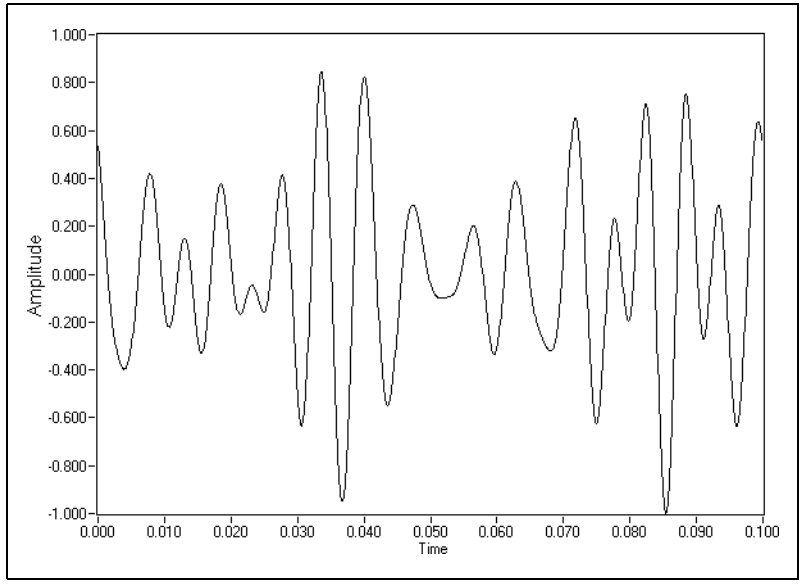


Figure 17-4. Multitone Signal with Random Phase Difference between Adjacent Tones

In addition to being more noise-like, this signal is also much less sensitive to phase distortion. Multitone signals with this sort of phase relationship generally achieve a crest factor between 10 and 11 dB.

Swept Sine versus Multitone

To characterize a system you often must measure the response of the system at many different frequencies. There are several methods to do this, including swept/stepped sine and multitone.

The swept sine is a process of continuously and smoothly changing the frequency of a sine wave across a range of frequencies. The stepped sine approach provides a single sine tone of fixed frequency as the stimulus for a certain time and then increments the frequency by a discrete amount. This process is continued until all the frequencies of interest have been reached.

A multitone signal composed of multiple sine tones has significant advantages over the swept sine and stepped sine approaches. For a given range of frequencies, the multitone approach can be much faster than the equivalent swept sine measurement, due mainly to settling time issues. For a stepped sine measurement, for each sine tone, you must wait for the settling time of the system to be over before starting the measurement. The settling time issue for a swept sine can be even more complex. If the system has low frequency poles/zeroes, or high Q resonances then the system may

take a relatively long time to settle. For a multitone signal, you must wait only once for the settling time. A multitone signal containing one period of the lowest frequency, actually one period of the highest frequency resolution, is enough. Once the response to the multitone signal is acquired, the processing can be very fast. A single FFT may be used to measure many frequency points (amplitude and phase) simultaneously.

There are situations for which a swept sine approach is more appropriate than the multitone. Each measured tone within a multitone signal is more sensitive to noise because the energy of each tone is lower than that in a single pure tone. Consider, for example, a single sine tone of amplitude 10 V peak and frequency 100 Hz. A multitone signal containing 10 tones, including the 100 Hz tone, may have a maximum amplitude of 10 V, but the 100 Hz tone component will have an amplitude somewhat less than this. This lower amplitude is due to the way that all the sine tones (with different phases) sum. Assuming the same level of noise, the signal-to-noise ratio (SNR) of the 100 Hz component is therefore better for the case of the individual tone. It is possible to mitigate this reduced SNR by adjusting the amplitudes of the tones, applying higher energy where needed and lower at less critical frequencies.

When viewing the response of a system to a multitone stimulus, any energy between FFT bins is due to noise or unit-under-test (UUT) induced distortion. The frequency resolution of the FFT is limited by your measurement time. If you only want to measure your system at 1.000 kHz and 1.001 kHz, two independent sine tones is the way to go. The measurement can be done in a few milliseconds while a multitone measurement requires at least 1 second. This is because you must wait for enough time so as to obtain the required number of samples to achieve a frequency resolution of 1 Hz. Some applications, like finding the resonant frequency of a crystal, combine a multitone measurement for coarse measurement and a narrow-range sweep for fine measurement.

Noise Generation

Noise signals may be used to perform frequency response measurements, or to simulate certain processes. Several types of noise are typically used, namely Uniform White Noise, Gaussian White Noise, and Periodic Random Noise.

The term white in the definition of noise refers to the frequency domain characteristic of noise. Ideal white noise has equal power per unit bandwidth, resulting in a flat power spectral density across the frequency range of interest. Thus, the power in the frequency range from 100 Hz to

110 Hz is the same as the power in the frequency range from 1000 Hz to 1010 Hz. In practical measurements, to achieve the flat power spectral density would require an infinite number of samples. Thus, when making measurements of white noise, the power spectra are usually averaged, with more number of averages resulting in a flatter power spectrum.

The terms uniform and Gaussian refer to the probability density function (PDF) of the amplitudes of the time domain samples of the noise. For uniform white noise, the PDF of the amplitudes of the time domain samples is uniform within the specified maximum and minimum levels. Another way to state this is to say that all amplitude values between some limits are equally likely or probable. Thermal noise produced in active components tends to be uniform white in distribution. Figure 17-5 shows the distribution of the samples of uniform white noise.

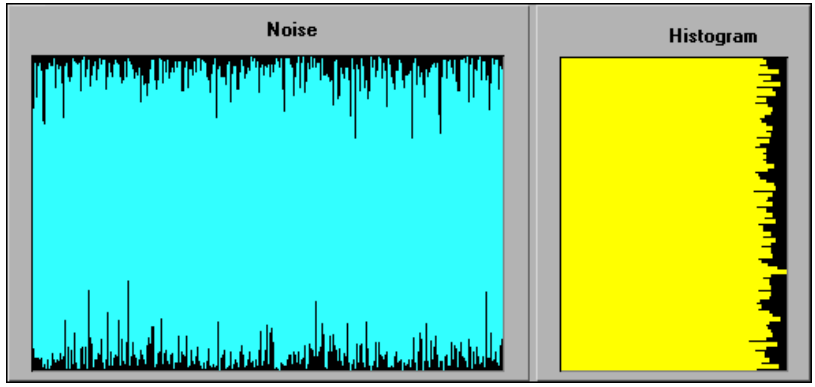


Figure 17-5. Uniform White Noise

For Gaussian white noise, the PDF of the amplitudes of the time domain samples is Gaussian. If uniform white noise is passed through a linear system, the resulting output will be Gaussian white noise. Figure 17-6 shows the distribution of the samples of Gaussian white noise.

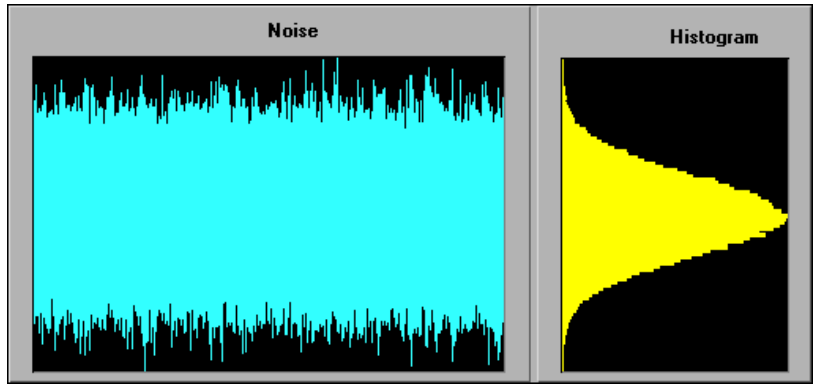


Figure 17-6. Gaussian White Noise

Periodic Random Noise (PRN) is a summation of sinusoidal signals with the same amplitudes but with random phases. It consists of all sine waves with frequencies that can be represented with an integral number of cycles in the requested number of samples. Since PRN contains only integral-cycle sinusoids, you do not need to window PRN before performing spectral analysis because PRN is self-windowing and therefore has no spectral leakage.

PRN does not have energy at all frequencies, as white noise does, but only at discrete frequencies which correspond to harmonics of a fundamental frequency which is equal to the sampling frequency divided by the number of samples. However, the level of noise at each of the discrete frequencies is the same.

PRN can be used to compute the frequency response of a linear system with one time record instead of averaging the frequency response over several time records, as you must for nonperiodic random noise sources. Figure 17-7 shows the spectrum of periodic random noise and the averaged spectra of white noise.

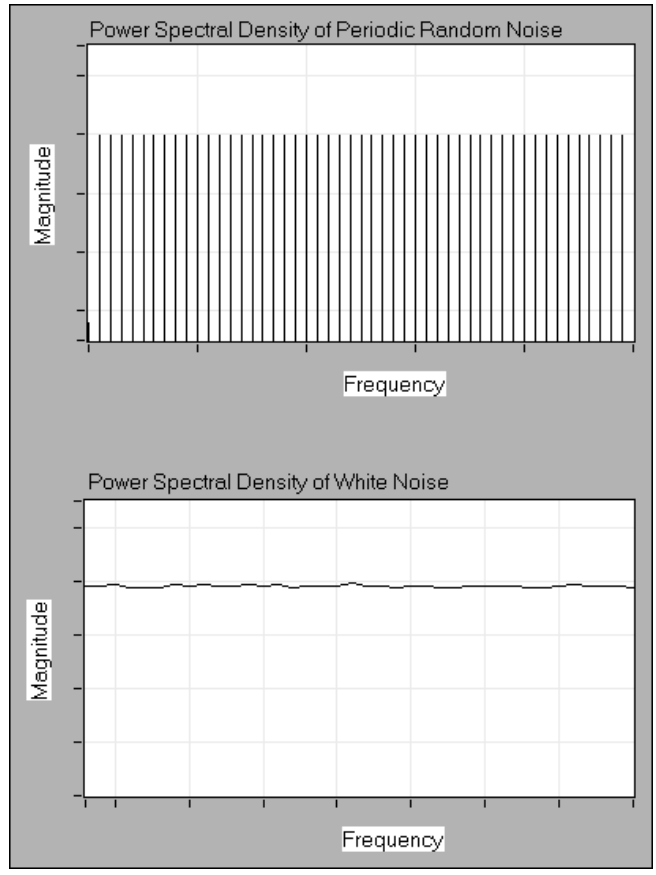


Figure 17-7. Spectral Representation of Periodic Random Noise and Averaged White Noise

Part IV

Instrument Control in LabVIEW

This part explains how to control instruments in LabVIEW.

Part IV, *Instrument Control in LabVIEW*, contains the following chapters:

- Chapter 18, *Using LabVIEW to Control Instruments*, introduces LabVIEW as a way to control instruments.
- Chapter 19, *Instrument Drivers in LabVIEW*, explains what instrument drivers are, where to find them, and how to use them.
- Chapter 20, *VISA in LabVIEW*, explains the basic concepts of VISA in LabVIEW.

18

Using LabVIEW to Control Instruments

This chapter explains how to communicate with instruments and introduces instrument drivers and VISA.

How Do You Use LabVIEW to Control Instruments?

In the simplest sense, instrument control is accomplished by sending commands and data between the instrument and the PC. With LabVIEW, you can use an instrument driver for your instrument, or you can write your own VIs using VISA.

An instrument driver is a set of software routines that control a programmable instrument. Each routine corresponds to a programmatic operation such as configuring, reading from, writing to, and triggering the instrument. Instrument drivers simplify instrument control and reduce test program development time by eliminating the need to learn the programming protocol for each instrument. The LabVIEW Instrument Library contains instrument drivers for a variety of programmable instrumentation, including GPIB, VXI, and RS-232/422 instruments. Because instrument driver VIs contain high-level functions with intuitive front panels, end users can quickly test and verify the remote capabilities of their instrument without the knowledge of device-specific syntax. The end user can easily create instrument control applications and systems by programmatically linking instrument driver VIs in the block diagram.

LabVIEW instrument drivers usually communicate with instruments using Virtual Instrument Software Architecture (VISA) functions. VISA is the underlying protocol used when talking to instruments. You can use VISA for many different instrument types, such as GPIB, Serial, VXI, and PXI. Once you learn how to communicate using VISA for one type of instrument, you do not have to learn a different way to communicate when you need to use another type of instrument. You do have to learn about the specific command set for the two instruments, but the method by which you send and receive the commands does not change.

When you begin to develop an instrument control application with LabVIEW, you have the option to use an instrument driver or to communicate directly using VISA.

Where Should You Go Next for Instrument Control?

LabVIEW has more than 700 instrument drivers from more than 50 vendors. A list is available on the National Instruments Developer Zone, `zone.ni.com/idnet` You should always check to see if there is an instrument driver available for your instrument. If you have an instrument not on the list, you can find a similar instrument on the list and easily modify its driver. Refer to Chapter 19, *Instrument Drivers in LabVIEW*, for more information about using instrument drivers.

If you cannot find an instrument driver for your instrument, refer to Chapter 20, *VISA in LabVIEW*, for more information about VISA.

19

Instrument Drivers in LabVIEW

This chapter describes what instrument drivers are, how to install and use instrument drivers from the Instrument Driver Library.

Installing Instrument Drivers

This section describes where to locate and install LabVIEW instrument drivers.

Where Can I Get Instrument Drivers?

Instrument drivers can be installed from an instrument driver CD or downloaded from the National Instruments Web site. You can download drivers using the Instrument Driver Network, available at `zone.ni.com/idnet`

If an instrument driver for your particular instrument does not exist, you can try the following:

- Use a driver for a similar instrument. Often similar instruments from the same manufacturer have similar if not identical command sets.
- Create a simple instrument driver.
- Develop a complete, fully functional instrument driver. To develop a National Instruments quality driver, you can download Application Note 006, *Developing a LabVIEW Instrument Driver*, from our web site. This application note will help you to develop a complete instrument driver.

Where Should I Install My LabVIEW Instrument Driver?

Instrument drivers should be installed as a subdirectory of your `labview\instr.lib`. For example, the HP34401A instrument driver, included with LabVIEW, is installed in the `labview\instr.lib\hp34401a` directory.

Within this directory you will find the menu files and VI libraries that make up an instrument driver. The menu files allow you to view your instrument driver VIs on the **Functions** palette. The VI libraries contain the instrument driver VIs.

Organization of Instrument Drivers

Figure 19-1 shows the organization of a typical instrument driver. This model applies to numerous instrument drivers.

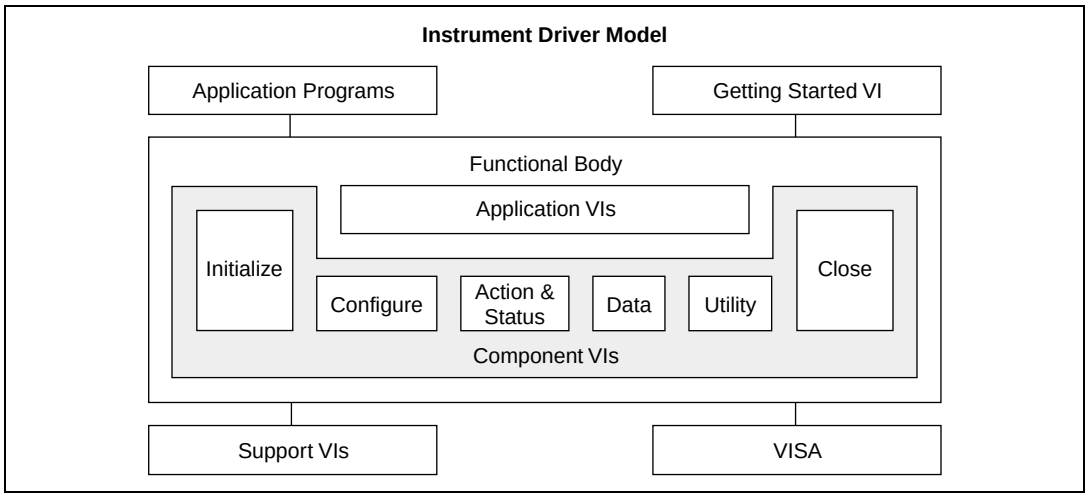


Figure 19-1. Instrument Driver Model

The Getting Started VIs are simple application VIs you can use without modification. Run this VI to verify communication with your instrument. Typically you only need to change the instrument address before running the VI from the front panel. However, many also require you to specify the VISA Resource name, for example, GPIB::2. Refer to Chapter 20, *VISA in LabVIEW*, for more information about VISA Resource names.

The Getting Started VI generally consists of three sub-VIs: the Initialize VI, the Application VI, and the Close VI.

The Application VIs are high-level examples of grouping together low-level component functions to execute a typical programmatic instrument operation. For example, the Application VIs might include VIs to control the most commonly used instrument configurations and measurements. These VIs serve as a code example to execute a common operation such as configuring the instrument, triggering, and taking a measurement.

Because the application VIs are standard VIs with icons and connector panes, you can call them from any high-level application when you want a single, measurement-oriented interface to the driver. For many users, the

application VIs are the only instrument driver VIs needed for instrument control. The HP34401A Example VI, shown in Figure 19-2, demonstrates an application VI front panel and block diagram.

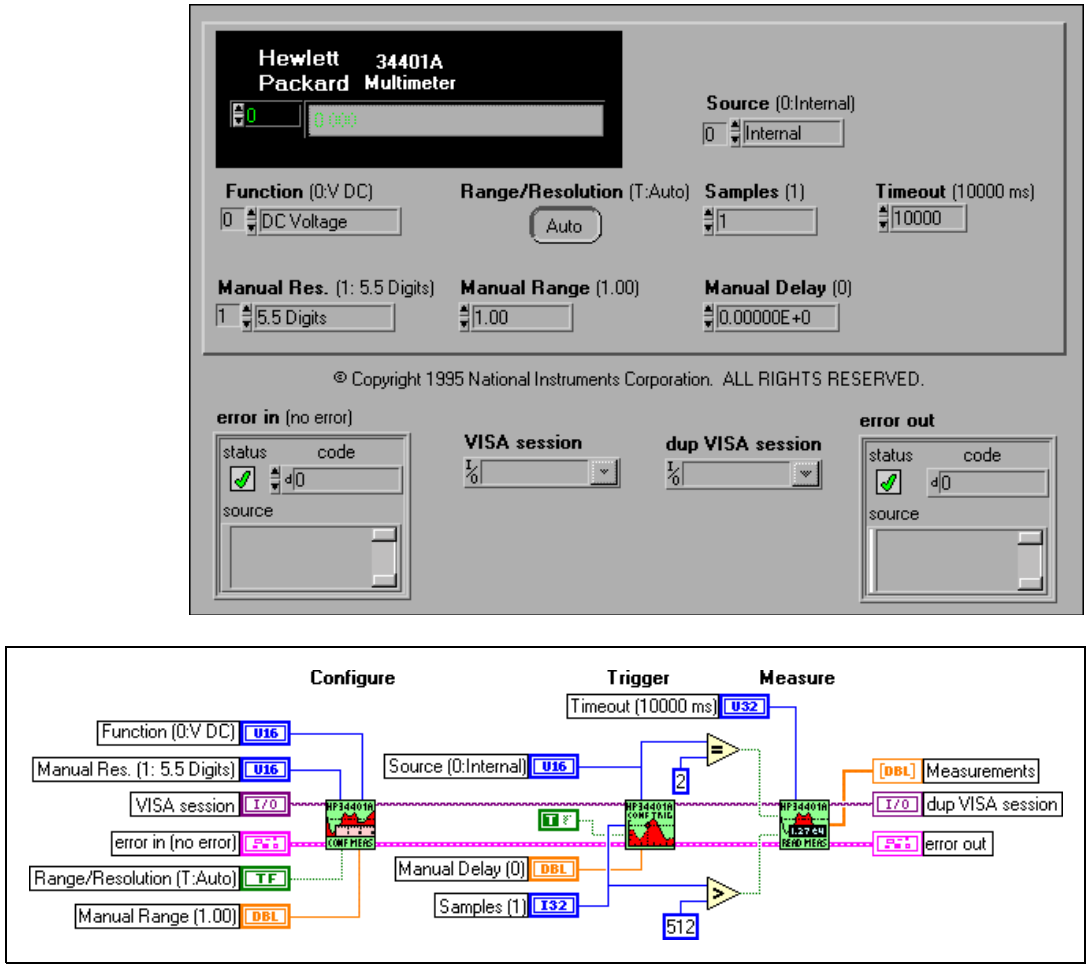


Figure 19-2. HP34401A Example

The Initialize VI, the first instrument driver VI called, establishes communication with the instrument. Additionally, it can perform any necessary actions to place the instrument either in its default power on state or in some other specific state. Generally, the Initialize VI only needs to be called once at the beginning of your application program.

The Configuration VIs are a collection of software routines that configure the instrument to perform the desired operation. There may be numerous Configuration VIs, depending on the particular instrument. After these VIs are called, the instrument is ready to take measurements or stimulate a system.

The *action/status* category contains two types of VIs. Action VIs initiate or terminate test and measurement operations. These operations can include arming the trigger system or generating a stimulus. These VIs are different from the Configuration VIs because they do not change the instrument settings, but only order the instrument to carry out an action based on its current configuration. The Status VIs obtain the current status of the instrument or the status of pending operations.

The Data VIs transfer data to or from the instrument. Examples include VIs for reading a measured value or waveform from a measurement instrument and VIs for downloading waveforms or digital patterns to a source instrument.

The Utility VIs perform a variety of operations that are auxiliary to the most often used instrument driver VIs. These VIs include the majority of the instrument driver template VIs such as reset, self-test, revision, error query, and error message, and might include other custom instrument driver VIs that perform operations such as calibration or storage and recall of setups.

The Close VI terminates the software connection to the instrument and frees up system resources. Generally, the Close VI only needs to be called once at the end of your application program or when you finish communication with your instrument. Make sure that for each successful call to the Initialize VI you have a matching Close VI. Otherwise you maintain unnecessary memory resources.

Note Application functions do not call Initialize and Close. To run an application function, you first must run the Initialize VI. The Getting Started VI calls Initialize and Close.

Kinds of Instrument Drivers

There are different kinds of instrument drivers. The difference is not as much in how you use them as in how they are implemented. The three kinds are usually called:

- LabVIEW instrument drivers
- VXI*plug&play* instrument drivers
- IVI drivers

The order here represents the evolution of instrument driver standards over the last several years, though all three kinds of drivers are viable and available today.

LabVIEW drivers are so called because they are written entirely with LabVIEW functions. The other two kinds are almost always written in C and have VI “wrappers” around each C function call. LabVIEW instrument drivers are easier to modify and debug than the other kinds of drivers, and they are easily converted from one computer hardware platform to another. However, many of these drivers are older and do not conform to any standard interface. For example, the functions you find in a driver for one brand of multimeter may be completely different from those found in a driver for a different multimeter.

The LabVIEW instrument driver library contains instrument drivers for a variety of programmable instruments that use GPIB, VXI or serial interfaces. You can use a library driver for your instrument as is. However, LabVIEW instrument drivers are distributed with their block diagram source code, so you can customize them for your specific application if need be.

The VXI*plug&play* consortium attempted to improve this situation by standardizing on VISA as the communications interface, as well as other details, such as where drivers would be installed. While the VXI*plug&play* standards admit drivers written entirely in LabVIEW, almost all VXI*plug&play* drivers are written in C and then converted for use in LabVIEW. To modify such a driver, you have to use a C-based development environment, such as LabWindows/CVI, and then convert the driver to LabVIEW again. The drivers also must be recompiled if you want to use them on another computer hardware platform. Refer to `www.vxipnp.org` for more information about the VXI*plug&play* consortium.

While the VXI*plug&play* standards did bring some consistency to instrument drivers, they did not address certain applications, such as production testing. The IVI Foundation was formed to bring even more standardization to instrument drivers. This time, instrument-specific programming interfaces were explicitly defined for the C language. This means that you can write a program that can work with any of several different brands of oscilloscope without needing special code for each model in your application. To change from one model to another just requires a change to configuration. The IVI Foundation also addressed other issues, such as simulation of missing instruments and performance. Refer to `www.ivifoundation.org` for more information about the IVI Foundation.

Since IVI drivers are C-based, they have most of the same issues as VXI*plug&play* drivers. However, they are the best solution if performance, interchangeability, and simulation are paramount. Refer to `www.ni.com/ivi` for more information about IVI. Refer Application Note 140, *Using IVI Drivers in LabVIEW*, for more information about using IVI in LabVIEW.

All three kinds of drivers can be found on `zone.ni.com/idnet`

Inputs and Outputs Common to Instrument Driver VIs

Now you can start using instrument driver VIs to build applications. Just as all instrument drivers share a common set of functions, they also share common inputs and outputs. This section covers these common parameters and how to use them.

Resource Name/Instrument Descriptor

Before you can communicate with an instrument, you need to open a communication link to the instrument with the Initialize Instrument Driver VI. After you are finished communicating with the instrument, you can call the Close Instrument Driver VI, and all references or resources for the instrument are closed. If you do not explicitly call the Close Instrument Driver VI, all references are closed when you close LabVIEW.

When you initialize an instrument, you need to know the Resource Name or Instrument Descriptor.

- *Resource*—VISA Alias or IVI Logical Name
- *Instrument Descriptor*—The exact name and location of a resource having a format:

  `Interface Type[board index]::Address::INSTR`.

  For example, `GPIB0::2::INSTR` is the instrument descriptor when using the first GPIB board to communicate with an instrument at device address `2`.

Use Measurement & Automation Explorer to determine what resources and instrument addresses are available. You can specify the VISA Alias for the Resource Name/Instrument Descriptor in the instrument driver VIs. Refer to the *Assigning VISA Aliases and IVI Logical Names* section in Chapter 3, *Installing and Configuring Your Measurement Hardware*, for more information about VISA Aliases.

Error In/Error Out Clusters

Error handling with instrument driver VIs is similar to error handling with other I/O VIs in LabVIEW. Each instrument driver VI contains **Error In** and **Error Out** terminals for passing error clusters from one VI to another. The error cluster contains a Boolean flag indicating whether an error has occurred, a number for the error code, and a string containing the location of the VI where the error occurred.

Each instrument driver VI is written so that when an error occurs previously (passed to the **Error In** terminal), the VI does not run. The error information is passed to the next VI through the **Error Out** terminal. You can find the Simple Error Handler VI on the **Functions»Time & Dialog** palette. This VI displays a dialog box if an error occurs and also looks up the error code to determine possible reasons for the error. You can use this VI at any time in your application to display any possible error conditions.

Verifying Communication with Your Instrument

Running the Getting Started VI Interactively

To verify communication with your instrument and test a typical programmatic instrument operation, first open the Getting Started VI. Look over each of the controls and set them appropriately. Generally, with the exception of the address field, the defaults for most controls are sufficient for your first run. You will need to set the address appropriately. On Windows, you can refer to Measurement & Automation Explorer for help if you do not know the address of your instrument. After running the VI, check to see that reasonable data was returned and an error was not reported in the error cluster. The most common reasons for the Getting Started VI to fail include the following:

- NI-VISA is not installed. If you did not choose this as an option during your LabVIEW installation, you must install it before running your Getting Started VI.
- The instrument address was incorrect. The Getting Started VI requires you to specify the correct address for your instrument. If you are not certain of your instrument address, use Measurement & Automation Explorer or the Find Resource function.
- The instrument driver does not support the exact model of instrument you are using. Double-check that the instrument driver supports the instrument model you are using.

Once you have verified basic communication with your instrument using the Getting Started VI, you probably want to customize instrument control for your needs. If your application needs are similar to the Getting Started VI, the simplest means of creating a customized VI is to save a copy of the Getting Started VI by selecting **File»Save As**. You can change the default values on the front panel by selecting **Operate»Make Current Values Default**. Block diagram changes might include changing the constants wired to the Application VI or other sub-VIs.

Verifying VISA Communication

If no VISA VIs appear to be working in LabVIEW, including instrument drivers, the first step to take is the VISA Find Resource VI. This VI runs without any other VISA VIs in the block diagram. If this VI produces strange errors such as nonstandard VISA errors, the problem is most likely that the wrong version of VISA is installed or that VISA is not installed correctly. If VISA Find Resource runs correctly, LabVIEW is working correctly with the VISA driver. The next step is to identify what sequence of VIs is producing the error in the LabVIEW program.

If it is a simple sequence of events that is producing the error, a good next step in debugging is to try the same sequence interactively with the VISAIC utility. It is generally a good idea to do initial program development interactively. If the interactive utility works successfully but the same sequence in LabVIEW does not, it is an indication that LabVIEW might have a problem interacting with the VISA driver. If the same sequence exhibits the same problem interactively in VISAIC it is possible that a problem exists with one of the drivers VISA is calling. You can use the interactive utilities for these drivers, such as IBIC for NI-488.2, to try to perform the equivalent operations. If the problems persist on this level, it is an indication that there might be a problem with the lower-level driver or its installation.

20

VISA in LabVIEW

This chapter is an overview of VISA in LabVIEW. It explains the basic concepts involved in programming instruments with VISA and gives examples demonstrating simple VISA concepts.

What Is VISA?

VISA is a standard I/O Application Programming Interface (API) for instrumentation programming. VISA can control VXI, GPIB, PXI, or serial instruments, making the appropriate driver calls depending on the type of instrument being used.

Types of Calls: Message-Based Communication versus Register-Based Communication

GPIB, serial, and some VXI instruments use message-based communication. Message-based instruments are programmed with high-level ASCII character strings. The instrument has a local processor that parses the command strings and sets the appropriate register bits to perform the desired functions. Message-based instruments are easy to program. To make things easier, SCPI standardizes the ASCII command strings used to program any instrument. All SCPI instruments with the defined function are programmed with the same commands. Instead of learning different command messages for each type of instrument from each manufacturer, you need to learn only one command set. The most common message-based functions are VISA Read, VISA Write, VISA Assert Trigger, VISA Clear, and VISA Read STB.

PXI and many VXI instruments use register-based communication. Register-based instruments are programmed at a low level using binary information that is directly written to the instrument control registers. Speed is the advantage of this type of communication because the instrument no longer needs to parse the command strings and convert the information to register level programming. Register-based instruments communicate literally at the level of direct hardware manipulation. The most common register-based functions are VISA In, VISA Out, VISA Move In, and VISA Move Out.

Writing a Simple VISA Application

For most simple instrument applications, only two VISA functions are needed: VISA Write and VISA Read.

The example shown in Figure 20-1 is very simple—just one VISA Write call and one VISA Read call. The instrument is specified using the VISA Resource Name Constant. The VISA Write function will check to see if a reference is already established with the specified instrument. If there is not an existing reference, a reference will automatically be opened. Then, the string `MEAS:DC?` is sent to the instrument. When reading from the instrument, you can simply wire the VISA Resource Name output from the VISA Write function to the VISA Read function to specify the desired instrument. You can then process and display the returned output from the VISA Read function as necessary for your measurement. The VISA Read is followed by the Simple Error Handler VI to process any errors that might have occurred with the VISA functions.

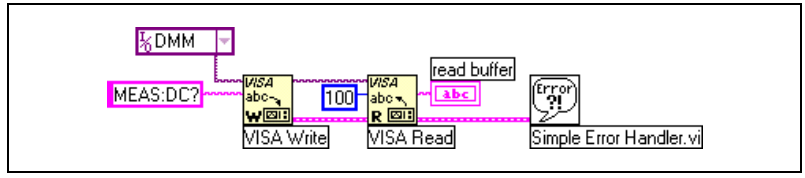


Figure 20-1. VISA Example

Using VISA Properties

VISA resources have a variety of properties (attributes) with values that can be read or set in a program. This section describes how to use VISA properties.

Using the Property Node

Property nodes are used to read or set the values of VISA properties. The property node is shown in Figure 20-2.

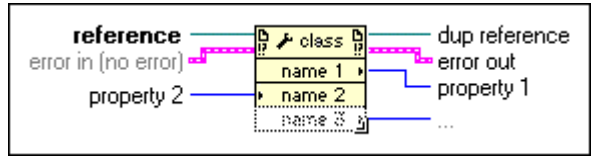


Figure 20-2. Property Node

Note The property node is a generic node that you also can use to set ActiveX and VI Server properties.

After placing the property node on the block diagram, wire a VISA Session to the reference input terminal of the property node.

The property node contains a single property terminal when it is initially placed on the block diagram. However, it can be resized to contain as many terminals as necessary. The initial terminal on the VISA property node is a read terminal. This means that the value of the property selected in that terminal will be read. This is indicated by the small arrow pointing to the right at the right edge of the terminal. Many terminals can be changed individually from a read terminal to a write terminal by right-clicking the property you wish to change.

Note Some properties are read only or write only. Their values cannot be set.

To select the property in each terminal of the property node, click on the property node terminal. This provides a list of all the possible properties that can be set in the program. The number of different properties shown under the Select Item choice of the VISA Property Node can be limited by changing the VISA Class of the property node.

To change the VISA class, right-click the VISA property node and select **VISA Class**. Several different classes can be selected under this option besides the default INSTR class which encompasses all possible VISA properties. These classes limit the properties displayed to those related to that selected class instead of all the VISA properties. Once a session is connected to the **Session** input terminal of the property node, the VISA Class is set to the class associated with that session.

Initially, the VISA properties will be somewhat unfamiliar. Refer to the *LabVIEW Help*, available by selecting **Help»Contents and Index**, for more information about the properties. Brief descriptions of individual properties are also available in the simple help window. To get a brief description of a specific property, select the property in one of the terminals of a property node and then open the **Context Help** window. The **Context Help** window is shown for the VXI LA property in Figure 20-3.

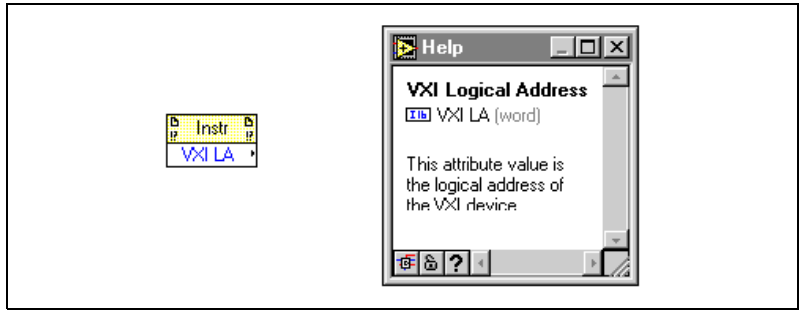


Figure 20-3. VXI Logical Address Property

Note that the help window shows the specific variable type of the property and gives a brief description of what the property does. In cases where it is not clear what variable type to use for reading or writing a property, remember that right-clicking a property node and selecting **Create Constant**, **Create Control**, or **Create Indicator** from the shortcut menu automatically selects the appropriate variable type.

There are two basic types of VISA properties: global properties and local properties. Global properties are specific to a resource while local properties are specific to a session. For example, the VXI LA property is a global property. It applies to all of the sessions that are open to that resource. A local property is a property that can be different for individual sessions to a specific resource. An example of a local property is the timeout value. Some of the common properties for each resource type are shown in the following lists.

Serial

Serial Baud Rate—The baud rate for the serial port.

Serial Data Bits—The number of data bits used for serial transmissions.

Serial Parity—The parity used for serial transmissions.

Serial Stop Bits—The number of stop bits used for serial transmissions.

GPIB

GPIB Readdressing—Specifies if the device should be readdressed before every write operation.

GPIB Unaddressing—Specifies if the device should be unaddressed after read and write operations.

VXI

Mainframe Logical Address—The lowest logical address of a device in the same chassis with the resource.

Manufacturer Identification—The manufacturer ID number from the device configuration registers.

Model Code—The model code of the device from the device configuration registers.

Slot—The slot in the chassis that the device resides in.

VXI Logical Address—The logical address of the device.

VXI Memory Address Space—The VXI address space used by the resource.

VXI Memory Address Base—The base address of the memory region used by the resource.

VXI Memory Address Size—The size of memory region used by the resource.

There are many other properties besides those listed here. There are also properties that are not specific to a certain interface type. The **timeout** property, which is the timeout used in message-based I/O operations, is an example of such a property.

The *LabVIEW Help*, available by selecting **Help»Contents and Index**, shows which type of interfaces the property applies to, whether the property is local or global, its data type, and what the valid range of values are for the property. It also shows related items and gives a detailed description of the property.

Using VISA Events

An event is a means of VISA communication between a resource and its applications. It is a way for the resource to notify the application that some condition has occurred that requires action by the application. Examples of different events are included in the following sections.

Types of Events

Handling GPIB SRQ Events Example

Figure 20-4 shows a block diagram for how to handle GPIB Service Request (SRQ) events with VISA.

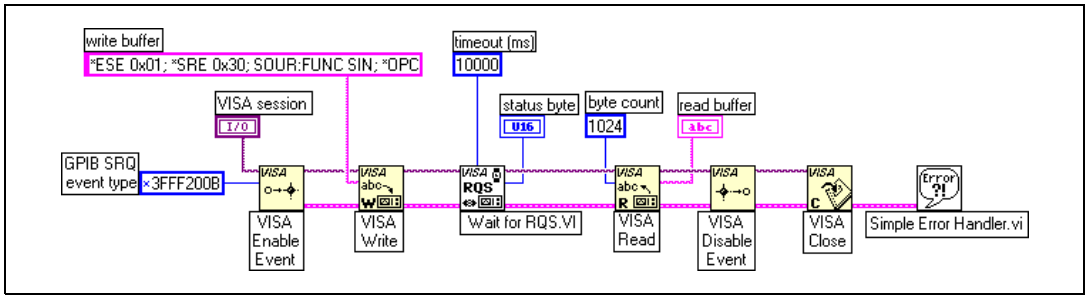


Figure 20-4. SRQ Events Block Diagram

The VI enables service request events and then writes a command string to the instrument. The instrument is expected to respond with an SRQ when it has processed the string. The Wait on Event Async VI waits for up to 10 seconds for the SRQ event to occur. After the SRQ occurs, the instrument status byte is read with the Read Status Byte VI. The status byte must be read after GPIB SRQ events occur, or later SRQ events may not be received properly. Finally the response is read from the instrument and displayed. The Wait on Event Async is different from the regular Wait on Event VI in that it continuously calls Wait on Event with a timeout of zero to poll for the event. This frees up time for other parallel segments of the program to run while waiting for the event.

Advanced VISA

Opening a VISA Session

As discussed previously, when you call VISA Read and/or VISA Write, LabVIEW checks to see if a reference has been opened for the instrument specified. If a reference is already open, the VISA call uses that reference. If there is not an open reference, VISA automatically opens one. You can choose to explicitly open references to your instruments using VISA Open. The VISA Open function is shown in Figure 20-5.

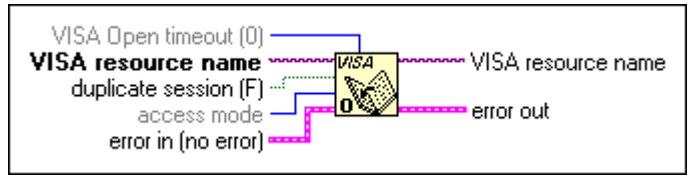


Figure 20-5. VISA Open Function

Closing a VISA Session

An open session to a VISA resource uses system resources within the computer. To properly end a VISA program, all of the opened VISA sessions should be closed. To do this, use the VISA Close VI, shown in Figure 20-6.

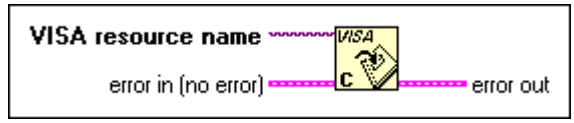


Figure 20-6. VISA Close VI

The VISA session input to the VISA Close VI is the session to be closed. This session originally comes from the output session terminal of the VISA Open VI or any other VISA VI. If a session is not closed when a VI is run, it remains open.

Note If a VI is aborted when you are debugging a VI, the VISA session is not closed automatically. You can use the Open VISA Session Monitor VI, available in `vi.lib\Utility`, to assist in closing such sessions.

Locking

VISA introduces locks for access control of resources. With VISA, applications can simultaneously open multiple sessions to the same resource and can access the resource through these different sessions concurrently. In some cases, applications accessing a resource must restrict other sessions from accessing that resource. For example, an application may need to execute a write and a read operation as a single step so that no other operations take place between the write and read operations. The application can lock the resource before invoking the write operation and unlock it after the read operation, to execute them as a single step.

The VISA locking mechanism enforces arbitration of accesses to resources on an individual basis. If a session locks a resource, operations invoked by other sessions are serviced or returned with a locking error, depending on the operation and the type of lock used.

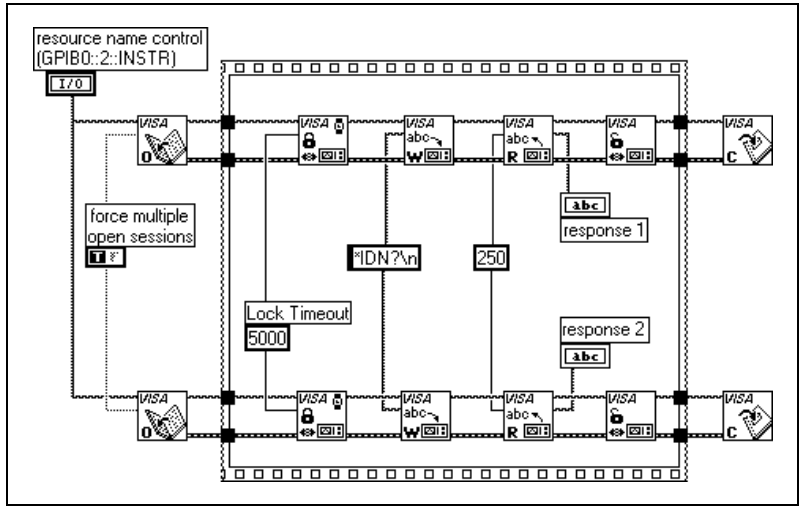


Figure 20-7. VISA Lock Async VI

The VISA Lock Async VI, shown in Figure 20-7 and available on the **Functions»Instrument I/O»VISA»VISA Advanced** palette, opens two sessions to the same resource and performs a query on each of them. This example uses a lock to guarantee that the write/read pairs happen in the expected order and are not interleaved. The lock is released after the write/read sequence is complete, thus allowing the other session's execution path to continue. There is no guarantee as to which session will receive the lock first. Locking is useful in cases where more than one application may be accessing the same resource, or where multiple modules may open multiple sessions to the same resource even within a single application.

Shared Locking

There might be cases where you want to lock access to a resource but selectively share this access. Figure 20-8 shows the Lock VI in complex help view.

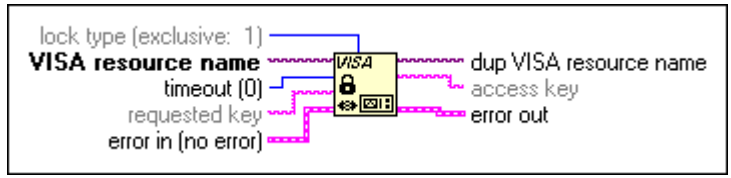


Figure 20-8. VISA Lock Function Icon

Lock type defaults to exclusive, but you can set it to shared. You can then wire a string to **requested key** to be the password needed for another application to access the resource. However, the VI assigns one in **access key** if you do not ask for one. You can then use this key to access a locked resource.

String Manipulation Techniques

You have learned that most LabVIEW instrument driver problems can be solved without modifying the instrument driver code. However, in a few situations, code modification is necessary. This section describes some fundamental methods of instrument communication and introduces you to some commonly-used functions in LabVIEW instrument drivers.

How Instruments Communicate

Recall that the two main types of instrument communication are *message-based* and *register-based*.

No standards exist for register-based instrument communication. Each device operates independently, and the instrument manual is the best resource for learning how to program it.

Building Strings

When communicating with a message-based instrument, you must format and build the correct command strings, or the instrument will not perform the appropriate operation or return a response.

Typically, a command string is a combination of text and numeric values. Because the instruments require that the entire command string be text, you

must find a way to convert the numeric values to text and append them onto the rest of the command string. The Format Into String function can be used to build the command strings that you need to send to your instrument. This function allows you to take an initial string and append other strings or numeric data types to it.

Removing Headers

This section describes how you read the information returned from a message-based instrument. The instrument user manual describes what header and trailer information to expect from each data transfer.

Most instruments return data with extra information attached as a header or a trailer. The header usually contains information such as the number of data points returned or the instrument settings. In some cases, some trailer information containing units or other instrument settings is at the end of the data string. You must first remove the header and trailer information before you can display or analyze the returned data.

Consider the example shown above in which the string contains a 6-byte header, the data points, and a 2-byte trailer. You can use the String Subset function, available on the **Functions»String** palette to remove the header as shown above. String Subset returns a substring of the string input beginning at offset and containing length number of characters. Length and offset must be scalar. The offset of `6` above removes the header of `CURVE<space>`, and the length of the string subset is the total length of the instrument response string minus the length of the header.

Note For strings, the offset value starts at zero, just as the index for arrays starts at zero.

Waveform Transfers

In addition to header and trailer information, instrument data can be returned in various formats. The instrument user manual describes what formats are available and how you can convert each one to usable data. The formats discussed in this section include ASCII, 1-byte binary, and 2-byte binary formats.

ASCII Waveforms

If data from an instrument is returned in ASCII format, you can view it as a character string. However, if numeric manipulation of the data is necessary or you need to graph the data, you must first convert the string data to numeric data. As an example, consider a waveform composed of 1,024 points, each point having a value between 0 and 255. Using ASCII

encoding, you would need a maximum of 4 bytes to represent each data value (a maximum of 3 bytes for the value and 1 byte for the separator, such as a comma). You would need a maximum of 4,096 bytes (4 bytes * 1,024) plus any header and trailer bytes to represent the waveform as an ASCII string.

Another example is shown above with the header information already removed —`10.2, 18.3, 8.91, 1.0, 5.`5. You can use the Extract Numbers VI to convert that ASCII string to a numeric array. The Extract Numbers VI is an example VI available in the *Search Examples Help*. It finds all numbers in the given ASCII string and puts them into a **Single Precision Array** of numbers. A non-numeric delimiter such as a comma, colon, line feed, etc. is assumed and all formats listed above are recognized. Any characters at the beginning of the ASCII string are ignored, so you do not need to strip off header information when you use the Extract Numbers VI. The example above shows how you can extract the five values from the string and place them into an array of numbers. You can now plot those values or use them in data analysis algorithms.

1-Byte Binary Waveforms

Some instruments do not have the option of sending data in ASCII format, or for performance reasons all waveform data is sent in binary format. No standard binary format exists, so you need to find out exactly how the data values are stored from the instrument user manual. One common binary format is 1-byte binary. With this type of data encoding, each data value is converted to an 8-bit binary value before being sent.

When you read 1-byte binary data from the bus, it is returned as a character string. However, the characters do not appear to have any correspondence to the expected data. The binary numbers are interpreted as ASCII character values and the corresponding characters are displayed. Some examples are shown in the table above. If a value of 65 is sent as one data value, you would read the character `A` from the bus. Notice that for a value of 13, there is no printable ASCII character; 13 corresponds to an invisible carriage return character.

You can display these invisible characters in a string indicator in LabVIEW if you view the **‘\’Codes Display**. For example, the top string indicator shown above displays the values in the default **Normal Display** and you cannot see the third character. However, if you right-click that front panel string indicator and choose **‘\’ Codes Display** from the shortcut menu, you will see the carriage return character as a `\r`, as shown above in the second string indicator.

By enabling the **'\' Codes Display** on the LabVIEW string, you can now see characters that may have been invisible before. However, you still must convert the binary string to a numeric array to graph or do mathematical operations with the data.

Suppose the instrument sends a binary string containing 1,024 1-byte binary encoded values as shown above. That waveform would require only 1,024 bytes plus any header and trailer information. Using binary encoding, you would need only 1 byte to represent the data value, assuming each value was an unsigned 8-bit integer.

Converting the binary string shown above to a numeric array is a little more complex than converting an ASCII string. You must first remove all header and trailer information using the String Subset function as previously described. Then you convert the remaining data string to an array of integers using the String To Byte Array function, available on the **Functions»String»String/Array/Path Conversion** palette.

Note Using binary data, it is better to extract the data using the data size rather than searching for the first character of the trailer information, because it is possible that the search character might also be contained as part of the binary values.

2-Byte Binary Waveforms

A third data format is 2-byte binary. When data is in 2-byte binary format, it is binary encoded and sent as ASCII characters just like the 1-byte binary. However, 16 bits of data (or two ASCII characters) represent each data value. Although this format uses twice as much space as the 1-byte binary data, it is still more efficiently packed than ASCII formatted data.

As an example, consider an oscilloscope that transfers waveform data in binary notation. For this example, the waveform consists of 1,024 data points where each value is a 2-byte signed integer. Therefore, the entire waveform requires 2,048 bytes plus a 5-byte header and a 2-byte trailer. Remove the 5-byte header and take the next 2,048 bytes. Then use the **Type Cast** function, available on the **Fuctions»Advanced»Data Manipulation** palette, to convert the waveform string to an array of 16-bit integers.

Byte Order

When data is transferred in 2-byte binary format, it is important to know the order of the bytes you receive. The 2-byte combination `qH` has the corresponding integer value of 29,000, but the opposite byte order of `Hq` has the corresponding integer value of 18,545.

If you receive the high byte first, you must reverse the order of the bytes before converting them to an integer value. Consider the example shown above. This 2-byte binary waveform data is the same size and contains the same header and trailer information as shown on the previous slide, but the data is sent with the high byte first. The diagram above shows that you still strip off the header and use the Type Cast function to convert the binary string to 16-bit integers. However, the **Swap Bytes** function, available on the **Functions»Advanced»Data Manipulation** palette, is needed to swap the high-order 8 bits and the low-order 8 bits for every element.

A

Types of Instruments

When you use a personal computer to automate your test system, you are not limited to the type of instrument that you can control. You can mix and match instruments from various categories, such as serial, GPIB, VXI, PXI, and computer-based instruments along with instruments that are not discussed, such as image acquisition, motion control, Ethernet, SCXI, CAMAC, parallel port, CAN, FieldBus, and other devices.

The things to be aware of with PC control of instrumentation are:

- What type of connector (pinouts) is on the instrument
- What kind of cable is needed (null-modem, number of pins, male/female)
- What electrical properties are involved (signal levels, grounding, cable length restrictions)
- What communication protocols are used (ASCII commands, binary commands, data format)
- What kind of software drivers are available

This appendix discusses the most common categories of instruments. You can use VISA to program or control all of these types of instruments.

Serial Port Communication

Serial communication is a popular means of transmitting data between a computer and a peripheral device such as a programmable instrument or another computer. Serial communication uses a transmitter to send data, one bit at a time, over a single communication line to a receiver. Use this method when data transfer rates are low or you must transfer data over long distances. Serial communication is popular because most computers have one or more serial ports, so no extra hardware is needed other than a cable to connect your instrument to the computer (or two computers to each other).

You must specify four parameters for serial communication: the *baud rate* of the transmission, the number of *data bits* encoding a character, the sense of the optional *parity* bit, and the number of *stop bits*. Each transmitted

character is packaged in a character frame that consists of a single start bit followed by the data bits.

Baud rate is a measure of how fast data moves between instruments that use serial communication.

A *start bit* signals the beginning of each character frame.

Data bits are transmitted "upside down and backwards." That is, inverted logic is used and the order of transmission is from least significant bit (LSB) to most significant bit (MSB). To interpret the data bits in a character frame, you must read from right to left, and read 1 for negative voltage and 0 for positive voltage.

An optional *parity bit* follows the data bits in the character frame. The parity bit, if present, also follows inverted logic. This bit is included as a simple means of error checking. You specify ahead of time whether the parity of the transmission is to be even or odd. If the parity is chosen to be odd, the transmitter then sets the parity bit in such a way as to make an odd number of 1's among the data bits and the parity bit.

The last part of a character frame consists of 1, 1.5, or 2 *stop bits*. These bits are always represented by a negative voltage. If no further characters are transmitted, the line stays in the negative (MARK) condition. The transmission of the next character frame, if any, begins with a start bit of positive (SPACE) voltage.

How Fast Can I Transmit Data over the Serial Port?

You can calculate the maximum transmission rate in characters per second for a given communication setting by dividing the baud rate by the bits per character frame.

Serial Hardware Overview

There are many different kinds (recommended standards) of serial port communication. The most common are the following:

- *RS-232* (ANSI/EIA-232) is used for many purposes, such as connecting a mouse, printer, or modem, as well as industrial instrumentation. Because of improvements in the line drivers and cables, applications often increase the performance of RS-232 beyond the distance and speed listed in the standard. RS-232 is limited to point-to-point connections between PC serial ports and devices.

- *RS-422* (AIA RS-422A Standard) uses a differential electrical signal, as opposed to the unbalanced (single ended) signals referenced to ground with RS-232. Differential transmission, which uses two lines each to transmit and receive signals, results in greater noise immunity and longer transmission distances as compared to RS-232. The greater noise immunity and transmission distance are big advantages in industrial environments.
- *RS-485* (EIA-485 Standard) is an improvement over RS-422 because it allows you to connect multiple devices (up to 32) to a single port and defines the electrical characteristics necessary to ensure adequate signal voltages under maximum load. With this enhanced multidrop capability, you can create networks of devices connected to a single RS-485 serial port. The noise immunity and multidrop capability make RS-485 the serial connection of choice in industrial applications requiring many distributed devices networked to a PC or other controller for data collection, HMI, and other operations.

Your System

If you have a serial device in your system, you first must obtain the pinout for that device and make sure you have the correct cable to connect it to your computer. Determine if the device is DCE or DTE and what settings it uses to communicate—baud rate, data bits, stop bits, parity, or handshaking (flow control).

GPIB Communications

GPIB instruments offer test and manufacturing engineers the widest selection of vendors and instruments for general-purpose to specialized vertical market test applications. GPIB instruments have traditionally been used as stand-alone benchtop instruments where measurements are taken by hand.

Controllers, Talkers, and Listeners

To determine which device has active control of the bus, devices are categorized as Controllers, Talkers, or Listeners and each device has a unique GPIB primary address between 0 and 30. The *Controller* defines the communication links, responds to devices requesting service, sends GPIB commands, and passes/receives control of the bus. *Talkers* are instructed by the Controller to talk and place data on the GPIB. Only one device at a time can be addressed to talk. *Listeners* are addressed by the Controller to listen and read data from the GPIB. Several devices can be addressed to listen.

Hardware Specifications

The GPIB is a digital, 24-conductor parallel bus. It consists of eight data lines (DIO 1-8), five bus management lines (EOI, IFC, SRQ, ATN, REN), three handshake lines (DAV, NRFD, NDAC), and eight ground lines. The GPIB uses an eight-bit parallel, byte-serial, asynchronous data transfer scheme. This means that whole bytes are sequentially handshaked across the bus at a speed that the slowest participant in the transfer determines. Because the unit of data on the GPIB is a byte (eight bits), the messages transferred are frequently encoded as ASCII character strings.

Additional electrical specifications allow data to be transferred across the GPIB at the maximum rate of 1 MB/sec because the GPIB is a transmission line system. These specifications are:

- A maximum separation of 4 m between any two devices and an average separation of 2 m over the entire bus.
- A maximum cable length of 20 m.
- A maximum of 15 devices connected to each bus with at least two-thirds of the devices powered on.

If you exceed any of these limits, you can use additional hardware to extend the bus cable lengths or expand the number of devices allowed.

Faster data rates can be obtained with HS488 devices and controllers. HS488 is an extension to GPIB and is supported by most National Instruments controllers.

VXI (VME eXtensions for Instrumentation)

VXI defines a standard communication protocol to certain devices. Through this interface, you can use common ASCII commands to control the instruments, just as with GPIB.

The VXIbus specification is an extension of the VMEbus (IEEE 1014) specification. As an electromechanical superset of the VMEbus, the VXIbus uses the same backplane connectors as VME, the same board sizes, and the same signals defined in the VMEbus specification. The VXIbus adds two board sizes, changes module width, and defines additional signals on the backplane.

VXI Hardware Components

A VXI system consists of a mainframe, a controller, instruments, and cables. The VXI mainframe is the chassis, cage, or crate that contains the power supply, cooling system, backplane connections, and physical mounting for VXIbus modules. Mainframes come in four sizes (A, B, C, and D) which correspond to the largest-size board you can plug into the mainframe.

VXI Configurations

You can use VXI in a variety of ways. You can integrate VXI into a system alongside other GPIB instruments, or you can build a system using only VXI instruments. Each system configuration has the following unique benefits:

- Embedded Controllers
  - Highest performance, smallest size
  - Direct access to VXIbus/fast interrupt response
- MXI, Multisystem eXtension Interface
  - Embedded performance with desktop computers
  - Use remote PCs to control VXI systems
  - MITE/DMA—23 Mbytes/s block transfers
- GPIB-to-VXI Translators
  - Control VXI mainframe with IEEE 488

The first configuration embeds a custom VXI computer directly inside the mainframe. Using this configuration, you can take full advantage of the high-performance capabilities of VXI because your computer can communicate directly with the VXI backplane.

The second configuration combines the performance benefits of a custom embedded computer with the flexibility of general-purpose desktop computers. With this configuration, you use a high-speed MXIbus link to connect an external computer directly to the VXI backplane.

The third configuration consists of one or more VXI mainframes linked to an external computer through GPIB. You can use this configuration to integrate VXI gradually into existing GPIB systems and to program VXI instruments using existing GPIB software.

PXI Modular Instrumentation

The new modular instrumentation system based on PCI eXtensions for Instrumentation (PXI) delivers a PC-based, high-performance measurement system.

PXI is completely compatible with CompactPCI and incorporates the advanced timing and triggering features associated with VXI. PXI fills the gap between low-cost desktop PC solutions and high-end VXI and GPIB solutions by combining the industry standards of Windows, PCI, CompactPCI, and VXI.

You design a PXI system by selecting everything, including the controller (an embedded Pentium class or higher computer and peripherals), the chassis, and the modules. PXI modules can be anything from analog-to-digital, digital-to-analog, digital I/O, and multifunction input/output boards to image acquisition, motion control, and instruments like oscilloscopes, multimeters, serial data analyzers, and other custom instruments.

Computer-Based Instruments

Computer-based instruments are made for several different platforms including PCMCIA (laptops), PCI (desktop computers), and PXI.

Computer-based instruments are an example of virtual instruments that consist of a PC-based instrument module, a computer, and application software. Traditional instruments are stand-alone instruments, where the functionality of the instrument is encapsulated within a “black box.” Because digitizer and packaging technology continues to evolve, today we have PC Card (PCMCIA) form factor instruments that give you the same functionality as a standalone instrument.

The computer-based instrument can take advantage of the processing power of the PC, expandable memory, high-resolution display options, and enterprise-wide connectivity to the corporate or world-wide Internet. You can measure voltage, current, and resistance using a computer-based instrument or expand the capabilities of your virtual instrument through application software. You can create a data logger and automatically analyze your acquired data. You can also generate reports on the fly. While you are making measurements, you can analyze and present information to make real-world decisions.

With application software, you can customize the capabilities of your virtual instrument to solve multiple test challenges. You also can upgrade the performance of your measurement system with evolving low-cost PC technology that offers a more economical instrumentation solution than purchasing an expensive, brand new, single-function stand-alone instrument.

B

Technical Support Resources

Web Support

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

NI Developer Zone

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

Customer Education

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

System Integration

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

Worldwide Support

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

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

Glossary

A

A/D	Analog-to-digital; analog/digital.
ADC	Analog-to-digital converter. An electronic device, often an integrated circuit, that converts an analog voltage to a digital number.
AI	Analog input.
AI device	Analog input device that has AI in its name, such as the NEC-AI-16E-4.
AIGND	Analog input ground pin on a DAQ device.
Am9513-based devices	These MIO devices do not have an E- in their names. These devices include the NB-MIO-16, NB-MIO-16X, NB-TIO-10, and NB-DMA2800 on the Macintosh; and the AT-MIO-16, AT-MIO-16F-5, AT-MIO-16X, AT-MIO-16D, and AT-MIO-64F-5 in Windows.
amplification	Type of signal conditioning that improves accuracy in the resulting digitized signal and reduces noise.
AMUX devices	*See* analog multiplexer.
analog multiplexer	Devices that increase the number of measurement channels while still using a single instrumentation amplifier. *Also called* AMUX devices.
analog trigger	Trigger that occurs at a user-selected level and slope on an incoming analog signal. You can set triggering to occur at a specified voltage on either an increasing or a decreasing signal (positive or negative slope).
ANSI	American National Standards Institute.
AO	Analog output.
Application Programming Interface (API)	Programming interface for controlling some software packages, such as Microsoft Visual SourceSafe.

B

Bessel filters

These filters have a maximally flat response in both magnitude and phase. The phase response in the passband, which is usually the region of interest, is nearly linear. Use Bessel filters to reduce nonlinear phase distortion inherent in all IIR filters.

Bessel function

The Bessel function of the first kind of order n $J_n(x)$ is defined by

$$J_n(x) = \left(\frac{1}{2}x\right)^n \sum_{k=0}^{\infty} \frac{\left(\frac{-1}{4}x^2\right)^k}{k!\Gamma(n+k+1)}$$

with $n = 0, 1, ...$

The Bessel function of the second kind of order n, $Y_n(x)$ is defined by

$$Y_n(x) = \frac{J_n(x)\cos(n\pi) - J_{-n}(x)}{\sin(n\pi)}$$

with $n = 0, 1, ...$

Bessel polynomial

The Bessel polynomial $P_n(x)$ of order n is defined by a recurrence relation

$$P_n(x) = P_{n-1}(x) + \frac{x^2}{4(n-1)^2 - 1} P_{n-2}(x)$$

for $n = 2, 3, ...$ where $P_0(x) = 1$ and $P_1(x) = 1 + x$

bipolar

Signal range that includes positive and negative values, for example, –5V to 5 V.

C

cascading

Process of extending the counting range of a counter chip by connecting to the next higher counter.

cast

To change the type descriptor of a data element without altering the memory image of the data.

channel clock

Clock that controls the time interval between individual channel sampling within a scan. Products with simultaneous sampling do not have this clock.

Chebyshev polynomial	The Chebyshev polynomial, for real numbers x, is given by $T_n(x) = \cos(n \arccos(x))$. This results in $T_0 = 1, T_1(x) = x, T_2(x) = 2x^2 - 1, T_3(x) = 4x^3 - 3x$ and so on.
circular-buffered I/O	Input/output operation that reads or writes more data points than can fit in the buffer. When LabVIEW reaches the end of the buffer, LabVIEW returns to the beginning of the buffer and continues to transfer data.
clock	Hardware component that controls timing for reading from or writing to groups.
code width	Smallest detectable change in an input voltage of a DAQ device.
column-major order	Way to organize the data in a 2D array by columns.
common-mode voltage	Any voltage present at the instrumentation amplifier inputs with respect to amplifier ground.
conditional retrieval	Method of triggering in which you simulate an analog trigger using software. *Also called* software triggering.
coupling	Manner in which a signal connects from one location to another.

D

D/A	Digital-to-analog.
DAC	Digital-to-analog converter. An electronic device, often an integrated circuit, that converts a digital number to a corresponding analog voltage or current.
DAQ Solution Wizard	Utility that guides you through specifying your DAQ application, and it provides a custom DAQ solution.
DAQ-STC	Data Acquisition System Timing Controller.
default input	Default value of a front panel control.
default load area	One of three parts of the SCXI EEPROM. The default load area is where LabVIEW automatically looks to load calibration constants the first time you access an SCXI module. When the module is shipped, this area contains a copy of the factory calibration constants. The other EEPROM areas are the factory area and the user area.

default setting	Default parameter value recorded in the driver. In many cases, the default input of a control is a certain value (often 0) that means *use the current default setting*. For example, the default input for a parameter can be *do not change current setting*, and the default setting can be *no AMUX-64T boards*. If you change the value of such a parameter, the new value becomes the new setting. You can set default settings for some parameters in the configuration utility.
device number	Slot number or board ID number assigned to the device when you configured it.
differential measurement system	A way to configure your device to read signals in which you do not need to connect either input to a fixed reference, such as a building ground.
digital trigger	TTL signal that you can use to start or stop a buffered data acquisition operation, such as buffered analog input or buffered analog output.
dimension	Size and structure of an array.
DIP	Dual Inline Package.
DMA	Direct Memory Access. A method by which you can transfer data to computer memory from a device or memory on the bus, (or from computer memory to a device), while the processor does something else. DMA is the fastest method of transferring data to or from computer memory.
down counter	Performs frequency division on an internal signal.

E

EEPROM	Electrically erased programmable read-only memory. Read-only memory that you can erase with an electrical signal and reprogram.
EISA	Extended Industry Standard Architecture.

F

factory area	One of three parts of the SCXI EEPROM. The factory area contains factory-set calibration constants. The area is read-only. The other EEPROM areas are the default load area and the user area.

FFT	Fast Fourier transform.
floating signal sources	Signal sources with voltage signals that are not connected to an absolute reference or system ground. Some common examples of floating signal sources are batteries, transformers, or thermocouples. *Also called* nonreferenced signal sources.

G

gain	Amplification or attenuation of a signal.
GATE input pin	Counter input pin that controls when counting in your application occurs.
grounded signal sources	Signal sources with voltage signals that are referenced to a system ground, such as a building ground. *Also called* referenced signal sources.

H

handshaked digital I/O	Type of digital acquisition/generation where a device or module accepts or transfers data after it receives a digital pulse. *Also called* latched digital I/O.
hardware triggering	Form of triggering where you set the start time of an acquisition and gather data at a known position in time relative to a trigger signal.
Hz	Hertz. Cycles per second.

I

immediate digital I/O	Type of digital acquisition/generation where LabVIEW updates the digital lines or port states immediately or returns the digital value of an input line. *Also called* nonlatched digital I/O.
input limits	Upper and lower voltage inputs for a channel. You must use a pair of numbers to express the input limits. The VIs can infer the input limits from the input range, input polarity, and input gain(s). Similarly, if you wire the input limits, range, and polarity, the VIs can infer the onboard gains when you do not use SCXI.

input range — Difference between the maximum and minimum voltages an analog input channel can measure at a gain of 1. The input range is a scalar value, not a pair of numbers. By itself, the input range does not uniquely determine the upper and lower voltage limits. An input range of 10 V could mean an upper limit of +10 V and a lower limit of 0 V or an upper limit of +5 V and a lower limit of –5 V.

The combination of input range, polarity, and gain determines the input limits of an analog input channel. For some products, jumpers set the input range and polarity, although you can program them for other products. Most products have programmable gains. When you use SCXI modules, you also need their gains to determine the input limits.

interrupt — Signal that indicates that the central processing unit should suspend its current task to service a designated activity.

interval scanning — Scanning method where there is a longer interval between scans than there is between individual channels that comprises a scan.

isolation — Type of signal conditioning in which you isolate the transducer signals from the computer for safety purposes. This protects you and your computer from large voltage spikes and makes sure the measurements from the DAQ device are not affected by differences in ground potentials.

L

Lab/1200 device — Devices, such as the Lab-PC-1200 and the DAQCard-1200, that use the 8253 type counter/timer chip.

latched digital I/O — Type of digital acquisition/generation where a device or module accepts or transfers data after it receives a digital pulse. *Also called* handshaked digital I/O.

Legacy MIO device — Devices, such as the AT-MIO-16, that typically are configured with jumpers and switches and are not Plug and Play compatible. They also use the 9513 type counter/timer chip.

limit settings — Maximum and minimum voltages of the analog signals you are measuring or generating.

linearization	Type of signal conditioning in which LabVIEW linearizes the voltage levels from transducers so the voltages can be scaled to measure physical phenomena.
LSB	Least Significant Bit.

M

MB	Megabytes of memory. 1 MB is equal to 1,024 KB.
multiplexed mode	SCXI operating mode in which analog input channels are multiplexed into one module output so that your cabled DAQ device can access the module's multiplexed output and the outputs on all other multiplexed modules in the chassis through the SCXI bus. *Also called* serial mode.
multiplexer	Set of semiconductor or electromechanical switches with a common output that can select one of a number of input signals and that you commonly use to increase the number of signals by one ADC measures.

N

nonlatched digital I/O	Type of digital acquisition/generation where LabVIEW updates the digital lines or port states immediately or returns the digital value of an input line. *Also called* immediate digital I/O.
non-referenced signal sources	Signal sources with voltage signals that are not connected to an absolute reference or system ground. Some common example of non-referenced signal sources are batteries, transformers, or thermocouples. *Also called* floating signal sources.
Non-referenced single-ended (NRSE) measurement system	All measurements are made with respect to a common reference, but the voltage at this reference can vary with respect to the measurement system ground.

O

onboard channels	Channels provided by the plug-in data acquisition board.
OUT output pin	Counter output pin where the counter can generate various TTL pulse waveforms.

output limits	Upper and lower voltage or current outputs for an analog output channel. The output limits determine the polarity and voltage reference settings for a board.

P

parallel mode	Type of SCXI operating mode in which the module sends each of its input channels directly to a separate analog input channel of the device to the module.
pattern generation	Type of handshaked (latched) digital I/O in which internal counters generate the handshaked signal, which in turn initiates a digital transfer. Because counters output digital pulses at a constant rate, you can generate and retrieve patterns at a constant rate because the handshaked signal is produced at a constant rate.
PCI	Peripheral Component Interconnect. An industry-standard, high-speed databus.
Plug and Play devices	Devices that do not require DIP switches or jumpers to configure resources on the devices. *Also called* switchless devices.
polling	Method of sequentially observing each I/O point or user interface control to determine if it is ready to receive data or request computer action.
postriggering	Technique to use on a data acquisition board to acquire a programmed number of samples after trigger conditions are met.
pretriggering	Technique to use on a data acquisition board to keep a continuous buffer filled with data so that when the trigger conditions are met, the sample includes the data leading up to the trigger condition.
pulse trains	Multiple pulses.
pulsed output	Form of counter signal generation by which produces a pulse output when a counter reaches a certain value.

R

read mark	Points to the scan at which a read operation begins. Analogous to a file I/O pointer, the read mark moves every time you read data from an input buffer. After the read is finished, the read mark points to the next unread scan. Because multiple buffers are possible, you need both the buffer number and the scan number to express the position of the read mark.
referenced signal sources	Signal sources with voltage signals that are referenced to a system ground, such as the Earth or a building ground. *Also called* grounded signal sources.
referenced single-ended (RSE) measurement system	All measurements are made with respect to a common reference or a ground. *Also called* a grounded measurement system.
RMS	Root Mean Square.
row-major order	Way to organize the data in a 2D array by rows.
RSE	Referenced Single-Ended.
RTD	Resistance Temperature Detector. A temperature-sensing device whose resistance increases with increases in temperature.
RTSI	Real-Time System Integration bus. The National Instruments timing bus that interconnects data acquisition devices directly by means of connectors on top of the devices for precise synchronization of functions.

S

S	Sample.
sampling period	Time interval between observations in a periodic sampling control system.
scan	One or more analog or digital input samples. Typically, the number of input samples in a scan equals the number of channels in the input group. For example, one pulse from the scan clock produces one scan that acquires one new sample from every analog input channel in the group.

scan clock	Clock that controls the time interval between scans. On products with interval scanning support (for example, the AT-MIO-16F-5), this clock gates the channel clock on and off. On products with simultaneous sampling (for example, the EISA-A2000), this clock times the track-and-hold circuitry.
scan rate	Number of times, (or scans), per second that LabVIEW acquires data from channels. For example, at a scan rate of 10 Hz, LabVIEW samples each channel in a group 10 times per second.
SCXI	Signal Conditioning eXtensions for Instrumentation. The National Instruments product line for conditional low-level signals within an external chassis near sensors, so only high-level signals in a noisy environment are sent to data acquisition boards.
sec	Seconds.
sensor	Device that produces a voltage or current output representative of a physical property being measured, such as speed, temperature, or flow.
settling time	Amount of time required for a voltage to reach its final value within specified limits.
signal conditioning	Manipulation of signals to prepare them for digitizing.
signal divider	Performing frequency division on an external signal.
simple-buffered I/O	Input/output operation that uses a single memory buffer big enough for all your data. LabVIEW transfers data into or out of this buffer at the specified rate, beginning at the start of the buffer and stopping at the end of the buffer. Use simple buffered I/O when you acquire small amounts of data relative to memory constraints.
software trigger	Programmed event that triggers an event, such as data acquisition.
software triggering	Method of triggering in which you simulate an analog trigger using software. *Also called* conditional retrieval.
SOURCE input pin	Counter input pin where the counter counts the signal transitions.
strain gauge	Thin conductor, which is attached to a material, that detects stress or vibrations in that material.

T

task — Timed I/O operation using a particular group. *See* task ID.

task ID — Number LabVIEW generates to identify the task at hand for the NI-DAQ drive the task at hand. The following table gives the function code definitions.

Functions Code	I/O Operation
1	analog input
2	analog output
3	digital port I/O
4	digital group I/O
5	counter/timer I/O

TC — Terminal count. The highest value of a counter.

timed digital I/O — Type of digital acquisition/generation where LabVIEW updates the digital lines or port states at a fixed rate. The timing is controlled either by a clock or by detection of a change in the pattern. Timed digital I/O is either finite or continuous. *Also called* pattern generation or pattern digital I/O.

TIO-ASIC — Timing I/O Application Specific Integrated Circuit. Found on 660*x* devices.

toggled output — Form of counter signal generation by which the output changes the state of the output signal from high to low or low to high when the counter reaches a certain value.

transducer excitation — Type of signal conditioning that uses external voltages and currents to excite the circuitry of a signal conditioning system into measuring physical phenomena.

trigger — Any event that causes or starts some form of data capture.

TTL — Transistor-Transistor Logic.

U

unipolar — Signal range that is either always positive or negative but never both. For example, 0 to 10 V, not –10 to 10 V.

update	One or more analog or digital output samples. Typically, the number of output samples in an update equals to the number of channels in the output group. For example, one pulse from the update clock produces one update that sends one new sample to every analog output channel in the group.
update rate	Number of output updates per second.
user area	One of the three parts of the SCXI EEPROM. The user area is where you store calibration constants that you calculate using the SCXI Cal Constants VI. If you want LabVIEW to load your constants automatically, you can put a copy of your constants in the default load area. The other EEPROM areas are the factory area and the default load area.
UUT	Unit under test.

V

V	Volts.
V_{ref}	Voltage reference.
VAC	Volts, Alternating Current.
VDC	Volts, Direct Current.
Virtual Instrument Software Architecture	Single interface library for controlling GPIB, VXI, RS-232, and other types of instruments.
VISA	*See* Virtual Instrument Software Architecture.

Index

Numbers

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W